

# 程序设计方法及在线实践导引

王 衍 王桂平 冯 睿 马雪英 编著

---

## 内容提要

本书以程序设计思想和方法为主线讲解 C/C++ 语言基础知识，并通过经典的程序设计竞赛题目为例题讲解基本的算法思想和应用问题。

本书内容分为五篇。第一篇介绍基础知识；第二篇引入 ACM/ICPC 程序设计竞赛题目的输入/输出方式，并介绍枚举、模拟、字符及字符串处理等基本算法和应用问题；第三篇介绍高精度计算、递归和搜索、排序和检索等较复杂的算法思想和应用问题；第四篇是课程设计；第五篇是附录。

本书作为教材可以适用于不同的教学对象和教学目标。学完第一篇的内容后，学生可以参加国家和浙江省高校计算机等级考试（二级 C）；学完第二篇后，学生具备了参加 ACM/ICPC 程序设计竞赛所需的基础知识；学完第三篇后，学生的程序设计和算法分析能力将得到进一步的提高，具备了参加全国计算机技术与软件专业技术资格考试（程序员级）中程序设计（C 语言）部分所需的基础知识。本书也可以作为程序设计竞赛爱好者的自学教材或培训教材。

# 前言

## 一、教材编写思路\*

程序设计基础课程是高校面向理工科专业开设的课程，采用 C、C++、Java 等语言讲授，课程名称一般为《程序设计基础》、《C 语言程序设计》、《C++程序设计》、《Java 程序设计》等等。课程的目的是培养学生基本的程序设计思想和方法。这些课程一般是在大学低年级开设，学生没有程序设计语言的基础，因此以往的教学方法多偏重于某种语言的语法知识教学，或者在讲述语言语法知识时少量地穿插讲解程序设计的基本思想和方法。然而我们在教学中体会到，一方面，由于语法体系的庞大与语法规则的严谨，不管是 54 课时，还是 72 课时，甚至 90 课时，都无法把某门语言的语法知识面面俱到地讲授。另一方面，语法内容讲得太多，对初学者来说，无疑是难以在短时间内理解和接受的。刚接触这门课的时候，学生的积极性大多都比较高，因为他们觉得学了这门课程后，可以自己编写程序，解决实际的问题。然而学生们很快发现，即使经过一个学期学究式的语法教学后，也只能编写一些很小的程序，这些小程序根本无法解决实际的问题。这极大地打击了学生学习后续相关课程的积极性。

另外，现在市面上现成的开发软件很多，如各种可视化开发工具，Delphi、VC、VC.NET、VB、PB 等。很多学生接触到这些软件后，热衷于开发一些简单的应用程序，而忽视了程序设计思想和方法、算法分析与设计能力和意识的培养，并对程序设计基础、数据结构、算法分析与设计等课程的认识陷入了一个误区。他们认为软件开发不过是拖动一些控件、编写简单的脚本而已，不需要理解程序设计思想和方法、不需要掌握数据结构知识，也不注重算法分析和设计能力的培养。

这些现象和问题引起了我们的思考：程序设计基础这些课程应该以什么为主线？是以程序设计语言语法知识的教学为主线？还是以程序设计思想和方法的培养为主线？我们认为，程序设计基础课程的主线应该以程序设计思想和方法的培养为主，以程序设计语言的教学为辅，原因有以下两点。

第一，语言语法的讲授应该以应用为导向，这些应用就具体体现了程序设计的思想和方法。针对大三、大四学生的调查表明，如果问他们程序设计基础这些课程学了什么？他们会回答学了指针、函数。但再具体一点地问，函数有什么用，指针有什么用，有些学生就不知道了。因此以程序设计语言语法知识为主线的教学方法无法让学生真正理解和掌握程序设计的思想和方法。而以程序设计思想和方法的培养为主线，既能在较大程度上避免枯燥的语法知识，也能引起学生的兴趣，从而接受和理解这些思想和方法。比如以数值型数据的处理为线索，就可以串起大部分 C/C++ 语言语法知识，而数值型数据的处理对学生来说是比较容易接受和理解的。

第二，程序设计基础课程中可以也必须向学生系统地讲授程序设计的思想和方法。以前的教学方法在讲授语言语法的同时，也会穿插讲一些程序设计的思想和方法，但对初学者来说，不经意的讲授往往不能引起重视。系统地讲授程序设计基础思想和方法，才是程序设计基础这些课程的主线。

举个例子，递归以前往往是放在函数这一章讲的，学生即使理解了递归函数的执行过程，掌握了递归函数定义方法，也难以明白在什么时候需要用到递归函数。其实递归是程序设计的重要思想，递归函数只不过是递归思想实现的手段。

再举个例子，以前在讲循环结构时，会例举很多使用一重循环、二重循环的例子。比如求  $x^2$

---

\*本文中关于程序设计基础课程教学改革的一些思想，已经发表在《计算机教育》第 22 期上，论文题目为《以在线实践为导向的程序设计课程教学新思路》，即参考文献[5]。

$x^2 + y^2 = 2000$  的正整数解。因为  $x$  和  $y$  都是正整数，因此  $x$  和  $y$  的取值范围只能是：1、2、...、44，其中 44 是小于等于  $\sqrt{2000}$  的最大整数。对于在这个范围内的所有  $(x, y)$  组合，都去判断一下是否满足  $x^2 + y^2 = 2000$ 。如果满足，则是一组解。即，当  $x$  取 1 时，考虑  $y$  取 1、2、...、44；然后当  $x$  取 2 时，又考虑  $y$  取 1、2、...、44；...；最后当  $x$  取 44 时，又考虑  $y$  取 1、2、...、44。这个过程需要用二重循环结构来实现，但从算法的角度看，这就是枚举的思想。初学者不会意识到，这就是算法，这就是程序设计的思想。这些思想才是程序设计的精髓，才是真正要掌握的。

确定了这门课程的主线以后，我们应该采取怎样的教学方法和手段来确保教学不偏离这条主线？我们的教学经验表明，可以通过以下三种教学方法和手段。

### （一）以程序设计竞赛为驱动

近十几年来，各种程序设计竞赛开展得如火如荼，尤其是国际大学生程序设计竞赛（ACM/ICPC）在各高校开展的规模与影响逐年扩大。这些竞赛不仅给众多程序设计爱好者提供了一个展示自己用计算机分析问题和解决问题的能力机会，也给程序设计初学者提供了一个实践程序设计思想和方法的平台。在程序设计基础课程中引入 ACM/ICPC 程序设计竞赛的训练方法与裁判规则，能极大地激发学生的学习兴趣 and 竞争意识，培养学生的创新思维能力，是一个非常好的教学新思路。

### （二）以在线实践为导向

传统的教学实践通常是由老师布置题目，学生编写程序，由老师来评判程序的正确与否。这种形式的教学实践既缺乏激励机制，难以引起学生的兴趣；又因为需要人工评判程序，评判结果不及时且带有主观性。随着各类程序竞赛的推广，各种程序在线评判（Online Judge，简称为 OJ）网站也应运而生，这为程序设计课程提供了一种新的实践方法——在线实践。在线实践不同于传统的教学实践，它指由试题网站提供试题，学生在线提交程序，试题网站的在线评判系统实时反馈评判结果。这些试题具有趣味性、挑战性，以及评判过程和结果公正及时，能引起学生的极大兴趣。

### （三）以课程设计进行强化

通常的观点是，程序设计基础这些课程教学的只是程序设计语言语法和基础的程序设计方法，学生无法完成实际意义上的课程设计任务。另一方面，很多学生到毕业设计阶段才发现还没掌握相关开发工具，需要花大量的时间去学习开发工具，或者由于没有一定的软件开发经历而对毕业设计无从下手。程序设计基础课程设置课程设计的目的是要从低年级开始就给学生提供软件开发经历。我们认为，低年级学生通过 3 人左右的团队协作，他们是可以完成 300~500 行代码的课程设计的。而且通过团队协作，可以让学得比较好的学生带动学得比较差的学生，激发他们的学习积极性。另外，对课程设计的考评，可以采用提交程序说明书、团队答辩等形式，既能让学生对今后的毕业论文答辩有较早的认识，也可以培养学生的口头表达能力、文档内容组织能力等。

一种新的教学思路和方法需要一本教材作为支撑，这就是我们编写这本教材的初衷。

## 二、教材内容安排

根据本教材知识结构的层次性，以及为适应不同的教学对象和读者对象，本教材将内容分为五篇，各篇的内容相对比较独立。

### 第一篇 基础知识

本篇介绍本教材的基础知识。包含了第 1 章和第 2 章，这两章内容安排如下。

第 1 章介绍 C/C++ 语言基础知识。这一章所讲授的语言知识只是用来编写一个 C/C++ 程序所需的最小语法知识集。这些语言知识的选取有两个原则。

第一个原则是：使学生在最短的时间里掌握编写一个完整程序所需的语法知识集，因此对于

可以在后续阶段学习的语法知识，都放在附录介绍（或者引导学生参考其他教材）。比如函数只要求能根据问题求解的需要去定义函数即可，像函数重载、函数模板、有默认参数的函数等等这些内容都不作介绍。再比如指针变量通常只有两个作用，一是用作函数参数，可以修改主调函数中变量的值；二是用来指向数组元素，从而可以很方便地访问数组中每个元素。初学者在学指针变量能掌握这两个用法就可以了。

第二个原则是：以数值型数据的处理为线索介绍 C/C++ 语言基础知识。其目的主要有两个：1) 以数值型数据的处理和数学应用为例讲解语言语法，学生更容易接受，因为这些数学应用问题学生已经在高等数学甚至初等数学中就已经学过了，现在只是用编写程序的方法去求解。2) 为了让学生在较短的时间里掌握 C/C++ 语言的基础知识，专门把字符及字符串处理的基础知识拿出来，放在第 2 章集中介绍。

第 2 章介绍字符及字符串处理的基础知识。字符型也是一种重要的数据类型，本章介绍的知识包括字符型数据（字符常量、字符变量）、字符串常量、字符数组、字符指针变量、字符串处理函数，这些内容有联系，也有区别。

## 第二篇 程序设计方法及在线实践（基础篇）

本篇及第三篇是本教材的重点，介绍程序设计的基本思想和方法。在介绍这些算法思想时，本教材是从最简单的例子开始引入算法思想，然后通过一些经典的 ACM/ICPC 程序设计竞赛题目对算法的思想进一步阐述。每章每节后面还布置了一些练习题，这些练习题也大多取材于实际的竞赛题目。

本篇介绍基本的算法思想和应用问题，包括第 3~6 章，各章的内容安排如下。

第 3 章介绍了 ACM/ICPC 程序设计竞赛与在线实践。从第 1 章的单组数据的处理，过渡到 ACM/ICPC 竞赛题目中的多组数据处理。本章详细地总结了 ACM/ICPC 题目的输入/输出方式，对每种输入方式通过分析讲解一道典型的竞赛题目来加深学生这些输入方式的理解。

第 4 章介绍枚举算法思想。在求解具体问题时，枚举是一种最容易想到的思路：枚举所有可能的结果，对枚举到的每个结果判断是否符合题目要求，符合题目的要求则是正确的解。本章介绍枚举的算法思想，并通过验证歌德巴赫猜想这一经典数学问题以及其他一些竞赛题目着重讲解在应用枚举算法时需要注意的问题。

第 5 章介绍模拟思想。现实中有些问题难以找到公式或规律来求解，只能按照一定的步骤不停的“模拟”下去，最后才能得到答案。这样的问题，用计算机来求解十分合适，只要能让计算机模拟人在解决此类问题时的行为即可。这种求解问题的思想，可以称为“模拟”。模拟也是求解竞赛题目时经常采用的方法。本章介绍模拟的算法思想，并通过约瑟夫环问题的模拟等竞赛题目着重讲解在应用模拟方法时需要注意的问题。

第 6 章讲解字符及字符串处理。涉及到的知识和应用问题包括：字符转换与编码、回文的判断与处理、子串的处理、字符串匹配等。

## 第三篇 程序设计方法及在线实践（提高篇）

本篇是第二篇的提高，介绍比较复杂的算法思想和应用问题，包括第 7~9 章，各章的内容安排如下。

第 7 章介绍高精度计算。高精度计算是字符及字符串处理知识的具体应用（有些题目采用整型数组处理更方便）。本章从大数运算引入高精度计算问题；继而介绍高精度计算涉及到的基础知识：进制转换、用数组实现算术运算，从而引出高精度计算的基本思路；最后通过经典竞赛题目讲解高精度数的基本运算以及其他一些问题的实现方法。

第 8 章介绍递归算法思想及其应用。递归思想是一种重要的算法思想，通过递归函数这种函数形式来实现。本章从求阶乘、Fibonacci 数列的递归求解等例题出发引入递归算法思想，再过渡到递归思想在实际竞赛题目中的应用。特别地，本章还介绍了求解问题的一种通用方法：搜索，并以排列组合问题的求解来阐述搜索方法在求解这一类问题中的应用。

第 9 章介绍排序及检索。对数据进行排序是数据处理中经常要用到的操作，本章介绍常用的、比较简单的排序算法的思想及其应用。另外，对一组已经排好顺序的数据进行检索也是经常要用到的操作，本章因此介绍了二分法的思想，以及二分法在检索中的应用。

#### 第四篇 课程设计

本篇的内容包括第 10 章，课程设计。学完程序设计基础课程后，学生最想知道的是能编程实现什么。课程设计的目的是要让学生知道，学了 C/C++ 这门语言，学了这些程序设计的思想和方法后，就能编写一个小软件，哪怕是字符界面的。等到掌握了可视化的开发工具后，把这些思想和方法移植过去，就是一个漂亮的软件。本章以“字符界面的扫雷游戏”为例，通过任务分解，循序渐进地讲解整个程序的开发过程。

#### 第五篇 附录

由于篇幅的限制，很多 C/C++ 语言的知识没有在教材的正文介绍。为了方便读者阅读和使用本教材，在第五篇中补充了丰富的相关知识。

附录 A~附录 C 为读者介绍在进行 C/C++ 程序设计实践时需要掌握的一些方法。特别地，附录 C 专门为程序设计竞赛初学者介绍了在做 ACM/ICPC 题目时需要掌握的一些基本方法和技巧。附录 D~附录 G 补充介绍一些有用的、在平时学习中经常需要查阅的 C/C++ 语言知识。

本教材力求通过较多的经典题目来介绍程序设计基本思想和方法，以及问题的求解方法，为此，本教材收录了 110 道左右的 ACM/ICPC 竞赛题目，例题和练习题各约占一半。对教材中的例题，都作了详细的分析和讲解，而对部分练习题，则给出相应的提示。为了方便读者借助国内外一些 OJ 来学习 C/C++ 语言并实践本教材介绍的程序设计思想和方法，在附录 H 中还列出了本教材例题和练习题在 ZOJ、POJ 及 UVA 上的题号。

### 三、教材使用建议

本书作为教材可以适用于不同的教学对象和教学目标。学完第一篇的内容后，学生可以参加国家和浙江省高校计算机等级考试（二级 C）；学完第二篇后，学生具备了参加 ACM/ICPC 程序设计竞赛所需的基础知识；学完第三篇后，学生的程序设计和算法分析能力将得到进一步的提高，具备了参加全国计算机技术与软件专业技术资格考试（程序员级）中程序设计（C 语言）部分所需的基础知识。本书也可以作为程序设计竞赛爱好者的自学教材或培训教材。

#### （一）没有 C/C++ 语言基础的学生

对于没有 C/C++ 语言基础的学生，如果课程内容安排在一学期（假设教学时间为 18 周，理论课时为 54，实验课时为 36），建议按表 1 所示的安排表进行教学。其中前面 12 周介绍 C/C++ 语言基础知识，即第 1 章和第 2 章，这两章很多例题本身已经包含了基础的编程思想和方法。接下来的 4 周介绍本书第二篇，即程序设计方法及在线实践（基础篇），每章一般只需精讲 3~4 道例题，其他例题以学生自学为主。最后 2 周集中讲解课程设计并向学生布置课程设计任务。

对于本书的第三篇，即程序设计方法及在线实践（提高篇），学生学完上述内容后，一些对程序设计竞赛感兴趣的学生，可以自学其中大部分内容。

表 1 教学安排表(没有 C/C++ 语言基础的学生)

章节	内 容	教学时数	实验时数
第一篇 基础知识	1.1 C/C++ 程序的基本框架 1.2 程序中的数据	3	2
	1.3 运算符和表达式 1.4 C/C++ 的语句	3	2
	1.5 数学函数的使用 1.6 算法及程序控制结构	3	2
	1.7 选择结构	3	2
	1.8 循环结构	6	4
	1.9 函数设计	6	4

第二篇 程序设计方法及在线实践（基础篇）	1.10 数组	3	2
	1.11 指针与指针变量	6	4
	第2章 字符及字符串基础知识	3	2
	第3章 ACM/ICPC 程序设计竞赛与在线实践	3	2
	第4章 枚举	3	2
	第5章 模拟	3	2
第四篇 课程设计	第6章 字符及字符串处理	3	2
	第10章 课程设计	6	4
合计		54	36

## （二）已经掌握 C/C++语言的学生

对于已经掌握了 C/C++语言的学生，建议按表 2 所示的安排表进行教学（理论课时为 54，实验课时为 36）。课程的主要目的是向学生介绍程序设计方法和基本的算法思想。其中前面 7 周介绍本书第二篇，中间 8 周介绍第三篇。对于已经掌握了 C/C++语言的学生，在布置课程设计任务时要求可以高一些，因此在课程的最后可以花 3 周的时间来安排课程设计。

表 2 教学安排表(已经掌握了 C/C++语言的学生)

章节	内 容	教学时数	实验时数
第二篇 程序设计方法及在线实践（基础篇）	第3章 ACM/ICPC 程序设计竞赛与在线实践	3	2
	第4章 枚举	6	4
	第5章 模拟	6	4
	第6章 字符及字符串处理	6	4
第三篇 程序设计方法及在线实践（提高篇）	第7章 高精度计算	6	4
	第8章 递归与搜索	9	6
	第9章 排序与检索	9	6
第四篇 课程设计	第10章 课程设计	9	6
合计		54	36

## （三）对程序设计竞赛爱好者

对于程序设计爱好者，在自学或者培训时，建议这些学生按照本书第二篇和第三篇章节的顺序来阅读和使用本教材，在理解了例题所包含的程序设计方法和算法思想后多做一些章节后面的练习题，以巩固对算法思想的理解。

## （四）C/C++语言语法知识进一步学习建议

本教材第一篇介绍的语法知识只是编写一个完整的 C/C++程序的最小语法知识集。读者掌握这些语法知识并具备编写完整的 C/C++程序的能力后，可以通过参考其他教材（如[2]、[3]等）来自学其他语法知识。

为了使读者更好地使用本教材和参考其他教材，在此列出本教材第一篇没有介绍到的 C/C++语言语法知识：

- 1) 变量：常变量、符号变量（也就是宏定义）等。
- 2) 变量的作用域：全局变量与局部变量；变量的存储期：静态存储期（static storage duration）和动态存储期（dynamic storage duration）；变量的存储类别：自动的（auto）、静态的（static）、寄存器的（register）和外部的（extern）等。
- 3) 运算符：位运算符、逗号运算符等等。
- 4) 预处理：宏定义、文件包含和条件编译等。
- 5) 函数：内置函数、函数的重载、有默认参数的函数、函数模板、内部函数与外部函数。
- 6) 指针：多维数组与指针、函数指针、返回值为指针的函数、指针数组与二级指针等。
- 7) 引用类型：变量的引用类型、引用作为函数参数等。
- 8) 结构体（本教材 9.2.1 节做了简单的介绍）、枚举类型、共用体。
- 9) C++语言的输入/输出流：本教材附录 D.2 对 C++语言的输入/输出做了简单的介绍，其他知识本书都没有介绍。

10) 所有面向对象程序设计的思想和语言语法知识。

#### 四、本书所附光盘说明

本书所附光盘包含了多媒体课件、例题代码、图片、实验报告等丰富的教辅资源。特别是在实验报告中，前面 7 个实验报告是专门为 C/C++ 语言基础知识设计的，每个实验报告都通过语法练习→程序调试→程序设计→综合设计这种比较完善的梯度训练让学生在较短的时间内掌握 C/C++ 语言知识。

#### 五、本教材约定

为了便于读者阅读和使用教材，本教材在排版上做了如下约定：

1) 对于 C/C++ 语言的语法格式，用粗体、斜体标明，如：

定义变量的一般形式是：

***变量类型 变量名列表;***

2) 对每道例题，分析完毕后，在段落的末尾用符号“□”结尾，以便与后面的内容分隔开。

#### 六、致谢

本教材在编写和修订过程中，赵辉、翁文庆、朱凌、万贤美等老师提出了宝贵的意见，在此编者对他们的辛勤工作表示诚挚的谢意。

由于我们水平有限，在编写教材时难免出错，欢迎读者指正，或者读者有什么好的建议，都可以联系编者：w\_guiping@163.com。不胜感激!!!

编者  
2009 年 5 月



# 目 录

## 第一篇 基础知识

第 1 章 C/C++语言基础 .....	3
1.1 C/C++程序的基本框架 .....	3
1.2 程序中的数据 .....	5
1.2.1 变量 .....	5
1.2.2 常量 .....	7
1.3 运算符和表达式 .....	8
1.3.1 运算符的优先级和结合性 .....	8
1.3.2 算术运算符及算术表达式 .....	9
1.3.3 赋值运算符及赋值表达式 .....	11
1.3.4 关系运算符及关系表达式 .....	12
1.3.5 逻辑运算符及逻辑表达式 .....	13
1.3.6 类型转换 .....	15
1.4 C/C++的语句 .....	17
1.5 数学函数的使用 .....	18
1.5.1 常用的数学函数 .....	18
1.5.2 数学函数的使用 .....	19
1.6 算法及程序控制结构 .....	21
1.6.1 算法及控制结构 .....	21
1.6.2 顺序结构 .....	22
1.7 选择结构 .....	23
1.7.1 if语句 .....	23
1.7.2 条件运算符与条件表达式 .....	27
1.7.3 switch语句 .....	27
1.8 循环结构 .....	32
1.8.1 3种循环语句 .....	32
1.8.2 break语句和continue语句 .....	37
1.8.3 循环的嵌套 .....	39
1.8.4 循环结构例子 .....	41
1.9 函数设计 .....	47
1.9.1 函数概述 .....	47
1.9.2 函数的定义 .....	47
1.9.3 函数参数 .....	49
1.9.4 函数的返回值 .....	51
1.9.5 函数的调用 .....	52
1.9.6 函数的嵌套调用 .....	53
1.9.7 函数的设计 .....	53
1.10 数组 .....	58
1.10.1 一维数组的定义与引用 .....	59
1.10.2 二维数组的定义和引用 .....	61
1.10.3 数组名作函数参数 .....	64
1.10.4 编写数组应用的综合程序 .....	65
1.11 指针与指针变量 .....	69
1.11.1 指针概述 .....	69

1.11.2 指针与指针变量 .....	70
1.11.3 指针变量作函数参数 .....	73
1.11.4 数组与指针变量 .....	76
1.11.5 编写指针应用的综合程序 .....	79
第2章 字符及字符串基础知识 .....	83
2.1 字符型数据 .....	83
2.1.1 字符型变量 .....	83
2.1.2 字符型常量 .....	84
2.1.3 字符型数据的输入/输出 .....	85
2.2 字符串常量 .....	87
2.3 字符数组 .....	88
2.3.1 字符数组的定义与初始化 .....	88
2.3.2 字符数组元素的引用 .....	89
2.3.3 字符数组的输入/输出 .....	90
2.3.4 字符数组与字符串常量的区别与联系 .....	92
2.4 字符指针变量 .....	93
2.4.1 字符指针变量的定义与引用 .....	93
2.4.2 字符指针变量、字符数组与字符串常量 .....	93
2.5 字符串处理函数 .....	95
2.6 编写处理字符型数据的程序 .....	100

## 第二篇 程序设计方法及在线实践（基础篇）

第3章 ACM/ICPC程序设计竞赛与在线实践 .....	105
3.1 程序设计竞赛与在线程序实践 .....	105
3.2 ACM/ICPC程序设计竞赛简介 .....	105
3.3 ACM/ICPC竞赛题目特点 .....	106
3.3.1 ACM/ICPC题目组成及特点 .....	106
3.3.2 ACM/ICPC题目的输入/输出 .....	106
3.3.3 ACM/ICPC题目类型 .....	108
3.4 ACM/ICPC竞赛题目解析 .....	108
第4章 枚举 .....	117
4.1 枚举的基本思路 .....	117
4.2 歌德巴赫猜想 .....	120
4.3 其他竞赛题目解析 .....	125
第5章 模拟 .....	138
5.1 模拟的基本思路 .....	138
5.2 模拟约瑟夫环 .....	139
5.3 游戏的模拟 .....	145
5.4 其他模拟题目解析 .....	158
第6章 字符及字符串处理 .....	165
6.1 字符转换与编码问题 .....	165
6.1.1 字符转换 .....	165
6.1.2 字符编码 .....	168
6.2 回文的判断与处理 .....	175
6.3 子串处理 .....	180
6.4 其他竞赛题目解析 .....	185

### 第三篇 程序设计方法及在线实践（提高篇）

第 7 章 高精度计算 .....	196
7.1 基础知识 .....	196
7.1.1 进制转换 .....	196
7.1.2 用字符型数组或整型数组实现算术运算 .....	199
7.1.3 高精度计算的基本思路 .....	201
7.2 高精度数的基本运算 .....	204
7.2.1 高精度数的加法 .....	204
7.2.2 高精度数的乘法 .....	206
7.2.3 高精度数的除法 .....	209
7.3 其他高精度题目解析 .....	214
7.3.1 数列问题 .....	214
7.3.2 其他题目 .....	216
第 8 章 递归与搜索 .....	221
8.1 递归的基本思想 .....	221
8.1.1 什么是递归 .....	221
8.1.2 例题解析及递归函数设计 .....	223
8.1.3 递归存在的问题 .....	229
8.2 递归思想在竞赛题目中的应用 .....	230
8.3 递归与搜索 .....	235
8.3.1 搜索算法思想 .....	235
8.3.2 递归函数的设计 .....	236
8.3.3 例题解析 .....	236
8.4 递归方法求解排列组合问题 .....	248
8.4.1 排列问题 .....	248
8.4.2 组合问题 .....	254
第 9 章 排序及检索 .....	264
9.1 排序算法 .....	264
9.1.1 插入排序法 .....	264
9.1.2 冒泡法排序 .....	266
9.1.3 简单选择法排序 .....	270
9.2 qsort 函数及其使用 .....	273
9.2.1 qsort 函数的用法 .....	274
9.2.2 qsort 函数应用例子 .....	276
9.3 竞赛题目解析 .....	280
9.3.1 数值型数据的排序 .....	280
9.3.2 字符型数据的排序 .....	283
9.3.3 混合数据的排序 .....	286
9.4 二分法思想及二分检索 .....	291
9.4.1 二分法的思想 .....	291
9.4.2 二分法检索 .....	291
9.4.3 竞赛题目分析 .....	294

### 第四篇 课程设计

第 10 章 课程设计：字符界面扫雷游戏的开发 .....	301
-------------------------------	-----

10.1 软件需求说明 .....	301
10.2 地图的表示与输出 .....	303
10.2.1 Windows操作系统扫雷游戏简介 .....	303
10.2.2 如何表示地图 .....	303
10.2.3 如何表示一个位置的 8 个相邻位置 .....	304
10.2.4 如何输出地图 .....	305
10.2.5 测试程序 .....	306
10.3 随机生成地图 .....	307
10.3.1 随机函数rand( ) .....	307
10.3.2 随机生成地图 .....	308
10.3.3 测试程序 .....	309
10.4 如何实现玩游戏 .....	311
10.4.1 显示给用户看的地图 .....	311
10.4.2 输出用户地图 .....	312
10.4.3 点开一片连续的没有地雷的区域 .....	313
10.4.4 游戏的玩法 .....	314
10.4.5 测试程序 .....	314
10.4.6 完善程序 .....	318

## 第五篇 附录

附录A C/C++程序的编写与运行 .....	323
附录B 程序测试与调试 .....	328
附录C ACM/ICPC入门指导 .....	340
附录D C/C++的输入/输出 .....	346
附录E ASCII编码表 .....	357
附录F C/C++关键字 .....	358
附录G 运算符及其优先级与结合性 .....	359
附录H 本教材例题和练习题在ZOJ、POJ及UVA上的题号 .....	360
参考文献 .....	364

## 第一篇 基础知识

本篇介绍本教材的基础知识。包含了第 1 章和第 2 章，这两章内容安排如下。

第 1 章以数值型数据的处理和简单的数学应用问题为线索介绍 C/C++ 语言基础知识。这些 C/C++ 语言知识只是用来编写一个 C/C++ 程序所需的最小语法知识集。初学者在学习程序设计时，应该把主要精力放在掌握程序设计的思想和方法上，不要拘泥于语言的语法细节，语言只是用来表达程序思想的工具而已。对于本章没有介绍的 C/C++ 语言语法知识，读者可以在需要用的时候参考其他教材。第 1 章的内容在叙述上具有以下 3 个特点，读者在阅读时请特别注意。

- 1) 第 1 章的内容是以数值型数据的处理为线索，以简单数学计算或数学应用为例子来讲解 C/C++ 语言语法知识，同时引入用程序求解具体问题的思想和基本方法。
- 2) 为了加深读者对 C/C++ 语法知识的理解，第 1 章经常对同一个应用问题采用不同的方法来实现，并对这些方法作对比。例如，对“交换两个数”的应用，分别在例 1.4、例 1.7、例 1.47 和例 1.49 采用不同的方法来实现。
- 3) 第 1 章的例题和习题都是本着“由浅入深”的思路来设置的，特别是习题的安排，是从模仿教材中的例题入手，逐步过渡到改写程序，再过渡到自己编写程序。

第 2 章介绍字符及字符串处理的基础知识。字符型也是一种重要的数据类型，在第 1 章中为了使读者在较短的时间里掌握 C/C++ 语言的基础知识，因此以数值型数据的处理为主线，并没有涉及到字符型数据的处理，第 2 章将集中介绍字符及字符串的基础知识。

### 程序实践提示

在学习 C/C++ 语法知识时，读者（特别是初学者）需要通过较多的程序实践来理解这些语法知识及其应用，建议读者通过验证例题→理解思考题→编程求解练习题等实践方式来理解 C/C++ 语言知识。

另外，本教材所附光盘包含的实验报告中，前面 7 个实验报告是专门为 C/C++ 语言基础知识设计的，每个实验报告都包含了语法练习→程序调试→程序设计→综合设计等四种实验任务，读者在学习时可以通过这种梯度训练在较短的时间内掌握 C/C++ 语言知识。



# 第 1 章 C/C++语言基础

C++语言是由 C 语言发展起来的，与 C 完全兼容。C++对 C 的“增强”，表现在两个方面：一是在原来面向过程机制的基础上，对 C 语言的功能做了一些扩充，使之用起来更方便；二是增加了面向对象的机制。但本教材中所讲述的编程思想和具体算法都不涉及到面向对象部分，这样 C 和 C++语言没有太大的区别。并且，现在很多编译器同时兼容 C 和 C++语言。因此本教材在描述某个语法知识点时，并不严格区分 C 和 C++语言，而统称为 C/C++语言。另外，由于本教材很多例题和习题都是取材于实际的 ACM/ICPC 程序设计竞赛题目，这些题目对程序运行时间都有严格的限制，而 C 语言的输入/输出方式（scanf 和 printf 函数）比 C++语言的输入/输出方式（cin 流和 cout 流对象）快得多，因此本教材的例题代码均采用 scanf 和 printf 函数进行输入/输出。

## 1.1 C/C++程序的基本框架

现以下面的简单程序为例来说明 C/C++程序的基本框架。这个程序只有输出，没有输入。

**例 1.1** 向屏幕上输出一行字符：Welcome to C/C++!

```
#include <stdio.h>    //包含头文件
void main( )          //主函数
{
    printf( "Welcome to C/C++!\n" );    //向屏幕上输出一行字符
}
```

程序在运行时会在屏幕上输出一行字符：

Welcome to C/C++!

这个程序包含了以下两个部分：

(1) 头文件包含

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

include 是 C/C++语言的关键字（关键字的含义请参考附录 F），表示要把另一个文件中的内容包含到本程序中，stdio.h 是被包含文件的文件名。扩展名为“.h”的文件一般称为头文件。C/C++语言提供了一些可以被直接拿来使用、能够完成某些特定功能的库函数，分别声明（函数及函数声明的概念详见 1.9 节）于不同的头文件中。例如，stdio.h 中定义了一些与输入/输出相关的函数。printf 就是一个能往屏幕上输出一串字符（或其他数据）的库函数。在 main 函数中使用了 printf 函数，所以要把头文件 stdio.h 包含进来，否则程序在编译时会给出编译错误，提示：“printf”是未定义的标识符（标识符的含义见 1.2.1 节）。

(2) 主函数

```
void main( )
{
    printf( "Welcome to C/C++!\n" );    //向屏幕上输出一行字符
}
```

main 函数是程序中的主函数。每个 C/C++程序都必须包含这个 main 函数，而且只能有一个 main 函数，但可以有多个其他函数。C/C++程序的最小独立单位是语句。程序运行时，总是从 main 函数的第一条语句开始执行，一直执行完 main 函数中的最后一条语句，整个程序才执行完毕。

在 C/C++语言中，分号是语句的标志。例如，main 函数中有一条语句：

```
printf( "Welcome to C/C++!\n" );
```

这条语句的作用是在屏幕上输出一串字符“Welcome to C/C++!”, 然后换行。“\n”的作用是换行。“//向屏幕上输出一行字符”是程序中的注释, 用来对程序作注解。C/C++规定, 一行中如果出现“//”, 则从它开始到本行末尾之间的全部内容都作为注释。这种注释称为**行注释**。注释内容在编译时会被忽略, 对运行不起作用。□

**思考 1.1:** 如果想在屏幕上输出“Hello world!”, 该如何修改例 1.1?

一个计算机程序通常包括输入数据、处理数据、输出结果等过程。在程序中要进行数据处理, 需要定义变量(变量的概念详见 1.2.1 节), 从键盘上输入的数据保存到变量中, 经过处理后, 程序将结果输出到屏幕上。为了让读者理解上述基本过程, 我们再举一个例子。

**例 1.2** 从键盘上输入两个整数 a 和 b, 计算 a 和 b 之和并输出。

```
/******
```

例 1.2 求两个数之和

author: Wang GuiPing

Date: 2007-11-30

```
*****/
```

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

```
void main( )
```

```
{
```

```
    int a, b, sum;           //定义变量
```

```
    scanf( "%d%d", &a, &b ); //输入语句, 从键盘上输入数据, 保存到变量 a 和 b 中
```

```
    sum = a + b;             //赋值语句
```

```
    printf( "a+b=%d\n", sum ); //输出语句
```

```
}
```

该程序的运行示例如下:

78 56✓(✓表示输入数据后回车, 以下同)

a+b=134

本程序开头的 5 行, 即从“/\*”开始到“\*/”结尾, 也是注释。C/C++规定, 程序某一行中如果出现“/\*”, 则从它开始到“\*/”结尾(注: 其他“\*”号仅仅是为了美观)之间的全部内容都作为注释。这种注释称为**块注释**, 块注释可以包含很多行。

在 main 函数中, 首先定义了 3 个整型变量 a、b 和 sum。其中 a 和 b 用来存放从键盘上输入的整数, sum 用来存放 a 和 b 的和。与 printf 函数类似, scanf 函数也是在头文件 stdio.h 中定义的, 用于从键盘上输入数据到变量中。如果从键盘上输入“78 56”, 然后回车, 那么数据 78 存储到变量 a 中, 数据 56 存储到变量 b 中。程序中的赋值语句将表达式“a+b”的值赋予给变量 sum。在 printf 函数中, 字符内容“a+b=”原封不动地输出来, “%d”部分是转换成变量 sum 的值输出来。□

**思考 1.2:** 在例 1.2 中, 如果想输出“78+56=123”(假设输入的数据为 78 56, 即在输出时显示变量 a 和 b 的值), printf 函数该如何修改?

一个 C/C++程序从编写到运行要经历编写程序、编译程序、连接程序、运行程序、分析程序输出结果等步骤, 详见附录 A。

另外, 有关输入/输出函数(scanf 和 printf 函数等)的详细介绍, 请参考附录 D。

## 练习

1.1 请模仿例 1.1, 编写程序, 向屏幕上输出以下 3 行字符信息。

```
*****
```



Welcome to C/C++!

\*\*\*\*\*

1.2 请模仿例 1.1，编写程序，向屏幕上输出下图所示的字符图形。

```

*
**
***
****
*****
*****
*****
***
*

```

1.3 请模仿例 1.2，编写程序，求函数值 $y = 4x^2 + 5x + 7$ 。自变量 $x$ 为整数，从键盘输入；经过计算后，将 $y$ 的值输出到屏幕上。（提示： $4x^2$ 在程序中必须表示成 $4*x*x$ ）

1.4 请模仿例 1.2，编写程序，从键盘上输入两个整数，分别求这两个整数的和、差、积、商并输出，请特别注意观察两个整数相除，得到的结果是否会保留余数。

## 1.2 程序中的数据

编写程序的目的是为了处理数据。在程序中，数据是以**常量**和**变量**两种形式存在的，而且程序中的数据是有特定类型的（例如整型、浮点型、字符型等）。为了存储数据，需要有变量；而为了给变量赋值，通常需要用常量。另外，在表达式中，变量和常量都是重要的数据形式。

### 1.2.1 变量

如果要在程序中表示和处理数据，首先要定义变量。程序中所使用的每个变量，在运行程序时都会在内存中占用一段存储空间，如图 1.1 所示，图中每一行代表内存中的一个字节。所有有关变量的操作都是针对这段存储空间的。变量由两个要素构成：**变量的名称**和**变量的类型**。变量的名称是这段存储空间的唯一标识；变量的类型决定了这段存储空间的大小、以及对所存储数据的类型要求等。

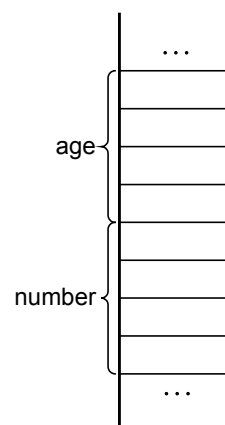


图1.1 变量名与变量所占的存储空间

#### 1. 变量的定义

定义变量的一般形式是：

**变量类型 变量名列表；**

变量名列表是指一个或多个变量名的序列。如：

```
int age, number;
```

变量名是标识符的一种。在程序设计语言中，用来标识变量、符号常量、函数、数组、类型等实体名字的有效字符序列称为**标识符**（identifier）。简单地说，标识符就是一个名字。**C/C++**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。而且，为了便于阅读和理解程序，变量名等标识符在命名时最好能“见名思义”，也就是根据变量名就能确定该变量的含义和作用。如下面的变量名就是很好的例子：

```
double score, average, max, min; //定义浮点型变量：分数，平均分，最高分，最低分
```

#### 2. 变量的类型

C/C++提供了 12 种基本数据类型，每种数据类型占用的字节数和表示范围各不一样。本章用到的数据类型有以下几种。

**int**：有符号的整型，占 4 个字节。其表示范围是 $-2^{31} \sim 2^{31}-1$ ，即 $-2147483648 \sim 2147483647$ 。

**unsigned int**：无符号的整型，占 4 个字节。其表示范围是 $0 \sim 2^{32}-1$ ，即 $0 \sim 4294967295$ 。

**float**: 单精度浮点型, 占 4 个字节。可以存储带小数部分的数值型数据。

**double**: 双精度浮点型, 占 8 个字节。可以存储带小数部分的数值型数据, 且精度和表示范围都比 float 类型高。

**bool**: 逻辑型, 占 1 个字节。**bool** 数据类型是 C++ 语言特有的, 详见 1.3.5 节。

需要说明的是, 字符型数据及其处理是在第 2 章讲述的。

在图 1.1 中, 变量 **age** 和 **number** 都是 **int** 型的, 所以给这两个变量分别分配了 4 个字节的存储空间。

### 3. 变量的赋值

给变量赋予一个新值的过程称为变量的**赋值**, 它是通过赋值运算符“=”以及赋值语句完成的。如:

```
number = 36;
```

表示把 36 赋值给变量 **number**, 最终是把 36 以二进制形式存放到变量 **number** 所占的 4 个字节中。下面再给出一些赋值的例子。

```
int count = 0; //在定义变量 count 时就赋予一个初始值 0
count = 1;    //将 count 重新赋值为 1
count = count + 1;
```

上面最后一条语句要特别注意, 请不要按照数学的思维去理解这条语句。在这条语句中, 符号“=”是赋值运算符(详见 1.3.3 节)。这条语句的执行过程是这样的: 从 **count** 变量所占用的 4 个字节中, 把 **count** 的值取出来, 参与算术运算“**count** + 1”, 并把该算术运算的结果重新赋值给 **count** 变量, 因此该变量的值变成 2。

**注意**: 在函数里定义的变量, 如果没有进行赋值, 则它的值是不确定的(或者称为是随机的值), 如下面的例 1.3。

**例 1.3** 没有赋值过的变量, 其值是不确定的。

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

该程序的输出结果可能为:

```
a=5, b=-858993460
```

在上面的程序中, 变量 **a** 在定义时就被初始化为 5, 而变量 **b** 始终没有被赋值。从输出结果可以看出, **a** 的值就是定义时被赋予的值, 而 **b** 的值是一个“奇怪”的值。□

如果赋值时, 赋值运算符“=”左右两边的数据类型不一样, 但可以进行转换, 则会自动进行转换。具体来讲, 如果将一个浮点型数据赋值给整型变量, 则只取整数部分; 如果将一个整型数据赋值给一个浮点型数据, 则小数部分为 0。这种在赋值时可以自动进行类型转换的数据类型称为是**赋值兼容**的。

下面是几个自动类型转换的例子。

```
int number, age = 17.5;    //赋值后, 整型变量 age 的值为 17
number = 3.14*3*3;        //赋值后整型变量 number 的值为 28
double area = number;      //赋值后浮点型变量 area 的值为 12.0
```

### 4. 从键盘上输入数据到变量

除赋值外, 也可以采用从键盘输入数据的方式给变量输入一定的值。如:

```
int age, number;
```

```
double score, average, max, min;
scanf( "%d%d", &age, &number );
scanf( "%lf%lf%lf%lf", &score, &average, &max, &min );
```

运行程序时，如果从键盘上输入如下数据：

20 45✓

89.5 76.2 92 30.5✓

则 age 的值为 20、number 的值为 45、score 的值为 89.5、average 的值为 76.2、max 的值为 92、min 的值为 30.5，等等。关于 scanf 函数的使用方法，详见附录 D。

## 5. 变量的引用

变量里存储的数据可以参与表达式的运算，或赋值给其他变量。这一过程称为变量的引用。

变量的引用具有“取之不尽，以新冲旧”的特点。具体来讲，将变量 a 的值赋值给 b 后，它的值保持不变，而变量 b 的值更新为变量 a 的值。下面的例 1.4 灵活地应用了变量引用的这个特点。

**例 1.4** 编写程序，实现交换程序中两个变量的值。

在程序中经常需要交换两个变量的值。可以采用的一种方法是通过中间变量 t（或称为临时变量，一般用 t、temp 等变量名），先把 a 的值暂时保存到 t 中，然后把 b 的值赋值给 a，最后把 t 的值赋值给 b。代码如下：

```
#include <stdio.h>
void main( )
{
    int a, b;        //定义两个整型变量
    scanf( "%d%d", &a, &b ); //从键盘上输入数据
    int t;           //用来保存变量 a 的值的临时变量
    //以下三条语句用于交换 a 和 b 的值
    t = a;           //(1)
    a = b;           //(2)
    b = t;           //(3)
    printf( "a=%d, b=%d\n", a, b );
}
```

该程序的运行示例如下：

5 7✓

a=7, b=5

在上面的程序中，要交换变量 a 和 b 的值，因此有赋值语句“a = b;”，然后再把变量 a 的值赋值给变量 b，但此时变量 a 的值已经不是原来的值，而是变量 b 的值了。因此，在语句(2)执行之前需要把变量 a 的值先保存到临时变量 t 中，然后在语句(3)中把临时变量 t 的值赋值给变量 b。交换 a 和 b 值的过程如图 1.2 所示，图(1)、图(2)、图(3)分别对应程序中 3 条语句执行后的效果。

因此，如果某个变量的值会更新，但该变量更新前的值对后面的程序有用，则需要把该变量更新前的值先保存到一个临时变量中。这种思想在程序中经常要用到。

**思考 1.3：**在例 1.4 中，变量引用的特点“取之不尽，以新冲旧”是如何体现的。

## 1.2.2 常量

在前面的一些赋值语句中，赋值运算符“=”右边的常数实际上就是常量。所谓**常量**，就是从字面上即可判别其值的量。在 C/C++语言中，常量有整型常量、浮点型常量、字符型常量等。

### 1. 整型常量

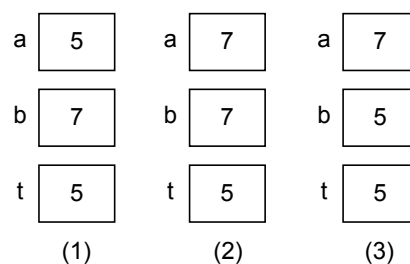


图1.2 交换两个变量值的过程

“127”，“-456”等都是整型常量。一个整型常量可以赋值给整型变量、浮点型变量、甚至字符型变量。

在 C/C++ 程序中，一个整型常量可以用 3 种不同的方式表示：

(1) 十进制整数。如 1357, -432, 0 等。

(2) 八进制整数。在常数的开头加一个数字 0，就表示这是以八进制数形式表示的常数。如 020 表示这是八进制数 20，即  $(20)_8$ ，它相当于十进制数 16。

(3) 十六进制整数。在常数的开头加一个数字 0 和一个英文字母 X (或 x)，就表示这是以十六进制数形式表示的常数。如 0X20 表示这是十六进制数 20，即  $(20)_{16}$ ，它相当于十进制数 32。

例如，有下面的赋值语句：

```
age = 023;
```

则“023”代表的是八进制，即十进制里的 19，因此 age 的值是  $(19)_{10}$ 。

## 2. 浮点型常量

“3.14159”，“-2.78”等都是浮点型常量。一个浮点型常量可以赋值给浮点型变量、整型变量、甚至字符型变量。

在 C/C++ 程序中，一个浮点型常量可以用两种不同的方式表示：

(1) 十进制小数形式。如 21.456, -7.98 等。

(2) 指数形式（即浮点形式）。一个浮点数可以写成指数形式，如 3.14159 可以表示为  $0.314159 \times 10^1$ ,  $3.14159 \times 10^0$ ,  $31.4159 \times 10^{-1}$ ,  $314.159 \times 10^{-2}$  等形式。在程序中应表示为：0.314159e1, 3.14159e0, 31.4159e-1, 314.159e-2，用字母 e 表示其后的整数是指数的幂（底为 10），如 1e-7 表示  $10^{-7}$ 。注意，字母 e 后面必须是整数，不能是浮点数，也不能是变量。例如，3.14159e2.5, 3.14159eN (N 为变量)，这些指数形式表示方法都是错误的。

## 练习

1.5 模仿例 1.4，编程实现：交换两个变量 a 和 b 的值。

提示：在例 1.4 中是先把变量 a 的值保存到临时变量 t 中，在这道练习题中，先把变量 b 的值保存到临时变量 t 中。注意将实现交换 a 和 b 的 3 条语句跟例 1.4 中的 3 条语句作对比。

1.6 请改写例 1.4，编程实现：从键盘上输入两个整数，保存到变量 a 和 b，借助 1 个中间变量 t，经过一定步骤后，变量 a 的值为输入的两个整数之和，变量 b 的值为输入的两个整数之差。

## 1.3 运算符和表达式

在进行数据处理时，经常要进行一定的运算，才能得到结果。运算是通过运算符和表达式来实现的。

初等数学提供了加(+)、减(-)、乘(\*)、除(/)运算符，而 C/C++ 提供了复杂得多的运算符。本章将介绍常用的运算符，其他运算符请参考附录 G。

所谓**表达式**，就是通过一些运算符将一些变量、常量等连接起来的式子。对表达式的理解，要特别注意以下两点：

- 1) 单个变量、单个常量是最简单的表达式；
- 2) 每个合法的表达式都有一个确定的值。

表达式和运算符的联系是密切的，表达式要通过运算符来实现，而运算符的作用是体现在表达式里的。

### 1.3.1 运算符的优先级和结合性

**优先级和结合性**是运算符的两个重要的特性。在求解表达式时，先按运算符的优先级别高低次序执行。如果一个操作数两侧的运算符的优先级别相同，则按结合性中规定的“结合方向”进行运算。

例如，在表达式“ $a + b * c$ ”中，先执行乘法运算“ $b * c$ ”，再执行加法，把  $a$  的值与乘法运算的结果加起来。这是因为操作数  $b$  的左右两侧的运算符分别是加法运算符“ $+$ ”和乘法运算符“ $*$ ”，而乘法运算符“ $*$ ”的优先级高于加法运算符“ $+$ ”。

又如，表达式“ $a = b = 5$ ”等效于“ $a = (b = 5)$ ”，即先执行“ $b = 5$ ”的赋值表达式，然后把该表达式的值（就是变量  $b$  的值）赋值给变量  $a$ 。这是因为操作数  $b$  的左右两侧都是赋值运算符“ $=$ ”，而赋值运算符“ $=$ ”的结合性是右结合性，所以先执行右边的赋值运算。

表 1.1 列出了常用的运算符的优先级和结合性，附录 G 列出了 C/C++所有的运算符。

表 1.1 常用运算符的优先级和结合性

优先级	运算符	含义	结合性
高	( )	括号、函数调用	自左向右
	++	自增运算符(后置)(单目运算符)	
	--	自减运算符(后置)(单目运算符)	
	++	自增运算符(前置)(单目运算符)	自右向左
	--	自减运算符(前置)(单目运算符)	
	!	逻辑非运算符	
	*	乘法运算符	自左向右
	/	除法运算符	
	%	求余运算符	
	+	加法运算符	自左向右
	-	减法运算符	
低	<、<=、>、>=	关系运算符	自左向右
	==	等于关系运算符	自左向右
	!=	不等于关系运算符	
	&&	逻辑与运算符	自左向右
		逻辑或运算符	自左向右
	=、+=、-=	赋值运算符、复合的赋值运算符	自右向左
	*=、/=、%=		

1.3.2 算术运算符及算术表达式

算术运算符用于数值运算。算术运算符包括加(+)、减(-)、乘(\*)、除(/)、求余数(%)、自增(++)、自减(--)共 7 个。下面介绍几个特殊的算术运算符及算术表达式。

1. 模运算符

求余数的运算符“%”称为**模运算符**。它是双目运算符，即需要有两个操作数，并且**两个操作数都必须是整数类型**。 $a \% b$  的结果就是  $a$  除以  $b$  后的余数。

另外，任意一个正整数 NUM 对正整数 N 取余，结果总是落在 $[0, 1, 2, \dots, N-1]$ 的范围内。

**例 1.5** 模运算符的运用。有 7 个人在一起玩报数游戏，从第 1 个人从 1 开始报数，第 7 个人报完 7 后，又回到第 1 个人从 8 开始报数，如此轮回往复。请问，输入一个正整数（如 2008），这个数是由第几个人报出的？

**分析：**报数的时候，如果无穷多个人，每个人的报数比前一个人报数的结果增加 1，即报数的结果是：1, 2, 3, ...，这是一个线性序列。本题只有 7 个人，要求第 7 个人报数到 7 后，又回到第 1 个人从 8 开始报数，这构成了一个环状序列。如图 1.3 所示，图中圆圈中的数字代表这 7 个人的号码，边上的数字为他们依次报出来的数。

取模运算可以使线性序列构成环状序列，在实际问题中经常要用到取模运算。假设输入的数

NUM 是第  $j$  个人报出来,为了求  $j$ ,容易想到的是将 NUM 对 7 取余,即:  $j = \text{NUM} \% 7$ 。但是  $\text{NUM} \% 7$  的范围是 0, 1, ..., 6。这不符合题目的要求。为了使取余的结果落入到范围 1, 2, ..., 7, 将取余的结果加 1, 即:  $j = \text{NUM} \% 7 + 1$ ; 同时为了抵消这个加 1 的效果, 必须在 NUM 对 7 取余之前先减 1, 即:  $j = (\text{NUM}-1) \% 7 + 1$ 。可以举几个数来验证结果是正确的: 当  $\text{NUM} = 1$  时,  $j = 1$ , 是第 1 个人报出来的; 当  $\text{NUM} = 9$  时,  $j = 2$ , 是第 2 个人报出来的, 结果正确。关于模运算符的类似应用, 请参考例 2.15。

程序代码如下:

```
#include <stdio.h>
void main( )
{
    int NUM, j; //变量定义
    scanf( "%d", &NUM );
    j = (NUM-1)%7 + 1; //求模
    printf( "j=%d\n", j );
}
```

该程序的运行示例如下:

2008 ✓

j=6

□

**思考 1.4:** 设有两个正整数  $N$  和  $a$ , 如果要使得任意的正整数  $M$  对  $N$  取余的结果落在范围  $[a, a+N-1]$ , 则应该如何处理; 如果范围为  $[a, b]$ ,  $a$  和  $b$  均为整数, 且  $b > a$ , 又该如何处理。

## 2. 除法运算符

C/C++ 的除法运算符有个特殊之处, 即如果  $a$ 、 $b$  是两个整数类型的变量或者常量, 那么  $a/b$  的结果是  $a$  除以  $b$  的商, 不保留余数, 例如, 表达式 “5/2” 的结果是 2, 而不是 2.5, 如果希望得到的结果保留余数, 可以使用表达式 “1.0\*a/b”。

模运算符和除法运算符广泛应用于整数数据的处理, 请看下面的例子:

已知 NUM 是一个 3 位数的整数, 我们可以分别得到其个位、十位和百位上的数字: 其个位是  $\text{NUM} \% 10$ , 其百位是  $\text{NUM} / 100$ , 其十位是  $(\text{NUM} / 10) \% 10$ 。具体应用见例 1.23、例 1.25 等。

## 3. 自增和自减运算符

自增运算符 “++” 用于将整型或浮点型变量的值加 1, 只有一个操作数, 是单目运算符, 并且该操作数必须是变量, 而不能是常量或表达式等。自增运算符有两种用法。

用法 1: 变量名++, 此时 “++” 称为 “后置++”。

用法 2: ++变量名, 此时 “++” 称为 “前置++”。

这两种用法都会使得变量的值加 1, 但它们是有区别的, 请看下面的例子。

**例 1.6** 前置++和后置++的用法。

```
#include <stdio.h>
void main( )
{
    int n1 = 3, n2;
    n2 = n1++; // (1) “后置++” 运算符
    printf( "n1=%d, n2=%d\n", n1, n2 );
    n1 = 3; // 重新将 n1 赋值为 3
    n2 = ++n1; // (2) “前置++” 运算符
    printf( "n1=%d, n2=%d\n", n1, n2 );
}
```

该程序的输出结果为:

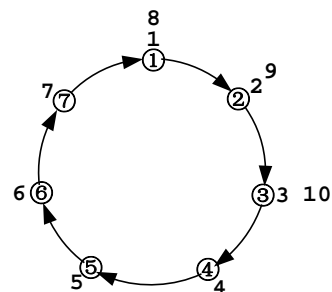


图1.3 取模运算构成环状序列

```
n1=4, n2=3
```

```
n1=4, n2=4
```

语句(1)使用了“后置++”运算符，其执行过程是先把 n1 的值赋值给 n2，然后使得 n1 的值加 1，因此程序首先输出“n1=4, n2=3”。之后，给 n1 重新赋值为 3。语句(2)使用了“前置++”运算符，其执行过程是先使得 n1 的值加 1，即 n1 的值变为 4，然后把此时 n1 的值赋值给 n2，因此程序的第 2 行输出是“n1=4, n2=4”。□

自减运算符“--”用于将整型或浮点型变量的值减 1，也有前置和后置之分。它的用法和“++”相同，不再赘述。

自增和自减运算符广泛使用于 C/C++ 程序中，特别是以下两种应用：

- 1) 用于循环语句中（详见 1.8 节），使循环变量自增 1；
- 2) 用于指针变量（详见 1.11 节），指向数组元素的指针变量自增 1 后将指向下一个元素。

自增和自减运算符在和其他运算符一起混合运算时，比较难理解，如“x = ++i+j++”，甚至有时会产生歧义。因此，在程序中，除了以上两种应用外，通常不会刻意去使用自增和自减运算符。

#### 4. 算术表达式

算术表达式就是通过算术运算符把变量、常量等连接起来的式子，如“b%10 + 25”、“a\*3 - 4\*c”等都是算术表达式。

算术表达式的值就是算术运算的结果。假设变量 a、b、c 的值分别为 27、15、2，则前面两个表达式的值分别为 30 和 73。

### 1.3.3 赋值运算符及赋值表达式

#### 1. 赋值运算符

赋值运算符“=”用于对变量进行赋值。**赋值**的含义就是赋予变量一个新的值，替换原有的值。如：

```
int a = 5, b = 7;
```

```
a = b + 3;           //把表达式 b+3 的值赋值给变量 a，则 a 的值变为 10
```

在赋值运算符“=”前面加上其他运算符，还可以构成**复合的赋值运算符**，如“+=”、“-=”、“\*=”、“/=”、“%=”等。

```
a *= b           等价于：a = a * b
```

如果“+=”等复合赋值运算符右边是包含若干项的表达式，则还要加上括号。如：

```
a *= b + 3       等价于：a = a * (b+3)
```

赋值运算符左侧的操作数称为“**左值**”（left value，简称为 lvalue）。

**注意：**并不是任何对象都可以作为左值的，变量可以作为左值，而常量不能作为左值，表达式“a + b”也不能作为左值。因此，表达式“a = 3”是合法的，但“a + b = 3”是非法的，这是很显然的，因为在内存中并没有一段存储空间用来存储表达式“a + b”，因此也就无法将 3 赋予给表达式“a + b”。

#### 2. 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**

赋值表达式的值就是赋值后左边变量的值。

赋值运算符“=”的结合性是右结合性。因此表达式：

```
a = b = 17+3
```

等价于：

**a = (b = 17+3)**

其执行过程是：先求解表达式“17+3”，其值为 20，然后将该值赋值给变量 **b**，因此变量 **b** 的值为 20；且赋值表达式“**b = 17+3**”的值为变量 **b** 的值，即为 20，最后将该表达式的值赋值给变量 **a**，因此变量的值也为 20。

**例 1.7** 编写程序，实现交换程序中两个变量的值。

**例 1.4** 采用中间变量的方法交换 **a** 和 **b** 的值，现在不借助中间变量，而是通过灵活运用算术运算符和赋值运算符来实现。代码如下：

```
#include <stdio.h>
void main( )
{
    int a, b;
    scanf( "%d%d", &a, &b ); //(1)输入语句
    a = a + b;    //(2) 此时 a 的值为 a 与 b 的和
    b = a - b;    //(3) 赋值后，b 的值为最初的 a 的值
    a = a - b;    //(4) 赋值后，a 的值为最初的 b 的值
    printf( "a=%d, b=%d\n", a, b );
}
```

该程序的运行示例如下：

5 7 ✓

a=7, b=5

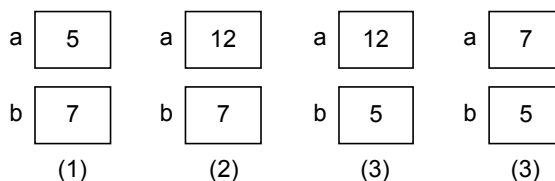


图1.4 交换两个变量值的过程(不借助中间变量)

上述代码中，先在语句(2)中把变量 **a** 和变量 **b** 的值加起来，赋值给变量 **a**。然后在语句(3)中将 **a** 减去 **b**，得到的是原来 **a** 的值，将这个值赋值给 **b**。在语句(4)中将 **a** 减去 **b**，得到的是原来 **b** 的值，将这个值赋值给 **a**。这样也实现了交换 **a** 和 **b** 两个变量的值。具体执行过程见图 1.4，图(1)、图(2)、图(3)、图(4)分别对应程序中 4 条语句执行后的效果。 □

**思考 1.5：**在例 1.7 的代码中，交换语句(2)和(3)的顺序，程序执行过程中变量 **a** 和 **b** 的值是如何变化的，能实现交换这两个变量的值吗？

### 1.3.4 关系运算符及关系表达式

#### 1. 关系运算符

关系运算符用于数值大小的比较。包括以下运算符：

<        (小于)  
 <=      (小于或等于)  
 >        (大于)  
 >=      (大于或等于)  
 ==      (等于)  
 !=      (不等于)

这里特别要提醒的是，在判断两个表达式是否相等时，一定要用两个等于号“==”，而不能用一个等于号“=”，后者表示赋值。



## 2. 关系表达式

用关系运算符将两个表达式连接起来的式子，称为关系表达式。关系表达式的一般形式可以表示为：

**表达式 关系运算符 表达式**

关系表达式的结果是逻辑型数据（详见 1.3.5 节），其值只有两种：0 或者 1。0 代表关系判断不成立，1 代表关系判断成立。

**例 1.8** 关系运算符的例子。

```
#include <stdio.h>
void main( )
{
    int a = 4, b = 5, c;
    c = a > b;    //(1)把关系表达式"a > b"的值赋值给变量 c
    printf( "c=%d", c );
    c = a < b;    //(2)把关系表达式"a < b"的值赋值给变量 c
    printf( "c=%d", c );
    c = a != b;   //(3)把关系表达式"a != b"的值赋值给变量 c
    printf( "c=%d ", c );
    c = a == b;   //(4)把关系表达式"a == b"的值赋值给变量 c
    printf( "c=%d ", c );
    c = a = b;    //(5)把变量 b 的值赋值给 a，然后把 a 的值赋值给 c
    printf( "c=%d\n", c );
}
```

该程序的输出结果为：

c=0 c=1 c=1 c=0 c=5

在上面的例子中，要特别注意语句(4)和语句(5)。语句(4)把关系表达式“a == b”的值（其值为 0）赋值给变量 c，“==”是关系运算符；而语句(5)把变量 b 的值赋值给 a，然后把赋值表达式“a=b”的值（即变量 a 的值）赋值给 c。 □

### 1.3.5 逻辑运算符及逻辑表达式

#### 1. 逻辑型数据

C 语言没有提供逻辑型数据类型，关系表达式的值（真或假）分别用数值 1 和 0 代表。C++ 增加了逻辑型数据类型，又称为布尔型（bool）。

逻辑型常量只有两个，即 false（假）和 true（真）。

逻辑型变量要用类型标识符 bool 来定义，它的值只能是 true 和 false 之一。如：

```
bool found, flag = false;    //定义逻辑变量 found 和 flag，并使 flag 的初值为 false
found = true;                //将逻辑常量 true 赋给逻辑变量 found
```

编译系统在处理逻辑型数据时，将 false 处理为 0，将 true 处理为 1，在内存中占一个字节，而不是将字符串“false”和“true”存放在内存中。因此，逻辑型数据可以与数值型数据进行运算。具体如下：

- 1) 如果逻辑型数据与其他数值型数据一起参与算术运算，则 true 为 1，false 为 0。例如假设变量 btemp 是逻辑型变量，它的值为 true，那么，赋值语句“a = 2 + btemp + false;”执行完后，a 的值为 3。
- 2) 将一个表达式的值赋值给一个逻辑型变量，则只要表达式的值为非 0，就按“真(true)”处理，如果表达式的值为 0，按“假(false)”处理。如：  
bool flag = 123 + 25; //赋值后 flag 的值为 true，即为 1

`bool found = 0.0;`      //赋值后 `found` 的值为 `false`，即为 0

### 2. 逻辑运算符

逻辑运算符用于数据的逻辑操作，可以将多个关系表达式组合成复杂的关系判断。逻辑运算符包括逻辑“与”（&&）、逻辑“或”（||）、逻辑“非”（!）3 种。前两者是双目运算符，第三个是单目运算符。其运算规则如下：

逻辑“与”（&&）运算：如表 1.2 所示，当且仅当表达式 `exp1` 和表达式 `exp2` 的值都为真（非 0）时，“`exp1 && exp2`”的结果才为真；其他情况，“`exp1 && exp2`”的结果均为假。例如，如果 `n` 的值为 4，那么“`n>4 && n<5`”的值就是假；“`n>=2 && n<5`”的值就是真。

逻辑“或”（||）运算：如表 1.2 所示，当且仅当表达式 `exp1` 和表达式 `exp2` 的值都为假（就是 0）时，“`exp1 || exp2`”的结果才为假；其他情况，“`exp1 || exp2`”的结果均为真。例如，如果 `n` 的值为 4，那么“`n>4 || n<5`”的值就是真；“`n<=2 || n>5`”的值就是假。

逻辑“非”（!）运算：如表 1.2 所示，如果表达式 `exp` 的值为真，那么“`! exp`”的值就是假；如果 `exp` 的值为假，那么“`! exp`”的值就是真。例如表达式“`!(4<5)`”的值就是假。

表 1.2 逻辑运算的真值表

exp1	exp2	exp1&& exp2	exp1    exp2	! exp1
真	真	真	真	假
真	假	假	真	假
假	真	假	真	真
假	假	假	假	真

### 3. 逻辑表达式

将两个关系表达式用逻辑运算符连接起来就成为一个逻辑表达式。其一般形式为：

**表达式 逻辑运算符 表达式**

注意：数学上的关系式“`3<x<5`”，在 C/C++ 语言中要表示成“`x>3 && x<5`”。

**逻辑表达式的值是一个逻辑值**，即为“真”或“假”。例如上面的例子中，如果 `x` 的值为 4，则关系表达式“`x>3`”的值为 `true`，关系表达式“`x<5`”的值也为 `true`，因此整个逻辑表达式的值为 `true`。

熟练掌握 C/C++ 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。如下面的例子。

#### 例 1.9 闰年的判定。

符合下面两个条件之一的年份为闰年：① 能被 4 整除，但不能被 100 整除；② 能被 100 整除，又能被 400 整除。例如 2004、2000 年是闰年，2005、2100 年不是闰年。

以图 1.5 为例，假设整个圆代表所有年份构成的集合。用条件(1)“能否被 4 整除”，将整个集合一分为二，其中子集(I)表示不能被 4 整除，该子集代表的年份不是闰年。对圆中剩下的部分，再施加条件(2)“能否被 100 整除”，又一分为二，其中子集(II)表示的年份能被 4 整除，但不能被 100 整除，这些年份是闰年。对圆中剩下的部分，再施加条件(3)“能否被 400 整除”，又一分为二，其中子集(III)表示的年份能被 400 整除，是闰年；子集(IV)表示的年份能被 100 整除，但不能被 400 整除，不是闰年。因此在图 1.5 中，子集(II)和(III)表示的年份是闰年。

假设用变量 `year` 表示年份，则子集(II)所表示的年份可以用下面的关系表达式表示：

`year % 4 == 0 && year % 100 != 0。`

子集(III)所表示的年份可以用关系表达式“`year % 400 == 0`”表示。二者是逻辑“或”的关系，因此可以用下面的逻辑表达式来判定闰年：

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

说明：如果上述逻辑表达式的值为 1，则 `year` 所表示的年份为闰年；如果该逻辑表达式的值

为 0，则 year 所表示的年份为平年。

□

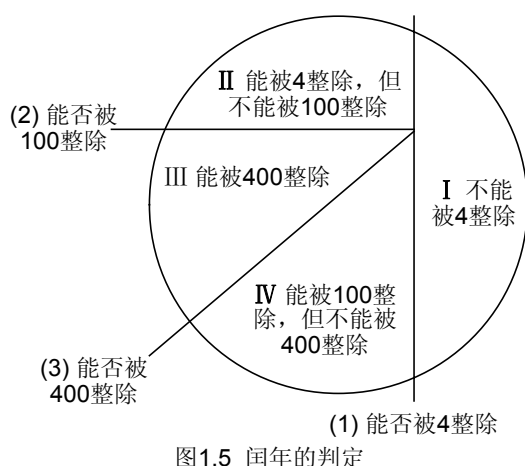


图1.5 闰年的判定

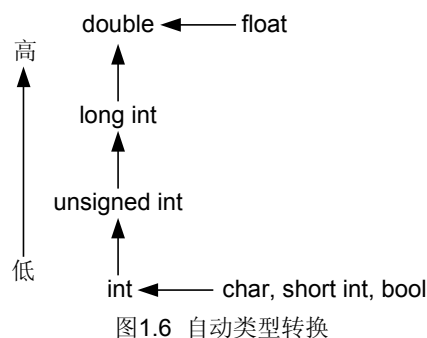


图1.6 自动类型转换

### 1.3.6 类型转换

在 C/C++ 语言中有两种类型转换：自动类型转换和强制类型转换。每种类型转换适用范围、使用方法、处理规则等都不一样。

#### 1. 自动类型转换

在表达式中经常会出现不同类型数据之间进行运算，如 `10+'a'+1.5-8765.1234*'b'`。

不同类型的数据要先转换成同一类型，然后进行运算。其目的是尽量保证精度，不丢失数据。转换的规则是朝精度高的数据类型转换，如图 1.6 所示。

横向向左的箭头表示必定的转换。例如：两个 `float` 型数据的运算也是要把它们都转换成 `double` 型再运算。

纵向的箭头表示当运算对象为不同的类型时转换的方向。例如：一个 `int` 型的数据加上一个 `double` 型的数据，先把 `int` 型的数据转换成 `double` 型，再相加。

自动类型转换是自动进行的。

假设要求圆锥体的体积，圆锥体的高为 2.5，底面圆半径为 1.5，如果用表达式 “`1/3*3.14*1.5*1.5*2.5`” 去求，则结果必为 0。因为先执行运算 “`1/3`”，不会进行自动类型转换，结果为 0。将表达式改写成 “`1.0/3*3.14*1.5*1.5*2.5`” 就正确了，因为在执行运算 “`1.0/3`” 时，会把整数 3 转换成 `double` 型数据 3.0，这样得到的结果就不是 0 了。

#### 2. 强制类型转换

除了自动类型转换，在有的时候，也需要强制将某种类型的数据强制转换成另一种类型。强制类型转换运算符的使用形式是：

**(类型名)(表达式)**

或者：

**类型名(表达式)**

具体应用见下面的例子。

**例 1.10** 自动类型转换与强制类型转换。

```
#include <stdio.h>
void main( )
{
    int a = 3, b = 4;
    double d1 = a/b; //(1)
```

```

        double d2 = (double)a/b; //(2) 或: d2 = double(a)/b;
        printf( "d1=%fnd2=%f\n", d1, d2 );
    }

```

该程序的输出结果为:

d1=0.000000

d2=0.750000

在语句(1)中, 两个整数相除, 不保留余数, 因此得到的结果是 0.000000。而在语句(2)中, 先是把操作数 **a** 强制转换成 **double** 型数据, 这样就是一个 **double** 型数据除以一个 **int** 型数据, 要进行自动类型转换, 将 **b** 由 **int** 型转换成 **double** 型后再运算, 结果是 0.750000。□

## 练习

- 1.7 阅读下面的程序, 分析程序的执行结果。然后运行该程序, 记录程序的输出, 看看程序的输出跟你的分析是否吻合。

```

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

```

- 1.8 阅读下面的程序, 分析各表达式执行完后变量 **a** 的值。然后运行该程序, 记录程序的输出, 看看程序的输出跟你的分析是否吻合。

```

#include <stdio.h>
void main( )
{
    int a = 3; a += a; printf( "a=%d\n", a );
    a = 3; a *= 2 + 3; printf( "a=%d\n", a );
    a = 3; a /= a + a; printf( "a=%d\n", a );
    int n = 5; a = 3; a %= ( n %= 2 ); printf( "a=%d\n", a );
    a = 3; a += a -= a *= a; printf( "a=%d\n", a );
}

```

- 1.9 阅读下面的程序, 分析程序的执行结果。然后运行该程序, 记录程序的输出, 看看程序的输出跟你的分析是否吻合。

```

#include <stdio.h>
void main( )
{
    int x = 5, y = 7, z = 9;
    printf( "%d\n", (x>5 && x<9) || (z>8) );
    printf( "%d\n", !(y==5) && (x>3 && z<12) );
    printf( "%d\n", x && (y>8) );
    printf( "%d\n", !y || y<9 );
    printf( "%d\n", x<y<z );
    printf( "%d\n", z>y>x );
}

```

- 1.10 编程求解: 已知 2009 年 1 月 1 号是星期四, 请问 2009 年 1 月 31 号是星期几?

- 1.11** 请改写例 1.7，编程实现：从键盘上输入两个整数，保存到变量 **a** 和 **b**，不借助中间变量，经过一定步骤后，变量 **a** 的值为输入的两个整数之和，变量 **b** 的值为输入的两个整数之差。
- 1.12** 请分析下面各表达式的值，然后编写程序，用程序计算每个表达式的值，比较你分析的结果跟计算机求解的结果是否吻合。设在执行**每个**表达式之前，**a, b, c** 的值分别为：**a = 3, b = 4, c = 5**。
- 提示：如果想让程序计算某个表达式的值，可以直接放到 **printf** 函数进行输出，或者把表达式赋值给某个变量，然后把该变量的值输出来，代码如下：
- ```
printf( "%d", 表达式 ); //如果表达式的值是整数，用%d；如果是浮点数，用%f 或%lf  
int exp = 表达式; printf( "%d", exp ); //根据需要将 exp 定义成整型或浮点型
```
- (1) **a+b > c**                                  (2) **b==c**
- (3) **a || b+c**                                (4) **b+c || b-c**
- (5) **!(b=a) && !(c==a)**                      (6) **a/b && b/a**
- (7) **(double)a/b && b/a**
- 1.13** 请用一个逻辑表达式来判断一个年份 **year** 是否是平年。要求：如果表达式的值为 1，则 **year** 所表示的年份是平年；否则（即表达式的值为 0），**year** 所表示的年份不是平年（即闰年）。请至少采用两种不同的方法来表示。
- 1.14** 请用一个逻辑表达式来判断一个正整数 **num** 能否同时被 2、3、7 整除。要求：如果表达式的值为 1，则 **num** 能同时被 2、3、7 整除；否则（即表达式的值为 0），**num** 不能同时被 2、3、7 整除。

## 1.4 C/C++的语句

C/C++程序中最小的独立单位是语句，C/C++的语句是用分号结束的。C/C++语句可以分为以下4种。

## 1. 声明语句

例如声明变量的语句:

```
int a, b;
```

又如对函数进行声明（详见 1.9.5 节）的语句：

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

在 C/C++ 中，对变量等的声明可以出现在函数中的任何行（只需要保证出现在引用该变量或调用该函数的语句前面就可以了），也可以放在函数之外。对变量而言，在函数里面声明的变量为**局部变量**（其有效范围为变量声明处到函数结束），在函数外声明的变量为**全局变量**（其有效范围为变量声明处到整个文件结束），关于局部变量和全局变量的进一步描述，请参考其他教材。

## 2. 执行语句

所谓执行语句就是能完成一定操作的语句。执行语句包括：

(1) 控制语句，完成一定的控制功能。C/C++有 9 种控制语句，主要是 1.7、1.8 节将介绍的、用于实现一些程序控制结构的语句。即：

- 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** 语句或循环语句)
- ⑦ **switch** (多分支选择语句)
- ⑧ **goto** (转向语句)
- ⑨ **return** (从函数返回语句)

(2) 函数调用语句。函数调用语句由一次函数调用加一个分号构成一个语句，例如 C/C++ 语言中实现输入/输出的 **scanf** 函数和 **printf** 函数调用语句：

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

(3) 表达式语句。由一个表达式加一个分号构成一个语句。最典型的是：由赋值表达式构成一个赋值语句。如下面的例子：

```
i = i + 1 //是一个赋值表达式
```

```
i = i + 1; //是一个赋值语句
```

任何一个表达式加一个分号都可以成为一个语句。表达式能构成语句是 C/C++ 语言的一个重要特色。C/C++ 程序中大多数语句是表达式语句（包括函数调用语句）。

### 3. 空语句

空语句是指只有一个分号的语句，即：

```
;
```

空语句什么也不做。空语句可以用来做为循环语句中的循环体，如例 2.4 程序中的循环体。

### 4. 复合语句

可以用花括号（“{” 和 “}”）把一些语句括起来，成为复合语句。如下面是一个复合语句。

```
int x = 5, y = 7;
{
    int z;
    z = x + y;
    printf( "%d\n", z);
}
```

注意，在复合语句中声明的变量，其有效范围仅限于复合语句内，在复合语句后面不能引用这些变量。因此如果将上述代码中的 **printf** 语句移至复合语句后面，则将有语法错误。

## 1.5 数学函数的使用

在进行数据处理时，仅有 1.3 节介绍的运算符和表达式往往是不够的，还经常需要使用数学函数。

### 1.5.1 常用的数学函数

在头文件 **math.h** 中声明了许多数学函数，下面列出了常用数学函数的原型（函数的声明及函数的原型详见 1.9 节）。

```
int abs( int x );           //求整型数据 x 的绝对值
double fabs( double x );   //求浮点型数据 x 的绝对值
double sqrt( double x );    //求 x 的平方根
double exp( double x );     //求 ex，e 为自然对数的底
double pow( double x, double y ); //求指数 xy
double sin( double x );     //求 x(弧度)的正弦值
double cos( double x );     //求 x(弧度)的余弦值
```

```
double asin( double x );    //求反三角函数 arcsin(x), 得到的结果是弧度
double acos( double x );    //求反三角函数 arccos(x), 得到的结果是弧度
double log( double x );     //求 x 的自然对数, 即底为 e
double log10( double x );   //求 x 的对数(底为 10)
```

另外要注意, 头文件math.h提供了开平方根的函数sqrt, 但没有开三次方根的函数, 求x的三次方根要使用pow函数, 因为对x开三次方根等价于 $x^{1/3}$ 。

### 1.5.2 数学函数的使用

在程序中如果要使用数学函数, 首先要把头文件 math.h 包含进来, 其次要注意调用数学函数的方法和形式。以 pow 函数为例加以说明, pow 函数的原型是:

```
double pow( double x, double y );
```

函数名是pow; 函数名后面圆括号内用逗号隔开的是两个double型的参数, 即在调用pow函数时, 需要带两个参数; 函数名前面有double类型说明符, 表示该函数执行完毕会返回一个double型的数据, 即 $x^y$ 的结果。例如要对 2.5 开 3 次方根, 即要求  $2.5^{1/3}$ , 可以使用下面的代码:

```
double x = 2.5, y = 1.0/3, z;    //注意不能写成 1/3, 否则 y 的值为 0
```

```
z = pow(x,y);    //对 x 开 3 次方根, 并把结果赋值给变量 z
```

注意, 初学者容易将函数调用写成如下的形式, 这些都是错误的:

```
z = double pow(x, y);
```

```
z = double pow( double x, double y);
```

数学函数具体使用方法见下面的例 1.11 和 1.12。

**例 1.11** 已知三角形的三条边的长度为 a、b 和 c (从键盘输入), 计算三角形三个内角 (角度)。

如图 1.7 所示, 记角  $\angle CAB$  为  $\angle 1$ , 则由余弦定理可求得  $\angle 1$  的余弦  $\cos \angle 1$ , 然后利用反余弦函数可以求得  $\angle 1$  (弧度), 再转换成角度输出。

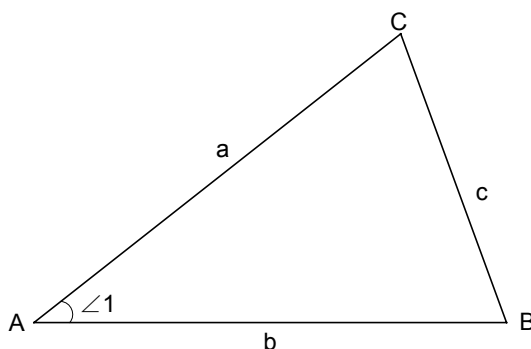


图1.7 计算三角形三个内角

代码如下:

```
#include <stdio.h>
#include <math.h>
void main( )
{
    double a, b, c;
    printf( "Please enter three float numbers : " );
    scanf( "%lf%lf%lf", &a, &b, &c );    //假定输入的 3 条边能构成三角形
    double cos1, angle1;
    double pi = 2*asin(1); //用反三角函数把 π 精确地表示出来
    cos1 = (a*a+b*b-c*c)/(2*a*b);
```

```

    angle1 = acos( cos1 );
    angle1 = angle1*180/pi;
    printf( "%.4f\n", angle1 );
}

```

该程序的运行示例如下：

Please enter three float numbers : 8.7 6.5 9.2 ✓

72.8767

上面的程序只求了 $\angle 1$ ，其他两个内角的求法类似。注意，数学上反余弦函数通常是用 `arccos` 来表示，但在 `math.h` 中，反余弦函数函数名为 `acos`。□

另外要注意，在例 1.11 的 `printf` 语句中，“%.4f”表示控制输出的浮点型数据保留小数点后 4 位有效数字，详见附录 D。

**例 1.12** 当海狸咬一棵树的时候，它从树杆咬出一个特别的形状。树杆上剩下的部分好像 2 个圆锥载体用一个直径和高相等的圆柱体连接起来一样。有一只很好奇的海狸它关心的不是要把树咬断，而是想计算出在给定要咬出一定体积的木屑的前提下，圆柱体的直径应该是多少。

如图 1.8 所示，假定树杆是一个直径为  $D$  的圆柱体，海狸咬的那一段高度也为  $D$ 。那么给定要咬出体积为  $V$  的木屑，内圆柱体的直径  $d$  应该为多少？其中  $D$  和  $V$  都是整数。

题目中有 3 个量： $D$ 、 $V$  和  $d$ ，其中  $D$  和  $V$  是从键盘输入，要计算  $d$  的值并输出。推导出来的关系式是： $V = (D^3 - d^3) * \pi / 6$ 。代码如下：

```

#include <stdio.h>
#include <math.h>
void main( )
{
    int V, D; double d, d3;    //变量 d3 用于存储 d 的 3 次方，然后对其开 3 次方根求 d
    double pi = 2*asin(1);    //把 π 用数学函数表示出来
    scanf( "%d%d", &D, &V ); //输入 D 和 V 的值
    d3 = D*D*D-6*V/pi;        //计算 d 的 3 次方
    d = pow( d3, 1.0/3 );      //计算 d
    printf( "%.3f\n", d );     //输出到小数点后 3 位
}

```

该程序的运行示例如下：

10 250 ✓

8.054

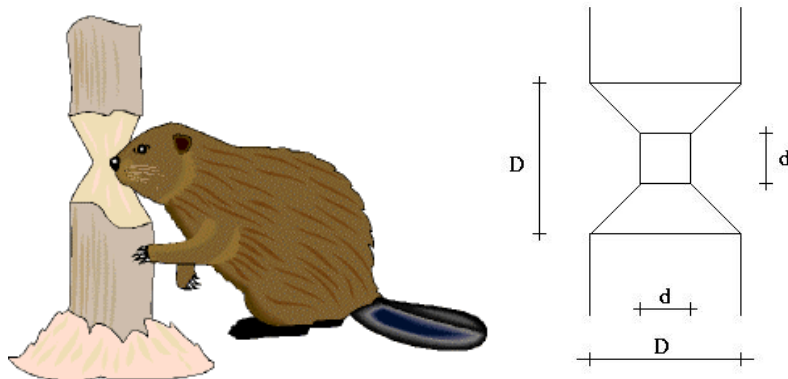


图 1.8 海狸咬树示意图

初学者容易犯的错误是在得到上述数学式子后，直接将这个数学式子表示在程序中：

```
V = pi*(D*D*D - d*d*d)/6;
```

以为程序会去求解这个数学式子。实际上这只是数学上的一个式子，要让程序去计算  $d$ ，必须先



把  $d$  的数学表示式表示出来, 即:  $d = \sqrt[3]{D^3 - 6 * V / \pi}$ , 然后在程序中把该数学表示式表示成合法的表达式, 并把该表达式的值赋值给变量  $d$ , 即  $d = \text{pow}(D * D * D - 6 * V / \text{pi}, 1.0 / 3)$ 。上述代码为了简化这个式子, 先求出了  $d$  的 3 次方  $d^3$ , 然后再用  $\text{pow}$  函数求  $d$ 。□

## 练习

1.15 编写程序, 计算下列数学式子的值。

$$(1) \sqrt{(\sin(x \cdot \pi))^{2.5}}$$

$$(2) \frac{1}{2} \left( a \cdot x + \frac{a + x}{4a} \right)$$

$$(3) \frac{c^{x^2}}{\sqrt{2 \cdot \pi}}$$

其中  $x = 0.2$ ,  $a = 2$ ,  $c = 3.5$

1.16 已知三角形三条边长  $L_1$ 、 $L_2$  和  $L_3$  (均为浮点型数据, 从键盘输入), 求其外接圆面积。

提示: 如图 1.9 所示, 利用公式  $R = (L_1 / 2) / \sin \angle 1$  求外接圆半径, 其中  $\angle 1$  可通过余弦定理求得。

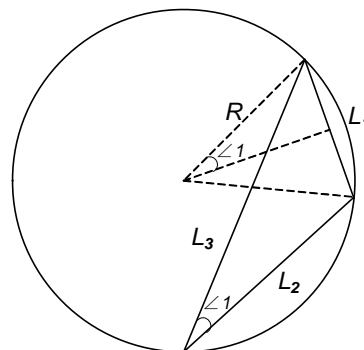


图1.9 求三角形外接圆面积

## 1.6 算法及程序控制结构

处理数据的过程通常是按照一系列的步骤来进行的。这些步骤的执行顺序就是算法, 而这些步骤的执行顺序是通过程序控制结构来控制的。本节介绍算法的概念及程序控制结构。

### 1.6.1 算法及控制结构

什么是算法? **算法**就是为解决某个问题而采取的一系列步骤。算法必须具体地指出在执行时每一步应当怎样做。表示算法的方式有很多种。本教材中采用**传统流程图**的方式表示算法, 这种方式用图的形式表示算法, 比较形象直观。传统流程图使用的符号如表 1.3 所示。

表 1.3 流程图中使用到的符号及其含义

| 符 号 | 含 义                | 符 号 | 含 义                           |
|-----|--------------------|-----|-------------------------------|
|     | 起止框, 表示算法的开始和结束    |     | 处理框, 表示初始化或赋值等操作              |
|     | 输入输出框, 表示数据的输入输出操作 |     | 判断框, 表示根据一个条件决定执行两种不同操作中的其中一个 |
|     | 流程线, 表示流程的方向       |     | 连接点, 用于流程的分页连接                |

例如, “求一个数的绝对值”的算法用流程图来表示就是图 1.10。从这个例子可以看出, 用流程图表示的算法结构很清晰, 很直观。

一个良好的算法通常由 3 种基本**程序控制结构**组成: **顺序结构**、**选择结构**和**循环结构**。1.6.2 节介绍顺序结构, 1.7 节和 1.8 节将介绍选择结构和循环结构。

## 1.6.2 顺序结构

所谓**顺序结构**，就是从上到下顺序执行各语句，没有其他分支。如前面所有的例子及下面的例 1.13。

**例 1.13** 从键盘上输入三角形的 3 条边的边长  $a$ 、 $b$ 、 $c$ （假定输入的 3 条边长可以构成三角形），根据公式  $S = \sqrt{p * (p - a) * (p - b) * (p - c)}$ ，其中  $p = (a + b + c) / 2$ ，求三角形的面积。

代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    double a, b, c, area, p;
    printf( "please enter a,b,c: " );
    scanf( "%lf%lf%lf", &a, &b, &c );    //输入三角形三条边的边长
    p = (a+b+c)/2;
    area = sqrt( p*(p-a)*(p-b)*(p-c) );    //计算三角形面积
    printf( "area=%f\n", area );    //输出面积
}
```

该程序的运行示例如下：

```
please enter a,b,c: 4.5 8.8 6.7 ✓
area=14.758049
```

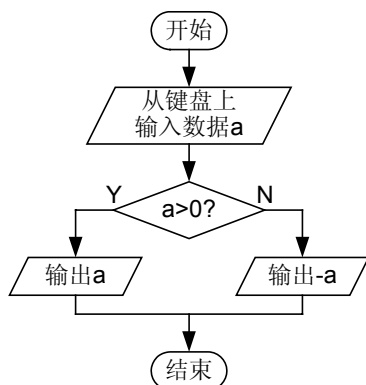


图1.10 流程图的例子

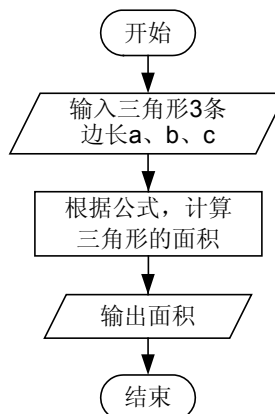


图1.11 求三角形面积的流程图

程序的流程图如图 1.11 所示。从流程图可以看出，该程序从输入三角形边长、计算三角形面积，到输出三角形面积，从上到下顺序执行，没有其他分支，所以是顺序结构。□

**思考 1.6：**如果要采用公式  $area = (a \times b \times \sin \angle 1) / 2$  来求三角形面积（ $\angle 1$  为边  $a$  和  $b$  的夹角），例 1.13 的程序该如何修改。

### 练习

- 1.17 已知圆柱体的底面半径  $r$  为 2.7，高  $h$  为 3.5，求圆柱体的体积并输出，保留小数点后面两位有效数字。
- 1.18 编程实现：求平面上一个凸四边形的面积。凸四边形 4 个顶点的坐标按顺时针顺序输入，如图 1.12 所示，这四个点的坐标可以按 A、B、C、D 的顺序输入，也可以按 C、D、A、B 的顺序输入，等等。

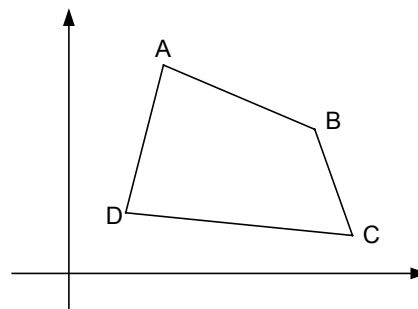


图1.12 求平面上凸四边形的面积

1.7 选择结构

在程序中经常需要根据一定的条件选择执行的操作，这就要用到**选择结构**（或称为**分支结构**）。在例 1.13 中，如果不知道输入的 3 条边长 a、b、c 是否能构成三角形，则在程序中就应该进行判断：如果能构成三角形，则计算面积；否则，输出“这三条边不能构成三角形”的信息。

在 C/C++语言中，用于实现选择结构的语句有：if 语句和 switch 语句。另外，C/C++语言还提供了条件表达式，也可以实现条件判定。

1.7.1 if 语句

C/C++语言提供了 3 种形式的 if 语句。表 1.4 列出了这 3 种形式，对这 3 种形式的语法格式、执行过程等作了对比，并给出了实例。

表 1.4 if 语句的 3 种形式

|                  | 形式一                                                       | 形式二                                                    | 形式三                                                                                                                                                               |
|------------------|-----------------------------------------------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 语<br>法<br>格<br>式 | <b>if(表达式) 语句</b>                                         | <b>if(表达式) 语句 1</b><br><b>else 语句 2</b>                | <b>if(表达式 1) 语句 1</b><br><b>else if(表达式 2) 语句 2</b><br><b>else if(表达式 3) 语句 3</b><br>...<br><b>else if(表达式 m) 语句 m</b><br><b>else 语句 n</b>                        |
| 执<br>行<br>过<br>程 | 先执行表达式，如果表达式的值为真(非 0)，则执行 if 结构中的语句，否则不执行。其流程图见图 1.13(a)。 | 先执行表达式，如果表达式的值为真(非 0)，则执行语句 1，否则执行语句 2。其流程图见图 1.13(b)。 | 先执行表达式 1，如果表达式的值为真(非 0)，则执行语句 1，整个 if 结构执行完毕；如果表达式 1 的值为假(0)，则继续判断表达式 2 的值是否为真，如果为真，则执行表达式 2，整个 if 结构执行完毕；如果表达式 2 的值为假(0)，则继续判断表达式 3，...。其流程图见图 1.13(c)。          |
| 例<br>子           | if(x>y)<br>printf("%d", x);                               | if(x>y) printf("%d", x);<br>else printf("%d", y);      | if( number>500 ) cost = 0.15;<br>else if( number>300 ) cost = 0.10;<br>else if( number>100 ) cost = 0.075;<br>else if( number>50 ) cost = 0.05;<br>else cost = 0; |

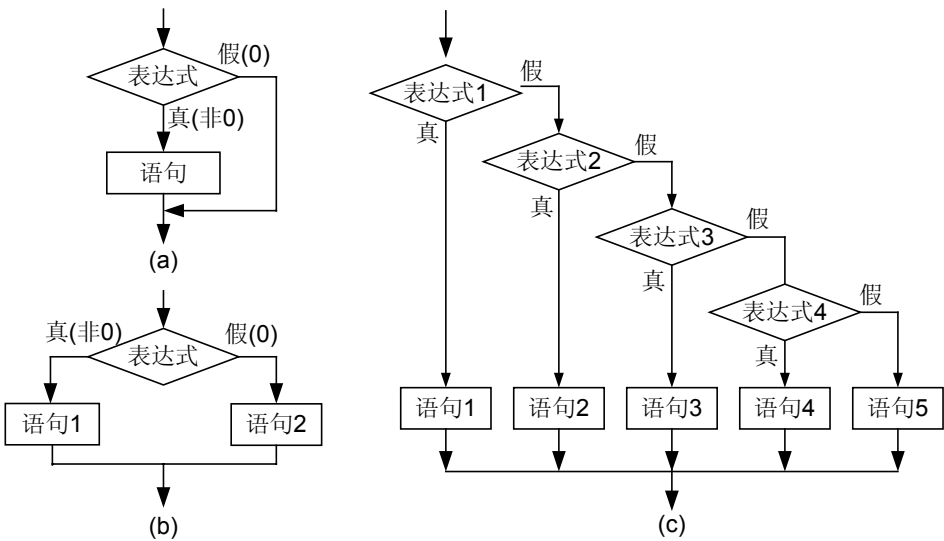


图1.13 if语句的3种形式

第 1 种形式的 if 语句详见例 1.14，第 2 种形式的 if 语句详见例 1.15、1.16，第 3 种形式的 if 语句详见例 1.17。

**例 1.14** 从键盘上输入一个年份，输出该年 2 月份的天数。

本题的思路是，平年的 2 月份有 28 天，闰年的 2 月份多 1 天。因此，在下面的程序中，变量 `days` 在定义时初始化为 28，用 if 语句判断输入的年份是否满足闰年的条件，如果满足则 `days` 加 1。代码如下：

```
#include <stdio.h>
void main( )
{
    int year, days = 28;
    printf( "Please input a year : " );
    scanf( "%d", &year );
    if( (year % 4 == 0 && year % 100 != 0) || year % 400 == 0 )
        days = days + 1; //闰年 2 月份要多 1 天
    printf( "days = %d\n", days );
}
```

该程序的运行示例如下：

Please input a year : 2009

days = 28

在上面程序的 if 语句中，只有一个分支，当条件成立时，变量 `days` 的值加 1；如果条件不成立，则不执行操作。 □

**思考 1.7：**请画出上述程序的流程图。

**例 1.15** 从键盘上输入一个整数，判断该整数是奇数还是偶数。

对输入的整数，将其对 2 取余，如果余数为 0，则该整数为偶数；否则（即余数为 1），该整数为奇数。代码如下：

```
#include <stdio.h>
void main( )
{
    int a;
    printf( "please enter an integer : " );
    scanf( "%d", &a );
    if( a%2==0 ) printf( "%d is an even number.\n", a );
    else printf( "%d is an odd number.\n", a );
}
```

该程序的流程图如图 1.14 所示，以下给出两个运行示例。

该程序的运行示例 1 如下：

please enter an integer : 58✓

58 is an even number.

该程序的运行示例 2 如下：

please enter an integer : 13✓

13 is an odd number.

在上面程序的 if 语句中，有两个分支，当条件成立时，输出“是偶数”的信息；当条件不成立时，输出“是奇数”的信息。 □

**思考 1.8：**例 1.15 中 if 语句的条件还可以用怎样的表示式来表示？还可以将 if 语句改成怎样的形式（将两个分支的顺序调换一下）？

**例 1.16** 从键盘上输入三角形的 3 条边的边长  $a$ 、 $b$ 、 $c$ ，判断是否能构成三角形，如果能构成三角形，则根据公式求三角形面积；否则输出“这三条边不能构成三角形”。

代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    double a, b, c;
    printf( "please enter a,b,c: " );
    scanf( "%lf%lf%lf", &a, &b, &c );
    //判断是否能构成三角形
    if( a+b>c && b+c>a && c+a>b )
    { //复合语句开始
        double s, area; //复合语句中的变量
        s = (a+b+c)/2;
        area = sqrt( s*(s-a)*(s-b)*(s-c) );
        printf( "area=%0.4f\n", area );
    } //复合语句结束
    else printf( "it is not a trilateral!\n" );
}
```

该程序的流程图如图 1.15 所示，以下给出两个运行示例。

该程序的运行示例 1 如下：

```
please enter a,b,c: 3.4 2.2 4.7 ✓
area=3.4589
```

该程序的运行示例 2 如下：

```
please enter a,b,c: 2.3 5.5 10.1 ✓
it is not a trilateral!
```

在上述程序的 if 语句中，充当条件判断的是逻辑表达式“ $a+b>c \ \&\& \ b+c>a \ \&\& \ c+a>b$ ”。这个 if 语句有 2 个分支：如果该表达式的结果为 1，则表示 3 条边能够成三角形，计算面积并输出；否则输出“不能构造三角形”的提示信息。 □

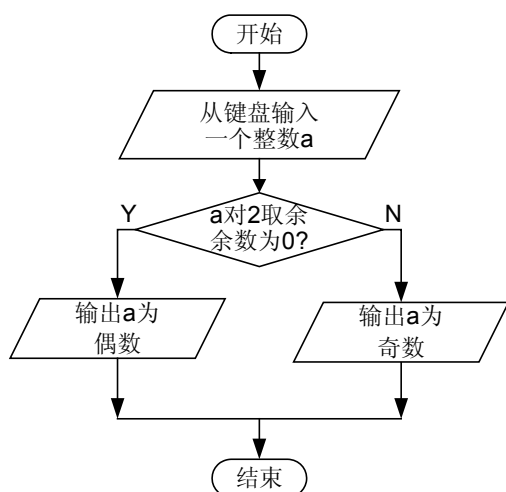


图1.14 判断一个整数是奇数还是偶数

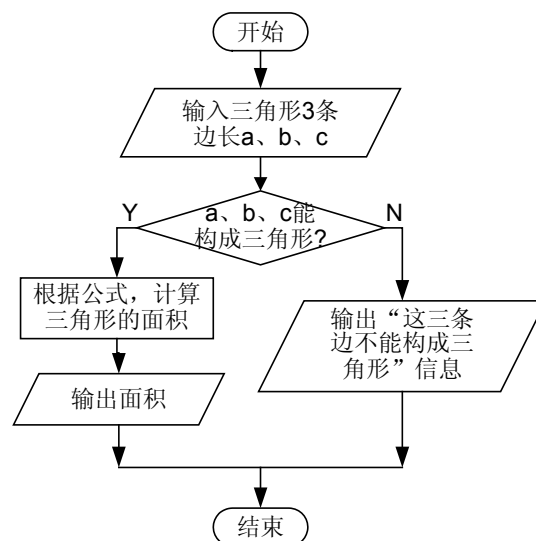


图1.15 求三角形面积(分支结构)流程图

请特别注意：如果在 if 结构的某个分支中有多条语句，那么必须用花括号扩起来，成为一个复合语句。在复合语句外面，不能使用复合语句中定义的变量（如上述代码中的变量  $s$  和  $area$ ）。

第 3 种形式的 if 语句也可以实现多个分支的选择，如下面的例子。

**例 1.17** 数学上的分段函数。编写程序，输入 x，输出 y 的值。

$$y = \begin{cases} x^2 & (x < 0) \\ \sqrt{x} & (0 \leq x < 9) \\ x - 6 & (x \geq 9) \end{cases}$$

代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    double x, y;
    printf( "Please input x : ");
    scanf( "%lf", &x );
    if( x<0 ) y = x*x;
    else if( x<9 ) y = sqrt(x);
    else y = x - 6;
    printf( "y=%f\n", y );
}
```

该程序的运行示例如下：

Please input x : 2.5 ✓

y=1.581139

□

**思考 1.9：**初学者容易将上述 if 结构写成如下几种情形：

情形 1：

```
if( x<0 ) y = x*x;
if( 0<=x<9 ) y = sqrt(x);
if( x>=9 ) y = x-6;
```

情形 2：

```
if( x<0 ) y = x*x;
else if( 0<x<9 ) y = sqrt(x);
else y = x-6;
```

情形 3：

```
if( x<0 ) y = x*x;
else if( x>0 && x<9 ) y = sqrt(x);
else ( x>=9 ) y = x-6;
```

请分析上述 if 结构错在哪里。

**例 1.18** 编写程序，求三个数据的最大值（或最小值）。

在程序中经常要求两个数据或三个数据、甚至多个数据的最大值（或最小值）。方法是先设最大值 max 为第一个数；然后将剩下的每个数都跟当前 max 的值进行比较，如果该数比 max 的值还大，则将 max 的值更新为该数；最后求得的 max 就是最大值。代码如下：

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

```

    if( b>max )   max = b;
    if( c>max )   max = c;
    printf( "max=%d\n", max );
}

```

该程序的运行示例如下：

```
78 56 99✓
```

```
max=99
```

□

**思考 1.10：**上述程序中两个 if 结构是否存在 if-else 的关系，也就说能否将这两个 if 语句合并成一个 if 语句（第 3 种形式）？

### 1.7.2 条件运算符与条件表达式

如果在 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给同一个变量赋值时，可以用简单的条件运算符和条件表达式来表达。例如，若有以下 if 语句：

```

if( a>b )   max = a;
else   max = b;

```

则可以用条件运算符( ? : )和条件表达式来表达：

```
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，则条件表达式的值就取“:”前面的值，即条件表达式的值为 a，否则条件表达式的值为“:”后面的值，即 b。

条件运算符要求有 3 个操作对象，称为三目运算符，它是 C/C++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**

条件运算符的执行顺序是：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。

条件运算符优先于赋值运算符，因此，表达式“max = (a>b) ? a : b”的求解过程是：先求解条件表达式，其值为 a 和 b 二者中的较大者；再将该值赋值给 max。

**例 1.19** 编写程序，求三个数据的最大值（条件表达式实现）。

例 1.18 的程序可以用另外一种思路来编写：先用条件表达式求两个数的较大者，再将这个较大者跟第 3 个数进行比较。代码如下：

```

#include <stdio.h>
void main( )
{
    int a, b, c;
    scanf( "%d%d%d", &a, &b, &c );
    int max = a>b ? a : b;    //先求两个数的较大者
    if( c>max )              //将第 3 个数与前面求得的 max 进行比较
        max = c;
    printf( "max=%d\n", max );
}

```

该程序的运行示例如下：

```
78 56 99✓
```

```
max=99
```

□

### 1.7.3 switch 语句

**switch** 语句用来实现多分支选择结构。虽然嵌套结构的 **if** 语句也可以实现多分支，但如果分支较多，则嵌套的 **if** 语句层数多，程序就会繁琐，而 **switch** 语句直接处理多分支选择，形式更简洁。**switch** 语句的一般形式如下：

```
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，某公司根据客户的信用等级 **credit** 来计算折扣 **discount**，等级越高，折扣越大。可以用 **switch** 语句实现（注意，该 **switch** 语句存在逻辑错误，下面会说明）：

```
switch( credit )
{
    case 1: discount = 0.01;
    case 2: discount = 0.015;
    case 3: discount = 0.022;
    case 4: discount = 0.03;
    case 5: discount = 0.04;
}
```

说明：

(1) **switch** 后面括号内的“表达式”，允许为任何类型（算术表达式、关系表达式、逻辑表达式等），但是其值必须是整型或者字符型，**case** 后面的常量表达式的值也必须是整数或者字符型。

(2) 当 **switch** 表达式的值与某一个 **case** 子句中的常量表达式的值相匹配时，就执行此 **case** 子句中的内嵌语句，若所有的 **case** 子句中的常量表达式的值都不能与 **switch** 表达式的值匹配，就执行 **default** 子句的内嵌语句。**default** 子句可以没有，如上面的例子。

(3) 在执行 **switch** 语句时，根据 **switch** 表达式的值找到与之匹配的 **case** 子句，从此 **case** 子句开始执行下去，不再进行判断。例如，上面的例子中，若 **credit** 的值等于 3，则最终求得的折扣 **discount** 为 0.04。

因此，在执行一个 **case** 子句后，应该根据需要使流程跳出 **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( credit )
{
    case 1: discount = 0.01; break;
    case 2: discount = 0.015; break;
    case 3: discount = 0.022; break;
    case 4: discount = 0.03; break;
    case 5: discount = 0.04; break;
}
```

最后一个子句也可以不加 **break** 语句。这时若 **credit** 的值等于 3，则求得的折扣 **discount** 为 0.022，这是正确的。各分支加了 **break** 语句后的 **switch** 结构流程图见图 1.16。

(4) 在 **case** 子句中虽然包含一个以上执行语句，但可以不必用花括号括起来，会自动顺序执行本 **case** 子句中所有的执行语句。



(5) 多个 **case** 可以共用一组执行语句，如：

```
...
case 1:
case 2:
case 3: discount = 0.022; break;
...
```

则信用等级为 1、2、3 的客户，折扣都是 0.022。

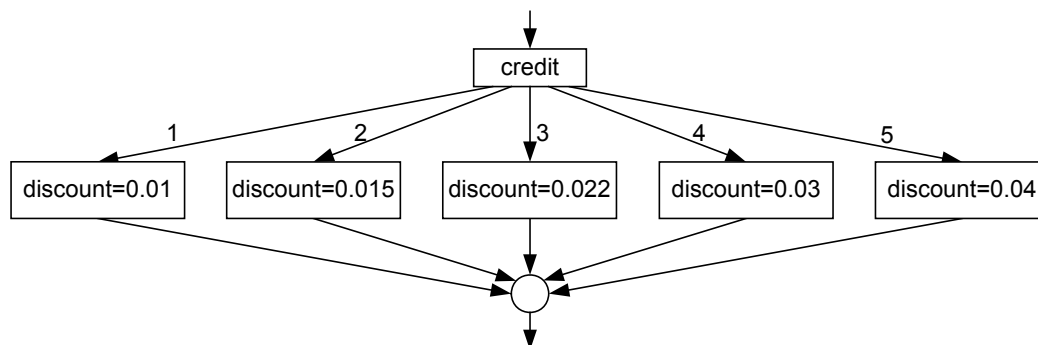


图1.16 switch结构流程图

使用 **switch** 语句关键在于如何构造括号内的表达式，现以下的例子加以说明。

**例 1.20** 输入百分制的成绩 **score**（假定输入的成绩范围在 0~100 之间，允许有 1 位小数，如 72.5 分），要求输出对应的五分制成绩，90.0~100 分为 A，80.0~89.9 为 B，70.0~79.9 为 C，60.0~69.9 为 D，60 分以下为 E。

首先要注意 **switch** 语句的格式。**switch** 括号后面的表达式的值必须是整数类型（或字符型），**case** 后面的常量表达式的值也必须是整数（或字符型）。因此不能直接把 **score** 放到 **switch** 后面的括号里。

其次，要尽量减少 **switch** 语句中的 **case** 分支。即使把 **score** 取整，符合 **switch** 语句的要求，如果直接对得到的整数进行判断：100 分为 A，99 分为 A，…，1 分为 E，0 分为 E。结果有 101 个 **case** 分支，如下面的程序所示，这显然是不现实的。

```
switch( (int)score )
{
case 100: printf( "A" ); break;
case 99:  printf( "A" ); break;
...
case 1:   printf( "E" ); break;
case 0:   printf( "E" ); break;
}
```

如何减少 **case** 分支而不影响程序功能呢？首先可以将 **score** 除以 10 取整，这样就将 101 个分支降为 11 个分支。然后我们发现 **score** 除以 10 取整后，得到的 5、4、3、2、1、0 共 6 个分支可归入到 **default** 分支。这样分支数大大减少了。最终的程序如下：

```
#include <stdio.h>
void main( )
{
    double score;
    int grade;
    printf( "please input a score : " );
    scanf( "%lf", &score );    //输入百分制成绩
    grade = score/10;
```

```

switch( grade )
{
case 10:
case 9: printf("A\n"); break;    //2 个 case 子句共用一组执行语句
case 8: printf("B\n"); break;
case 7: printf("C\n"); break;
case 6: printf("D\n"); break;
default: printf("E\n");
}
}

```

该程序的运行示例如下：

please input a score : 89.5 ✓

B

在上面程序的 switch 结构中，“case 10”和“case 9”两个分支共用一段执行语句。 □

**思考 1.11：**例 1.20 中的 switch 结构是否可以改成 if 结构，如何改。

为了让读者进一步掌握 switch 语句的使用，下面再讲解了一个应用 switch 结构的例子。

**例 1.21** 输入一个职工的月薪 salary（假定 salary 为大于 0.0 的浮点数，不需判断），计算并输出他应交的个人所得税  $\text{tax} = \text{rate} \times (\text{salary} - 850)$ 。其中 rate 的计算方式如下：

当  $\text{salary} \leq 850$  时， $\text{rate} = 0$ ；

当  $850 < \text{salary} \leq 1350$  时， $\text{rate} = 5\%$ ；

当  $1350 < \text{salary} \leq 2850$  时， $\text{rate} = 10\%$ ；

当  $2850 < \text{salary} \leq 5850$  时， $\text{rate} = 15\%$ ；

当  $\text{salary} > 5850$  时， $\text{rate} = 20\%$ ；

分析：通过观察发现，salary 的每个区间端点减去 850 以后都是 500 的整数倍，因此我们可以将  $(\text{salary} - 850)$  取整后除以 500，得到的结果（设为 d）取值为负数、0~10、大于 10 的正整数，对负数，统一按 -1 处理，对大于 10 的正整数，统一按 10 处理。这样就可以在 switch 结构中按 d 的取值得到不同的税率，然后求出个人所得税并输出。

代码如下：

```

#include <stdio.h>
void main( )
{
    double tax, rate, salary;
    printf( "Please input salary : " );
    scanf( "%lf", &salary );
    int d = (int)(salary - 850)/500;
    if( d < 0 ) d = -1; //对 salary < 850 的特殊处理
    if( d > 10 ) d = 10; //对 salary > 5850 的特殊处理
    switch( d )
    {
        case -1: rate = 0.0; break;
        case 0: rate = 0.05; break;
        case 1:
        case 2:
        case 3: rate = 0.10; break;
        case 4:
        case 5:
        case 6:

```

```

        case 7:
        case 8:
        case 9: rate = 0.15; break;
        case 10: rate = 0.20; break;
    }
    tax = rate * ( salary - 850 );
    printf( "tax = %.4f\n", tax );
}

```

该程序的运行示例如下：

Please input salary : 2555.5

tax = 170.5500

□

## 练习

- 1.19 请改写例 1.15，实现：从键盘上输入一个整数，判断该整数是否能被 5 整除。请按以下两种思路分别实现：

思路一：将该整数对 5 取余，如果余数为 0 则表示能被 5 整数。

思路二：如果该整数的个位为 0 或者为 5，则能被 5 整数，注意 if 条件的表示。

- 1.20 用 if 结构实现例 1.5。（提示，当 NUM 的值为 1, 2, 3, 4, 5, 6, 7, 8, 9 时，NUM%7 的结果分别是 1, 2, 3, 4, 5, 6, 0, 1, 2，因此只有当 NUM 的值为 7 的倍数时，NUM 对 7 取余的结果不符合题目的要求，这时 NUM%7 为 0，而根据题目的意思，得到的结果应该为 7。所以只需判断 NUM 是否为 7 的倍数，如果是，则是第 7 个人报出来的，否则，用 NUM 对 7 取余。）

- 1.21 编写程序，实现闰年的判定。从键盘输入一个年份，然后判定是否为闰年。程序的输入/输出格式见下面的运行示例。

运行示例 1：

1996✓

1996 is a leap year.

运行示例 2：

2001✓

2001 is a not leap year.

- 1.22 求一元二次方程  $a \cdot x^2 + b \cdot x + c = 0$  的根。系数 a、b、c 为浮点数，其值在运行时由键盘输入。须判定  $\Delta$  的情形。

- 1.23 请用第 3 种形式的 if 语句实现：根据输入的百分制成绩 score（假定范围在 0.0~100.0，不用判断），输出等级。其中  $90.0 \leq \text{score} \leq 100.0$  为优秀， $80.0 \leq \text{score} < 90.0$  为良， $70.0 \leq \text{score} < 80.0$  为中， $60.0 \leq \text{score} < 70.0$  为及格， $\text{score} < 60.0$  为不及格。

- 1.24 某物流公司对客户托运的货物计算运费。路程(s)越远，每公里运费折扣越高、客户需要支付的每公里运费就越低。折扣的标准如下：

$s < 250\text{km}$             没有折扣

$250 \leq s < 500$         2%折扣

$500 \leq s < 1000$       5%折扣

$1000 \leq s < 2000$      8%折扣

$2000 \leq s < 3000$      10%折扣

$3000 \leq s$             15%折扣

设每公里每吨货物的运费为 p，货物重为 w，距离为 s，折扣为 d，则总运费 f 的计算公式为： $f = w \times s \times p \times (1 - d)$ 。

从键盘上输入 w、s 和 p 的值，计算总运费 f 并输出。

1.25 某商场在 10 周年店庆时打出了诱人的优惠广告，根据消费者购物总金额 `money`，有两种优惠方案：

方案一：

|                                     |       |
|-------------------------------------|-------|
| <code>money &lt; 250</code> 元       | 1%折扣  |
| <code>250 ≤ money &lt; 500</code>   | 2%折扣  |
| <code>500 ≤ money &lt; 1000</code>  | 5%折扣  |
| <code>1000 ≤ money &lt; 2000</code> | 8%折扣  |
| <code>2000 ≤ money &lt; 3000</code> | 10%折扣 |
| <code>money ≥ 3000</code>           | 12%折扣 |

方案二：

|                                  |         |
|----------------------------------|---------|
| 总金额 <code>money</code> 在 200 元以下 | 没有优惠    |
| 总金额 <code>money</code> 满 200 元   | 返 12 元  |
| 总金额 <code>money</code> 满 400 元   | 返 30 元  |
| 总金额 <code>money</code> 满 800 元   | 返 75 元  |
| 总金额 <code>money</code> 满 1600 元  | 返 175 元 |
| 总金额 <code>money</code> 满 2400 元  | 返 280 元 |
| 总金额 <code>money</code> 满 3000 元  | 返 375 元 |

输入任意购物总金额 `money`(正整数)，问消费者最多可以省多少钱，以及是根据哪个优惠方案进行优惠的。

## 1.8 循环结构

在处理问题时常常需要反复执行某一操作，这就需要用到**循环结构**。例如在判断一个正整数 `m` 是否为素数时，要判断 2 是否能整除 `m`，判断 3 是否能整除 `m`，...；求一个数列的前 `n` 项和时，需要反复把前 `n` 项中的每一项累加起来；等等。

### 1.8.1 3 种循环语句

在 C/C++ 语句中提供了 3 种循环语句：`while` 语句、`do-while` 语句和 `for` 语句，下面分别介绍。

#### 1. `while` 循环语句

`while` 语句的一般形式如下：

**`while(表达式) 语句`**

其执行过程是：如果充当条件判断的表达式为真（非 0）时，执行循环体语句；循环体执行完后又返回到循环条件判断处；如果循环条件仍然成立，重复上述过程；如果循环条件不成立，则整个循环结构执行完毕。其流程图见图 1.17。

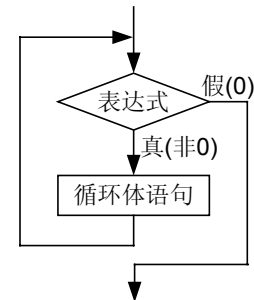


图1.17 `while`循环结构图

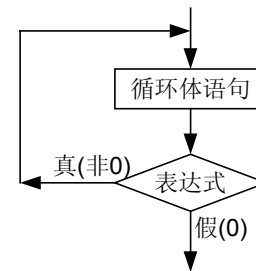


图1.18 `do-while`循环结构流程图

2. do-while 循环语句

do-while 语句的一般形式为：

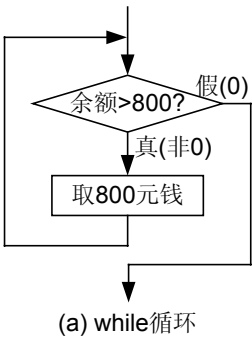
**do**

*语句*

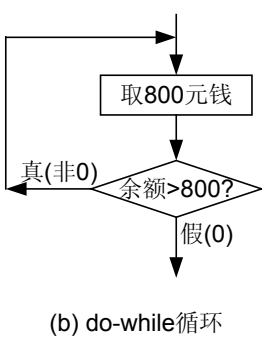
**while(表达式);**

其执行过程是：先执行一次循环体语句，然后判别充当条件判断的表达式；如果表达式为真（表达式的值为非 0），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。其流程图见图 1.18。

这里先对 while 循环和 do-while 循环作个简单的对比：假定开学初你家里往你的帐号上存一定金额，每月花费 800；每月只在月初取出钱；假定不可以透支。问可以取多少次？取钱一般有两种习惯：先查询再取钱；或者先取钱再查询。这两种取钱方式分别相当于 while 循环和 do-while 循环，如图 1.19 所示。“先查询再取钱”的方式总是先判断余额是否大于 800 元，如果是，则可以取钱，相当于 while 循环；而“先取钱再查询”的方式是先取钱，然后判断余额是否大于 800 元，如果是，则下次还可以取钱，相当于 do-while 循环。由这个例子可以看出，do-while 循环的逻辑有时是不合理的，比如这个例子中，如果帐号上的金额初始时就少于 800 元，按照 do-while 循环的逻辑则总是可以取到第一笔 800 元。



(a) while 循环



(b) do-while 循环

图1.19 while循环与do-while循环的对比

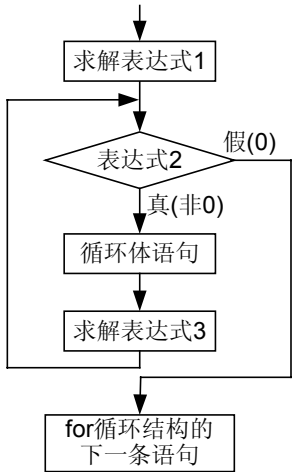


图1.20 for循环结构流程图

3. for 循环语句

for 语句的一般格式为：

**for(表达式 1; 表达式 2; 表达式 3) 语句**

它的执行过程如下（如图 1.20 所示）：

- (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 语句的循环体语句，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。

在使用 for 循环语句时，一定要注意 for 语句 4 个部分（3 个表达式和循环体语句）的执行顺序和次序，例如：表达式 1 最先执行，且只执行一次；循环体语句先于表达式 3 执行等等。

一般来说，以上 3 种循环语句是等效的，读者在例 1.22 中可以体会到。但对于循环次数已知的循环操作，用 **for** 语句实现比较方便，如例 1.23、1.24；如果是根据某个状态来决定是否要循环下去，用 **while** 循环比较方便，如例 1.28。这些细节请读者在实际应用中慢慢去体会。

**例 1.22** 分别用 3 种循环求  $\sum_{n=1}^{100} n$ ，即  $1 + 2 + 3 + \dots + 100$ 。

用 **while** 循环实现的流程图如图 1.21 所示。代码如下：

```
#include <stdio.h>
void main( )
{
    int i = 1, sum = 0; //给循环变量和累加变量初始化
    while( i<=100 )
    {
        sum = sum + i;
        i++;
    }
    printf( "sum=%d\n", sum );
}
```

注意：循环条件中的变量 **i** 通常称为**循环变量**，而且循环变量的变量名通常取为 **i**、**j**、**k**，表达式中的“**100**”通常称为循环变量的**终值**，即循环条件是判断循环变量的值是否超过终值。

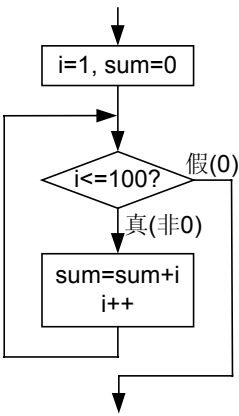


图1.21 用while循环实现累加的流程图

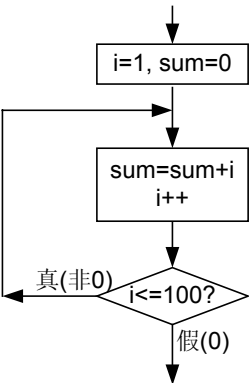


图1.22 用do-while循环实现累加的流程图

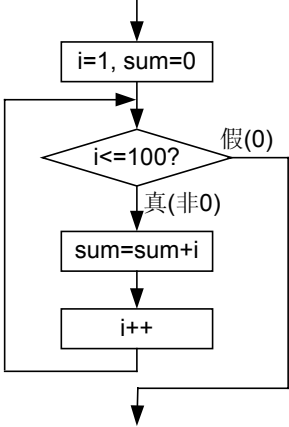


图1.23 用for循环实现累加的流程图

用 **do-while** 循环实现的流程图如图 1.22 所示。代码如下：

```
#include <stdio.h>
void main( )
{
    int i = 1, sum = 0;
    do
    {
        sum = sum + i;
        i++;
    }while( i<=100 );
    printf( "sum=%d\n", sum );
}
```

注意：do-while 语句后有一个分号。

用 **for** 循环实现的流程图如图 1.23 所示。代码如下：

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

```

void main( )
{
    int i, sum;
    for( i = 1, sum = 0; i<=100; i++ )
        sum = sum + i;
    printf( "sum=%d\n", sum );
}

```

以上程序的输出结果均为：

sum=5050

□

从上面的代码可以看出，for 循环更简洁，而且 for 循环可以有很多种写法（请参考其他教材），更灵活。但是，正如上面的代码的一样，for 语句最简单、也是最容易理解的格式为：

**for( 循环变量赋初值; 循环条件; 循环变量增值 ) 循环体语句**

为了让读者理解 for 循环语句的使用，下面再举两个例子。

**例 1.23** 输出所有的“水仙花数”。所谓“水仙花数”是指一个 3 位数，其各位数字的立方和等于该数本身。例如，153 是一个“水仙花数”，因为  $153 = 1^3 + 5^3 + 3^3$ 。

对每一个 3 位数，要判断其各位数字的立方和是否等于它本身，即要反复执行同样的操作，所以要用到循环；而且循环的次数是已知的（从 100 循环到 999），所以用 for 循环实现比较方便。代码如下：

```

#include <stdio.h>
void main( )
{
    int a, b, c;    //分别表示某个数的个位、十位和百位
    for( int NUM=100; NUM<=999; NUM++ )
    {
        a = NUM%10;    //取个位
        c = NUM/100;    //取百位
        b = (NUM/10)%10;    //取十位
        if( a*a*a + b*b*b + c*c*c == NUM ) printf( "%d\n", NUM );
    }
}

```

该程序的输出结果为：

153

370

371

407

注意，1.3.2 节曾经提到“模运算符和除法运算符广泛应用于整数数据的处理”，本题灵活地使用了这两个运算符。 □

**例 1.24** 数列的第 1 项为 81，此后各项均为它前一项的正平方根，统计并输出该数列前 30 项之和。保留小数点后面 6 位精度。

在本题中，需要累加数列前 30 项的和，但并不需要定义 30 个变量来表示每一项，只需要定义一个变量 an，初始时表示第 1 项，累加 an 当前的值后，利用 an 递推到下一项，如此直至前 30 项累加完毕。代码如下：

```

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

```

```

double an = 81, sum = 0; //an 数列的每一项, sum 为前 30 项和
int i;
for( i=1; i<=80; i++ ) //累加前 30 项和
{
    sum += an;
    an = sqrt( an );
}
printf( "sum = %.6f\n", sum );
}

```

该程序的输出结果为:

sum = 171.335860

□

**思考 1.12:** 在例 1.24 的程序中, 如果将 sum 初始化为 an, 即第 1 项的值, 那么 for 循环该如何修改?

上面例 1.23 灵活地使用了取余和整除两个运算符, 下面再讲解一个更灵活的例子。

**例 1.25** 从键盘上输入一个正整数 (范围不超过无符号整数的表示范围), 判断该数是否能被 3 整除。

判断一个正整数是否能被 3 整除的方法是: 将该正整数每位上的数字累加起来, 判断累加和是否能被 3 整除。在本题中, 需要取出正整数每位上的数字。通常取出个位上的数字是最简单的, 因此本题采取的方法是通过整除反复把其他位上的数字变成个位。

代码如下:

```

#include <stdio.h>
void main( )
{
    unsigned int a;
    printf( "please enter an integer : " );
    scanf( "%d", &a );
    unsigned int tmp = a, sum = 0;
    while( tmp )
    {
        sum += tmp%10;
        tmp = tmp/10;
    }
    if( sum%3==0 ) printf( "%d is a multiple of 3.\n", a );
    else printf( "%d is not a multiple of 3.\n", a );
}

```

该程序的运行示例如下:

```

please enter an integer : 147869235
147869235 is a multiple of 3.

```

如上面的运行示例所示, 输入的正整数为 147869235, 第 1 次取余数时取的是个位上的数字; 然后将该正整数除以 10, 则该整数变成了 14786923, 即十位降个位, 下一次取余数时得到的是原来十位上的数字。循环结束的条件是最后一次取得最高位上的数字 1 (此时 tmp 变量的值为 1), 将 tmp 除以 10, tmp 的值为 0, 此后循环条件不成立, 循环结束。 □

注意, 通常程序中不希望改变输入数据的值, 因为后面的语句可能还需要用到该变量的值, 比如例 1.25 中最后的输出语句还需要输出变量 a 的值。因此在例 1.25 中先将 a 的值赋值给临时变量 tmp, 此后变量 a 的值保持不变, 而 tmp 的值不断变化。

#### 4. 循环语句总结



关于循环语句，以下几点需特别注意：

1) 与选择结构类似，如果循环体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。如下面的例子，“**i++;**”这条语句是不属于循环结构的。

```
#include <stdio.h>
void main( )
{
    int i=1, sum=0;
    while( i<=100 )
        sum = sum + i; //循环体语句
    i++; //循环体外的语句
    printf( "sum=%d\n", sum );
}
```

2) 在循环体中应有使循环趋向于结束的语句。否则就会陷入**死循环**，即循环永不停止地执行。比如上面的例子，由于“**i++;**”这条语句不属于循环结构。**while** 循环结构中没有是循环趋向于结束的语句，所以该循环是一个死循环。

3) 如果循环条件永远为真，这种循环称为**永真循环**，如下面的两个例子。当然，这种循环应该在适当时候用 **break** 语句（详见 1.8.2 节）退出循环。

```
while( 1 )
{
    ...
}

while( true )
{
    ...
}
```

## 1.8.2 break 语句和 continue 语句

**break** 语句除了可以在 **switch** 结构中使用外，还可以用于循环体内。其作用为使流程从循环体内跳出循环体，即提前结束循环，接着执行整个循环语句后面的语句。**break** 语句只能用于循环语句和 **switch** 语句内，不能单独使用或用于其他语句中。

**continue** 语句只能用于循环结构内，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。

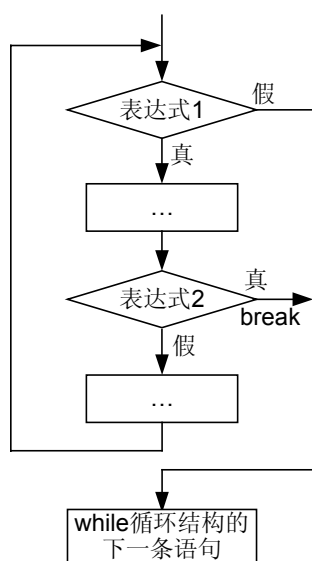


图1.24 在循环中使用break语句

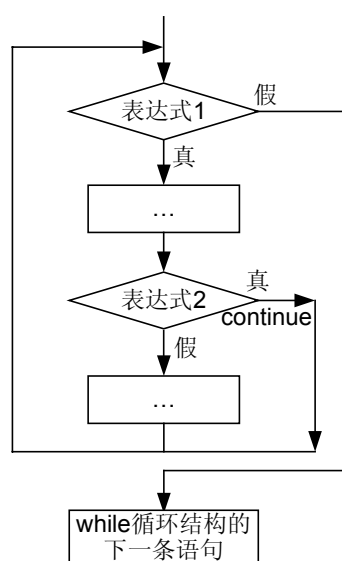


图1.25 在循环中使用continue语句

**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：

|                                                                      |                                                                         |
|----------------------------------------------------------------------|-------------------------------------------------------------------------|
| <pre>(1) while(表达式 1) {     ...     if(表达式 2) break;     ... }</pre> | <pre>(2) while(表达式 1) {     ...     if(表达式 2) continue;     ... }</pre> |
|----------------------------------------------------------------------|-------------------------------------------------------------------------|

程序(1)的流程图如图 1.24 所示，而程序(2)的流程如图 1.25 所示。请特别注意图 1.24 和图 1.25 中当“表达式 2”为真时流程的转向。

在循环里到底该用 **break** 语句还是 **continue** 语句，应视具体情况而定，详见下面的例 1.26。

**例 1.26** 判断  $m=199$  是否为素数。素数（也称为质数）的定义是：若该数除 1 和本身之外不能被其他数整除，则该数为素数，否则为合数。

**分析：**在判断  $m$  是否为素数的过程中，要判断 2 是否能整除  $m$ ，如果能整除则  $m$  是合数，要提前退出循环（所以应该用 **break** 语句），不再继续判断；否则继续判断 3 是否能整除  $m$ ，如果能整除则  $m$  是合数，要提前退出循环，不再继续判断；...；一直判断到小于等于  $m^{1/2}$  的最大整数为止。如果在这个过程中始终没有找到能整除  $m$  的数，则  $m$  是素数。这个过程需要反复执行判断  $m$  是否能被  $i$  整除的操作（ $i = 2, 3, \dots$ ），所以要用到循环。循环结构执行完毕后，如何判断  $m$  是否为素数，有两种方法。

**方法一，使用状态变量**（用来标志某种状态的变量，通常为 **bool** 型，也可以为 **int** 型）。在程序中定义一个 **bool** 型的状态变量 **prime**，其初值为 **true**；当找到第一个能整除  $m$  的  $i$  时，设置 **prime** 为 **false**，并提前退出循环；如果始终没有找到能整除  $m$  的  $i$ ，则状态变量 **prime** 的值仍为初值 **true**。当 **for** 循环执行完毕后，根据 **prime** 的值可判定  $m$  是否为素数。因此状态变量 **prime** 的含义是表征  $m$  是否为素数，如果 **prime** 为 **true**，则  $m$  为素数；否则  $m$  为素数。其流程图如图 1.26(a)所示。代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    int m, i, k;
    bool prime = true; //状态变量
    scanf( "%d", &m );
    k = (int)sqrt(m); //k 为小于等于根号 m 的最大正整数
    for( i=2; i<=k; i++ )
    {
        if( m%i==0 )
        {
            prime = false; //当找到第一个能整除 m 的 i，设置 prime 为 0
            break; //并提前退出循环
        }
    }
    if( prime ) printf( "%d 是素数\n", m );
    else printf( "%d 是合数\n", m );
}
```

**方法二，不使用状态变量**，而是根据 **for** 循环执行完后循环变量的值及循环变量的终值的关系来判定  $m$  是否为素数。其流程图如图 1.26(b)所示。该循环有 2 个出口，一是在判断  $m$  是否为素数的过程中，已找到一个能整除  $m$  的整数  $i$ ，要提前退出循环，这样循环变量  $i$  的值是  $\leq k$  的；

二是一直判断到  $k$  都没找到能整除  $m$  的整数  $i$ ，这时循环条件不再成立，要退出循环，此时循环变量  $i$  的值是  $k+1$ 。因此循环执行完后判断  $i$  是否大于  $k$  即可判别  $m$  是否为素数。代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    int m, i, k;
    scanf( "%d", &m );
    k = (int)sqrt(m);
    for( i=2; i<=k; i++ )
    {
        if( m%i==0 ) //当找到第一个能整除 m 的 i，提前退出循环
            break;
    }
    if( i>k ) printf( "%d 是素数\n", m );
    else printf( "%d 是合数\n", m );
}
```

以上程序的输出结果均为：

199 是素数

□

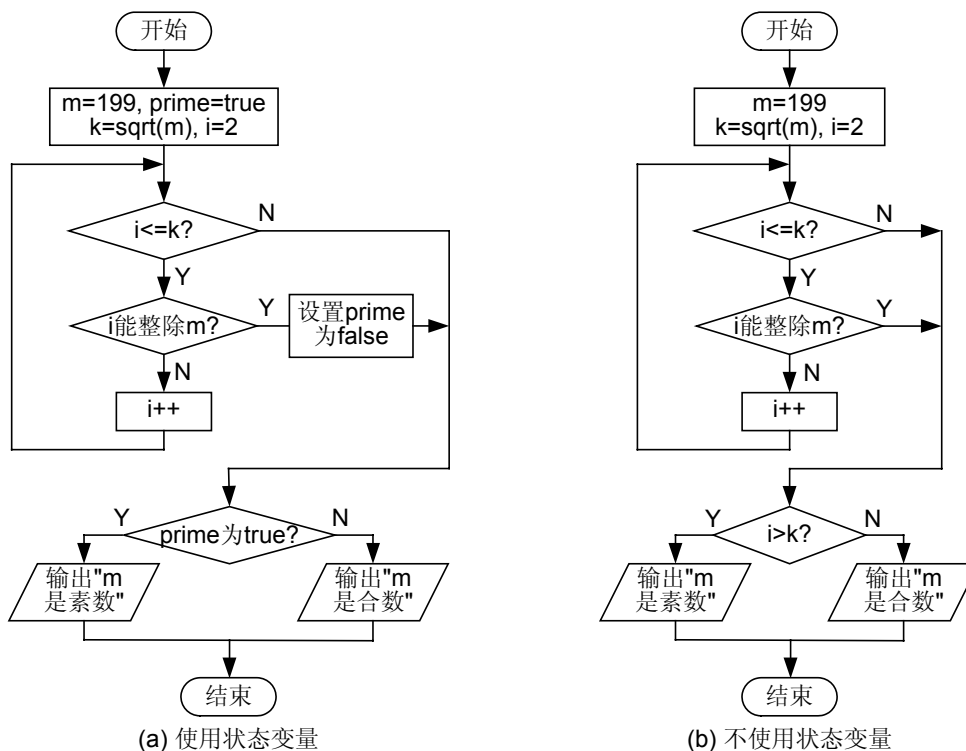


图1.26 判断 $m=199$ 是否为素数的流程图

例 1.26 是使用 `break` 语句的例子，对于在循环中使用 `continue` 语句的例子，这里不列举。读者有兴趣的话，可以先看一下例 3.4 的程序。

### 1.8.3 循环的嵌套

如果一个循环结构的循环体内又包含另一个完整的循环结构，则称为**循环的嵌套**。内嵌的循环中还可以再嵌套循环，这就是多层循环的嵌套。2 层循环的嵌套通常称为 2 重循环，3 层循环的嵌套通常称为 3 重循环，等等。

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:

在实际的程序中经常要用到 2 重循环甚至 3 重循环结构, 详见下面的例子。

|                                                                |                                                                   |                                                               |
|----------------------------------------------------------------|-------------------------------------------------------------------|---------------------------------------------------------------|
| (1) while( )<br>{<br>...<br>while( )<br>{ ... }<br>...<br>}    | (2) do<br>{<br>...<br>do<br>{ ... }while( );<br>...<br>}while( ); | (3) for(;;)<br>{<br>...<br>for(;;)<br>{ ... }<br>...<br>}     |
| (4) while( )<br>{<br>...<br>do<br>{ ... }while( );<br>...<br>} | (5) for(;;)<br>{<br>...<br>while( )<br>{ ... }<br>...<br>}        | (6) do<br>{<br>...<br>for(;;)<br>{ ... }<br>...<br>}while( ); |

**例 1.27** 输出 6~10000 之间的完数。完数的定义: 若该数除本身之外的所有因子之和等于该数, 则为完数。如:  $1+2+3=6$ , 所以 6 为完数;  $1+2+4 \neq 8$ , 所以 8 不是完数。

与例 1.26 判断素数不同的是, 在判断完数时, 对于找到能够整除  $m$  的  $i$ , 要累加起来, 而不是提前退出循环; 另外循环变量  $i$  是从 1 取值, 一直到  $m-1$ 。判断  $m=496$  是否为完数的流程图如图 1.27(a)所示。要输出 6~10000 之间的完数, 需要使  $m$  从 6 变化到 7, 再变化到 8, ..., 一直到 10000。这样需要在原有循环的基础上再套一层循环。因此该程序的代码用到了 2 重循环, 其中内循环用于判断  $m$  是否为完数, 外循环使得  $m$  从 6 变化到 10000。其流程图如图 1.27(b)所示。

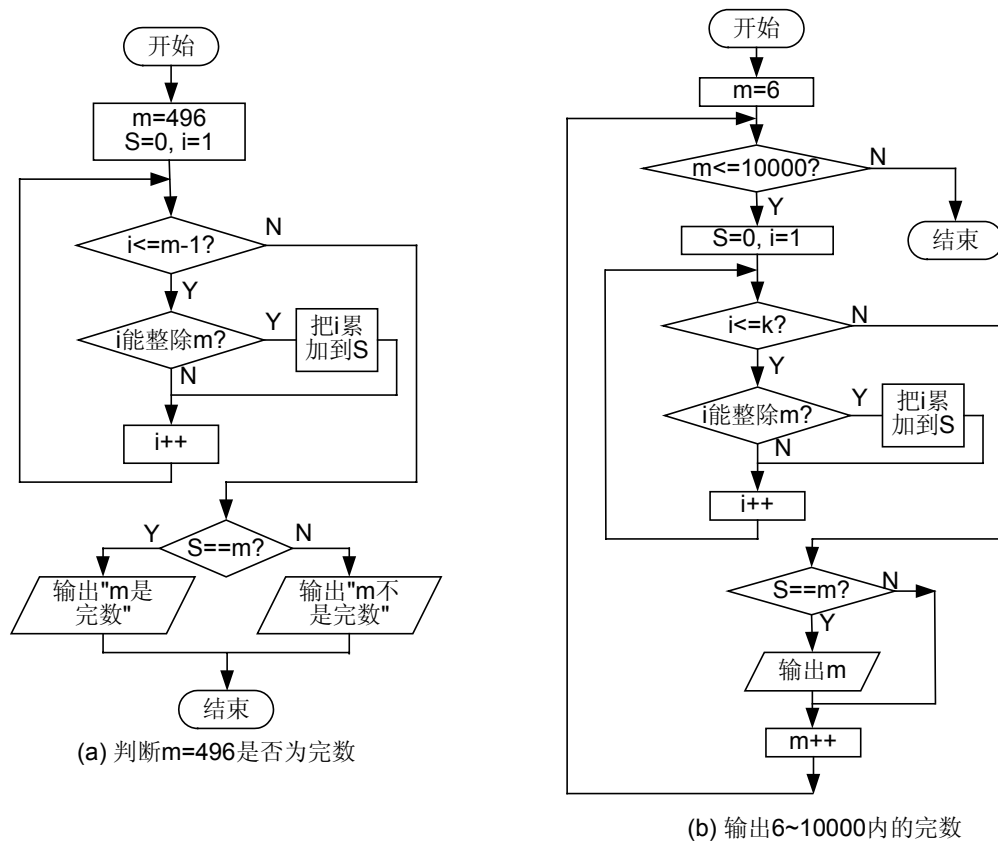


图1.27 判断完数的流程图

代码如下：

```
#include <stdio.h>
void main( )
{
    int m, i, S;
    for( m = 6; m<=10000; m=m+1 )
    {
        S = 0; //累加 m 的因子
        for( i=1; i<m; i++ )
        {
            if( m%i==0 ) //把 m 的因子累加起来
                S = S+i;
        }
        if( S==m ) //输出完数 m
            printf( "%d\n", m );
    }
}
```

该程序的输出结果为：

```
6
28
496
8128
```

□

**思考 1.13：**例 1.27 的程序中，对变量 S 的赋值能否只在定义处就初始化为 0，而把语句“S = 0;”去掉？

### 1.8.4 循环结构例子

为了让读者更好地理解和使用循环结构，接下来再举几个循环结构的例子。

#### 例 1.28 用迭代法求平方根

头文件里 `math.h` 提供了求平方根的函数 `sqrt`。但你知道平方根是怎么求出来的吗？迭代法是方法之一。用迭代法求  $x = \sqrt{a}$ ，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  $10^{-5}$ 。浮点数  $a$  的值从键盘输入，将求得的  $a$  的平方根  $x$  输出来。迭代的初值  $x_0$  可任取，比如可以取  $a/2$ 。

**分析：**所谓迭代法，就是由一个初始值  $x_0$ ，递推出  $x_1$ ，再根据  $x_1$  递推出  $x_2$ ，...，一直递推到精度符合要求，或者满足某个条件，比如本题中要求前后迭代出来的  $x$  差的绝对值小于  $10^{-5}$ 。程序需要用到循环结构，因为要反复迭代，但具体需要迭代多少次，不知道，所以用 `while` 循环结构比较方便。

另外，程序里也无法用很多变量去表示每次迭代出来的结果。其实只需要用到两个变量  $x_0$  和  $x_1$ ，表示前后两次迭代的结果即可。 $x_0$  的初值可以随便取，一般取  $a/2$ ， $x_1$  的初值就是根据  $x_0$  的初值迭代出来的  $x$ 。`while` 循环的循环条件就是  $x_1$  和  $x_0$  差的绝对值

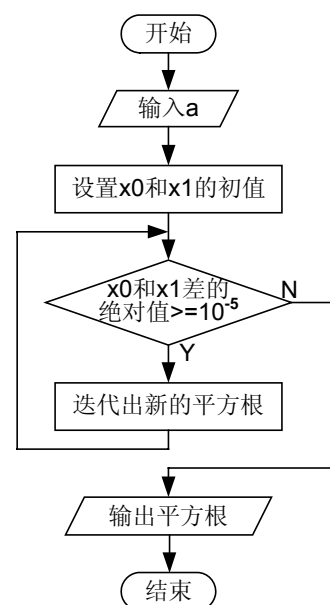


图1.28 迭代法求平方根的流程图

大于  $10^{-5}$ ，只要循环条件满足，就反复迭代，流程图如图 1.28 所示。程序代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    double a;
    double x0, x1;
    printf( "Please enter a positive number : a=" );
    scanf( "%lf", &a ); //输入 a，要求根号 a
    x0 = a/2;
    x1 = (x0 + a/x0)/2;
    //当前后两次迭代结果之差的绝对值大于 10 的-5 次方,就继续迭代
    while( fabs(x1-x0)>=1e-5 )
    {
        x0 = x1;    //更新 x0 的值
        x1 = (x0 + a/x0)/2;    //从前一项 x0 递推到后一项 x1
    }
    printf( "The square root of %0.6f is %0.6f\n", a, x1 );
}
```

该程序的运行示例如下：

Please enter a positive number : a=2✓

The square root of 2.000000 is 1.414214

□

**例 1.29** 输出 Fibonacci 数列前 40 个数。这个数列有如下特点：第 1、2 个数为 1、1。从第 3 个数开始，每个数是其前面两个数之和。即：

F1=1                    (n=1)  
F2=1                    (n=2)  
Fn=Fn-1+Fn-2    (n≥3)

**分析：**要求 Fibonacci 数列前 40 个数，可以定义 40 个变量表示这 40 个数据，但变量太多，程序繁琐，不切实际。而采用递推的方法，只需要两个变量 f1 和 f2，就可以递推出 40 个数。f1 和 f2 表示数列中前后两个数，初始时，f1 表示第 1 个数，f2 表示第 2 个数。有两种递推方法。

**方法一：每次递推出 1 项。**如图 1.29 所示。f1 和 f2 表示该数列连续 3 项中的前两项，新递推出来是第 3 项，即为 f1+f2，用变量 f2 来表示这一项，因此有语句(2)，然后使得 f1 表示第 2 项，因此要把原来 f2 的值赋值给变量 f1。注意这时 f2 的值已经改变了，现在表示的是第 3 项，因此在语句(2)之前要把 f2 的值先保存到临时变量 t 中。这种思想在例 1.4 中也采用过。代码如下：

```
#include <stdio.h>
void main( )
{
    int f1, f2; //f1,f2 分别代码 Fibonacci 数列中前后两个数
    int t;      //临时变量
    int i;      //循环变量
    f1 = f2 = 1;
    printf( "%12d%12d", f1, f2 );
    for( i=3; i<=40; i++ )
    {
        t = f2;          //(1)先把 f2 保存起来
        f2 = f1+f2;       //(2)f2 是新的项
        f1 = t;           //(3)此时的 f1 是上一次的 f2
        printf( "%12d", f2 );
    }
}
```

```
        if( i%5==0 ) printf( "\n" );
    }
}
```

该程序的输出结果如下：

|          |          |          |          |           |
|----------|----------|----------|----------|-----------|
| 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  | 5702887  | 9227465   |
| 14930352 | 24157817 | 39088169 | 63245986 | 102334155 |

即每行输出 5 个 Fibonacci 数，每个数输出时占 12 字符宽度，且右对齐。从以上输出结果可以看出，Fibonacci 数列的增长速度是很快的。

注意，上面的代码中实现递推的 3 条语句是通过中间变量 t 来实现的，其实也可以不使用中间变量，仅采用下面两条语句，也可以实现递推过程。请读者仔细推敲、理解这两条语句。

```
f2 = f1 + f2;
f1 = f2 - f1;
```

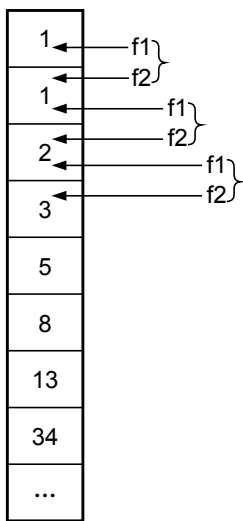


图1.29 每次递推出一个数

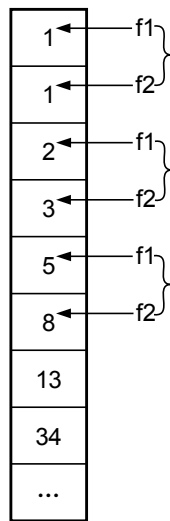


图1.30 每次递推出两个数

**方法二：每次递推出 2 项。**如图 1.30 所示。f1 和 f2 原先表示该数列的第 1 项和第 2 项。第 3 项的值是 f1+f2，把该项的值赋值给 f1，即语句(1)，因为此后第 1 项的值在递推过程中不再使用，这样 f1 此时表示的是第 3 项，第 4 项的值是第 2、3 项之和，分别是变量 f2 和 f1，因此把 f2+f1 的值赋值给 f2，即语句(2)，因为此后第 2 项的值在递推过程中也不再使用。这样，经过语句(1)和(2)，f1 和 f2 分别表示第 3 项和第 4 项，仍然是前后两项。代码如下：

```
#include <stdio.h>
void main( )
{
    int f1, f2;
    int i;
    f1 = f2 = 1;
    for( i=1; i<=20; i++ )
    {
        printf( "%12d%12d", f1, f2 );    //设置输出字段宽度为 12，每次输出两个数
```

```

        if( i%2==0 ) printf( "\n" ); //每输出完 4 个数后换行, 使每行输出 4 个数
        f1 = f1+f2;      //(1)左边的 f1 代表第 3 个数, 是第 1、2 个数之和
        f2 = f2+f1;      //(2)左边的 f2 代表第 4 个数, 是第 2、3 个数之和
    }
}

```

该程序输出 Fibonacci 数列前 40 个数, 每 4 个数占一行。 □

**例 1.30** 已知数列 $a_n=(-1)^{n+1} \cdot 1/n$ 。数列前 $n$ 项和为 $S_n$ 。编写程序, 求 $S_n$ 的最大值和最小值,  $100 \leq n \leq 10000$ , 输出 15 位有效数字。

**分析:** 单独求某个  $S_n$ , 比如  $S_{100}$ , 需要用 1 重循环。现在  $n$  需要从 100 变化到 10000, 所以为了求出  $S_{100}$  到  $S_{10000}$ , 需要用 2 重循环。设  $\max S_n$ 、 $\min S_n$  分别代表求得的  $S_n$  最大值和最小值, 在计算  $S_n$  过程当中, 每求出一个  $S_n$ , 需要跟  $\max S_n$ 、 $\min S_n$  进行比较, 如果  $S_n$  比当前的  $\max S_n$  还大, 则将  $\max S_n$  更新为  $S_n$ ; 如果  $S_n$  比  $\min S_n$  还小, 则将  $\min S_n$  更新为  $S_n$ 。

对于  $\max S_n$ 、 $\min S_n$  的初值, 有两种取法: 第一种方法是先求出  $n=100$  时的  $S_n$ ,  $\max S_n$ 、 $\min S_n$  的初值都取  $n=100$  时的  $S_n$ 。

第二种方法是先估计出  $S_n$  的值,  $100 \leq n \leq 10000$  时,  $S_n$  的值在  $[0, 1]$  范围内, 所以, 让  $\max S_n$  取一个较小的初值, 如 0.0; 让  $\min S_n$  取一个较大的初值, 如 2.0。为什么呢? 假设要在一片苹果园中找一个最大的苹果, 如果手里拿一个南瓜, 跟苹果园里的苹果去比较, 能找到最大的苹果吗? 不能, 所以手里先要拿一个小的, 如樱桃。同样的道理, 假设要在一片苹果园中找一个最小的苹果, 如果手里拿一个樱桃, 跟苹果园里的苹果去比较, 能找到最小的苹果吗? 不能, 所以手里先要拿一个大的, 如西瓜。

程序的代码 ( $\max S_n$ 、 $\min S_n$  初值的取法采用第二种方法) 如下:

```

#include <stdio.h>
void main( )
{
    double maxSn = 0.0; //最大值
    double minSn = 2;   //最小值
    for( int n=100; n<=10000; n++ )//求 S100~S10000 并判断
    {
        double Sn = 0;
        int flag = 1;
        for( int i=1; i<=n; i++ ) //求 Sn
        {
            Sn += flag*(1.0/i);
            flag = -flag;
        }
        if( Sn>maxSn ) maxSn = Sn;
        if( Sn<minSn ) minSn = Sn;
    }
    printf( "maxSn=%0.15f\nminSn=%0.15f\n", maxSn, minSn );
}

```

该程序的输出结果如下:

```

maxSn=0.698073169409205
minSn=0.688172179310195

```

□

**例 1.31** 鸡兔同笼问题: 一个笼子里关了鸡和兔子 (鸡有 2 只脚, 兔子有 4 只脚)。已知笼子里面脚的总数  $nFeet$  (假设  $nFeet$  为正整数, 从键盘输入), 问笼子里可能有多少只鸡, 有多少



只兔子。

分析：本题的求解要采用一种被称为枚举（也称为穷举，本教材第 4 章将专门讨论这种思路）的编程思路。假定鸡的数目和兔子的数目分别为 `hens` 和 `rabbits`，当 `nFeet` 为偶数时，`hens` 的取值范围是  $0 \sim nFeet/2$ ，我们需要枚举 `hens` 在  $0 \sim nFeet/2$  范围内的每个取值；如果 `hens` 的某个取值使得  $(nFeets - 2 \times hens)$  能被 4 整除，则  $rabbits = (nFeets - 2 \times hens)/4$ ，这样 `hens` 只鸡、`rabbits` 只兔子是一个满足条件的解。当然，当 `nFeet` 为偶数时，满足条件的解可能不止一个。

另外，在本题中，如果输入的脚的总数为奇数，则是不可能，应输出“impossible!”。

代码如下：

```
#include <stdio.h>
void main( )
{
    int nFeet;    //脚的总数
    int hens, rabbits;
    printf( "Please input the total feet : " );
    scanf( "%d", &nFeet );
    if( nFeet%2 ) //nFeet 为奇数，不可能
        printf( "impossible!\n" );
    else
    {
        for( hens = 0; hens <= nFeet/2; hens++ )//鸡的数目最少为 0，最多为 nFeet/2
        {
            if( (nFeet-hens*2)%4 == 0 )    //除去鸡的脚数，剩下的脚数为 4 的倍数
            {
                rabbits = (nFeet-hens*2)/4; //求兔子的数目
                printf( "hens = %d, rabbits = %d\n", hens, rabbits );
            }
        }
    }
}
```

该程序的运行示例 1 如下：

```
Please input the total feet : 13 ✓
impossible!
```

该程序的运行示例 2 如下：

```
Please input the total feet : 20 ✓
hens = 0, rabbits = 5
hens = 2, rabbits = 4
hens = 4, rabbits = 3
hens = 6, rabbits = 2
hens = 8, rabbits = 1
hens = 10, rabbits = 0
```

□

## 练习

- 1.26 求  $n$  的阶乘： $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$ ，分别用 `while`，`do-while`，`for` 循环实现。
- 1.27 编程实现：累加 100~1000 之内能被 3 整除的偶数。
- 1.28 求 Fibonacci 数列中大于  $n$  的第一个数及其在 Fibonacci 数列中的序号，以及求 Fibonacci 数列中不大于  $n$  的最大的数及其在 Fibonacci 数列中的序号， $n$  从键盘输入。
- 1.29 求数列： $a, aa, aaa, \dots, aa \dots a$  前  $N$  项和，第  $N$  项有  $N$  个  $a$ ， $a$  和  $N$  均为正整数， $0 \leq a$

$\leq 9$ ,  $0 \leq N \leq 9$ ,  $a$  和  $N$  的值都是从键盘输入的。

- 1.30 有一分数数列：

$$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \dots$$

求这个数列前 20 项和。

- 1.31 用下面的公式求  $\pi$  的近似值。

$$\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

直到最后一项的绝对值小于  $10^{-7}$  为止（该项不累加）。

- 1.32 编写程序：计算数列  $1, -1/3!, 1/5!, -1/7!, 1/9!, \dots$  的和，直到最后一项的绝对值小于  $10^{-7}$  为止，分别处理以下两种情形：累加与不累加第一个小于  $10^{-7}$  的项。

- 1.33 头文件 `math.h` 提供了求  $e^x$ （其中  $e$  为自然对数的底）的函数 `exp`。  $e^x$  也可以采用下面的式子来求：

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$

从键盘上输入浮点数  $x$ ，根据上面的式子，累加前 20 项求  $e^x$ 。

- 1.34 请为某公司设计一个电话语音系统。当用户拨通电话时给出提示信息，按 1 接通总经理办公室、按 2 接通客户服务部门、按 3 接通人力资源部门、按 0 退出系统，其他按键均给出错误提示。在本题中，用输入整数代表用户按键，用输出提示信息代表接通电话，请用 `switch` 结构判断用户的按键情况。（需要用到永真循环）

程序的运行示例如下：

```
Welcome to our company.
1 - Manager's Office.
2 - Customer Service department.
3 - Human Resource department.
0 - Exit System.
Please input your choice : 1✓
Here is Manager's Office.
Please input your choice : 2✓
Here is Customer Service department.
Please input your choice : 3✓
Here is Human Resource department.
Please input your choice : 4✓
Error Input!
Please input your choice : 0✓
Thanks Your Visit!
```

- 1.35 输出 100~200 之间的素数，并画出流程图。

- 1.36 求  $1! + 2! + 3! + 4! + \dots + 20!$ 。提示：20! 阶乘太大了，超出了 `int` 类型数据的范围，因此需要用 `float` 型或 `double` 型。

- 1.37 请利用循环结构设计程序，输出九九乘法表，如下表：

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
.....
```

9\*1=9 9\*2=18 9\*3=27 9\*4=36 9\*5=45 9\*6=54 9\*7=63 9\*8=72 9\*9=81

- 1.38 百钱百鸡问题。中国古代数学家张丘建在他的《算经》中提出了著名的“百钱买百鸡问题”：鸡翁一，值钱五，鸡母一，值钱三，鸡雏三，值钱一，百钱买百鸡，问翁、母、雏各几何？意思是说：1 只公鸡值 5 钱，1 只母鸡值 3 钱，3 只小鸡值 1 钱，某人用 100 钱买了 100 只鸡，问公鸡、母鸡、小鸡各有多少只。请编程求解该问题。

## 1.9 函数设计

### 1.9.1 函数概述

“函数”这个名词的英文原文是“function”，而“function”的原意是“功能”。顾名思义，函数就是用来实现某个功能的，而且通常只实现一个功能（而不会把多个功能糅合到一个函数里）。也就是说，在程序设计语言里引入函数的概念，就是为了进行功能分解。

例如，要输出 100~200 之内的素数，可以用一个 2 重循环实现。但如果有一个函数 `prime`，能够实现判断一个正整数 `m` 是否为素数，其调用形式是：`prime(m)`。调用该函数后返回值如果为 1，则 `m` 为素数；如果为 0，则 `m` 为合数。因此我们只需要用如下的代码就可以输出 100~200 之内的素数：

```
for( int m =100; m<=200; m++ )
{
    if( prime(m) )
        printf( "%d\n", m );
}
```

在这个例子中，我们把“输出 100~200 之内所有素数”的功能需求进行分解，把“判断一个正整数是否为素数”的功能用 `prime` 函数去实现。这就是函数的功能所在。

我们可以从不同的角度对函数进行分类。从用户使用的角度看，函数有两种：

(1) 系统函数，即库函数：这是由编译系统提供的，用户不必自己定义这些函数，可以直接调用它们，但必须把相关的头文件包含进来。例如，求平方根的函数 `sqrt`，是在头文件 `math.h` 中声明的，因此调用 `sqrt` 函数时必须把 `math.h` 包含进来。

(2) 用户自定义函数：用以解决用户的专门需要，需要用户自己定义。

从函数的形式看，函数分为两类：

(1) 无参函数：在调用函数时，函数名后面的圆括号内没有参数。

(2) 有参函数：在调用函数时，要在圆括号内给出参数。主调函数和被调函数之间是通过参数来进行数据传递的。

另外，如果在 A 函数中调用了 B 函数，则称 A 函数为主调函数，称 B 函数为被调函数。

### 1.9.2 函数的定义

#### 1. 无参函数的定义

定义无参函数的一般形式为：

**类型标识符 函数名([void])**

```
{
    函数中的语句
}
```

类型标识符是指函数值的类型，所谓函数值就是函数的返回值（如果没有返回值，则类型标

标识符为 `void`)。函数名必须是合法的标识符。圆括号内的 `void` 加上了方括号 “[ ]”，表示 `void` 可以省略，因此，本章所有例子的 `main` 函数中都省略了圆括号内的 `void`。

无参函数的定义和使用详见下面的例子。

**例 1.32** 定义两个函数，分别输出一行字符，在 `main` 函数中调用这两个函数。

代码如下：

```
#include <stdio.h>
void printstar( void )
{
    printf( "*****\n" );
}
void printmessage( void )
{
    printf( "    Welcome to C/C++!\n" );
}
void main( )
{
    printstar( );      //调用 printstar 函数
    printmessage( );   //调用 printmessage 函数
    printstar( );      //调用 printstar 函数
}
```

该程序的输出结果如下：

```
*****
    Welcome to C/C++!
*****
```

在例 1.32 中，`printstar( )` 函数和 `printmessage( )` 都没有参数，也没有返回值，其功能是向屏幕上输出一行字符。在主调函数中调用了这两个函数。 □

## 2. 有参函数的定义

定义有参函数的一般形式为：

**类型标识符 函数名( 类型名 形式参数 1, 类型名 形式参数 2, …… )**

```
{
    函数中的语句
}
```

定义有参函数时，要将参数以列表形式在圆括号内列出，有多少个参数就必须列出多少个，而且每个参数都必须用类型名修饰，尽管这些参数类型可能是相同的。请注意以下两种用法：

```
int x, y, z;    //(√)定义变量时，可以用一个类型名定义多个同类型的变量
int f( int x, y, z )  //(×)在定义函数时，每个参数都必须用类型名修饰
{
    ...
}
```

数学上的函数大多数都是带参数的，例如： $\sin(x)$ ， $f(x) = 4x^2 - 9x + 7$  等等。

有参函数的定义和使用详见下面的例子。

**例 1.33** 定义有参函数，实现计算函数值  $f(x) = 4x^2 - 9x + 7$ ，在 `main` 函数中根据输入的 `x`，调用 `f` 函数求函数值 `y`。

代码如下：

```
#include <stdio.h>
double f( double x )
{
    double y;
    y = 4*x*x - 9*x + 7;
    return y;
}
void main( )
{
    double x, y;
    printf( "Please input x : " );
    scanf( "%lf", &x );
    y = f( x );
    printf( "y = %.4f\n", y );
}
```

该程序的运行示例如下：

Please input x : 2.7 ✓

y = 11.8600

注意，在例 1.33 中，f 函数中的变量 x 和 y，与 main 函数中的变量 x 和 y 是不同的变量。特别是定义 f 函数时的参数 x 和调用 f 函数时的参数 x，有不同的含义，详见下面 1.9.3 节。 □

### 1.9.3 函数参数

大多数函数都是带参数的。参数有两种：形式参数和实际参数。

在定义函数时函数名后面括号中的变量名称为**形式参数**（formal parameter，简称**形参**）。

在主调函数中调用被调函数时，函数名后面括号中的参数（可以是表达式）称为**实际参数**（actual parameter，简称**实参**）。

主调函数与被调函数之间的数据传递是通过实参和形参来进行的。下面通过例 1.34 和例 1.35 来详细描述这种数据传递过程。

**例 1.34** 通过自定义函数的方式求两个数的较大者。代码如下：

```
#include <stdio.h>
int max( int x, int y ) //定义有参函数 max：求两个数的较大者
{
    int m;
    m = x>y ? x : y;
    return(m);
}
void main( )
{
    int a, b, c;
    printf( "please enter two integer numbers: " );
    scanf( "%d%d", &a, &b );
    c = max(a,b); //调用 max 函数，给定实参为 a,b。函数值赋给 c
    printf( "max=%d\n", c );
}
```

该程序的运行示例如下：

45 67 ✓

max=67

如图 1.31(a)所示, 该程序在执行时, 首先从 `main` 函数第一条代码开始执行。当执行到语句“`c=max(a,b);`”时, 因为有函数调用, 所以要转而去执行 `max` 函数。在执行 `max` 函数里的语句之前, 把实参 `a` 的值传递给形参 `x`, 把实参 `b` 的值传递给形参 `y`, 如图 1.31(b)所示。这样 `max` 执行完毕后, 变量 `m` 的值为 67, 将这个值返回到 `main` 函数中, 即返回到“`c=max(a,b);`”语句处, 把 67 赋值给 `c`。然后输出 `c` 的值, 最终程序从 `main` 函数结束。□

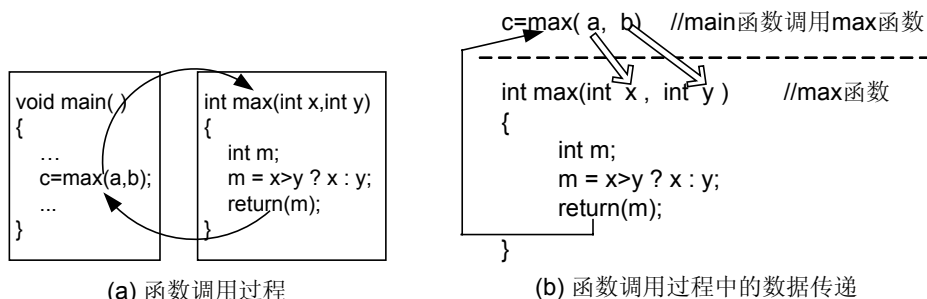


图1.31 函数调用过程

有关形参和实参的说明:

- 1) 在定义函数时指定的形参, 在未出现函数调用时, 它们并不占内存中的存储单元, 并不是实际存在的数据, 因此称它们是形式参数或虚拟参数。
- 2) 除带默认参数的函数和重载函数 (本书都没有介绍, 请参考其他教材) 外, 实参个数必须与形参个数一致。此外, 每个实参的类型必须跟对应的形参的类型一致或赋值兼容。
- 3) 实参可以是常量、变量或表达式。如 `max(3, a+b)`, 但要求 `a` 和 `b` 有确定的值。如果实参是变量, 为避免混淆, 建议初学者将形参和实参用不同的变量名表示。
- 4) 实参变量对形参变量的数据传递是“值的传递”, 即单向传递, 只由实参传给形参, 而不能由形参传回来给实参。在例 1.34 中, 将实参 `a` 和 `b` 的值分别传给形参 `x` 和 `y`。(在值传递之后, `a` 和 `x` 就没有任何关联了)

**思考 1.14:** 请分析下面的程序是否能实现交换变量 `a` 和 `b` 的值 (请参考 1.11.3 节的内容)。

```
#include <stdio.h>
void exchange( int x, int y )
{
    int t = x;  x = y;  y = t;
}
void main( )
{
    int a, b;
    scanf( "%d%d", &a, &b );
    exchange( a, b );
    printf( "a=%d, b=%d", a, b );
}
```

关于形参与实参, 以及主调函数与被调函数之间的数据传递, 下面再讲解一个例子。

**例 1.35** 定义函数, 求平面上两个点的距离。

本题将定义一个函数 `d`, 用于求平面上两个点的距离, `main` 函数调用函数 `d` 实现计算输入的两个点的距离。整个程序的代码如下:

```
#include <stdio.h>
#include <math.h>
double d( double xx1, double yy1, double xx2, double yy2 )
{
```

```

    return sqrt( (xx1-xx2)*(xx1-xx2) + (yy1-yy2)*(yy1-yy2) );
}
void main( )
{
    double x1, y1, x2, y2;
    printf( "Please input the coordinates of 2 points : " );
    scanf( "%lf%lf%lf%lf", &x1, &y1, &x2, &y2 );
    printf( "the distance of the 2 points is %.4f\n", d(x1,y1,x2,y2) );
}

```

该程序的运行示例如下：

Please input the coordinates of 2 points : 2.5 3.7 4.2 7.8✓

the distance of the 2 points is 4.4385.

注意，在上述代码中，函数 `d` 有 4 个形参，分别表示第 1 个点的 `x` 坐标和 `y` 坐标、第 2 个点的 `x` 坐标和 `y` 坐标，在调用函数 `d` 时，4 个实参也必须按照这样的顺序给出。如果以“`d(x1,x2,y1,y2)`”的形式去调用函数 `d`，得到的结果肯定是错误的。□

### 1.9.4 函数的返回值

函数的返回值是通过函数中的 `return` 语句返回的。`return` 语句将被调用函数中的一个确定值带回主调函数中。

一个函数中可以有一个以上的 `return` 语句，执行到哪个 `return` 语句，该 `return` 语句起作用。执行到某个 `return` 语句，函数就执行完毕，之后的语句就不执行了。

如果函数值的类型（定义函数时函数名前的类型标识符）和 `return` 语句中表达式值的类型不一致，则以函数值类型为准，即函数值类型决定返回值的类型。对数值型数据，可以自动进行类型转换。

在 1.7.1 节的例 1.17 中用 `if` 结构实现了数学上的分段函数，在下面的例 1.36 中用函数形式来改写例 1.17 的程序。

**例 1.36** 多个 `return` 语句的例子：通过自定义函数的方式实现如下的分段函数。

$$y = \begin{cases} x^2 & (x < 0) \\ \sqrt{x} & (0 \leq x < 9) \\ x - 6 & (x \geq 9) \end{cases}$$

代码如下：

```

#include <stdio.h>
#include <math.h>
double f( double x1 )
{
    if( x1<0 ) return (x1*x1);
    else if( x1<9 ) return (sqrt(x1));
    else return (x1 - 6);
}
void main( )
{
    double x, y;
    printf( "Please input x : " );
    scanf( "%lf", &x );
    y = f( x );
    printf( "y = %.4f\n", y );
}

```

```
}

```

该程序的运行示例如下：

Please input x : 2.7 ✓

y = 1.6432

在上面的程序中，函数 `f` 里有多条 `return` 语句：根据形式参数 `x1` 的取值执行不同的 `return` 语句。要注意的是，执行到一个 `return` 语句，函数 `f` 就执行完毕，后面的分支不会再判断下去，`if` 结构后面如果还有代码也不会执行了。□

注意，`return` 语句只能向主调函数返回一个值，如果函数执行后得到两个以上的结果，是无法通过 `return` 语句将结果返回到主调函数的。具体例子见例 1.38。

### 1.9.5 函数的调用

#### 1. 函数调用的一般形式

函数调用的一般形式为：

**函数名 ( [实参列表] )**

如果是调用无参函数，则“实参列表”可以没有，但括号不能省略。如果实参列表包含多个实参，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（相同或赋值兼容）。实参与形参按顺序对应，一对一地传递数据。

#### 2. 函数调用的方式

按函数在语句中的作用来分，可以有以下 3 种函数调用方式：

##### 1) 函数语句

在函数调用后面加一个分号，构成一个语句，即把函数调用单独作为一个语句，并不要求函数带回一个值，只是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，这时要求函数带回一个确定的值以参加表达式的运算。如 `c = 2*max(a,b)`。

注意，如果函数没有返回值，则不能将函数调用放到一个表达式进行运算。

3) 作为函数参数：函数调用可以作为另一个函数调用的实参，同样要求函数有返回值。如：  
`m = max( a, max(b,c) );` //`max(b,c)`是函数调用，其值作为外层 `max` 函数调用的一个实参

#### 3. 对被调用函数的声明和函数原型

如果函数的定义出现在函数调用语句之后，则必须在函数调用语句之前对函数进行声明，如在例 1.34 的代码中，如果 `max` 函数的定义出现在 `main` 函数之后，则必须在函数调用语句“`c=max(a,b);`”之前对 `max` 函数进行声明，否则会出现编译错误，提示：标识符“`max`”未声明。对 `max` 函数进行声明的语句可以放在 `main` 函数里，也可以放在 `main` 函数之前，只要保证出现在 `max` 函数调用语句之前就可以了。代码如下：

```
#include <stdio.h>
int max( int x, int y ); //对 max 函数进行声明
void main( )
{
    int a, b, c;
    printf( "please enter two integer numbers: " );
    scanf( "%d%d", &a, &b );
    c=max( a, b );           //调用 max 函数，给定实参为 a, b。函数值赋给 c
    printf( "max=%d\n", c );
}
```



```
int max( int x, int y )  //定义有参函数 max: 求两个数的较大者
{
    int m;
    m = x>y ? x : y;
    return(m);
}
```

从上面代码中可以知道，函数声明其实就是取了函数定义的第一行，然后加分号。这种函数声明就是**函数原型**（function prototype）。

具体来说，函数的声明有两种格式：

**(1) 函数类型 函数名( 参数类型 1, 参数类型 2, …… );**

**(2) 函数类型 函数名( 类型名 参数名 1, 类型名 参数名 2, …… );**

也就是说，在对函数进行声明时，形参名是可以省略的。即使写上，甚至声明时的形参名跟定义时的形参名不一致，也是可以的，因为声明时的形参名在编译时是忽略的。

### 1.9.6 函数的嵌套调用

例 1.34 及图 1.31 只有一层函数调用，实际程序中可能有多层函数调用，即被调函数又作为主调函数去调用其他函数，这种多层函数调用称为**嵌套调用**。

图 1.32 所示的嵌套调用，其执行过程是：

- ① 执行 main 函数的开头部分；
- ② 执行到调用 f1 函数的语句，流程转去执行 f1 函数；
- ③ 执行 f1 函数的开头部分；
- ④ 执行到调用 f2 函数的语句，流程转去执行 f2 函数；
- ⑤ 执行 f2 函数，如果再无嵌套调用其他函数，则执行完 f2 函数的全部语句；
- ⑥ 返回 f1 函数中调用 f2 函数的语句处；
- ⑦ 继续执行 f1 函数中尚未执行的部分，直到 f1 函数执行完毕；
- ⑧ 返回到 main 函数中调用 f1 函数的语句处；
- ⑨ 继续执行 main 函数的剩余部分直到整个程序执行完毕。

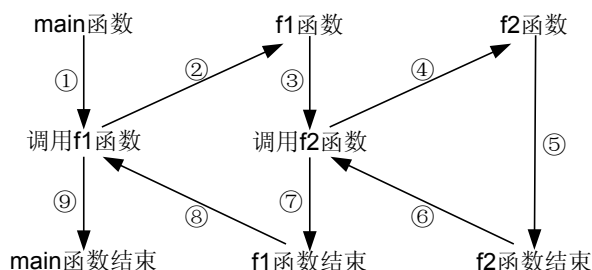


图1.32 函数嵌套调用过程

嵌套调用的例子，请参考 1.9.7 节的例 1.39。

### 1.9.7 函数的设计

很多初学者对函数比较头疼，不知道该如何设计函数。具体体现在：

- 1) 不知道函数是否有参数，有几个参数，是否有返回值。
- 2) 不明确函数要处理的数据是哪些，不明白函数形参的作用是什么，形参的值是在什么时候被“赋予”的。初学者经常在函数里通过输入语句给形参输入数据。

对于第 1 个问题，程序设计者希望采用怎样的形式去调用函数，这种函数调用形式里有几个参数，分别是什么类型，是以此来确定函数的形参数数和类型；程序设计者希望函数执行以后是

否得到一个结果，这个结果是什么类型的，是什么含义，是否需要返回到主调函数中，以此来确定函数的返回值及其类型、含义等。

对于第 2 个问题，函数形参是在函数调用时，通过实参与形参之间的数据传递，从而“被赋予”了值。其实前面也提到了，只要没有函数调用发生，就不会给形参分配存储空间。当函数调用发生时，为形参分配存储空间，并把实参的值赋值给形参。

下面举一些例子进一步阐述上述方法。

**例 1.37** 输出 100~200 间的全部素数。要求如下：

- 1) 定义一个函数 `prime`，用于判断 `x` 是否为素数，如果为素数，返回 1，否则返回 0。
- 2) 在主函数中调用 `prime` 函数，用于判断 100~200 之间的每个数是否为素数。

**分析：**根据题目的意思，主调函数中调用 `prime` 函数的形式是 `prime(199)`，即判断 199 是否为素数，如果为素数则返回 1，否则返回 0。因此，`prime` 函数的原型为：

```
int prime( int x );
```

另外，在 `prime` 函数里，是要判断形参 `x` 是否为素数，这个 `x` 的值不是在 `prime` 函数里通过输入语句输进去的，也不是采用赋值的方式“赋予”给它的，而是在主调函数调用 `prime` 函数时，如 `prime(199)`，把实参 199 的值传递给形参的，因此这时执行 `prime` 函数，形参 `x` 的值就是 199，调用 `prime` 函数就是要判断 199 是否为素数。整个程序的代码如下：

```
#include <stdio.h>
#include <math.h>
int prime( int x ) //如果 x 为素数；则返回 1，否则返回 0
{
    int k = sqrt(x);
    for( int i=2; i<=k; i++ )
    {
        if( x%i==0 ) //采用例 1.26 中的方法二判断素数
            break;
    }
    if( i>k ) return 1; //x 为素数，返回 1
    else return 0; //x 为合数，返回 0
}
void main( )
{
    int count = 0; //控制每行输出 10 个素数的变量
    for( int m=100; m<=200; m++ )
    {
        if( prime(m) ) //调用 prime 函数，判断 m 是否为素数
        {
            printf( "%d ", m ); //输出素数
            count++;
            if( count%10==0 ) //控制每行输出 10 个素数
                printf( "\n" );
        }
    }
}
```

该程序的输出结果为：

```
101 103 107 109 113 127 131 137 139 149
151 157 163 167 173 179 181 191 193 197
199
```

□

**例 1.38** 求 Fibonacci 数列中大于  $n$  的第一个数及其在 Fibonacci 数列中的序号，要求：

1) 定义函数 **fibonacci**：用于求解并输出 Fibonacci 数列中大于  $m$  的第一个数及其在 Fibonacci 数列中的序号。

2) 在 **main** 函数中输入  $n$ ，调用 **fibonacci** 函数输出结果。

**分析：**根据题目的意思，主调函数中调用 **fibonacci** 函数的形式是 **fibonacci(1000)**，即求 Fibonacci 数列中大于 1000 的第一个数及其在 Fibonacci 数列中的下标，求得的结果有两个，因此不能以返回值的方式返回这两个结果，只能在 **fibonacci** 函数中进行输出，这样 **fibonacci** 函数就没有返回值。因此，**fibonacci** 函数的原型为：

```
void fibonacci( int m );
```

其中形参  $m$  的值也是在函数调用时通过实参与形参之间的数据传递从而被“赋予”的。

另外，在例 1.29 中介绍了递推 Fibonacci 数列每一项的两种方法，在这里采用第一种方法比较方便：每次递推出一项。

```
#include <stdio.h>
void fibonacci( int m )
{
    int f1 = 1, f2 = 1; //fibonacci 数列中相邻的 2 个数
    int t;             //用来保存 f2 的临时变量
    int sn = 2;        //f2 在 fibonacci 数列中的序号
    while( f2<=m )     //当新递推出来的这一项 f2 的值小于等于 m，反复递推
    {
        t = f2;
        f2 = f1 + f2;
        f1 = t;
        sn++;
    } //注意 while 循环执行完毕时，f2 就是大于 m 的第 1 个数
    printf( "Fibonacci 数列中，第一个大于%d 的数是第%d 个数，其值为%d\n", m, sn, f2 );
}
void main( )
{
    int n;
    printf( "Please input n : " );
    scanf( "%d", &n );
    fibonacci( n ); //调用 fibonacci 函数，没有返回值
}
```

该程序的运行示例如下：

Please input n : 1000✓

Fibonacci 数列中，第一个大于 1000 的数是第 17 个数，其值为 1597

□

**说明：**形参和实参可以用相同的变量名（不会有语法错误，因为这是在不同函数中定义的变量），但为了避免混淆，建议实参和形参用不同的变量名。如本题中，形参名为  $m$ ，实参名为  $n$ 。

**思考 1.15：**在本题中，如果要求 Fibonacci 数列中小于  $m$  的最大项，**fibonacci** 函数该如何修改？

前面两个例子中，都只有一层函数调用，下面再举一个多层函数调用的例子。

**例 1.39** 抛物线  $y = x^2 / (2 \cdot p)$  绕它的对称轴  $x = 0$  旋转所成的曲面就是旋转抛物面。放在焦点  $F(0, p/2)$  处的光源所发出的光，经过抛物面各点反射之后就成为平行光束。可以利用这一性质制造需要发射平行光的灯具，例如：探照灯，汽车的车前灯等。请编写程序验证这个性质。

**分析：**题目的意思是，如图 1.33 所示，从焦点  $F$  发射的任意光线，比如图中的两条光线  $L$  和  $L'$ ，经过抛物面反射后，反射光线  $R$  和  $R'$  都平行  $y$  轴。

要证明反射光线  $R$  平行  $y$  轴，只要证明  $\angle 1 = \angle 3$ ，而  $\angle 1$  和  $\angle 2$  是相等的，所以只要证明  $\angle 2 = \angle 3$  即可，即只要证明  $FC = FT$ ，这里点  $C$  是光线  $L$  与抛物线的交点，点  $T$  是抛物线在  $C$  点的切线与  $y$  轴的交点。

以下编写程序，实现：任意给定抛物线参数  $p$  和发射光线斜率  $k$ ，输出线段  $FC$  和  $FT$  的长度。求解过程如下：

- 1) 求直线  $L: y = k*x + p/2$  与抛物线的交点  $C$ （有两个交点，只取横坐标小于 0 的交点，即图中的  $C$ ；另一个交点的处理方法类似）的坐标为  $(x_c, y_c)$ ，其中  $x_c = p * (k - \sqrt{k^2 + 1})$ 。
- 2) 抛物线在点  $C$  处的切线斜率： $k_c = x_c / p = k - \sqrt{k^2 + 1}$ 。
- 3) 切线方程为： $y = k_c(x - x_c) + y_c$ ，切线与  $y$  轴的交点  $T$  的坐标为  $(x_t, y_t)$ ，其中  $x_t = 0$ ， $y_t = -k_c x_c + y_c$ 。
- 4) 求解线段  $FC$  和  $FT$  的长度并输出。

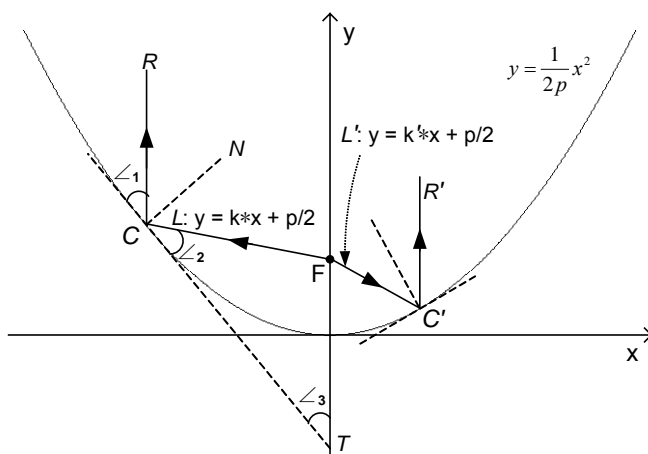


图1.33 从焦点发出的光线经抛物面反射后平行光轴

设计函数如下：

- 1) **main** 函数：在 **main** 函数中输入抛物线参数  $p$  和直线参数  $k$ ，接下来所有工作都是通过调用 **solve** 函数实现的。
- 2) **solve** 函数：求交点  $C$  和交点  $F$  的坐标，并调用 **length** 函数求线段  $FC$  和  $FT$  的长度并输出。**solve** 函数有两个形参，即抛物线参数  $p$  和直线参数  $k$ ，没有返回值。**solve** 函数的原型为：

```
void solve( double p, double k );
```

- 3) **length** 函数：求平面上两点  $(x_1, y_1)$  和  $(x_2, y_2)$  的距离，即连接这两点的线段的长度。该函数有 4 个形参，为两个点的坐标；返回值为求的线段长度。**length** 函数的原型为：

```
double length( double x1, double y1, double x2, double y2 );
```

代码如下：

```
#include <stdio.h>
#include <math.h>
double length( double x1, double y1, double x2, double y2 )    //线段的长度
{
    return sqrt( (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) );
}
void solve( double p, double k )//求解线段 FC 和 FT 的长度并输出
{
    double xc, yc, kc, xt, yt;    //直线 L 与抛物线的交点为 C，及切线与 y 轴的交点为 T
```

```

    xc = p*( k - sqrt( k*k+1 ) );  yc = xc*xc/(2*p);
    kc = xc/p;
    xt = 0;  yt = -kc*xc + yc;
    printf( "the length of the segment FC is %0.8f.\n", length(0,p/2,xc,yc) );
    printf( "the length of the segment FT is %0.8f.\n", length(0,p/2,xt,yt) );
}
void main( )
{
    double p, k;
    printf( "Please input the parameters, p and k : ");
    scanf( "%lf%lf", &p, &k ); //输入抛物线和直线的参数
    solve( p, k ); //调用函数求解并输出线段 FC 和 FT 的长度
}

```

该程序的运行示例 1 如下：

```

Please input the parameters, p and k : 2 -1 ✓
the length of the segment FC is 6.82842712.
the length of the segment FT is 6.82842712.

```

该程序的运行示例 2 如下：

```

Please input the parameters, p and k : 3 2 ✓
the length of the segment FC is 1.58359214.
the length of the segment FT is 1.58359214.

```

从以上运行示例可以看出，对不同的抛物线参数  $p$  和直线参数  $k$ ，求得的两条线段长度都是相等的，从而验证了这一性质。 □

## 练习

- 1.39 设计一个函数 **edge**，实现：根据直角三角形的两条直角边长度，求斜边长度。在 **main** 函数中输入两条直角边的长度，调用 **edge** 函数求斜边长度。
- 1.40 设计一个函数 **root**，求一元二次方程  $ax^2 + bx + c = 0$  的根，并在 **main** 函数中调用该函数。  
注意：由于方程可能有两个根，因此无法采用返回值的方式将求得的根返回到主函数中，所以应在 **root** 函数中输出根的信息。
- 1.41 编写程序，实现闰年的判断。从主函数输入一个年份，然后调用 **leap** 函数判断该年份是否为闰年。**leap** 函数是自定义函数，用于判断一个年份是否为闰年。
- 1.42 编程实现：任意输入 3 个整数，输出其中的最大值。函数 **maximum()** 用来求 3 个数中的最大值，主函数输入三个数并调用 **maximum** 函数求三个数的最大数并输出。
- 1.43 模仿例 1.37，输出 6~10000 之间的完数。
  - 1) 定义 **perfect** 函数，用于判断  $x$  是否为完数。如果为完数，返回 1，否则返回 0。函数原型为： **int perfect(int x)**;
  - 2) 在主函数中调用 **perfect** 函数，判断 6~10000 之间的每个数是否为完数。
- 1.44 模仿例 1.38，求 Fibonacci 数列中不大于  $m$  的前  $n$  项和的最大值及  $n$  并输出。要求：
  - 1) 定义函数 **fibo**：用于求解并输出 Fibonacci 数列中不大于  $s$  的前  $n$  项和的最大值及  $n$ 。  
**fibo** 函数的原型如下：

**void fibo( int s );** //s 为形参

- 2) 在主函数中输入一个正整数  $m$ ，调用 **fibo** 函数求解。**fibo** 函数的调用形式为：**fibo(m)**，即  $m$  作为实参。

整个程序的运行示例如下：

```
100 ✓
```

88 9

表示, fibonacci 数列中不大于 100 的前 n 项和是前 9 项和, 其值为 88。

- 1.45 编程实现: 输入一个正整数, 统计该正整数的各位上数字中, 0 出现了几次, 1 出现了几次, ...。如没有出现, 则不输出该位数字的信息。该程序的运行示例如下:

566675801 ✓

0:1

1:1

5:2

6:3

7:1

8:1

提示: 定义函数 f, 用于统计整数 number 中, 数字 digit 在 number 的各位上出现了几次, 函数的原型如下:

int f( int number, int digit );

在主函数中输入一个整数, 然后调用 f 函数分别统计 0 出现了几次, 1 出现了几次, ..., 并输出相关信息; 如没有出现, 则不输出该位数字的信息。

- 1.46 编写程序: 在主函数任意输入 3 个整数, 调用 sort 函数对这三个整数按从小到大的顺序进行输出。

提示: 因为 return 语句只能返回一个值, 因此在 sort 函数中对这 3 个整数排好后, 是不能把这 3 个数都返回到 main 函数的, 因此只能在 sort 函数中进行输出。

- 1.47 设计函数 reverse, 实现将一个整数 number (符号可以为负) 逆序。在 main 函数中调用 reverse 函数实现将一个输入的整数逆序并输出 (要求能去掉前导 0, 比如 900 逆序后变成 9; 还要求能保留符号, 如 -5908 逆序后变成了 -8095)。提示: 例 1.25 已经实现了将一个正整数各位上的数字取出来, 将一个正整数逆序的方法是在取出各位数字时将它组合成逆序的正整数。reverse 函数的原型为:

int reverse( int number );

- 1.48 某客户为购房办理商业贷款, 选择了按月等额本息还款法, 在贷款本金 (loan) 和月利率 (rate) 一定的情况下, 住房贷款的月还款额 (money) 取决于还款月数 (month), 计算公式如下。客户打算在 5~30 年的范围内选择还清贷款的年限, 想得到一张“还款年限——月还款额表”以供参考。

$$money = loan \times \frac{rate(1 + rate)^{month}}{(1 + rate)^{month} - 1}$$

- (1) 定义函数 cal\_power( x, n ) 计算 x 的 n 次幂 (即  $x^n$ ), 函数返回值类型是 double。
- (2) 定义函数 cal\_money( loan, rate, month ) 计算月还款额, 函数返回值类型是 double, 要求调用函数 cal\_power( x, n ) 计算 x 的 n 次幂。
- (3) 定义函数 main( ), 输入贷款本金 loan (元) 和月利率 rate, 输出“还款年限——月还款额表”, 还款年限的范围是 5~30 年, 输出时分别精确到年和元。要求调用函数 cal\_money( loan, rate, month ) 计算月还款额。

## 1.10 数组

如果程序中有若干个数据, 我们通常需要为每个数据定义变量。但是如果这些数据具有相同的数据类型, 并且存在一定的内在联系, 比如 100 个学生的数学成绩数据, 我们就可以把这些数

据放到同一个数组中。用数组名代表这一批数据，用下标来区分这些数据。这样处理的好处有两个：① 可以省略很多个变量名，使得程序精炼；② 便于对这些数据作统一处理。

具体来讲：**数组是一组有顺序的数据的集合，用统一的名字代表这一组数据，用序号来区分这组数据中的各个数据；数组中每个数据称为数组的元素。**

数组的特点：

- 1) 具有类型属性。在定义数组时必须指定数组的类型，与普通变量的定义类似。
- 2) 一个数组在内存中占一片连续的存储单元。

元素的特点：

- 1) 同一数组中的每一个元素都必须属于同一数据类型。
- 2) 用数组名和下标惟一地标识每个数组元素。
- 3) 在 C/C++中用方括号来表示下标，并且下标是从 0 开始的。

例如，假设在程序中有一个整型数组 **a**，有 5 个元素，并且假设第 0 个元素的起始地址是 2000（十进制），则数组中 5 个元素的存储情形如图 1.34 所示。

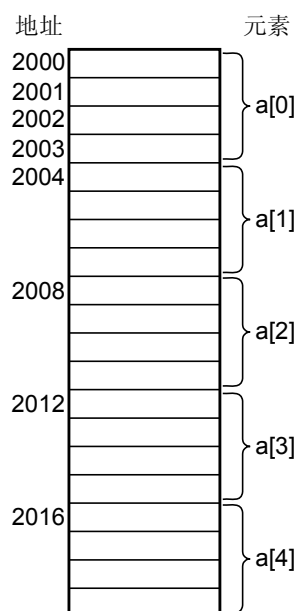


图1.34 数组各元素存储情形

### 1.10.1 一维数组的定义与引用

#### 1. 定义一维数组

定义一维数组的一般格式为：

**类型标识符 数组名 [常量表达式];**

例如：

```
int a[10];
```

定义了一个整型数组，数组名为 **a**，有 10 个元素。

说明：

- (1) 数组名也是标识符，和变量名一样，数组名的命名必须遵循标识符的命名规则。
- (2) 数组名表示整个数组所占存储空间的首地址（其实也是第 0 个元素的首地址），是一个表示地址的常量，它的值是不能改变的。比如，“**a++**”就是非法的。
- (3) 用方括号括起来的常量表达式表示数组元素个数，即数组长度。如下面的写法是合法的：  

```
int a[10];    //数组 a 中有 10 个元素
```

```
int a[2*5];
```
- (4) 数组元素的下标从 0 开始。例如，定义了“**int a[10];**”，则这 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]**。注意最后一个元素是 **a[9]**，而不是 **a[10]**。
- (5) 常量表达式中可以包括常量、常变量和符号常量（常变量的概念请参考文献[2]2.2.4 节，符号常量的概念请参考文献[2]2.3.5 节），但不能包含变量。也就是说，C/C++不允许对数组的大小作动态定义，即数组的大小不能依赖于程序运行过程中变量的值。详见下面的例子。

情形 1：

```
int n;
scanf( "%d", &n );    //输入 a 数组的长度
int a[n]; //企图根据 n 的值决定数组的长度
```

情形 2：

```
const int n = 5;    //n 为常变量
```

```
int a[n];
```

情形 3：

```
int a = 5;
```

```
int a[n];
```

其中情形 1、3 是非法的，情形 2 是合法的。

C/C++为什么要作这样的规定？因为在编译阶段编译系统需要知道数组的长度（以便确定数组占多大的存储空间），因此表示数组长度的表达式必须在编译阶段有确定的值。

## 2. 引用一维数组的元素

数组必须先定义，再使用。并且只能逐个引用数组元素而不能一次性引用整个数组中的全部元素。如下面的代码试图一次性把数组中 10 个元素的值输出来，这是错误的：

```
int a[10];
printf( "%d", a );
```

引用数组元素的形式为：

**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]
```

如果要对数组各元素进行输入/输出，可以使用循环。如下面的例子：

```
int i, a[10];
for( i=0; i<10; i++ )    //输入 10 个数据到数组(各元素)中
    scanf( "%d", &a[i] );
for( i=0; i<10; i++ )    //输出 10 个数组元素的值
    printf( "%d", a[i] );
```

## 3. 一维数组的初始化

所谓**初始化**，就是在定义数组时，就给数组元素赋初始值。对一维数组的初始化可以采用下面的方式：

(1) 在定义数组时分别对每个数组元素赋予初值。例如：

```
int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

(2) 可以只给一部分元素赋值，对于没有赋值的元素编译器会自动赋值为 0。例如：

```
int a[10] = { 0, 1, 2, 3, 4 }; //a[5]~a[9]的值由编译器自动赋值为 0
```

(3) 在对全部数组元素赋初值时，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 };

编译器会根据花括号内初始值的个数确定数组的长度。

## 4. 一维数组程序举例

本节分析两道应用一维数组的题目。

**例 1.40** 用数组存储从键盘上输入的 10 个整数，并求这 10 个整数的最大值、最小值和平均值。

在本例中，采用一重 for 循环实现访问数组中的每个元素并做一些处理。代码如下：

```
#include <stdio.h>
void main( )
{
    int a[10], i;
    int max, min; //最大值、最小值
    double average; //平均值
    printf( "Please input 10 numbers :\n" );
    for( i=0; i<10; i++ )
        scanf( "%d", &a[i] );
```



```

max = min = a[0];
average = a[0];
for( i=1; i<10; i++ )
{
    if( a[i]>max ) max = a[i];
    if( a[i]<min ) min = a[i];
    average += a[i];
}
printf( "max = %d, min = %d, average = %.1f\n", max, min, average );
}

```

该程序的运行示例如下：

Please input 10 numbers :

28 39 -11 13 -26 98 135 -37 52 72 ✓

max = 135, min = -37, average = 36.3

在上面的程序中，max、min、average 的初始值均为 a[0]，因此第 2 个 for 循环是从 i=1 开始循环。另外，在统计平均值时，是先将 10 个整数累加起来，然后除以 10。 □

**例 1.41** 用数组处理例 1.29 的问题：输出 Fibonacci 数列前 40 个数。

在例 1.29 中提到，要表示 Fibonacci 数列中前 40 个数，可以用 40 个变量，不切实际。在例 1.29 中采用的是递推方法。而在本例中采用一个数组来存放 Fibonacci 数列中的 40 个数，前两个数是在初始化时赋值的，其他 38 个数的值是通过递推求得的，如图 1.35 所示。

```

#include <stdio.h>
void main( )
{
    int i;
    int f[40] = { 1, 1 }; //前 2 个数通过初始化方法赋值
    for( i=2; i<40; i++ ) //递推后面 38 个数
        f[i] = f[i-2] + f[i-1];
    for( i=0; i<40; i++ )
    {
        if( i%5==0 ) printf( "\n" ); //每行输出 5 个数据
        printf( "%12d", f[i] ); //每个数据占 12 个字符的宽度
    }
    printf( "\n" );
}

```

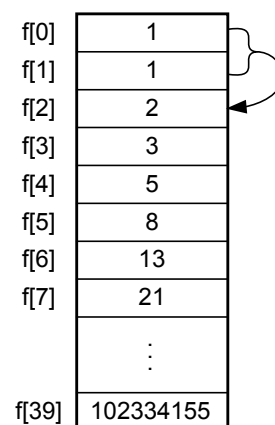


图1.35 用数组处理 fibonacci 数列问题

该程序首先输出一个空行（当 i 等于 0 时），然后输出 Fibonacc 数列前 40 个数，每行输出 5 个。 □

**思考 1.16：** 如果要避免输出第一个空行，程序该如何改写？

## 1.10.2 二维数组的定义和引用

实际应用中有些数据需要依赖两个因素才能唯一地确定。例如有 3 个学生，每个学生有 4 门课程的成绩，显然，成绩数据是一个二维表，如表 1.5 所示。

表 1.5 学生成绩数据

| 学生学号   | 数学 | 大学语文 | 英语 | 计算机导论 | 程序设计基础 |
|--------|----|------|----|-------|--------|
| 200501 | 80 | 79   | 86 | 82    | 79     |
| 200502 | 78 | 81   | 87 | 84    | 82     |
| 200503 | 85 | 74   | 83 | 90    | 74     |

如果想表示第 3 个学生第 4 门课程的成绩，就需要指出学生的序号和课程的序号两个因素，

在数学上以 $S_{3,4}$ 表示。在C/C++中要用 $S[2][3]$ 来表示（注意下标从0开始），它代表数据90。

具有两个下标的数组称为二维数组。

1. 定义二维数组

定义二维数组的一般形式为：

**数据类型 数组名 [常量表达式] [常量表达式];**

如：

`int a[3][4];` //定义a为3行4列的二维数组

`float b[2][5];`

对上面的数组a，一共有 $3 \times 4 = 12$ 个元素，其中第0行的4个元素是： $a[0][0]$ 、 $a[0][1]$ 、 $a[0][2]$ 、 $a[0][3]$ ，第1行的4个元素是： $a[1][0]$ 、 $a[1][1]$ 、 $a[1][2]$ 、 $a[1][3]$ ，第2行的4个元素是： $a[2][0]$ 、 $a[2][1]$ 、 $a[2][2]$ 、 $a[2][3]$ 。

这12个元素在内存中存储的顺序是：按行存放，即在内存中先按顺序存放第0行的4个元素，然后是第1行的4个元素，最后是第2行的4个元素。如图1.36所示。而且这12个元素在内存中也是连续存放的，如果 $a[0][0]$ 的地址是2000（十进制），则 $a[0][1]$ 的地址是2004，...，最后一个元素 $a[2][3]$ 的地址是2044。如图1.37所示。

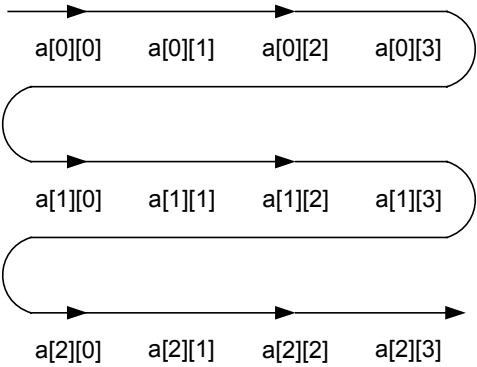


图1.36 二维数组各元素存储顺序

|      |           |
|------|-----------|
| 2000 | $a[0][0]$ |
| 2004 | $a[0][1]$ |
| 2008 | $a[0][2]$ |
| 2012 | $a[0][3]$ |
| 2016 | $a[1][0]$ |
| 2020 | $a[1][1]$ |
| 2024 | $a[1][2]$ |
| 2028 | $a[1][3]$ |
| 2032 | $a[2][0]$ |
| 2036 | $a[2][1]$ |
| 2040 | $a[2][2]$ |
| 2044 | $a[2][3]$ |

图1.37 二维数组在内存中的存储情形

2. 引用二维数组的元素

二维数组元素的引用形式为：

**数组名 [行下标] [列下标];**

如： $a[2][3] = b[3][2] + 1$ ;

3. 二维数组的初始化

二维数组的初始化有两类方法：分行初始方法和按元素排列顺序初始化方法。

1) 分行初始方法：在花括号内再以花括号的形式将每行元素列举出来，有3种形式。

① 全部元素初始化：

例如：`int a[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };`

② 部分元素初始化：

例如：`int a[2][3] = { { 1, 2 }, { 4 } };` // $a[0][2]$ 、 $a[1][1]$ 、 $a[1][2]$ 由编译器自动赋值为0

③ 初始化时第一维长度省略：

例如：`int a[ ][3] = { { 1 }, { 4, 5 } };`

第一维下标是根据最外围花括号内花括号的个数来确定的，比如上述例子等效于：

`int a[2][3] = { { 1 }, { 4, 5 } };`

注意：第二维的长度不能省略！

2) 按元素排列顺序初始化方法：把所有元素按元素的排列顺序在花括号内列举出来，同样也有 3 种形式。

① 全部元素初始化：

例如：int a[2][3] = { 1, 2, 3, 4, 5, 6 };

② 部分元素初始化：

例如：int a[2][3] = { 1, 2, 4 };

③ 初始化时第一维长度省略：

例如：int a[ ][3]={ 1, 2, 3, 4, 5, 6, 7, 8 };

花括号内有 8 个数据，前面 3 个是第 0 行的元素，中间 3 个是第 1 行的元素，最后两个是第 2 行的元素，编译器对第 2 行的最后一个元素 a[2][2] 自动赋值为 0。这样该数组一共有 3 行，即上述语句等效于：

```
int a[3][3]={1, 2, 3, 4, 5, 6, 7, 8};
```

同样要注意：第二维的长度不能省略！

#### 4. 二维数组程序举例

遍历二维数组需要采用二重循环。所谓**遍历**，就是访问每个元素一次且仅一次，不能重复也不能由遗漏。例如，输出二维数组的每个元素，就需要访问每个元素一次且仅一次。假设有一个 3×4 的二维数组，则遍历该数组要用到下面的 2 重循环：

```
int i, j;    //循环变量
for( i=0; i<3; i++ ) //遍历第 0 行~第 2 行
{
    for( j=0; j<4; j++ ) //遍历每行的第 0 列~第 3 列元素
    {
        //在此处访问 a[i][j]，保证能访问每个元素一次且仅一次
    }
}
```

**例 1.42** 有一个 3×4 的矩阵（即二维数组），要求程序求出其中值最大的那个元素的值，以及该元素所在的行号和列号。

求一组数中的最大数，要用到“**摆擂台**”的思想。其实这种思想在例 1.18 和例 1.30 中已经用过了。

“摆擂台”思想的具体思路是：初始时把 a[0][0] 的值赋给变量 max，然后让每一个元素与 max 比较。如果这个元素比 max 的当前值还要大，则把该元素的值赋值给 max，并且用 row 和 column 记录该元素的两个下标。这样 max 就是当前找到的最大的数。一直比较到最后一个元素为止。max 最后的值就是矩阵所有元素中的最大值。代码如下：

```
#include <stdio.h>
void main( )
{
    int i, j, row, column, max;
    int a[3][4] = { { 5, 12, 23, 56 }, { 19, 28, 37, 46 }, { -12, -34, 6, 8 } };
    max = a[0][0], row = 0, column = 0;    //使 max 开始时取 a[0][0] 的值
    for( i=0; i<=2; i++ )                //从第 0 行~第 2 行
    {
        for( j=0; j<=3; j++ )            //从第 0 列~第 3 列
        {
            if( a[i][j]>max )              //如果某元素大于当前的 max
```

```

        {
            max = a[i][j]; //max 将取该元素的值
            row = i; column = j; //记下该元素的行号 i 和列号 j
        }
    }
}
printf( "max=%d, row=%d, column=%d\n", max, row, column );
}

```

该程序的输出结果为：

max=56, row=0, column=3

□

### 1.10.3 数组名作函数参数

除了常量、变量、表达式等可以作函数实参外，数组名也可以作函数的实参。如果函数的实参是数组名，则函数的形参也必须是数组名（或指针变量，详见 1.11.4 节）。如下面的例子。

**例 1.43** 将数组 `array` 中的 10 个元素按相反顺序存放。要求用函数实现。

**分析：**将数组各元素按相反顺序存放，只需要把 `array[0]` 和 `array[9]` 交换，`array[1]` 和 `array[8]` 交换，...，`array[4]` 和 `array[5]` 交换。题目要求用函数实现，所以定义 `reverse` 函数，实现将数组 `a` 中的 `n` 个元素按相反顺序存放。在主函数中输入数组 `array` 各元素值，然后调用 `reverse` 函数实现逆序存放，最后在主函数中输出 `array` 各元素的值。`reverse` 函数的原型为：

```
void reverse( int x[ ], int n );
```

`reverse` 函数的第 1 个形参为数组名的形式，这种形式的形参之前没有使用过，下面会详细讲解。程序的代码如下：

```

#include <stdio.h>
void reverse( int a[ ], int n );    //函数声明
void main( )
{
    int array[10], i;
    for( i=0; i<10; i++) scanf( "%d", &array[i] );//输入
    reverse( array, 10 );    //(1)
    for( i=0; i<10; i++) printf( "%d ", array[i] );//输出
    printf( "\n" );
}
//reverse 函数用于将 a 数组中的 n 个元素按逆序存放
void reverse( int a[ ], int n )
{
    int i, j;    //循环变量
    int temp;    //交换 a[i]和 a[j]时用到的临时变量
    for( i=0; i<n/2; i++ )
    {
        j = n-1-i;
        temp = a[i]; a[i] = a[j]; a[j] = temp;    //这 3 条语句交换 a[i]和 a[j]
    }
}

```

该程序的运行示例如下：

```

12 7 -89 120 55 79 3 11 66 -45 ✓
-45 66 11 3 79 55 120 -89 7 12

```

**注意：**因为 `reverse` 函数并不“知道”`main` 函数中的 `array` 数组有多少个元素，所以需要将

array 数组的元素个数(10)这个信息传递给 reverse 函数，而 reverse 函数中的第 2 个形参 n 就是用来接收这个信息的。

**思考 1.17:** 如果程序中语句(1)修改成 reverse(array, 5)，还是输入上述数据，程序的输出结果是什么？

从程序的运行结果可以看到，main 函数中调用 reverse 函数中后，数组中各元素的值发生了变化，比如原来 array[0]的值为 12，reverse 函数执行后，array[0]的值变为-45 了。这一点是否跟 1.9.3 中提到的“实参对形参的数据传递是单向的，只由实参传给形参，不能由形参传回来给实参”是否有矛盾？

在 1.10.1 曾经提到：数组名表示整个数组所占存储空间的首地址，是一个表示地址的常量。因此在 reverse 函数调用语句：

```
reverse( array, 10 );
```

中，数组名作函数实参，传递给形参 a，这样，形参 a 也表示这段存储空间的首地址。如图 1.38 所示。在 reverse 函数中交换 a[i]和 a[j]的值，实际上交换的就是 array[i]和 array[j]的值。

那么是否可以直接在 reverse 函数中直接使用 array 数组呢？答案是否定的，因为数组 array 是在主函数中定义的，因此它的有效范围只限于主函数，在 reverse 函数中是不能使用数组 array 的。 □

需要说明的是：

(1) 如果函数实参是数组名，形参也应为数组名（或指针变量，详见 1.11.4 节），形参不能声明为普通变量（如 int a）。

(2) 形参采用数组名的形式，但实际上，声明形参数组并不意味着真正建立一个包含若干元素的数组，在调用函数时也不对它分配存储单元，只是用 a[]这样的形式表示 a 是一维数组名，以接收实参传来的地址。因此 a[]中方括号内的数值并无实际作用，编译系统对一维数组方括号内的内容不予处理。形参一维数组的声明中可以写元素个数，也可以不写。

因此，例 1.43 程序中，reverse 函数首部的以下几种写法都合法，作用相同。

```
void reverse ( int a[10], int n ) //指定元素个数与实参数组相同
void reverse ( int a[ ], int n ) //不指定元素个数
void reverse ( int a[5], int n ) //指定元素个数与实参数组不同
```

学了 1.11.4 节就会知道，C/C++实际上只把形参数组名作为一个指针变量来处理，用来接收从实参传过来的地址。

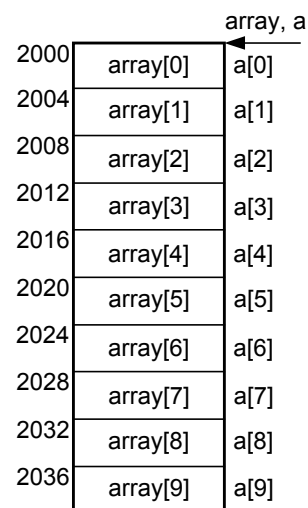


图1.38 用数组名作函数参数

#### 1.10.4 编写数组应用的综合程序

本节讲解两道综合应用数组知识的程序，其中例 1.44 应用了一维数组，例 1.45 可以用一维数组实现，也可以用二维数组实现。

**例 1.44** 输入一个日期（年、月、日），计算出该日期是该年的第几天。

分析：要计算某年某月某日是当年的第几天，就必须先知道该月份之前的各个月份具体的天数，还必须弄清该年是否为闰年这一重要信息。因此，在本题中，可以用一个一维数组存储平年中 12 个月份的天数。另外，本题还需要设计两个函数：

- 1) sum\_day 函数用来计算输入日期中的某月某日在平年里是第几天，方法是把该月份前每个月份的天数累加起来，再加上年月日中的日；
- 2) leap 函数用来判断输入的年份是否为闰年。

在下面的程序中，在 main 函数里输入日期，并调用 sum\_day 函数计算该日期在平年是第几

天；如果输入的年份是闰年并且月份大于等于 3，则天数还要加 1。

代码如下：

```
#include <stdio.h>
void main( )
{
    int sum_day( int , int );//函数声明
    int leap( int year );      //函数声明
    int year, month, day, days = 0;
    printf( "input date(year month day):" );
    scanf( "%d%d%d", &year, &month, &day );
    printf( "%d/%d/%d", year, month, day );
    days = sum_day(month,day);    //调用 sum_day 函数求该日期是第几天
    //如果年份是闰年且月份大于等于 3，则天数还要加 1
    if( leap(year) && month>=3 )    //调用 leap 函数判断是否闰年
        days = days + 1;
    printf( " is the %dth day in this year.\n", days );
}
int sum_day( int month, int day )
{
    int i;
    //用一维数组存储平面 12 个月份每月的天数
    int day_tab[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    for( i=0; i<month-1; i++ )
        day += day_tab[i];
    return (day);
}
int leap( int year ) //判断是否为闰年
{
    int leap;
    leap = (year%4==0&&year%100!=0||year%400==0);
    return( leap );
}
```

该程序的运行实例如下：

input date(year month day):2009 6 12✓

2009/6/12 is the 163th day in this year.

□

**例 1.45** 输入平面上 5 个点的坐标，求 5 个点各点间距离总和。要求分别用一维数组和二维数组实现。

分析：如果用一维数组实现，可以分别用  $x$  数组和  $y$  数组表示 5 个点的横坐标和纵坐标， $x[i]$ 、 $y[i]$  分别表示第  $i$  个点的横坐标和纵坐标；如果用二维数组实现，则可以用  $point[5][2]$  表示 5 个点的坐标， $point[i][0]$  和  $point[i][1]$  分别表示第  $i$  个点的横坐标和纵坐标。

用二维数组存储 5 个点坐标的代码如下：

```
#include <stdio.h>
#include <math.h>
void main( )
{
    int i, j;
    double length, sum = 0, point[5][2];
    for( i=0; i<5; i++ )
```

```

    {
        scanf( "%lf%lf", &point[i][0], &point[i][1] );
    }
    for( i=0; i<4; i++ )
    {
        for( j=i+1; j<5; j++ )
        {
            length = sqrt( (point[i][0]-point[j][0])*(point[i][0]-point[j][0])
                          + (point[i][1]-point[j][1])*(point[i][1]-point[j][1]) );
            sum += length;
        }
    }
    printf( "sum=%.8lf\n", sum );
}

```

如果要用一维数组存储 5 个点的坐标，则应该将 main 函数修改成：

```

void main( )
{
    int i, j;
    double length, sum = 0, x[5], y[5];
    for( i=0; i<5; i++ )
    {
        scanf( "%lf%lf", &x[i], &y[i] );
    }
    for( i=0; i<4; i++ )
    {
        for( j=i+1; j<5; j++ )
        {
            length = sqrt( (x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j]) );
            sum += length;
        }
    }
    printf( "sum=%.8lf\n", sum );
}

```

该程序的运行示例如下：

```

0 0 2 3 3 2 -2 2 -3 -3 ✓
45.539008

```

□

**思考 1.18：** 在上面的程序中，如果将二重 for 循环中第 2 重循环循环变量 j 的初值改成 0，则得到的距离总和正确吗？

## 练习

- 1.49 设计程序：用一维数组存储从键盘上输入的 10 个整数，统计这 10 个整数中偶数的个数。
- 1.50 用“筛选法”求 100 之内的素数。  
提示：将 1~100 存放到一个数组中，数组中的第 0 个元素不用；然后分别用 10 以内的素数 2,3,5,7 去判断第 1~100 个数组元素，如果能整除它，则将其置为 0(2,3,5,7 本身除外)。最后，元素值不为 0 的就是素数了。
- 1.51 设计程序：从键盘上输入一个正整数，输出其逆序，如果有前导 0，还需去掉前导 0。例如，如果输入的整数为 73152400，则输出 425137。提示：利用例 1.25 的方法取出该正整数中

的每位并存储到一个数组中（已经逆序了），在输出时从第 1 个非 0 的元素开始输出。

- 1.52 编程实现：从键盘上输入一个十进制正整数，将该正整数转换成二进制输出。提示：十进制转换成二进制的方法是将十进制数反复除以 2 取余数，在实现时需要将得到的余数保存到一个数组中并按相反的顺序输出。
- 1.53 设计程序：输入 10 个点的坐标；输出 10 个点中同时处于圆  $(x-1)*(x-1) + (y+0.5)*(y+0.5) = 25$  和  $(x-0.5)*(x-0.5) + y*y = 36$  内的点数 k。提示：用一维数组 x 和 y 实现，元素  $x[i], y[i]$  表示第 i 个点的坐标。
- 1.54 编程实现：分别统计  $4 \times 4$  矩阵中正数、负数、0 的个数并输出，矩阵元素值从键盘上输入。
- 1.55 编程实现：分别求  $4 \times 4$  矩阵主对角线和次对角线元素和并输出，矩阵元素值从键盘上输入。
- 1.56 分别按如下的格式要求输出杨辉三角形。
  - 1) 按图 1.39(a)所示的格式输出杨辉三角形前 10 行。提示：① 用  $10 \times 10$  的二维数组存储杨辉三角形；② 前面 10 行中最大的数为 126，为保证足够的间隔，每个整数应该按 4 位宽度进行输出。
  - 2) 通常杨辉三角形是以图 1.39(b)所示的形式来表示的，请按图(b)所示的格式输出杨辉三角形前 10 行。提示：这 10 行中，从第 0 行开始计起，也就是说最后一行为第 9 行。第 9 行前面没有空格，第 8 行比第 9 行在前面空出了半个数的宽度，也就是 2 个空格；第 7 行又比第 8 行空出了半个数的宽度：…。

[illegible]

图 1.39 杨辉三角形

- 1.57** 编程实现对二维数组进行转置，即：将二维数组行列元素互换，存到另一个数组中。例如，如果一个  $3 \times 5$  的二维数组存储情形如图 1.40(a)所示，则转置后将得到一个  $5 \times 3$  的二维数组，其存储情形如图 1.40(b)所示。

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| 78  | 57  | 13  | -25 | 66  |
| 65  | -52 | 34  | 7   | -55 |
| -33 | 19  | -27 | 22  | 91  |

(a)

|     |     |     |
|-----|-----|-----|
| 78  | 65  | -33 |
| 57  | -52 | 19  |
| 13  | 34  | -27 |
| -25 | 7   | 22  |
| 66  | -55 | 91  |

(b)

图 1.40 二维数组的转置

- 1.58 将以下 10 个数放到一个数组中。实现：查找一个数是否为该数组的元素，如果是，输出它在数组中第一次出现时的下标，如果不是，则输出-1；要求能反复查找若干个数。

12, 7, -89, 120, 55, 79, 3, 11, 66, -45

提示：要能反复查找若干个数，比如要陆续查找 29、66、57、13、...，每查找一个数，需要用循环，很麻烦。所以专门定义一个函数 **find**，实现在一个数组中查找某个数。然后在 **main** 函数中，每查找一个数只需要调用一次 **find** 函数即可。



## 1.11 指针与指针变量

指针是 C/C++ 语言中的一个重要概念。正确而灵活地使用指针变量，可以使程序简洁、紧凑、高效；可以有效地表示复杂的数据结构；可以实现动态分配内存；可以得到多于一个的函数“返回值”（如果不使用指针变量，被调函数最多只能通过 `return` 语句向主调函数返回一个值）等等。但是指针的概念也比较复杂，而且容易出错。本节只介绍指针及指针变量的基本概念、定义和引用指针变量的方法、以及将指针变量作为函数参数的方法等内容。

### 1.11.1 指针概述

#### 1. 变量的地址

要理解指针的概念，必须弄清楚变量的地址。程序中定义的每个变量，在编译时都会给它分配内存单元。编译系统根据变量的类型，从而给变量分配一定字节数的内存空间。例如，在 32 位计算机中，C/C++ 编译系统一般为整型变量分配 4 个字节，为字符型变量分配 1 个字节等等。每个内存单元（即字节）都有唯一的地址。假设在程序中有下面的变量定义语句：

```
int a = 5, b = 7;
```

则在内存中给变量 `a` 和 `b` 各分配了 4 个字节，如图 1.41 所示。假设分配给变量 `a` 的 4 个字节的地址是 2000~2003，分配给变量 `b` 的 4 个字节的地址是 2004~2007。一般以变量的低字节地址称呼为它的地址。因此，我们可以说变量 `a` 的地址是 2000。在变量 `a` 的有效期间，变量名 `a` 与分配给变量 `a` 的这一段存储空间就建立了一一对应的关系，所有对变量 `a` 的操作实际上就是对这 4 个字节进行操作。因此给变量 `a` 赋值为 5，实际上就是把 5（以二进制形式）存放到这 4 个字节里。

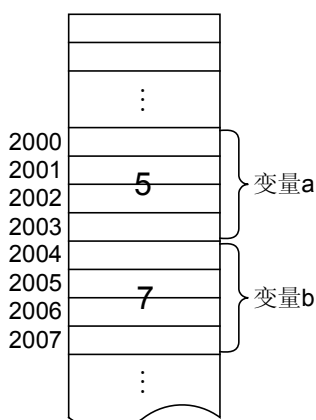


图1.41 变量名、变量的地址、变量的值

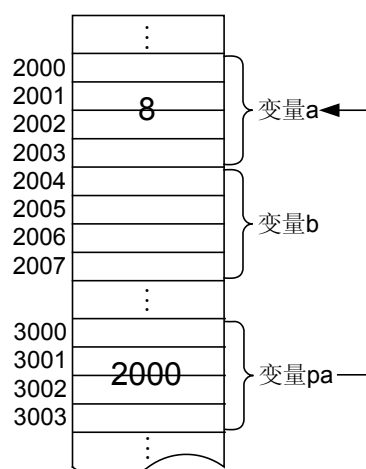


图1.42 直接存取与间接存取

另外，如果想从键盘上输入一个数据到变量 `a`，应该使用 `scanf` 函数，语句为：

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

在该语句中，`&a` 表示的就是 `a` 的地址，即把从键盘上输入的数据保存到变量 `a` 所在的内存空间中。

#### 2. 直接存取与间接存取

前面已提到，由于变量名与分配给变量的存储空间是一一对应的，因此如果想给变量 `a` 赋值，可以直接使用变量名 `a`。如：

```
a = 3 + 5;
```

上面这条赋值语句把表达式“`3 + 5`”的值存放到变量 `a` 所占用的存储空间中，替换原有的值。

如图 1.42 所示。这种直接通过变量名访问变量所占存储空间的方式，称为**直接存取方式**，或**直接访问方式**。

还可以采用另一种称为**间接存取（间接访问）**的方式访问变量。可以在程序中定义这样一种特殊的变量，它是专门用来存放变量的地址。如图 1.42 所示，变量 **pa** 就是这种变量。如果要把变量 **a** 的地址赋值给变量 **pa**，可以使用下面的语句：

```
pa = &a;
```

**&**是**取地址运算符**，**&a** 就是变量 **a** 的地址。执行上述语句后，变量 **pa** 的值就是 **a** 的地址，即 2000。如果要取出变量 **a** 的值，除了可以采用直接访问方式外，还可以采用**间接方式**：先找到存放“**a** 的地址”的变量 **pa**，从中取出变量 **a** 的地址（即 2000），然后到 2000 开始的 4 个字节中取出变量 **a** 的值。

如图 1.42 所示，变量 **pa** 和变量 **a** 构成了一种**指向关系**，这种指向关系是通过在变量 **pa** 中存放变量 **a** 的地址来实现的。

一个变量的地址称为该变量的**指针**。对于专门来存放另一个变量地址的变量，如 **pa**，称为**指针变量**。

指针变量是一种特殊的变量，它和以前学过的其他类型变量的不同之处是：它的内容不是一种数值信息，而是地址信息；并且由于它保存了另外一个变量的地址，从而指向了该变量。为了表示指针变量和它所指向的变量之间的联系，在 C/C++ 中用“**\***”运算符表示指向。例如，**pa** 是一个指针变量，而 **\*pa** 表示 **pa** 所指向的变量。因此，如果要采用间接访问方式访问变量 **a**，可以采用下面的语句：

```
*pa = 3 + 5;
```

该语句把表达式“**3 + 5**”的值存放到指针变量 **pa** 所指向的变量，即变量 **a** 中。

### 1.11.2 指针与指针变量

如前所述，**变量的指针就是变量的地址**，用来存放另一个变量地址的变量称为**指针变量**。指针变量与所指向的变量之间的指向关系可以用图 1.43 来表示。

#### 1. 定义指针变量

定义指针变量的一般形式为：

**基类型 \* 指针变量名;**

如下面的定义语句：

```
int a, b;
```

```
int *pa, *pb;
```

第 2 行开头的 **int** 是指：所定义的指针变量 **pa** 和 **pb** 是**指向整型数据**的指针变量。也就是说，指针变量 **pa** 和 **pb** 只能用来指向整型数据（例如 **a** 和 **b**），而不能指向浮点型变量。这个 **int** 就是指针变量的**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。

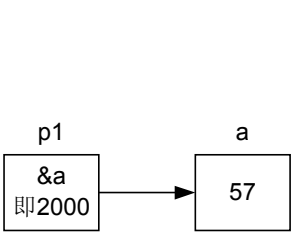


图1.43 指针变量与所指向的变量

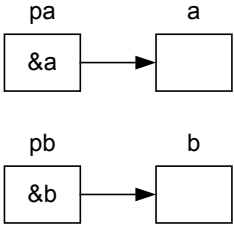


图1.44 pa指向a，pb指向b

注意定义指针变量时“**\***”号仅仅是说明定义的变量是指针变量，“**\***”号并不是指针变量名的一部分。另外，“**\***”号跟前面的基类型之间可以没有空格，跟后面的指针变量名之间也可以没有空格，如“**int\*pa;**”、“**int \*pa;**”、“**int\* pa;**”、“**int \* pa;**”等都是定义一个指针变量，变量名为

pa。但要注意区分下面两条定义语句：

- (1) `int* p1, p2;` //没有语法错误，但建议写成：`int *p1, p2;`
- (2) `int *p1, *p2;`

第(1)条语句定义了一个指针变量 `p1`，和一个整型变量 `p2`；第(2)条语句定义了两个指针变量 `p1` 和 `p2`。因此建议在定义指针变量时，“\*”号跟后面的指针变量名连在一起。

## 2. 使指针变量指向其他变量

定义好指针变量以后，要使指针变量指向另一个变量，只需要把被指向的变量的地址赋给指针变量即可。例如：

```
pa = &a;
pb = &b;
```

这样 `pa` 就指向了变量 `a`，`pb` 就指向了变量 `b`。可以用图 1.44 来表示。再强调一遍，这种指向关系是通过在指针变量里保存另一个变量的地址来实现的。

在定义指针变量时要注意：

- (1) 不能用一个整数给一个指针变量赋初值。如下面的语句是错误的：

```
int *pa = 2000;
```

- (2) 在定义指针变量时必须指定基类型。

另外，我们知道，在 32 位机器中，一个 `int` 型变量占 4 个字节，一个 `char` 型变量占 1 个字节，等等。那么指针变量占几个字节呢？答案是，指针变量占 4 个字节。为什么呢？这是因为在 32 位机器中，地址是 32 位的。1 个 `int` 型变量的地址是 32 位的，1 个 `char` 型变量的地址也是 32 位的。要存储地址信息，必须用 4 个字节，所以指针变量都是占 4 个字节的。

举个通俗的例子，宾馆有很多种房间，每种房间的大小不一样（对应于 32 机器中各种类型的变量所占的存储空间大小不一样）；但各种房间的钥匙（对应于各种类型变量所占存储空间的地址）型号是一样的，所以存放钥匙的盒子大小（对应于指针变量所占的存储空间大小）也是一样的。

## 3. 引用指针变量

有两个与指针变量有关的运算符：

- (1) `&` 取地址运算符。

(2) `*` 指针运算符（或称间接访问运算符）。如果要引用指针变量所指向的变量，需要使用指针运算符。

`&a` 为变量 `a` 的地址，`*pa` 为指针变量 `pa` 所指向的存储单元（即所指向的变量）。如：

```
int a, *pa;
pa = &a;
*pa = 3 + 5; //相当于 a = 3 + 5
```

请注意以下两点事项：

(1) 注意区分定义指针变量时的“\*”号及引用指针变量时的“\*”号。定义指针变量 `pa` 时，“\*”号只是用来说明变量 `pa` 是一个指针变量；而引用指针变量 `pa` 时，要用“`*pa`”这种形式，这里的“\*”号实际上是一个运算符，表示通过指针变量 `pa` 去引用它所指向的变量，即变量 `a`。

(2) 定义好指针变量后，要及时给指针变量赋值，如果一个指针变量没有赋值，它的值是未知的，即它指向哪段存储空间也是未知的，从而通过指针变量去存取它所指向的存储空间是很危险的。如下面的例子：

```
int *pa; //定义指针变量 pa，但 pa 指向哪个变量并不知道
*pa = 5; //把常量 5 赋值给指针变量 pa 所指向的存储空间，是很危险的
```

通常，如果一个指针变量暂时不会指向某个变量，可以将该指针变量赋值为 `NULL`。如：

```
pa = NULL;
```

`NULL` 的值实际为 0，`NULL` 也称为空指针。在内存中，地址为 0 的字节不被任何程序使用。所

以，即使通过指针变量 `pa` 对该字节进行赋值也不会造成什么影响。

下面通过例 1.46 和例 1.47 介绍指针及指针变量的使用方法。

**例 1.46** 同一个指针变量在不同时刻指向不同的变量。

代码如下：

```
#include <stdio.h>
void main( )
{
    int a, b;
    int *p1;
    scanf( "%d%d", &a, &b );
    p1 = &a;    //(1)p1 指向 a
    printf( "a = %d", *p1 );
    p1 = &b;    //(2)p1 指向 b
    printf( ", b = %d\n", *p1 );
}
```

该程序的运行示例如下：

5 7 ✓

a = 5, b = 7

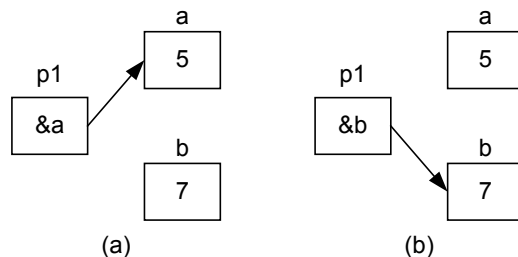


图1.45 同一个指针变量在不同时刻指向不同的变量

上面的程序在执行完语句(1)后，指针变量 `p1` 指向变量 `a`，如图 1.45(a)所示。执行完语句(2)后，指针变量 `p1` 指向变量 `b`，如图 1.45(b)所示。因此，指针变量可以在不同时刻指向不同的变量。 □

**例 1.47** 交换两个数，用指针变量实现。

前面在例 1.4 和例 1.7 介绍了两种交换两个变量 `a` 和 `b` 的方法，现在再介绍第 3 种方法：使用指针变量实现。代码如下：

```
#include <stdio.h>
void main( )
{
    int *p1, *p2;
    int t, a, b;
    scanf( "%d %d", &a, &b ); //输入两个整数
    p1 = &a; //使 p1 指向 a
    p2 = &b; //使 p2 指向 b
    t = *p1;    //(1)
    *p1 = *p2;  //(2)
    *p2 = t;    //(3)
    printf( "a=%d, b=%d\n", a, b );
}
```

该程序的运行示例如下：

57 43 ✓

a=43, b=57

如图 1.46 所示。初始时指针变量 `p1` 指向变量 `a`，指针变量 `p2` 指向变量 `b`。程序中语句(1)、(2)和(3)实现了交换两个变量 `*p1` 和 `*p2`，即变量 `a` 和 `b`，因此执行完后，变量 `a` 的值变为 43，变量 `b` 的值变为 57。 □

下面的程序在输入“57 43”时，输出来的也是“a=43, b=57”，但是交换的不是变量 `a` 和 `b`，而是指针变量 `p1` 和 `p2`。

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

```

{
    int *p1, *p2, *p;
    int a, b;
    scanf( "%d %d", &a, &b ); //输入两个整数
    p1 = &a; //使 p1 指向 a
    p2 = &b; //使 p2 指向 b
    p = p1; //(1)
    p1 = p2; //(2)
    p2 = p; //(3)
    printf( "a=%d, b=%d\n", *p1, *p2 );
}

```

程序中语句(1)、(2)和(3)交换的是两个指针变量 **p1** 和 **p2**。如图 1.47 所示。交换完以后，指针变量 **p1** 的值为变量 **b** 的地址，指针变量 **p2** 的值为变量 **a** 的地址，即指针变量 **p1** 指向变量 **b**，指针变量 **p2** 指向变量 **a**。变量 **a** 和 **b** 的值本身没有发生变化，在输出时，按顺序输出 **\*p1** 和 **\*p2**，即变量 **b** 和 **a**，因此输出结果也是 “a=43, b=57”。

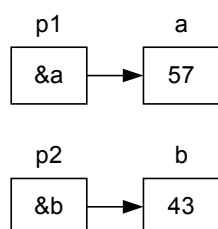


图1.46 交换变量a和变量b(通过指针变量实现)

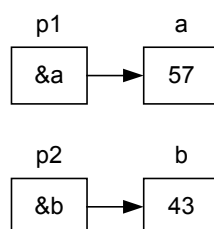
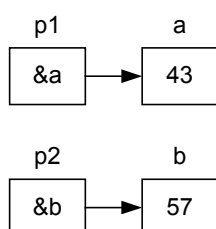
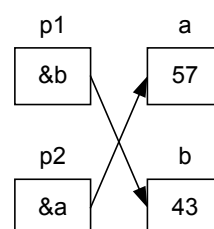


图1.47 交换指针变量pa和pb



### 1.11.3 指针变量作函数参数

指针变量的一个重要作用就是用作函数参数。在 1.9.3 提到用普通变量作函数形参，无法改变主调函数中实参变量的值。如果要采用函数的形式改变主调函数中变量的值，必须采用指针变量作函数参数。本节先通过一个简单的例子（例 1.48）演示指针的这个重要作用，然后在例 1.49 中用函数和指针参数实现交换主函数中两个变量的值。

**例 1.48** 指针变量作函数参数，改变主函数中变量的值。

代码如下：

```

#include <stdio.h>
void f( int* p )
{
    (*p)++;
}
void main( )
{
    int a = 0, i;
    int *pa = &a;
    for( i=1; i<=10; i++ )
    {
        f( &a );
        printf( "%d ", a );
    }
}

```

该程序的输出结果如下：

1 2 3 4 5 6 7 8 9 10

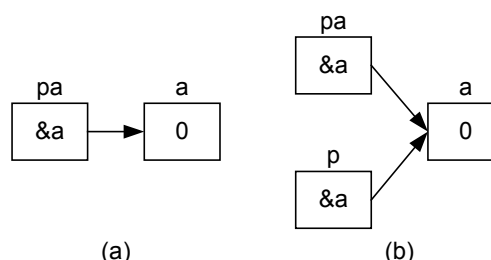


图1.48 指针变量作函数参数，使得实参和形参指向同一个变量

在上面的程序中，main 函数的 for 循环每执行一次，变量 a 的值将递增 1，而变量 a 的值递增 1 的操作是在 f 函数中实现的。main 函数中指针变量 pa 指向变量 a，如图 1.48(a)所示，在 for 循环中调用 f 函数时，将实参 pa 的值传递给形参 p，因此 p 也指向 a，如图 1.48(b)所示，因此 “(\*p)++” 将使得 a 的值递增 1。 □

**思考 1.19:** f 函数唯一一条语句中的表达式 “(\*p)++” 能否改成 “\*p++”？即去掉括号。

**思考 1.20:** 在 f 函数能否通过语句 “a++;” 来使得 a 的值递增 1？

**例 1.49** 交换两个数，用函数实现，其中实参和形参均为指针变量。

代码如下：

```
#include <stdio.h>
void main( )
{
    void swap( int *p1, int *p2 );    //函数声明
    int *pa, *pb, a, b; //定义指针变量 pa,pb，整型变量 a,b
    scanf( "%d%d", &a, &b );
    pa = &a;    //使 pa 指向 a
    pb = &b;    //使 pb 指向 b
    swap( pa, pb );    //使*pa 和*pb 互换
    printf( "a=%d, b=%d\n", a, b );//a,b 已交换
}
void swap( int *p1, int *p2 )//函数的作用是将*p1 的值与*p2 的值交换
{
    int temp;
    temp = *p1; //(1)
    *p1 = *p2;    //(2)
    *p2 = temp;    //(3)
}
```

该程序的运行示例如下：

97 65 ✓

a=65, b=97

该程序的运行过程及分析如下：

(a) 在主函数里，指针变量 pa 指向变量 a，指针变量 pb 指向变量 b，如图 1.49(a)所示。

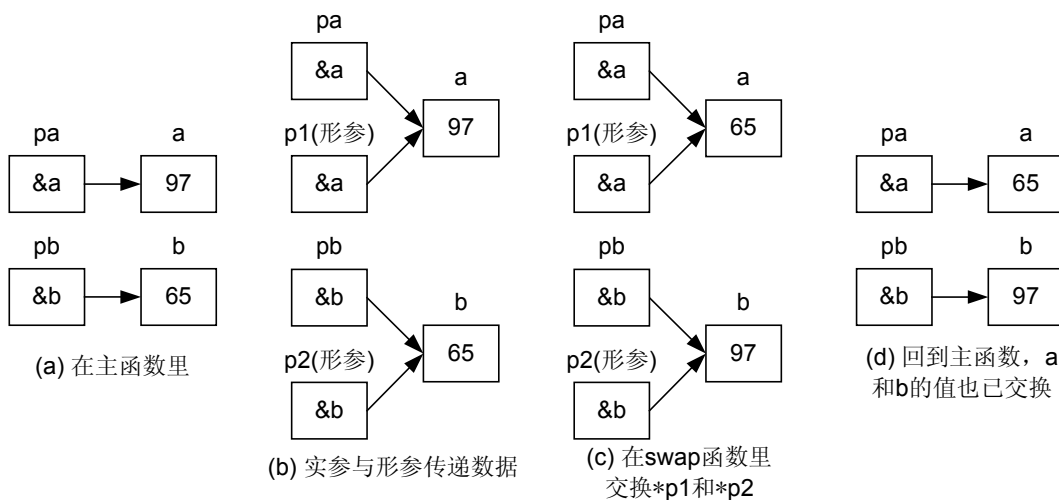


图1.49 交换两个数，用函数实现，参数为指针变量

(b) 调用 `swap` 函数时，实参的值传递给形参，即实参 `pa` 的值传递给形参 `p1`，因此指针变量 `p1` 的值也为变量 `a` 的地址，即 `p1` 也指向了变量 `a`，同样的道理，指针变量 `p2` 指向了变量 `b`，如图 1.49(b)所示。

(c) 在执行 `swap` 函数时，交换了 `*p1` 和 `*p2` 的值，即交换了变量 `a` 和 `b` 的值，如图 1.49(c)所示。

(d) `swap` 函数执行完毕，回到主函数里，变量 `a` 和 `b` 的值已经交换了，如图 1.49(d)所示。

分析到这里，读者可能有一点疑问：1.9.3 曾提到，实参变量对形参变量的数据传递是“值传递”，即单向传递，只由实参传给形参，而不能由形参传回来给实参，用指针变量作函数参数是不是违反了这一规则。其实用指针变量作函数参数，也是遵守这一规则的。如图 1.49(b)所示，实参 `pa` 的值传递给形参 `p1`，传递之后，`pa` 和 `p1` 也是没有任何关系了，如果在 `swap` 函数中改变了形参 `p1` 的值，并不影响实参 `pa`（详见以下对例 1.49 程序的第 1 次改动）。例 1.49 的程序在 `swap` 函数交换的是形参 `p1` 和 `p2` 所指向的变量。□

**思考 1.21：**例 1.49 程序的 `main` 函数中，指针变量 `pa` 和 `pb` 能不能省略不用，直接采取“`swap( &a, &b )`”这种形式调用 `swap` 函数？

为了进一步理解例 1.49 的程序，下面对例 1.49 的程序作两次改动，再分析为什么两次改动都不能实现交换主函数中的变量 `a` 和 `b`。

(1) `swap` 函数的形参也为指针变量，但在 `swap` 函数里交换的是形参指针变量，而不是形参指针变量所指向的变量。

```
#include <stdio.h>
void main( )
{
    void swap( int *p1, int *p2 );    //函数声明
    int *pa, *pb, a, b; //定义指针变量 pa,pb, 整型变量 a,b
    scanf( "%d%d", &a, &b );
    pa = &a;    //使 pa 指向 a
    pb = &b;    //使 pb 指向 b
    swap( pa, pb );
    printf( "a=%d, b=%d\n", a, b );
}
void swap( int *p1, int *p2 )
{
    int *temp;
    temp = p1;    //(1)
    p1 = p2;    //(2)
    p2 = temp;    //(3)
}
```

程序的运行示例如下：

97 65✓

a=97, b=65

程序的运行过程如图 1.50 所示，该程序并没有实现交换 `a` 和 `b` 的值。对比图 1.49 就发现，问题出现图(c)，即 `swap` 函数里的 3 条语句交换的是指针变量的值，交换完毕后，指针变量 `p1` 指向变量 `b`，指针变量 `p2` 指向变量 `a`，而变量 `a` 和 `b` 的值是没有交换的（请特别注意，形参 `p1` 的值改变后，并不影响对应的实参 `pa`，`pa` 仍然是指向变量 `a` 的）。回到主函数，输出 `a` 和 `b` 的值，同样也是没有改变的。

(2) 将 `swap` 函数的形参改为普通变量

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

```

void main( )
{
    void swap( int x, int y );
    int a, b;
    scanf( "%d%d", &a, &b );
    swap( a, b );
    printf( "a=%d, b=%d\n", a, b );
}

void swap( int x, int y )
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

```

程序的运行示例如下：

97 65✓

a=97, b=65

这个程序不能实现交换主函数中变量 **a** 和 **b** 的值，还是因为实参跟形参之间的数据传递是“单向”的，形参值的改变并不影响实参。

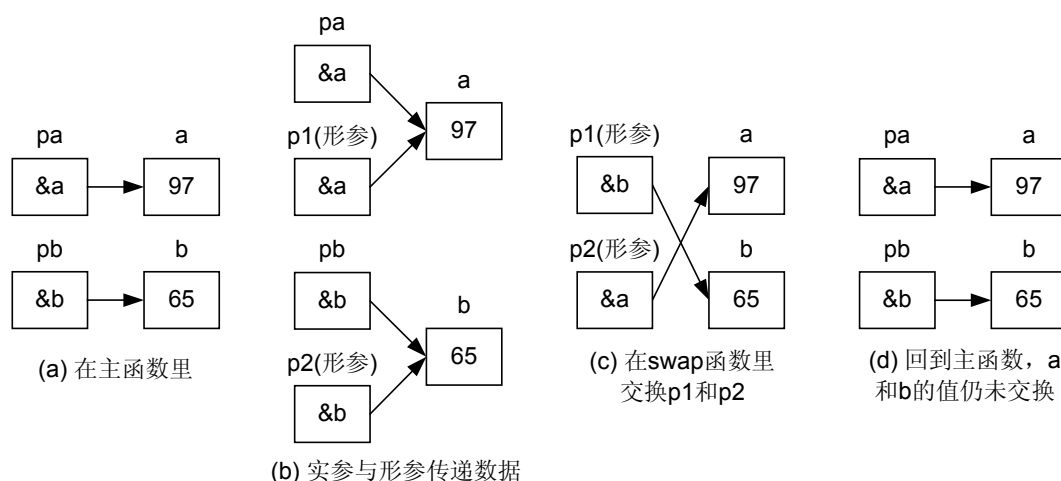


图1.50 在swap函数里交换两个指针变量p1和p2的值

#### 1.11.4 数组与指针变量

在 1.10.1 中提到过，数组名是代表整个数组起始地址的常量，因此数组和指针之间有紧密的联系。指针变量的第二个重要作用是：使指针变量指向数组首地址，从而可以很方便地遍历数组元素。

##### 1. 指向数组元素的指针变量

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（把某一元素的地址存放在一个指针变量中）。数组元素的地址也就是数组元素的指针。如下面的例子：

```

int a[10];    //定义一个整型数组 a，它有 10 个元素
int *p;       //定义一个基类型为整型的指针变量 p
p = &a[0];    //将元素 a[0]的地址赋给指针变量 p，使 p 指向 a[0]

```

在 C/C++ 中，数组名代表数组中第 0 个元素的地址。因此，下面两个语句等价：



```
p = &a[0];
```

```
p = a;
```

说明:

(1) 既然指针变量可以指向数组元素，那么就可以通过指针变量来引用数组元素。如:

```
int *p = &a[5];    //指针变量 p 指向数组元素 a[5]
```

```
*p = 25;          //这条语句等效于 a[5] = 25;
```

(2) 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。

另外，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，它们指向数组  $a$  的第  $i$  个元素，图 1.51 所示。

(2)  $*(p+i)$  或  $*(a+i)$  是  $p+i$  或  $a+i$  所指向的数组元素，即  $a[i]$ 。

(3) 指向数组元素的指针变量也可以带下标，如  $p[i]$  与  $*(p+i)$  等价。

因此，如果指针变量  $p$  指向数组  $a$  第 0 个元素  $a[0]$ ，则以下 4 个式子等价:

$$a[i] \Leftrightarrow p[i] \Leftrightarrow *(p+i) \Leftrightarrow *(a+i)$$

根据以上叙述，引用一个数组元素，可以采用以下方法:

(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。如果已经给  $p$  赋值为  $a$ ，则  $*(p+i)$  就是  $a[i]$ 。可以通过指向数组元素的指针变量找到所需的元素。

另外，用指针法访问数组元素的效率比下标法的效率更高，因为下标法中的“ $[]$ ”实际上是运算符，通过运算求得出  $a[i]$  的地址，再访问  $a[i]$ ；而指针法中的加法运算“ $p+i$ ”可以更快地求出  $a[i]$  的地址。

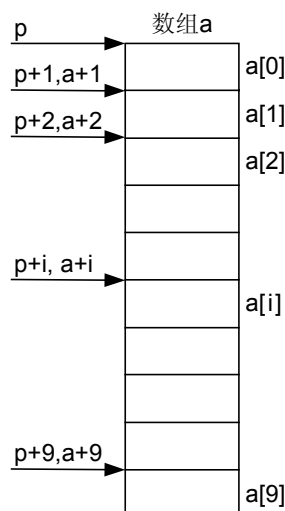


图1.51 指向数组元素的指针变量

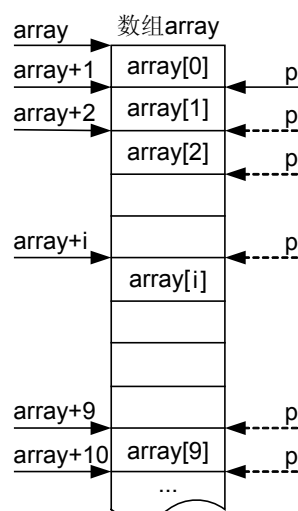


图1.52 通过修改指针变量的值，实现遍历数组

**例 1.50** 求一维数组的 10 个元素中的最大值。

这道题目需要遍历一维数组，遍历时可以采用下标法或指针法访问每个数组元素。其中采用下标法的代码如下:

```
#include <stdio.h>
void main( )
{
    int array[10] = { 12, 7, -89, 120, 55, 79, 3, 11, 66, -45 };
    int max = array[0];    //max 初始化为第 0 个元素
    for( int i=1; i<10; i++ )
    {
        if( array[i] > max )//如果 a[i]比当前最大值 max 还大
```

```

        max = array[i];    //将 max 更新为 a[i]
    }
    printf("the maximum element is : %d\n", max);
}

```

该程序的输出结果为：

the maximum element is : 120

**思考 1.22：** 请将上面的程序修改成指针法。

另外，还可以专门定义一个指针变量 *p* 指向数组元素，初始时 *p* 指向第 0 个元素，然后在 *for* 循环中每次循环后使得 *p* 自增 1，这样 *p* 指向下一个元素，通过这种方式也可以遍历到数组里每个元素。代码如下：

```

#include <stdio.h>
void main( )
{
    int array[10] = { 12, 7, -89, 120, 55, 79, 3, 11, 66, -45 };
    int max = array[0];
    int *p;    //定义指针变量 p，并在 for 循环中首先赋值为数组首地址
    for( p=array; p<(array+10); p++ ) //p++会使得指针变量 p 指向下一个数组元素
    {
        if( (*p) > max )    //如果 a[i]比当前最大值 m 还大
            max = (*p);    //将 max 更新为 a[i]
    }
    printf("the maximum element is : %d\n", max);
}

```

在上面的程序中，指针变量 *p* 初始时指向 *array[0]*，然后在 *for* 循环中不断修改 *p* 的值，使得指针变量 *p* 指向数组中的每个元素。另外请特别注意，该 *for* 循环的循环条件中，*array+10* 表示最后一个元素 *array[9]* 后面的第 1 个字节的地址。当指针变量 *p* 指向到 *array[9]* 后再加 1，则 *p* 的值实际为 *array+10*，这时循环条件不再成立，循环结束。整个过程如图 1.52 所示。 □

## 2. 用指针变量作函数参数接收数组地址

在 1.10.3 节中介绍过用数组名作函数的参数，传递的是数组首地址。其实用指针变量作函数形参，同样可以接收从实参传递来的数组首地址（实际上，即使是数组名的形式，编译器也是按指针变量处理的），如例 1.51。

**例 1.51** 设计函数 *sub*，实现两个一维数组对应元素相减。在主函数中分别输入 10 个整数到数组 *a* 和 *b* 中，调用 *sub* 函数分别实现 *a - b* 和 *b - a*。

分析：在本题中，*main* 函数需要定义两个一维整型数组用于接收从键盘输入的 20 个整数，在调用 *sub* 函数时，需要将数组名 *a* 和 *b* 作为实参传递给 *sub* 函数中的形参。在例 1.43 中是用数组名的形式做函数形参，在本题中采用指针形式做函数形参。

代码如下：

```

#include <stdio.h>
void sub( int *p1, int* p2, int n )
{
    for( int i=0; i<n; i++, p1++, p2++ )    //指针变量递增，指向不同的数组元素
    {
        printf( "%d ", (*p1) - (*p2) );
    }
}
void main( )

```

```

{
    int a[10], b[10], i;
    printf( "Please input 10 numbers for array a : " );
    for( i=0; i<10; i++ )
        scanf( "%d", &a[i] );
    printf( "Please input 10 numbers for array b : " );
    for( i=0; i<10; i++ )
        scanf( "%d", &b[i] );
    printf( "the results of a - b are : " );
    sub( a, b, 10 );    //调用 sub 函数实现 a-b
    printf( "\n" );
    printf( "the results of b - a are : " );
    sub( b, a, 10 );    //调用 sub 函数实现 b-a
}

```

该程序的运行示例如下：

Please input 10 numbers for array a : 28 79 -13 17 -22 6 55 -98 1 25 ✓

Please input 10 numbers for array b : 53 -1 62 79 81 -57 26 -25 -3 9 ✓

The results of a - b are : -25 80 -75 -62 -103 63 29 -73 4 16

The results of b - a are : 25 -80 75 62 103 -63 -29 73 -4 -16

如图 1.53 所示，当第 1 次调用 sub 函数时，第 1 个实参（数组名 a）传递给指针形参 p1，因此 p1 的值为数组 a 的首地址，即 p1 指向数组 a；同样的道理，p2 指向数组 b。因此初始时，(\*p1)和(\*p2)分别是 a[0]和 b[0]。在 for 循环执行过程中，随着 p1 和 p2 的不断递增，p1 和 p2 分别指向数组 a 和数组 b 中的每个元素，从而实现数组 a 和数组 b 对应元素相减。

当第 2 次调用 sub 函数时，p1 指向数组 b，p2 指向数组 a，从而实现数组 b 和数组 a 对应元素相减，如图 1.53 所示。 □

**思考 1.23：**在例 1.51 的程序的 sub 函数中，如果要求不修改形参 p1 和 p2 的值，则 sub 函数该如何修改？

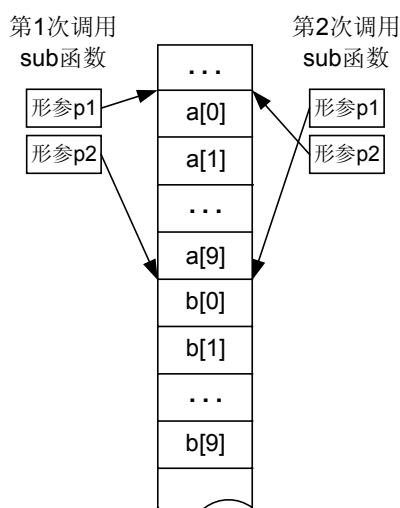


图1.53 用指针变量作函数参数接收数组地址

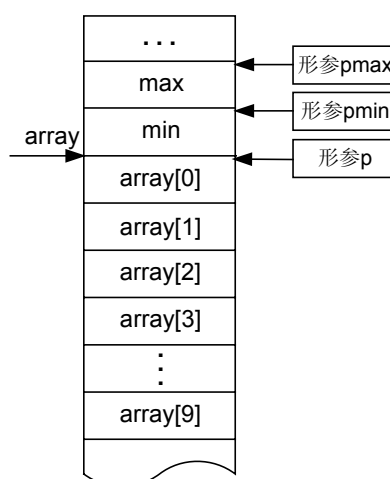


图1.54 用指针变量作函数参数接收数组地址和变量地址

### 1.11.5 编写指针应用的综合程序

本节讲解一道综合应用函数、数组和指针知识的程序。

**例 1.52** 求一维数组 10 个元素中的最大值和最小值，要求用函数实现。

本节开头提到了，使用指针变量作函数参数，可以得到多于一个的函数“返回”值，而如果不使用指针变量作函数参数，被调函数最多只能通过 `return` 语句向主调函数返回一个值。

在本例中，被调函数需要向主调函数“返回”2 个值：最大值和最小值。自定义函数 `mmfun` 用于求一维数组 `a` 的 `n` 个元素中的最大值和最小值，以指针变量参数的形式将这两个结果“带回”到主函数中，并在主函数中进行输出。代码如下：

```
#include <stdio.h>
void main( )
{
    void mmfun( int *p, int n, int *pmax, int *pmin );    //函数声明
    int array[10] = { 12, 7, -89, 120, 55, 79, 3, 11, 66, -45 };
    int max, min; //最大值及最小值
    mmfun( array, 10, &max, &min );
    printf("the maximum element is %d\n", max);
    printf("the minimum element is %d\n", min);
}
void mmfun( int *p, int n, int *pmax, int *pmin )
{
    int i; //循环变量
    *pmax = *pmin = *p; //首先假定最大的数和最小的数就是第 0 个元素
    for( i=0; i<n; i++, p++ )
    {
        if( *p > *pmax )//如果*p 比当前最大值还大，则将 max 更新为*p
            *pmax = *p;
        if( *p < *pmin )//如果*p 比当前最小值还小，则将 min 更新为*p
            *pmin = *p;
    }
}
```

该程序的输出结果为：

the maximum element is 120

the minimum element is -89

这里着重解释 `mmfun` 函数的 4 个形参：

**int \*p:** 指针变量作函数形参，在函数调用时接收的是数组 `array` 的首地址。这样，形参 `p` 初始时指向的是数组 `array` 的第 0 个元素，如图 1.54 所示。在 `mmfun` 函数中的 `for` 循环中，每执行一次循环，`p` 的值自增 1，这样使得 `p` 能够指向数组 `array` 中的每个元素。

**int n:** 表示数组元素的个数，该参数与例 1.43 中 `reverse` 函数的形参 `n` 的作用类似。

**int \*pmax:** 指针变量作函数形参，在函数调用时它接收的是变量 `max` 的地址，这样指针变量 `pmax` 指向变量 `max`，如图 1.54 所示。并且指针变量 `pmax` 的值在 `mmfun` 函数中保持不变，即指针变量 `pmax` 始终指向变量 `max`。在 `mmfun` 函数中，`*pmax` 实际上就是主函数中的变量 `max`。这样在 `mmfun` 函数中求得的最大值可以“带回”到主函数中。

**int \*pmin:** 与形参 `pmax` 类似，其作用是：通过该指针变量可以修改主函数中变量 `min` 的值，从而将求得的最小值“带回”到主函数中。

`mmfun` 函数正是通过两个指针变量形参 `pmax` 和 `pmin`，将求得的两个结果“带回”到主函数中的。 □

## 练习

1.59 分析下面程序的输出。然后运行程序，对比程序的输出，看看跟你自己的分析是否吻合。

```

#include <stdio.h>
void main( )
{
    int *p, a=10, b=1;
    p=&a;
    (*p)++;
    a=*p+b;
    p=&b;
    *p=*p+a;
    printf("%d %d\n", a, b);
}

```

- 1.60 分析下面程序的输出。然后运行程序，对比程序的输出，看看跟你自己的分析是否吻合。

```

#include <stdio.h>
void fun( int *c, int d )
{
    *c=*c+1; d=d+1;
    printf( "%d %d\n", *c, d );
}
void main( )
{
    int a = 65, b = 97;
    fun( &a, b );
    printf( "%d %d\n", a, b );
}

```

- 1.61 分析下面程序的输出。然后运行程序，对比程序的输出，看看跟你自己的分析是否吻合。

```

#include <stdio.h>
void f1(int x, int y)
{
    int t; t=x; x=y; y=t;
}
void f2( int *x, int *y )
{
    int t; t=*x; *x=*y; *y=t;
}
void main( )
{
    int i, x1, x2;
    int a[5] = { 1, 2, 3, 4, 5 };
    x1 = x2 = 0;
    for( i=1; i<5; i++ )
    {
        if( a[i]>a[x1] ) x1=i;
        if( a[i]<a[x2] ) x2=i;
    }
    f2( &a[x1], &a[0] );
    for( i=0; i<5; i++ ) printf( "%2d", a[i] );
    printf( "\n" );
    f1( a[x2], a[1] );
    for( i=0; i<5; i++ ) printf( "%2d", a[i] );
}

```

```

    printf( "\n" );
    f2( &a[x2], &a[4] );
    for( i=0; i<5; i++ ) printf( "%2d", a[i] );
    printf( "\n" );
    f1( a[x1], a[3] );
    for( i=0; i<5; i++ ) printf( "%2d", a[i] );
    printf( "\n" );
}

```

1.62 编程实现：从键盘上 **a**、**b**、**c** 三个整数，按由大到小的顺序输出。要求在交换两个整数时调用 **swap** 函数，该函数采取指针参数的形式实现交换主调函数中两个变量的值。

1.63 使用指针法实现：将以下 10 个数存放到一个数组中。然后从键盘上输入一个数，查找该数是数组中第几个元素，如果没有查找到，则输出“无此数”信息。

12, 7, -89, 120, 55, 79, 3, 11, 66, -45

1.64 编程实现：分别累加一个一维数组中正数和、负数和。一维数组元素值从键盘输入，在累加时通过一个指针变量访问数组的每个元素。

1.65 设计一个函数 **count**，统计一个一维数组中第 **s**~**t** 个元素的和，要求在 **count** 函数中采用指针形式访问数组元素。在 **main** 函数中用一维数组存储从键盘上输入的 20 个整数，三次调用 **count** 函数求不同元素段的和。**count** 函数的原型为：

```
void count( int a[], int n, int s, int t );
```

1.66 设计一个函数 **move**，将一个一维数组前面各数后移 **m** 个位置，最后 **m** 个数变成最前面 **m** 个数，如图所示 1.55 所示，要求在 **move** 函数中采用指针形式访问数组元素。在

**main** 函数中用一维数组存储从键盘上输入的 20 个整数，调用 **move** 函数移动数组中的元素。**move** 函数的原型为：

```
void move( int a[], int n, int m );
```

1.67 统计一维数组的 10 个元素中，正数、负数、0 分别有多少个：在主函数中从键盘上输入 10 个数，存放到一个数组中，然后调用 **count** 函数进行统计。**count** 函数的原型为：

```
void count( int a[], int n, int *ppositive, int *pzzero, int *pnegative );
```

即要求以指针变量形参的方式将统计的结果带回主函数中。

程序的运行示例如下：

12 7 -89 120 55 79 3 11 66 -45 ✓

positive=8, zero=0, negative=2

1.68 编程实现：分别统计一个  $3 \times 4$  二维数组各元素中偶数和与奇数和。二维数组各元素的值在主函数中从键盘输入。定义函数 **sum** 累加二维数组中偶数和与奇数和。**sum** 函数的原型为：

```
void sum( int a[ ][4], int *peven, int *podd );
```

即要求以指针变量形参的方式将累加的和带回主函数中。

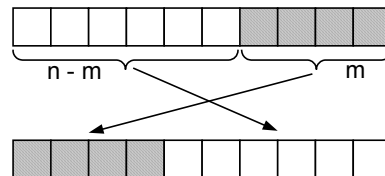


图1.55 移动数组元素

## 第2章 字符及字符串基础知识

字符及字符串的基础知识包括字符型数据、字符数组、字符串常量和字符指针变量，这些内容有联系，也有区别。本章将集中介绍这些知识。

### 2.1 字符型数据

在程序里，字符型数据同样是以变量和常量两种形式存在的。

#### 2.1.1 字符型变量

C/C++语言提供了2种字符型类型：有符号的字符型和无符号的字符型。

**char**：有符号的字符型。其表示范围是-128 ~ +127，占1个字节。

**unsigned char**：无符号的字符型。其表示范围是0 ~ 255，占1个字节。

如果在程序里定义了一个字符型变量 **ch**，它的值是大写字母字符'A'，实际上并不是将该字符本身存放到内存单元中，而是将该字符的 ASCII 编码以二进制形式存放到存储单元中。从附录 E 可知，字符'A'的 ASCII 编码值为 65，对应的二进制为 01000001。因此该字符型变量在内存中占一个字节，假设其地址为 2000，它的内容为“01000001”。如图 2.1 所示。

既然字符型数据是以 ASCII 编码值存储的，它的存储形式就与整数的存储形式类似。因此，在 C/C++语言中，字符型数据和整型数据之间可以通用。一个字符型数据可以赋给一个整型变量，反之，一个整型数据也可以赋给一个字符型变量（截取整型数据的最低字节赋值给字符型变量）。也可以对字符型数据进行算术运算，此时相当于以它们的 ASCII 编码值进行算术运算。如例 2.1。

**例 2.1** 大小写字母字符转换。

```
#include <stdio.h>
void main( )
{
    char c1, c2; //定义两个字符型变量
    c1 = 97;      //把整型常量赋值给字符型变量
    c2 = 98;
    c1 = c1 - 32; //(1)字符型数据与整型数据一起运算
    c2 = c2 - 32; //(2)
    printf( "%c %c\n", c1, c2 ); //输出转换后的字符
}
```

该程序的输出结果如下：

A B

从附录 E 的 ASCII 编码表可知，大写字母在 ASCII 编码表中是按顺序排列的，小写字母也是按顺序排列的，而且，每一个小写字母比相应的大写字母的 ASCII 编码值大 32。ASCII 编码值为 97 的是小写字母字符'a'，编码值为 98 的是小写字母字符'b'，依此类推。执行完程序中的语句(1)和(2)以后，c1 的值为 65，对应的字符是大写字母字符'A'，c2 的值为 66，对应的字符是大写字母字符'B'。□

**思考 2.1：**在上述代码中，如果把 printf 语句中的格式控制符改成“%d”，程序将会输出什么？

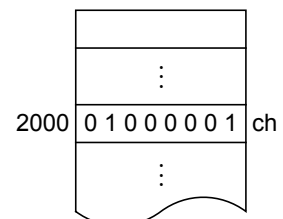


图2.1 字符型变量在内存中的存储形式

### 2.1.2 字符型常量

在给字符型变量赋值时，以及在一些涉及字符运算的表达式里，往往要用到字符型常量。用单撇号括起来的一个字符就是**字符型常量**。如'a'、'#'、'%','D'都是合法的字符型常量，在内存中占一个字节。字符常量可以赋给一个 char 型变量或者整型变量。如：

```
char c = 'D';
int a = 'A';    //变量 a 的值为 65，即字符'A'的 ASCII 编码值
```

除了以上形式的字符常量外，C/C++还允许使用一种特殊形式的字符型常量，就是以右斜杆“\”开头的字符序列，这些字符型常量称为**转义字符**，表示将右斜杆(\)后面的字符转换成另外的意义。例如，'\n'代表一个“换行”字符。

表 2.1 列出了 C/C++语言中提供的所有转义字符及其含义。

表 2.1 转义字符及其含义

| 字符形式 | 含义                       | ASCII 编码 |
|------|--------------------------|----------|
| \a   | 响铃                       | 7        |
| \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       |
| \v   | 竖向跳格                     | 8        |
| \\   | 反斜杆字符“\”                 | 92       |
| \'   | 单引号(撇号)字符                | 39       |
| \"   | 双引号字符                    | 34       |
| \0   | 空字符(字符串结束标志)             | 0        |
| \ddd | 1-3 位八进制 ASCII 编码所代表的字符  |          |
| \xhh | 1-2 位十六进制 ASCII 编码所代表的字符 |          |

表 2.1 最后两行要特别注意。有了这两行就意味着在程序中要表示一个字符常量'a'，可以采用'a'，还可以用'\141'或'\x61'，因为字符 a 的 ASCII 编码为 97，其八进制形式为“141”，十六进制形式为“61”。但是要注意不能采用十进制形式'\97'去表示字符常量'a'。

**例 2.2** 转义字符的使用。

```
#include <stdio.h>
void main( )
{
    char c1 = 'a', c2 = 'b', c3 = 'c', c4 = '\101', c5 = '\116', c6 = '\x42';
    printf( "%c%c%c\n", c1, c2, c3 );
    printf( "\t%c%c\t%c\n", c4, c5, c6 );
}
```

该程序的输出结果如下：

```
abc
      AN      B
```

其中第 2 行字符“A”之前有 8 个空格的宽度，字符“N”和字符“B”之间有 6 个空格的宽度。 □

要理解为什么有这么多空格数，需要理解水平制表位。

关于水平制表位的说明：

- 1) 制表位是 DOS 操作系统延续下来的概念，用于绘制表格时的对齐操作。
- 2) 一般来说，一个水平制表位表示 8 个空格的宽度，但在有的系统（如 Visual C++ 6.0



编辑器)中,一个水平制表位也可能表示4个空格的宽度。

3) 在键盘上有一个“tab”键,可以输入水平制表位符号。

例如,假设这里一个水平制表位表示8个空格的宽度,下面两行中字符a是第一列,字符b是第二列,字符c是第三列:

```
aaaaabbbbcccc
```

```
aaabc
```

要使得两行中每一列都对齐,可以在字符a和字符b之间,字符b和字符c之间按下tab键。结果如下:

```
aaaaa  bbbbb  ccc
```

```
aaa    b      c
```

第一行中字符a和字符b之间有3个空格的宽度,因为在第一个水平制表位中已经有5个字符a了,需要再跳过3个空格的宽度,才能到下一个水平制表位。第二行中字符a和字符b之间有5个空格的宽度,因为在第一个水平制表位中已经有3个a了,需要再跳过5个空格的宽度,才能到下一个水平制表位。

### 2.1.3 字符型数据的输入/输出

在C/C++语言中除了可以使用scanf和printf函数对字符型数据进行输入/输出(使用%c格式控制符)外,还可以使用getchar函数输入一个字符,使用putchar函数输出一个字符。

getchar函数的作用是从终端(输入设备,通常为键盘)读入一个字符。这个函数没有参数;返回值为读入字符的ASCII编码值。该函数的原型如下:

```
int getchar( void );
```

putchar函数的作用是向终端(输出设备,通常为显示器)输出一个字符。该函数有一个参数,含义是待输出字符(的ASCII编码值)。该函数也有返回值,其含义就是待输出字符的ASCII编码值。该函数的原型如下:

```
int putchar( int c );
```

putchar函数也可以输出转义字符,如例2.3。

**例 2.3** getchar 和 putchar 函数的使用。

```
#include <stdio.h>
void main( )
{
    char c1 = getchar( ); //读入一个字符,假设输入小写字母
    c1 = c1 - 32;          //转换成大写字母
    char c2 = putchar(c1); //用 putchar 输出字符,并把返回的字符赋值给 c2
    putchar( '\n' );
    putchar(c2); putchar( '\n' );
    putchar( "\117" ); putchar( "\113" ); putchar( '\n' ); //输出转义字符
}
```

该程序的运行示例如下:

```
a✓
A
A
OK
```

从例2.3的输出结果可以看出,putchar函数的返回值就是待输出字符的ASCII编码值。上述输出的第一行,即字符'A',是putchar(c1)输出出来的,把该函数调用的返回值赋值给字符型变量c2,因此c2的值也是字符'A';输出的第二行,也是字符'A',是putchar(c2)输出出来的。□

`getchar` 函数只能读入一个字符，但是如果把 `getchar` 函数放到循环里，也能实现读入一串字符。例如“要读入一串字符，直到按下回车键表示结束”，可以使用下面的结构：

```
char ch;
while( ( ch=getchar( ) ) != '\n' )
{
    ... //处理该字符 ch
}
```

上述循环的执行过程是：先从键盘上读入一个字符，将该字符赋值给 `ch`，然后判断 `ch` 是否为换行符 `'\n'`，如果是，循环就结束了；如果不是，则执行循环体。因此该结构可以读入多个字符，一直到按下回车键为止（按下回车键实际上输入了两个字符：“回车”字符和“换行”字符，ASCII 编码值分别为 13 和 10）。具体应用详见例 2.4。

**思考 2.2：**上述 `while` 循环中，充当条件判断的表达式 `“( ch=getchar( ) ) != '\n' ”` 中，最外围的一对圆括号能不能去掉？

**例 2.4** 使用 `getchar` 函数读入一串字符。

网络上一个经典的帖子：如果将字母 A 到 Z 分别编上 1 到 26 的分数（A = 1, B = 2, ..., Z = 26），你的知识（KNOWLEDGE）得到 96 分（11 + 14 + 15 + 23 + 12 + 5 + 4 + 7 + 5 = 96），你的努力（HARDWORK）也只得到 98 分（8 + 1 + 18 + 4 + 23 + 15 + 18 + 11 = 98），你的态度（ATTITUDE）才是左右你生命的全部（1 + 20 + 20 + 9 + 20 + 21 + 4 + 5 = 100）。现要求你编写一个程序，实现：从键盘输入任意一单词（假定该单词中只包含大写字母，不需判断），回车，计算并输出该单词的得分。

**分析：**使用上述循环结构读入每个字符 `c` 后，要得到它对应的得分，只需要减去 64 即可；再把每个字符的得分累加起来就是每个单词的得分。

```
#include <stdio.h>
void main( )
{
    int sum = 0;
    char c;
    for( ; (c=getchar( )) != '\n'; sum+=c-64 )
        ; //循环体为空语句
    printf( "sum=%d\n", sum );
}
```

该程序的运行示例如下：

```
HELLO✓
sum=52
```

**思考 2.3：**上述程序中，`for` 循环后的分号能否去掉？

**思考 2.4：**请将上述程序改由 `while` 循环实现。 □

## 练习

- 2.1 编程实现：从键盘上输入两个任意字符，把这两个字符的 ASCII 编码值加起来，并输出到屏幕上。
- 2.2 编程实现：从键盘上输入一个任意字符，如果该字符为小写字母字符，则转换成大写字母字符；如果该字符为大写字母字符，则转换成小写字母字符；如果是其他字符，则不转换；输出转换后的字符。
- 2.3 TOM 喜欢玩一种叠字母的游戏，游戏规则如下，从键盘上输入一个自然数 `N`（保证输入

的  $N$  满足  $1 \leq N \leq 13$ ，不需判断)，输出具有如下规律的图形，其中最后一行顶格输出，下面的图形是  $N=5$  时的情形。

AZ  
ABYZ  
ABCXYZ  
ABCDWXYZ  
ABCDEWXYZ

2.4 根据输入的正整数  $N$  ( $1 < N < 10$ ，不需判断) 的值，输出如下图所示的图形，该图形对应的  $N$  为 4。

      \*  
     \*\*\*\*\*  
     \*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
     \*\*\*\*\*  
     \*\*\*\*\*  
     \*

2.2 字符串常量

用双撇号括起来的字符序列就是字符串常量，如下面的 printf 函数调用语句中：

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

"Welcome!" 就是一个字符串常量，在输出时，对字符串常量，是把其中的字符原样输出来的。

如图 2.2 所示，编译系统在存储字符串常量时，将字符串常量存储在一段连续的存储空间中。每个字符占一个字节，存储的是字符的 ASCII 编码值（二进制形式），然后在字符串最后自动加上一个 '\0' 作为字符串的结束标志。因此 "Welcome!" 这个字符串常量在内存中占 9 个字节。

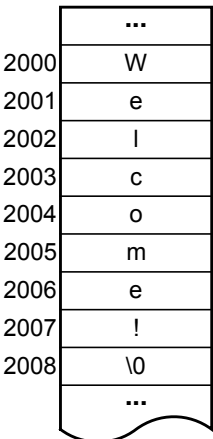


图2.2 字符串常量在内存中的存储情形

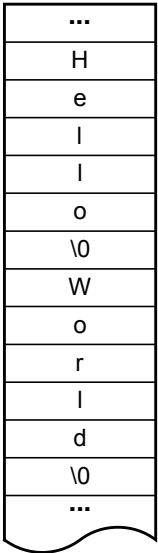


图2.3 字符串常量中包含多个字符串结束标志 '\0'

从附录 E 以及表 2.1 可知，转义字符 '\0' 所表示的字符，其 ASCII 编码值为 0，这个字符用作字符串常量的结束标志。在输出字符串常量时，输出每个字符直到 '\0' 为止，'\0' 这个字符不会输出出来。如果一个字符串常量中包含多于一个 '\0'，则在输出这个字符串常量时，只输出第一个 '\0' 之前的所有字符。例如在用下面的 printf 语句输出一个字符串常量时，只输出 “Hello”，该字符串常量的存储情形如图 2.3 所示。

```
printf("Hello\0world");
```

在字符串常量中也可以包含转义字符。例如字符串常量"abc\\n"包含了 5 个字符：'a'、'b'、'c'、'\\"和'\n'，其中后两个字符就是转义字符，分别表示右斜杆和换行符；这个字符串常量占 6 个字节，编译系统会在字符串常量后自动加上字符串结束标志'\0'。

**例 2.5** 输出字符串。

以下程序中每一个 printf 语句都输出了一个字符串。

```
#include <stdio.h>
void main( )
{
    printf( "Hello\0World!\n" );
    printf( "Hello\nWorld!\n" );
    printf( "Hello\tWorld!\n" );
    printf( "abc\\n" );
    printf( "Jack said : \"Hello World!\"\\n" );
}
```

该程序的输出结果如下：

HelloHello

World!

Hello    World!

abc\

Jack said : "Hello World!"

□

还有一点要注意，字符串常量所占的每个字节的内容都是不允许修改的。如果试图修改（修改方法见 2.4.2 节），编译可能没有错误，但运行时会出错。

## 练习

2.5 请分析以下 printf 语句的输出结果，并画出各字符串的存储情形：

```
printf( "a\\bre\\hi\\y\\b\\n" );
printf( "ab\\141\\142\\x61\\x62\\n" );
printf( "\\a\\n" );
printf( "\\\"Hello\\\"\\n" );
printf( "\\n\\n" );
```

## 2.3 字符数组

在第 1 章的 1.10 节介绍了整型的一维数组和二维数组，如果数组元素是字符型数据，则该数组就是字符数组。字符数组中的每个元素存放一个字符，占一个字节。字符数组具有数组的一般属性。字符数组和字符串联系是很紧密的，并且由于字符串应用广泛，C/C++ 专门为它提供了许多方便的用法和专门的处理函数。

### 2.3.1 字符数组的定义与初始化

字符数组的定义方法跟 1.10 节所描述的定义整型数组的方法完全一样。如：

```
char c[10], ch[3][4];
```

上面的语句分别定义了一个有 10 个字符元素的一维数组 c，以及有  $3 \times 4 = 12$  一共 12 个字符元素的二维数组。

字符数组有 2 种初始化方法：

- 1) 逐个元素赋值，这种初始化方法跟一般数组一样；
- 2) 用字符串常量初始化。

第 1 种方法的例子如下：

```
char c1[5] = { 'H', 'e', 'l', 'l', 'o' };
```

```
char c2[5] = { 'B', 'o', 'y' };
```

初始化后，c1 和 c2 的存储情况如图 2.4 所示。与整型数组初始化类似的是，如果在初始化时所提供的初值个数小于数组长度，对于没有初值的元素，编译器自动赋值为 0（注意，这些 0 在字符数组中有特殊含义，可以充当字符串结束标志）。因此字符数组 c2 中，后两个元素的值为 0。

注意在初始化时，所提供的初值个数不能大于数组长度，如下面的语句在编译时有错误。

```
char c3[4] = { 'H', 'e', 'l', 'l', 'o' }; // (×)
```

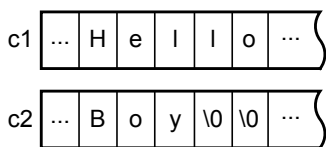


图2.4 字符数组初始化

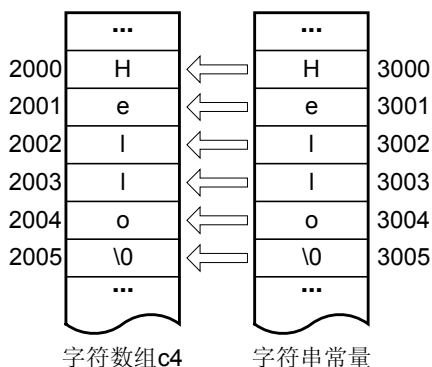


图2.5 用字符串常量初始化数组

第 2 种方法的例子如下：

```
char c4[6] = { "Hello" };
```

```
char c4[6] = "Hello";
```

```
char c4[ ] = "Hello";
```

以上 3 种初始化方法都是对的，而且是等效的。

在用字符串常量给字符数组初始化时，是把字符串常量的每一个字符赋值给字符数组中对应的元素。如图 2.5 所示。假设字符数组 c4 的起始地址是 2000，字符串常量的起始地址是 3000，则把地址为 3000 这个字节的内容赋值给地址为 2000 这个字节，依此类推。

上面在定义 c4 时，能不能将元素个数定义成 5？即：

```
char c4[5] = { "Hello" };
```

```
或：char c4[5] = "Hello";
```

答案是不可以的，因为字符串常量 "Hello" 实际上是占 6 个字节的，后面有串结束标志 '\0'。在初始化时，会把 '\0' 也赋值给某个数组元素。

### 2.3.2 字符数组元素的引用

和整型数组类似，只能引用字符数组中的某个元素，而不能引用整个数组。例如，假设已经定义好了两个字符数组 a 和 b：

```
char a[5], b[5];
```

下面两种用法都是错误的：

```
a = { 'C', 'h', 'i', 'n', 'a' }; // (×)
```

```
b = a; // (×)
```

引用某个数组元素是正确的，如：

```
a[0] = b[0] = 'A'; // 引用数组元素
```

### 2.3.3 字符数组的输入/输出

字符数组的输入/输出主要有三种方法：

1) 逐个字符输入/输出

在 `scanf` 和 `printf` 函数里，使用 “%c” 格式控制符可以对单个字符数据输入、输出，加上循环控制则可以对字符数组进行输入/输出。如下面的代码段可以对一个字符数组进行输入/输出：

```
char str[10];
for( int i=0; i<10; i++ )
    scanf( "%c", &str[i] );
for( i=0; i<10; i++ )
    printf( "%c", str[i] );
```

2) 将字符数组中的整个字符串一次性输入或输出

上面提到对字符数组“只能引用字符数组中的某个元素，而不能引用整个数组”，有一个例外，就是在输入/输出时，可以对字符数组一次性输入或输出。需要注意的是：① 在 `scanf` 和 `printf` 函数里使用 “%s” 格式控制符；② 输入/输出时使用数组名，而且在 `scanf` 函数里不要加上取地址运算符&。如上面代码中的输入/输出部分可改成：

```
scanf( "%s", str );
printf( "%s", str );
```

字符数组输入时要注意：必须保证输入字符串的长度小于数组长度。这是因为如果输入串的长度等于或大于数组长度，系统会把超出数组长度的字符存放在数组所占存储空间的后继字节里，而这些字节不属于数组范围，这是很危险的。如下面的例子。

**例 2.6** 字符数组输入时输入串长度大于数组长度。

```
#include <stdio.h>
void main( )
{
    char b[8] = {"LOVE"};
    char a[8];
    scanf( "%s", a ); //输入超过 8 个字符
    printf( "%s\n", a );
    printf( "%s\n", b );
}
```

该程序运行示例如下：

```
aaaaaaaaa✓
aaaaaaaaa
aa
```



图2.6 输入串的长度大于字符数组的长度

如图 2.6 所示，在输入字符数组 `a` 时，输入 10 个字符 “a”，再加上串结束标志 ‘\0’，一共 11 个字符，其中前面 8 个字符 “a” 存在在字符数组 `a` 中，后面 2 个字符 “a” 和串结束标志 ‘\0’ 覆盖了字符数组 `b` 前 3 个字符。输出字符数组 `a` 时，也不是只输出 8 个字符，而是输出 10 个

字符，一直输出每个字符直到串结束标志'\0'为止；输出字符数组 **b** 时，输出 2 个字符 **a**，后边的字符“**E**”不会输出来，因为前面有串结束标志'\0'。 □

通过这个例子可以看出，串结束标志'\0'对字符数组和字符串是非常重要的。

### 3) 使用 gets 函数和 puts 函数进行输入/输出

使用 scanf 函数读入字符数组时，是以空格或回车键表示输入结束的，因此，如果用 scanf 函数读入字符串“**abc de**”到一个字符数组中，则只能读取空格前的字符串“**abc**”。如果要读入包含空格的字符串，需要使用 gets 函数，该函数是以回车键表示输入结束的。

gets 函数的原型为：

```
char *gets( char * buffer );
```

调用 gets 函数时，实参可以是字符数组名或字符指针变量，表示存放读入字符的存储空间首地址。如果读入成功，该函数的返回值就是实参所表示的地址，否则（即读入不成功），返回空指针 NULL。

puts 函数的功能是输出一个字符串，puts 函数的原型是：

```
int puts( const char * string );
```

其中参数 string 为存储字符串的首地址。如果输出成功，puts 函数的返回值为非负整数，否则返回-1。

gets 函数具体使用方法如例 2.7。

**例 2.7** 输入一行字符（字符总数不超过 80 个），统计其中大写字母、小写字母、数字字符、空格、其他字符分别有多少个。

在该例子中，使用 gets 函数读入一行字符，存入到字符数组。然后对字符数组中字符串结束标志'\0'前的每个字符都判断是否是大写字母、小写字母、数字字符、空格、其他字符等。

```
#include <stdio.h>
void main( )
{
    char str[80];
    gets(str);    //读入一行可以包含空格的字符串
    int upper = 0, lower = 0, digit = 0, space = 0, others = 0;
    int i = 0;
    while( str[i]!='\0' ) //对字符串结束标志'\0'前的每个字符进行如下的判断
    {
        if( str[i]>='A' && str[i]<='Z' ) //大写字母字符
            upper++;
        else if( str[i]>='a' && str[i]<='z' ) //小写字母字符
            lower++;
        else if( str[i]>='0' && str[i]<='9' ) //数字字符
            digit++;
        else if( str[i]==32 )    //空格
            space++;
        else others++;    //其他字符
        i++;
    }
    printf( "upper=%d,lower=%d,digit=%d,space=%d,others=%d\n",
            upper, lower, digit, space, others );
}
```

该程序的运行示例如下：

```
Art78&* ~! 78AERsuv✓
upper=4,lower=5,digit=4,space=4,others=4
```

□

### 2.3.4 字符数组与字符串常量的区别与联系

字符数组与字符串常量这二者之间有一定的联系，但又有明确的区别。

联系：

- 1) 如图 2.2 所示，字符串常量中的每个元素都是字符型数据，都占一个字节，而且这些字节是连续的。如果有一组数据，其中每个数据的类型都相同，而且这些数据所占的存储空间是连续的，这就是数组的概念。因此字符串常量在内存中是以字符数组的形式存放的，自动在末尾加字符串结束标志'\0'。
- 2) 如果能获得字符串常量“所在数组的数组名”，或者能获得字符串常量存储空间的首地址，那么就可以访问字符串常量中的每个字符。方法见 2.4.2 小节中的第 2 点。
- 3) 可以用字符串常量给数组初始化，但要注意，字符串常量最后一个字节是字符串结束标志'\0'，这个字节也是要赋值给字符数组中对应元素的。

区别：

- 1) 字符串常量末尾一定是字符'\0'，并以此作为结束标志。
- 2) 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'。但如果字符数组中没有字符'\0'，那么输出时可能会出现异常。例如下面的字符数组：

```
char ch[5] = { 'H', 'e', 'l', 'l', 'o' };
printf( "%s", ch ); //输出字符数组，因为没有串结束标志'\0'，因此输出时会出现异常字符
```

**思考 2.5：**在用下面的语句定义并初始化字符数组 `ch` 后，`ch` 中是否包含串结束标志'\0'？

```
char ch[5] = { 'H', 'i' };
```

### 练习

- 2.6 编程实现：用字符数组存储从键盘上输入的字符串，然后将其中的元音字母复制到另一个字符串并输出。
- 2.7 编程实现：用字符数组存储从键盘上输入的字符串，并将字符串中大写字母字符转换成小写字母字符，小写字母字符转换成大写字母字符，其他字符不变，输出转换后的字符串。
- 2.8 编程实现：用字符数组存储从键盘上输入的字符串，将其逆序后输出。
- 2.9 编程实现：用字符数组存储从键盘上输入的字符串，求字符串中由 ASCII 编码值连续的字符组成的最长子串（子串是指由位置连续的若干个字符组成的字符串），如果存在长度相同的最长子串，则输出第 1 个字符 ASCII 编码值最大的最长子串。

程序的运行示例 1 如下：

```
abcbbdefgmnrsv✓
defg
```

程序的运行示例 2 如下：

```
abefjkmnrsvuv✓
uv
```

- 2.10 字符数组的使用。编写程序，实现：将 12 个月份的英文单词存入一个二维字符数组中；输入月份号，输出该月份的英文单词。程序的运行示例如下：

```
3✓
March
```



## 2.4 字符指针变量

在 1.11 介绍了指针及指针变量的概念，如果定义的指针变量的基类型是 `char` 型，那么该指针变量就是字符指针变量。

### 2.4.1 字符指针变量的定义与引用

字符指针变量定义的例子：

```
char *pc;
```

字符指针变量可以指向普通的字符型变量，如下面的例子：

```
char c;
pc = &c; //字符指针变量指向字符型变量 c
(*pc) = 'Z'; //这条语句等效于 c = 'Z';
```

字符指针变量也可以指向字符串的存储空间，如字符数组或字符串常量。

### 2.4.2 字符指针变量、字符数组与字符串常量

#### 1. 字符指针变量与字符数组

字符指针变量可以指向数组元素，如下面的例子：

```
char ch[20];
pc = ch; //pc 指向字符数组 ch 的首地址，即第 0 个元素
pc++; //pc 指向字符数组下一个元素，即第 1 个元素
```

注意理解：如果在 `printf` 函数里，输出表达式是一个字符数组名或字符指针变量，或者表达式的值表示的是一个字符数组中某个元素的地址，那么输出内容是从当前字符开始一直到字符串结束标志之间的字符串。如下面的例子：

```
char str[] = "I love CHINA!";
printf("%s\n", str); //输出：I love CHINA!
char* pc = str;
printf("%s\n", pc); //输出：I love CHINA!
printf("%s\n", str+7); //输出：CHINA!
```

字符指针变量与字符数组的灵活运用详见例 2.8。

**例 2.8** 使用字符指针变量输出字符数组中的字符。

```
#include <stdio.h>
void main()
{
    char ch[20];
    gets(ch); //输入一串字符
    char *pc;
    pc = ch; // (1) 字符指针变量 pc 指向字符数组第 0 个元素
    for( ; (*pc) != '\0'; pc++) // (2) 采用字符指针变量输出字符数组中的字符
        printf("%c", *pc);
    printf("\n");
    pc = ch; // (3) 字符指针变量 pc 重新指向字符数组第 0 个元素
    printf("%s", pc+11); // (4) pc+11 指向的是字符数组中的第 11 个元素
    printf("\n");
}
```

该程序的运行示例如下：

Welcome to C/C++! ✓

Welcome to C/C++!  
C/C++!

上面的程序中，要特别注意第(4)条语句，`pc+11` 指向的是字符数组中的第 11 个元素，即 `ch[11]`，如图 2.7 所示，因此这条语句是从第 11 个字符开始输出。

**思考 2.6：**例 2.8 中的语句(3)能不能去掉？

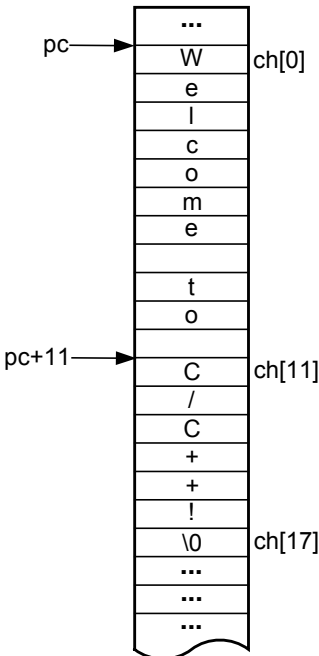


图2.7 通过字符指针访问字符数组

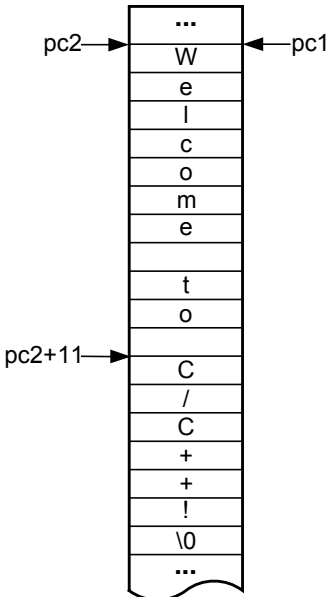


图2.8 通过字符指针访问字符串常量

## 2. 字符指针变量与字符串常量

2.3.4 小节中曾提到，“如果能获得字符串常量存储空间的首地址，那么就可以访问字符串常量中的每个字符”。字符指针变量就可以实现这一点。如果将一个字符串常量赋值给一个字符指针变量，则字符指针变量的值就是字符串常量存储空间的首地址。如：

```
char *pc;
```

```
pc = "Welcome to C/C++!";
```

`pc` 指向的就是这个字符串常量的第 0 个元素，通过这个字符指针变量就可以输出字符串常量中的每个字符，具体使用方法见例 2.9。

**例 2.9** 使用字符指针变量输出字符串常量中的字符。

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

```
void main( )
```

```
{
```

```
    char *pc1, *pc2;
```

```
    pc1 = pc2 = "Welcome to C/C++!"; //字符指针变量 pc1 和 pc2 指向字符串常量
```

```
    for( ; (*pc1)!='\0'; pc1++ ) //采用字符指针变量输出字符串常量中的每个字符
```

```
        printf( "%c", *pc1 );
```

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

```
    printf( "%s", pc2+11 ); // (1)从字符串常量的第 11 个字符开始输出
```

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

```
}
```

该程序的输出结果如下：

```
Welcome to C/C++!
C/C++!
```

如图 2.8 所示, 初始时, 字符指针变量 `pc1` 和 `pc2` 都指向字符串常量的第 0 个字符。在 `for` 循环中, 通过改变 `pc1` 的值可以访问字符串常量中的每个字符。`pc2+11` 指向的是字符串常量的第 11 个字符, 因此语句(1)从第 11 个字符开始输出。 □

### 3. 字符指针变量、字符数组、字符串常量的区别与联系

字符指针变量、字符数组、字符串常量这三者之间既有联系又有区别。下面两条语句分别定义了字符指针变量与字符数组:

```
char *pc;
char str[20] = "Welcome";
```

接下来以字符指针变量 `pc` 和字符数组 `str` 为例解释字符指针变量、字符数组、字符串常量这三者之间的联系与区别:

- 1) 字符数组 `str` 由若干元素组成, 每个元素放一个字符, 在初始化时可以用字符串常量对字符数组赋值, 除此之外, 不能把字符串常量赋值给字符数组; 字符指针变量 `pc` 存放的是字符数据的地址, 在任何时候, 都可以把字符串常量赋值给字符指针变量, 实际上是把字符串常量的首地址赋值给字符指针变量。如下面的语句:

```
str = "I love China!";    //(×)
pc = "I love China!";    //(√)
```

- 2) 数组名 `str` 是地址常量, 代表整个数组的首地址, 它的值不能改变; 指针变量 `pc` 是地址变量, 它的值可以改变。如 “`str++`” 是非法的, 而 “`pc++`” 是合法的。
- 3) 可以往字符数组 `str` 中输入字符串; 但是通过字符指针变量 `pc` 输入字符串时, 必须先让字符指针变量 `pc` 指向某个字符数组 (即某段存储空间), 才能输入字符串。

```
scanf( "%s", str );      //(√)
scanf( "%s", pc );      //(×)
pc = str; scanf( "%s", pc );    //(√)
```

- 4) 字符串常量所占的每个字节的内容都是不允许修改的 (2.2 节已经提到), 如下面的例子; 但如果把字符串常量赋值给字符数组后, 字符数组元素的值是可以改变的。

`char *pc = "Welcome"; (*pc) = 'a';` // (×) 通过字符指针变量修改字符串常量中的字符是非法的

`char c[ ] = "Welcome"; c[0] = 'a';` // (√) 修改字符数组元素的值是合法的

### 练习

- 2.11 编程实现: 用一字符数组存储从键盘输入的字符串, 输出 ASCII 编码最大的字符, 要求使用字符指针访问字符数组中的每个字符。
- 2.12 编程实现: 用一字符指针指向一个字符串常量, 并将其拷贝到一个字符数组。

## 2.5 字符串处理函数

由于字符串使用广泛, C/C++ 提供了一些字符串处理函数, 使得用户能很方便地对字符串进行处理。几乎所有版本的 C/C++ 都提供这些函数, 它们是放在函数库中的, 在 `string` 和 `string.h` 头文件中声明。如果程序中使用这些字符串函数, 应该用把 `string.h` 或 `string` 头文件包含到本文件中。下面介绍几个常用的函数。

### 1. 字符串连接函数 `strcat`

原型: `char* strcat(char *strDestination, const char *strSource);`

功能：把第 2 个参数（字符指针变量）**strSource** 所指向的字符串连到第 1 个参数（字符指针变量）**strDestination** 所指向的字符串后面。

返回值：返回连接后所得到的字符串的首地址，即第 1 个参数 **strDestination** 的值。

说明：① 第 1 个参数所指向的存储空间必须足够大（足以容纳这两个字符串）。

② 连接前，两串均以'\0'结束；连接后，串 1 的'\0'取消，新串最后加'\0'。

**strcat** 函数使用的例子如例 2.10，其执行过程如图 2.9 所示。

**例 2.10** **strcat** 函数的使用。

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char s1[20] = "Welcome to";
    char s2[ ] = " C/C++!";
    char *pc = strcat(s1, s2); //将字符串 s2 连接到 s1
    printf( "%s", pc ); //输出连接后的字符串
    //以上两条语句可合并成： printf( "%s", strcat(s1, s2) );
}
```

该程序的输出结果如下：

Welcome to C/C++!

如图 2.9 所示，把字符串 **s2** 连接到 **s1**，是从 **s2** 的第 0 个字符，即空格字符开始，连接到 **s1** 的串结束标志'\0'处。 □

**说明：****strcat** 函数及后面的 **strcpy** 函数返回值都是字符指针，而且是目标字符串的指针，返回这个指针有什么好处？答案是：返回目标字符串的指针，这样可以直接把函数执行结果输出来（如例 2.10）；甚至可以把函数执行结果放到表达式里参与运算，或者作为其他函数的参数。如下面的代码段中，把函数调用“**strcat(str1,str2)**”的返回值作为 **strcpy** 函数的参数。

```
char str1[30] = {0};
char str2[30] = "People's Republic of ";
char str3[ ] = "China";
//把 str3 连接到 str2，并把结果拷贝给 str1，最后输出 str1
printf( "%s\n", strcpy( str1, strcat(str2,str3) ) );
```

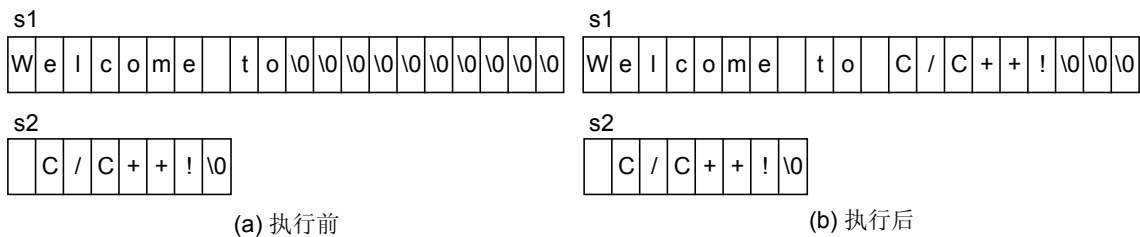


图2.9 **strcat**函数的执行过程

## 2. 字符串复制函数 **strcpy**

原型：**char\* strcpy(char \*strDestination, const char \*strSource);**

功能：把第 2 个参数 **strSource** 所指向的字符串拷贝到第 1 个参数 **strDestination** 所指向的存储空间里，将原有的字符覆盖。

返回值：返回拷贝后所得到的字符串的首地址，即第 1 个参数 **strDestination** 的值。

说明：① 第 1 个参数所指向的存储空间必须足够大（足以容纳第 2 个字符串）。

② 拷贝时'\0'一同拷贝过去。

③ 不能使用赋值语句为一个字符数组赋值！如：

```
char str1[20] = { "Hello!" }, str2[20];    //(✓), 初始化, 合法
str2 = str1;    //(×), 赋值, 非法
```

strcpy 函数使用的例子如例 2.11, 其执行过程如图 2.10 所示。

**例 2.11** strcpy 函数的使用。

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char s1[20] = "Welcome to C/C++!";
    char s2[ ] = "C/C++!";
    char *pc = strcpy(s1, s2); //(1)将字符串 2 拷贝到字符串 1
    printf( "%s", pc ); //输出拷贝后的字符串 1
    //以上两条语句可合并成: printf( "%s", strcpy(s1, s2) );
}
```

该程序的输出结果如下:

C/C++!

如图 2.10 所示, 拷贝时, 把字符串 s2 中的字符, 包括串结束标志'\0'拷贝到 s1 数组, 覆盖原有的字符。对于没有被覆盖的字符, 保持原值。因此, 如果在语句(1)后面用下面的语句输出 s1 数组中的第 8 个字符:

```
printf("%c", s1[8]);
```

输出的是字符 “t”。

□

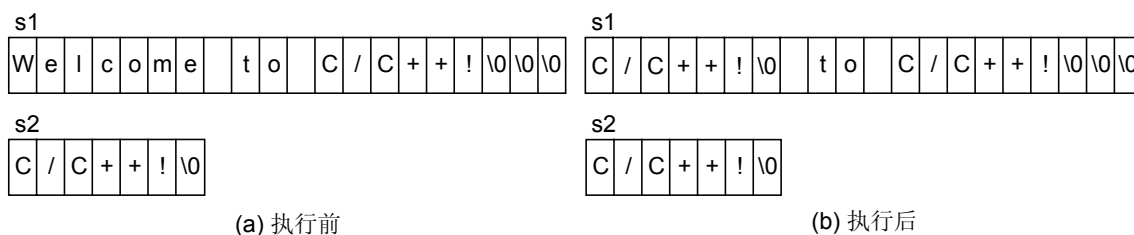


图2.10 strcpy函数的执行过程

### 3. 字符串比较函数 strcmp

原型: int strcmp(const char \*string1, const char \*string2);

功能: 比较两个字符串的大小。

比较规则: 对两字符串从左向右逐个字符比较 (ASCII 码), 直到遇到不同字符或'\0'为止。

返回值: 返回 int 型整数, a. 若字符串 1 < 字符串 2, 返回 -1

b. 若字符串 1 > 字符串 2, 返回 1

c. 若字符串 1 == 字符串 2, 返回 0

说明: ① 字符串比较不能用 “==”、“>”、“<” 等, 必须用 strcmp 函数。

② 这两个字符数组只参加比较而不改变其内容。

**例 2.12** strcmp 函数的使用。

```
#include <stdio.h>
#include <string.h>
void main( )
{
    printf( "%d\n", strcmp("aba", "abAc") );
    printf( "%d\n", strcmp("abca", "abcd") );
    printf( "%d\n", strcmp("abcd", "abcd") );
}
```

```
}
```

该程序的输出结果如下：

```
1
-1
0
```

□

#### 4. 字符串长度函数 `strlen`

原型： `unsigned int strlen(const char *string);`

功能：计算字符串长度。

返回值：返回字符串实际长度，不包括 '\0' 在内。

例如以下字符串的长度为 5，字符数组的长度为 10。

```
char str[10] = "China";
```

#### 5. 存储空间赋值函数 `memset`

原型： `void* memset( void *dest, int c, int count );`

功能：把空类型指针变量 `dest` 所指向的存储空间的前 `count` 个字节设置成整数 `c`。

返回值：返回值就是参数 `dest` 所指向的存储空间的地址。

该函数的最初功能是将某段字符数组存储空间每个元素的内容设置成某个同样的字符，如下面的例子。

**例 2.13** `memset` 函数的使用。

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char s[20] = "Welcome to C/C++";
    printf( "%s\n", s );
    memset( s, 'e', 5 );    //将字符数组 s 前 5 个字符设置成字符'e'
    printf( "%s\n", s );
}
```

该程序的输出结果如下：

```
Welcome to C/C++
eeeeeme to C/C++
```

□

`memset` 函数还有另外一个功能：给某段内存清零，通常用来给一个数组中每个元素赋值为 0。例如在例 3.2 中就使用了这个函数。再举一个例子：

```
int a[10];
memset( a, 0, 10*sizeof(int) );
```

上面的代码使得整型数组 10 个元素共 40 个字节，每个字节的内容设置为 0，这样就使得 10 个元素的值设置为 0。

**思考 2.7:** `memset` 函数能否用来给一个数组中每个元素赋值为一个非零的值？例如给一个整型数组各元素赋值为 1。

#### 6. 存储空间拷贝函数 `memcpy`

原型： `void *memcpy( void *dest, const void *src, size_t count );`

功能：把空类型指针变量 `src` 所指向的存储空间前 `count` 个字节的内容拷贝到空类型指针变量 `dest` 所指向的存储空间。

返回值：返回的就是参数 `dest` 所指向的存储空间的地址。

例如，以下代码将数组 `a` 的内容拷贝到数组 `b`：

```
int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }, b[10];
```

```
memcpy( b, a, sizeof(a) );
```

因此，`memcpy` 函数可以快速地将一个数组的内容备份到另一个数组。

理解了上述字符串处理函数的执行过程后，读者可以试着自己编写这些字符串处理函数，如例 2.14 就实现了字符串连接函数。

**例 2.14** 自己编写字符串处理函数。

编写一个函数 `mystrcat( )`，实现两个字符串连接。不使用标准函数 `strcat( )`。在主函数中输入两个字符串，调用 `mystrcat` 函数进行连接，并输出连接后的字符串。

```
#include <stdio.h>
char* mystrcat( char *s1, const char *s2 )
{
    int i = 0;
    char *ps1 = s1;
    for( ; (*ps1) != 0; ps1++ ) ; //找到字符串 1 的串结束标志位置
    char *ps2 = s2;
    //从字符串 1 的串结束标志开始把字符串 2 中的每个字符按顺序连接到 s1 后面
    for( ; (*ps2) != 0; ps2++, ps1++ )
        (*ps1) = (*ps2);
    (*ps1) = 0; //重要：最后一个字符设置成串结束标志'\0'
    return s1; //返回字符串 1 的首地址，即 s1 的值
}
void main( )
{
    char str1[20] = "Welcome to";
    char str2[ ] = " C/C++!";
    char *pc = mystrcat(str1, str2); //调用 mystrcat 函数进行字符串连接
    printf("%s", pc); //输出连接后的字符串
    //以上两条语句可合并成：printf( "%s", mystrcat(str1, str2) );
}
```

该程序的输出结果如下：

Welcome to C/C++!

在 `mystrcat` 函数中，首先找到字符串 `s1` 的串结束标志位置，然后从该位置开始将字符串 `s2` 中的字符依次拷贝到字符串 `s1` 中，最后将 `s1` 字符串中的最后一个字符设置成串结束标志 `'\0'`，从而实现了字符串的连接。□

**思考 2.8：**上述代码的 `mystrcat` 函数中，是通过 `for` 循环来查找字符串 1 的串结束标志位置，也可以通过求得字符串 1 的长度后找到这个位置，请修改代码。

## 练习

2.13 编写一个函数 `mystrcpy( )`，实现两个字符串的拷贝。不使用标准函数 `strcpy( )`。主函数输入字符串，调用 `mystrcpy` 函数实现字符串拷贝，并输出拷贝后的字符串。`mystrcpy( )` 函数的原型为：

```
char * mystrcpy(char *string1, const char *string2);
```

2.14 编写一个函数 `mystrcmp( )`，实现两个字符串的比较：对两串从左向右逐个字符比较(ASCII 码)，直到遇到不同字符或 `'\0'` 为止；该函数的返回值为 `int` 型整数，当两个字符串相等时，返回 0；当两个字符串不等时，返回的是两个字符串中第 1 对不相同字符的 ASCII 编码值

的差值。不使用标准函数 `strcmp( )`。主函数输入 2 个字符串，调用 `mystrcmp` 函数实现字符串的比较，输出比较结果。`mystrcmp( )`函数的原型为：

```
int mystrcmp(const char *string1, const char *string2);
```

- 2.15 字符数组和字符串函数的使用：从键盘上输入 3 个字符串（假设只包括大写字母字符），每个字符串的长度不超过 20，输出最大的字符串。要求：在主函数中读入 3 个字符串，存入一个  $3 \times 30$  的二维字符数组，调用 `max_string` 求 3 个字符串的最大者并输出；`max_string` 函数用于求 n 个字符串的最大者并输出。`max_string` 函数的原型为：

```
void max_string( char str[ ][20], int n );
```

程序的运行示例如下：

CHINA✓

GERMANY✓

FRANCH✓

The largest string is : GERMANY

- 2.16 请使用 `memcpy` 函数实现交换两个数组的内容：定义长度相同的 3 个整型数组 a、b 和 t，并给数组 a 和 b 赋值，数组 t 充当临时数组，用类似于例 1.4 的方法实现交换数组 a 和 b 的内容。

## 2.6 编写处理字符型数据的程序

本节分析两道处理字符型数据的程序。

**例 2.15** 字符转换（模运算符的运用）。

从键盘上输入 1 个字符（假定输入的是小写字母字符，不需判断），保存到变量 `ch` 中。对这个小写字母字符，转换成后面第 4 个字母字符，如'a'变成'e'，'b'变成'f'，...，'v'变成'z'，'w'、'x'、'y'、'z'分别变成'a'、'b'、'c'、'd'。如图 2.11 所示。

**分析：**正如第 1 章例 1.5 中分析的那样，这里需要用模运算符来实现。要将 `ch` 转换成 `ch` 后面的第 4 个字符，则需要将 `ch` 的值改变成 “`ch+4`”。要使得'w'变成'a'，即要构成环状序列，则要对 26 取余，要用到表达式 “`ch = (ch + 4)%26`”，但是 $(ch + 4) \% 26$  的范围是 0~25，我们希望 `ch` 取到 97~122。所以必须做修改。先让 `ch` 减去 97，因为 `ch - 97` 的范围是 0~25，`ch - 97 + 4` 对 26 取余的范围也是 0~25，求余之后还得加上 97，即 `ch = (ch - 97 + 4)%26 + 97`，这样处理后 `ch` 的范围是 97~122，满足要求。在本题中，取模运算使得一个线性序列构成环状序列。

代码如下：

```
#include <stdio.h>
void main( )
{
    char ch;
    scanf("%c", &ch);
    ch = 97 + (ch-97+4)%26;    //取模运算
    printf( "%c\n", ch );
}
```

该程序的运行示例如下：

w✓

a

□

**例 2.16** 将字符串 `str1` 复制给字符串 `str2`。

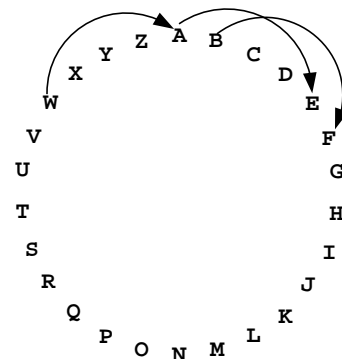


图2.11 取模运算构成环状序列



在本程序中，定义两个字符数组 `str1` 和 `str2`，再定义两个指针变量 `p1` 和 `p2`，分别指向两个字符数组中的第 0 个字符，通过改变指针变量的值使它们指向字符串中的不同的字符，以实现字符串的复制。

```
#include <stdio.h>
void main( )
{
    char str1[ ] = "I love CHINA!";
    char str2[15], *p1, *p2;
    p1 = str1;    //字符指针变量 p1 指向数组 str1 中的字符串
    p2 = str2;    //字符指针变量 p2 指向数组 str2 中的字符串
    for( ; *p1!='\0'; p1++, p2++ )    //把字符串 str1 赋值到 str2
        *p2 = *p1;
    *p2 = '\0';    //在 str2 后面加串结束标志
    p1 = str1;    //p1 回到字符串首
    p2 = str2;    //p2 回到字符串首
    printf( "str1 is: %s\n", p1 );
    printf( "str2 is: %s\n", p2 );
}
```

该程序的输出结果如下：

str1 is: I love CHINA!

str2 is: I love CHINA!

如图 2.12 所示，初始时，字符指针变量 `p1` 指向字符数组 `str1` 第 0 个元素，`p2` 指向 `str2` 第 0 个元素。在 `for` 循环中，通过赋值语句“`*p2 = *p1;`”，将 `p2` 所指向的元素的值赋值给 `p1` 指针变量所指向的元素。然后分别使得 `p1` 指向下一个元素，`p2` 也指向下一个元素。循环直至字符数组 `str1` 中的串结束标志 `'\0'`，从而实现字符串的复制。 □

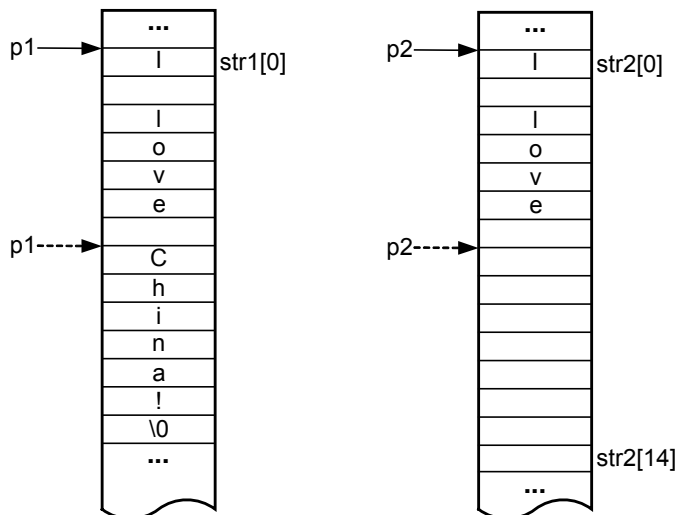


图2.12 使用字符指针实现字符串复制

## 练习

2.17 编程实现：从键盘输入一个字符（假定该字符是大写字母，不需判断），将该字符按如下规律转换：如果是'A'，则转换成'Z'；如果是'B'，则转换成'Y'；...；如果是'Z'，则转换成'A'，以此类推。输出转换后的字符。

提示：在 ASCII 编码表中，大写字母是按顺序排列的，如果一个字符型数据 `ch`，它的值是'A'，在这个值的基础上加 1，就变成了'B'，再加 1，就变成了'C'。`ch` 的值应该怎样变

换，才能使得'A'→'Z'、'Z'→'A'，...，'Z'→'A'？

- 2.18 编程实现：输入一行字符，统计其中有多少个单词。提示：以空格区分每个单词，但是不能简单的直接统计空格的个数，因为可以用多个空格隔开 2 个单词。
- 2.19 编写一个函数 `upper_lower`，实现将字符串中的小写字母转换成大写字母、大写字母转换成小写字母、其他字符保持不变。主函数中输入字符串，并调用 `upper_lower` 函数进行转换，并输出转换后的字符串。`upper_lower` 函数的原型为：  
`char* upper_lower( char *str );`
- 2.20 用一个二维字符数组存储 10 个长度均不超过 10 的字符串，然后从键盘上输入一个长度也不超过 10 的字符串，在二维数组中查找与输入字符串最接近的字符串。定义字符串 `s` 与两个字符串 `s1` 和 `s2` 接近程度的大小关系为：`s` 与 `s1` 从第 1 个字符开始比较，直至第 `d1` 个字符不相等，且 ASCII 编码值相差 `L1`（取绝对值）；`s` 与 `s2` 也从第 1 个字符开始比较，直至第 `d2` 个字符不相等，且 ASCII 编码值相差 `L2`（取绝对值）；如果 `d1` 与 `d2` 不等，则 `s` 与 `d1` 和 `d2` 较大者对应的字符串更接近，如果 `d1` 与 `d2` 相等，则 `s` 与 `L1` 和 `L2` 较小者对应的字符串更接近。（假设不存在两个字符串与 `s` 同样接近，即 `d` 值和 `L` 值均相等）

## 第二篇 程序设计方法及在线实践（基础篇）

本篇介绍基本的程序设计方法和算法，并通过在线实践实例分析讲解其中包含的程序设计方法和算法。本篇包括了第 3~6 章。各章的内容安排如下。

第 3 章介绍了 ACM/ICPC 程序设计竞赛及在线实践。详细地总结了 ACM/ICPC 题目的输入/输出方式，对每种输入方式通过分析讲解一道典型的竞赛题目来加深读者这些输入方式的理解。

第 4 章介绍枚举算法思想，并通过验证歌德巴赫猜想这一经典数学问题以及其他一些竞赛题目着重讲解枚举算法思想及在应用枚举算法时需要注意的问题。

第 5 章介绍模拟思想，并通过约瑟夫环问题的模拟等竞赛题目着重讲解模拟方法的思想及在应用模拟方法时需要注意的问题。

第 6 章讲解字符及字符串处理。涉及到的知识和应用问题包括：字符转换与编码、回文的判断与处理、子串的处理、字符串匹配等。

### 程序实践提示

在学习程序设计方法时，建议读者通过验证例题→理解思考题→完成练习题等这些程序实践来理解本篇介绍的程序设计方法和算法及其应用，对例题和练习题一定要提交到 OJ 网站上来验证算法的正确性。

另外，本教材在附录 C 专门为程序设计竞赛初学者介绍了在做 ACM/ICPC 题目时需要掌握的一些基本方法和技巧。

---

## 第 3 章 ACM/ICPC 程序设计竞赛与在线实践

近十几年来,各种程序设计竞赛开展得如火如荼。本章介绍的 ACM/ICPC 程序设计竞赛,就是其中规模最大的一类程序设计竞赛,本教材所收录的竞赛题目都是 ACM/ICPC 各级别的竞赛题目。这些竞赛不仅给众多程序设计爱好者提供了一个展示自己分析问题和解决问题的能力的机会,也给程序设计初学者提供了一个实践程序设计思想和方法的平台。

### 3.1 程序设计竞赛与在线程序实践

随着各类程序设计竞赛的推广,各种程序在线评判(Online Judge, 简称为 OJ)网站也应运而生,这为程序设计类课程提供了一种新的程序实践方法:在线程序实践。

在线程序实践是指由 OJ 网站提供题目,用户在线提交程序,OJ 网站的在线评判系统实时评判并反馈评判结果。这些题目一般具有较强的趣味性和挑战性,评判过程和结果也公正及时,因此能引起用户的极大兴趣。

用户在解题时编写的解答程序通过网页提交给在线评判系统称为提交运行,每一次提交运行会被判为正确或者错误,判决结果会及时显示在网页上。

用户从评判系统收到的反馈信息包括:

"Accepted" — 程序通过评判!

"Compile Error" — 程序编译出错。

"Time Limit Exceeded" — 程序运行超过该题的时间上限还没有得到输出结果。

"Memory Limit Exceeded" — 内存使用量超过题目里规定的上限。

"Output Limit Exceeded" — 输出数据量过大(可能是因为陷入死循环了)。

"Presentation Error" — 输出格式不对,可检查空格、空行等等细节。

"Run Time Error" — 程序运行过程中出现非正常中断,如数组越界等。

"Wrong Answer" — 用户程序的输出错误。

等等。

用户可以根据 OJ 系统反馈回来的评判结果反复修改程序,直到最终收获 **Accept**(程序正确)。这个过程不仅能培养用户独立分析问题、解决问题的能力,而且每成功解决一道题目都能给用户带来极大的成就感。

### 3.2 ACM/ICPC 程序设计竞赛简介

ACM/ICPC(ACM International Collegiate Programming Contest, 国际大学生程序设计竞赛)是由美国计算机协会 ACM(Association for Computing Machinery)主办的,世界上公认的规模最大、水平最高的国际大学生程序设计竞赛,其目的旨在使大学生运用计算机来充分展示自己分析问题和解决问题的能力。该项竞赛从 1977 年第一次举办世界总决赛以来,至今已连续举办 30 多届了。该项竞赛一直受到国际各知名大学的重视,并受到全世界各著名计算机公司的高度关注。

ACM/ICPC 竞赛分区域预赛和总决赛两个阶段进行,各预赛区第一名自动获得参加世界总决赛的资格。世界总决赛安排在每年的 3~4 月举行,而区域预赛安排在上一年度的 9~12 月在各大洲举行。

ACM/ICPC 竞赛以组队方式进行比赛,每支队伍由不超过 3 名队员组成,比赛时每支队伍

只能使用一台计算机。在 5 个小时的比赛时间里,参赛队伍要解答 6~10 道指定的题目。排名时,首先根据解题数目来排名,如果多支队伍解题数量相同,则根据队伍的总用时进行排名(用时越少,排名越靠前)。每支队伍的总用时为每道解答正确的题目的用时总和。每道解答正确的题目的用时为从比赛开始计时到该题目解答被判定为正确的时间,其间每一次错误的提交运行将被加罚 20 分钟时间。最终未正确解答的题目不记入总时间,其提交也不加罚时间。

ACM/ICPC 竞赛在公平竞争的前提下,提供了一个让大学生充分展示用计算机解决问题的能力与才华的平台。ACM/ICPC 竞赛鼓励创造性和团队协作精神,鼓励在编写程序时的开拓与创新,它考验参赛选手在承受相当大的压力下所表现出来的非凡能力。竞赛所触发的大学生的竞争意识为加速培养计算机人才提供了最好的动力。竞赛中对解决问题的苛刻要求和标准使得大学生对解决问题的深度和广度展开最大程度的追求,也为计算机科学的研究和发展作了一个最好的导向。

### 3.3 ACM/ICPC 竞赛题目特点

#### 3.3.1 ACM/ICPC 题目组成及特点

一道完整的 ACM/ICPC 题目通常包含 5 部分:题目描述,输入描述,输出描述,样例输入,样例输出等。

- 1) 题目描述:题目通常不会直接告知要求解一个什么问题,而是以一个故事或者一个游戏作为背景知识引入的,所以题目描述通常会比较繁琐。
- 2) 输入、输出描述:给出题目对输入/输出格式的要求。
- 3) 样例输入/输出:为了便于理解题目,以及测试程序的需要,题目中会给出几组正确的测试数据。

每道 ACM/ICPC 竞赛题目都有时间限制和内存空间限制。3.4 节给出了 4 道 ACM/ICPC 竞赛题目(例 3.1~例 3.4)。

另外,在服务器端,每道题目还会有输入数据文件和标准输出数据文件。输入数据文件用来测试用户提交的程序,该数据文件通常能测试到题目需要考虑的各种特殊情况。标准输出数据文件是由标准解答程序根据输入数据文件得到的正确的输出数据文件。评判系统就是将用户的输出文件与标准输出文件进行对比从而判定用户的解答程序是否正确。

ACM/ICPC 题目的特点是每道题目有多组测试数据。我们平时写的程序,只有一组数据,处理完这组数据,程序就结束了,但是 ACM/ICPC 题目都是需要处理多组数据的。其目的有两个:一是为了测试各种可能的情况,防止出现用户程序考虑不全面也能通过评判的情形,二是可以测试用户程序的运行时间、以及所采用算法的优劣。

初次接触这类题目的学生通常难以从一组数据的处理过渡到多组数据的处理(详见 3.3.2 及例 3.1 中的说明),所以本章的重点是分析这类题目输入/输出方式。

#### 3.3.2 ACM/ICPC 题目的输入/输出

##### 1. 输入

ACM/ICPC 题目的输入有 4 种基本情形:

- ① 输入数据文件中,第一行数据标明了测试数据的数目;
- ② 输入数据文件中,有标明输入结束的数据;
- ③ 输入数据文件中,测试数据一直到文件尾;
- ④ 没有输入数据,这种情形罕见。

表 3.1 列出了前 3 种情形的处理方法。这里要特别提醒的是,对于第 3 种情形,题目有时

不会明确告诉测试数据一直到文件尾，只要判断不是第 1、2、4 这三种情形，那么就是第 3 种情形，因为 ACM/ICPC 题目都是要处理多组测试数据的。

表 3.1 ACM/ICPC 题目输入情形及其处理方法

| 情形 1 的处理                                                                                                            | 情形 2 的处理                                                                                                                                                      | 情形 3 的处理                                                                                                      |
|---------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <pre>//kase 表示测试数据数目 int i, kase; scanf( "%d", &amp;kase ); for(i=1; i&lt;=kase; i++) {     //处理第 i 个输入 } ...</pre> | <pre>//假定每组测试数据包含 //两个数据: m n, 0 0 表示结束 int m, n; while( 1 ) {     scanf( "%d %d", &amp;m, &amp;n );     if( m==0&amp;&amp;n==0 ) break;     //处理 } ...</pre> | <pre>//假定每组测试数据包含两个数据: m n int m, n; while( scanf("%d %d", &amp;m, &amp;n)!=EOF ) {     //处理该测试数据 } ...</pre> |

初次接触 ACM/ICPC 的学生，通常的思维是试图把所有的测试数据先存储起来，再依次处理(详见例 3.1 中的说明)。这种处理方式存在的问题是，由于不知道输入数据文件中有多少组测试数据(无论是情形 1、情形 2，还是情形 3)，只能定义很大的数组来存储所有的测试数据，浪费存储空间，甚至有时会超出题目所规定的内存限制。其实是没有必要把所有测试数据都先存储起来再处理的，正确的处理方法是先读一组测试数据进来，处理完毕后，再读入下一组测试数据进行处理。所以只需要定义存储一组测试数据所需的变量即可。具体方法详见例 3.1。

另外，如果题目提到输入数据的范围，则输入数据文件中的数据肯定满足这个范围，不必判断!!!

需要说明的是，目前有些竞赛题目的输入情形是某两种情形的嵌套。比如第 5 章中的练习 5.8，其输入文件格式为：第一行为整数  $N$ ，表示输入文件中有  $N$  组测试数据；接下来是  $N$  组测试数据，每组测试数据至少 1 行，至多 21 行，除最后一行外每行代表一个测试数据，最后一行是 0，表示该组测试数据结束。这种输入格式中每组测试数据是第 2 种输入情形，整个输入文件是第 1 种输入情形——是在第 1 种输入情形的里面又嵌套了第 2 种输入情形。

## 2. 输出

ACM/ICPC 题目对输出要求是极其严格的，只要格式不对或者程序考虑不全面，程序就不可能通过。这就要求学生从简单的程序开始就全面考虑，养成良好的编程习惯。程序的输出通常需要考虑如下情况：

- 1) 正确输出空格，特别是有的题目要求在一行数据中除最后一个数据外，其他每个数据之后都输出空格，而最后一个数据之后不输出空格，处理方法与以下空行的处理方法类似。
- 2) 正确输出空行，特别是有的题目要求除最后一组测试数据外，每组测试数据的输出内容之后都输出空行，而最后一组测试数据的输出内容之后不输出空行。可以采用的方法是反其道而行：在除第一组测试数据外的每组测试数据的输出内容之前输出空行。具体方法是：设置一个状态变量 **flag**，初值为 **false**，其且仅当 **flag** 为 **true** 时，在测试数据的输出内容之前输出空行；当读入第一组测试数据时，因为 **flag** 为 **false**，不输出空行，然后把 **flag** 设置为 **true**；之后，在每组测试数据的输出内容之前都会输出空行了。
- 3) 正确地按照题目所要求的精度进行输出。

还有一点要注意，对于题目要求输出的一些提示信息，如果样例输出里有，可以拷贝过去，这样就不容易出错。

### 3.3.3 ACM/ICPC 题目类型

ACM/ICPC 竞赛题目所涉及到的算法主要有 3 大类：

- 1) 基础算法，如枚举、模拟、递归及搜索等；
- 2) 优化算法，如贪心、分治、动态规划等；
- 3) 图论、数论、计算几何等领域的基础算法。

这些题目对于培养学生算法分析与设计的意识和能力有很大的作用。据统计，ACM/ICPC 竞赛题目的题型及比例如表 3.2 所示。

表 3.2 ACM/ICPC 程序设计竞赛题目题型及比例

| 题型 | 搜索  | 动态规划 | 贪心 | 构造 | 图论  | 计算几何 | 纯数学问题 | 数据结构 | 其它  |
|----|-----|------|----|----|-----|------|-------|------|-----|
| 比例 | 10% | 15%  | 5% | 5% | 10% | 5%   | 20%   | 5%   | 25% |

本教材接下来的第 4 章~第 9 章陆续介绍第一大类算法思想。

## 3.4 ACM/ICPC 竞赛题目解析

本节分析 4 道 ACM/ICPC 竞赛题目，其中例 3.1 对应第 1 种输入情形，例 3.2 对应第 2 种输入情形，例 3.3 对应第 3 种输入情形，例 3.4 对应第 4 种输入情形。

### 例 3.1 数字阶梯(Number Steps)

题目来源：

Asia 2000, Tehran (Iran)

题目描述：

从坐标(0,0)出发，在平面上写下所有非负整数 0, 1, 2, ..., 如图 3.1 所示。例如 1、2 和 3 分别是在(1,1)、(2,0)和(3, 1)坐标处写下的。

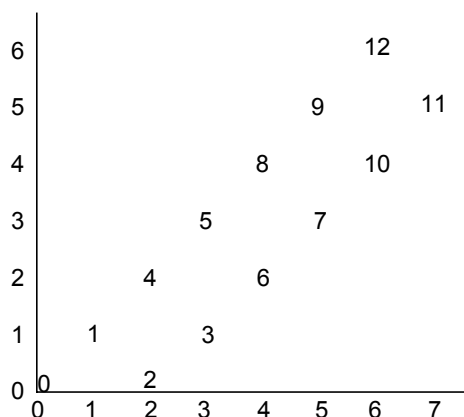


图3.1 数字阶梯

编写一个程序，给定坐标(x,y)，输出对应的整数(如果存在的话)，输入文件中 x、y 坐标范围都是 0~5000。

输入描述：

输入文件的第一行为正整数 N，表示输入文件中测试数据的数目。接下来是 N 个测试数据，每个测试数据占一行，包含两个整数：x y，代表平面上的坐标(x,y)。

输出描述：



对输入文件中每一行所表示的坐标点，输出在该点的非负整数，如果没有对应整数，输出“No Number”。

样例输入：

3  
4 2  
6 6  
3 4

样例输出：

6  
12  
No Number

分析：

在本题中，输入数据文件中第一行为一个整数  $N$ ，表示测试数据的个数，因此是第一种输入情形，程序要处理  $N$  个测试数据。每个测试数据表示平面上一个点的坐标，要输出该点对应的非负整数；如果没有对应的非负整数，则输出“No Number”。

非负整数  $0, 1, 2, \dots$  有规律地分布在两条直线上， $L_1: y = x$  和  $L_2: y = x - 2$ 。其中的规律为：

1. 如果输入的点坐标不满足这两条直线方程，则没有对应的非负整数。
2. 否则，非负整数在这两条直线上分布规律是：
  - 1) 在直线  $L_1: y = x$  上，如果坐标  $x$  是偶数，则对应的整数是  $2*x$ ，如果坐标  $x$  是奇数，则对应的整数是  $2*x-1$ ；
  - 2) 在直线  $L_2: y = x - 2$  上，如果坐标  $x$  是偶数，则对应的整数是  $2*x-2$ ，如果坐标  $x$  是奇数，则对应的整数是  $2*x-3$ 。

另外，正如 3.3.2 节指出的，本题的题目描述中提到  $x$  和  $y$  的范围，则输入数据文件中的数据肯定满足这个范围，在程序中不必判断!!!。

代码如下：

```
#include <stdio.h>
int main( )
{
    int N, i;
    scanf( "%d", &N );    //N 表示输入文件中测试数据的数目
    int x, y;
    for( i=0; i<N; i++ )    //处理输入文件中的 N 个测试数据
    {
        scanf( "%d %d", &x, &y ); //每个测试数据包含两个整数
        int Num;    //在(x,y)坐标点上的非负整数
        if( y!=x && y!=x-2 ) Num = -1; //(x,y)不在 L1，也不在 L2 上
        else
        {
            if( y==x && x%2==0 ) Num = 2*x; //(x,y)在 L1 上，且 x 为偶数
            else if( y==x && x%2!=0 ) Num = 2*x-1;    //(x,y)在 L1 上，且 x 为奇数
            else if( y==x-2 && x%2==0 ) Num = 2*x-2;    //(x,y)在 L2 上，且 x 为偶数
            else Num = 2*x-3;    //(x,y)在 L2 上，且 x 为奇数
        }
        if( Num==-1 ) printf( "No Number\n" );
        else printf( "%d\n", Num );
    }
    return 0;
}
```

说明：

1) 在 3.3.2 节中提到,初次接触 ACM/ICPC 竞赛的学生对输入数据通常的做法是把所有输入数据先存储起来,再处理。比如下面的方法:

```
int x[1000], y[1000]; //存储 N 个点的 x 坐标和 y 坐标
int i, N;
scanf( "%d", &N ); //N 表示输入文件中测试数据的个数
for( i=1; i<=N; i++ )
    scanf( "%d%d", &x[ i ], &y[ i ] );
for( i=1; i<=N; i++ )
{
    ... //处理第 i 个点的坐标并输出
}
```

如果输入数据文件中测试数据的数目小于 1000, 即  $N < 1000$ , 这样处理也是可以的。但如果测试数据的数目大于 1000, 则因为定义的数组太小, 导致数组越界、产生运行出错。由于本题并没有告诉  $N$  的取值范围, 不知道输入数据文件中测试数据有多少个, 所以这种方法是不可取的。

2) ACM/ICPC 竞赛及各 OJ 系统使用的是标准 C/C++ 语言的编译器, 而标准 C/C++ 规定 main 函数必须返回 int 值, 所以今后所有竞赛题目解答程序的 main 函数返回值类型都采用 int, 而不是像第 1 章中的程序那样没有返回值(即函数返回值为 void)。

### 例 3.2 假票(Fake Tickets)

**题目来源:**

South America 2002, Practice

**题目描述:**

舞会收到很多假票。要求编写程序, 统计所有门票中存在假票的门票数。

**输入描述:**

输入文件中包含多个测试数据。每个测试数据占两行。第 1 行为两个整数  $N$  和  $M$ , 分别表示发放门票的张数和参加晚会的人数( $1 \leq N \leq 10000$ ,  $1 \leq M \leq 20000$ )。第 2 行为  $M$  个整数  $T_i$ , 为收到的  $M$  张门票的号码( $1 \leq T_i \leq N$ )。输入文件最后一行为 0 0, 代表输入结束。

**输出描述:**

对每个输入测试数据, 你的程序要输出一行, 为一个整数, 表示收上来的门票中有多少张票被伪造过。

**样例输入:**

```
5 5
3 3 1 2 4
6 10
6 1 3 6 6 4 2 3 1 2
0 0
```

**样例输出:**

```
1
4
```

**分析:**

在本题中, 输入数据文件中每个测试数据的第 1 行为两个整数  $N$  和  $M$ , 分别表示发放门票的总数和收到的门票总数,  $N = M = 0$  代表输入结束, 程序读入到这一组数据后, 处理结束, 所以是第 2 种输入情形。

因为  $N$  不会超过 10000, 所以定义一个一维数组 ticket[10001], 各元素的初值为 0, 统计每张收到的门票: 设门票的号码为  $i$ , 在对应的数组元素上加 1, 即 ticket[i]++。M 张门票统计完毕后, 元素值大于 1 的就是存在伪造门票的。例如, 样例输入中的第 2 个测试数据, 收到 10

张门票，统计完以后，`ticket` 数组的存储情形如图 3.2 所示。其中门票号码为 6 的有 3 张，号码为 1 的有 2 张，号码为 3 的有 2 张，号码为 2 的有 2 张，这些门票都被伪造过，即有 4 张门票被伪造过。

|          |   |   |   |   |   |   |   |     |  |
|----------|---|---|---|---|---|---|---|-----|--|
| 元素序号→    | 0 | 1 | 2 | 3 | 4 | 5 | 6 |     |  |
| ticket数组 |   | 2 | 2 | 2 | 1 | 0 | 3 | ... |  |

图3.2 假票：统计结果

另外，在以下程序中用到了 `memset` 函数，该函数是在头文件 `string.h` 中声明的，其作用内存初始化，即给某一段存储空间中的每个字节赋值为同一个值，在该程序中，调用 `memset` 函数给数组 `ticket` 各元素清零。

代码如下：

```
#include <stdio.h>
#include <string.h>
int N, M; //N 是门票的总数，M 是收到的门票总数。
int ticket[10001]; //统计每个号码的门票有多少张
int main( )
{
    int i;
    while( scanf("%d %d", &N, &M) ) //每个测试数据的第 1 行为两个整数 N 和 M
    {
        if( !N && !M ) break; //当 N = M = 0 时，输入结束
        int tmp;
        memset( ticket, 0, sizeof(ticket) ); //对 ticket 数组清零
        for( i=1; i<=M; i++ ) // "登记"这 M 张门票，对每张门票，给对应数组元素加 1
        {
            scanf( "%d", &tmp );
            ticket[tmp]++;
        }
        int sum = 0;
        for( i=1; i<=N; i++ ) //统计被伪造过的门票的数目
            if( ticket[i]>1 ) sum++;
        printf( "%d\n", sum );
    }
    return 0;
}
```

### 例 3.3 纸牌(Deck)

题目来源：

South Central USA 1998

题目描述：

一张扑克牌可以放置在桌子的边缘，只要伸出桌子边缘的长度不超过整张牌长度的一半即可。 $n$  张牌叠起来放在桌子的边缘，其最长可伸出桌子边缘的长度为  $1/2 + 1/4 + \dots + 1/(2*n)$ ，如图 3.3 所示。输入  $n$ ，按照题目要求的格式输出  $n$  张牌可伸出桌子边缘的最大长度。

输入描述：

输入文件包括多个测试数据，每个测试数据占一行，为一个非负整数。每个整数都是小于 99999 的。

**输出描述:**

输出首先包含一个标题，即首先输出下面一行：

**# Cards Overhang**

注意"#"和"Cards"之间有一个空格，"Cards"和"Overhang"之间有两个空格。

然后对输入文件中的每个测试数据，首先输出该测试数据中牌的数目  $n$ ，再输出  $n$  张牌最长可伸出桌子边缘的长度，单位为一张牌的长度，保留小数点后 3 位有效数字。输出长度的格式必须在小数点前至少有一位数，在小数点后有 3 位。牌的数目  $n$  右对齐到第 5 列，长度中的小数点在第 12 列。

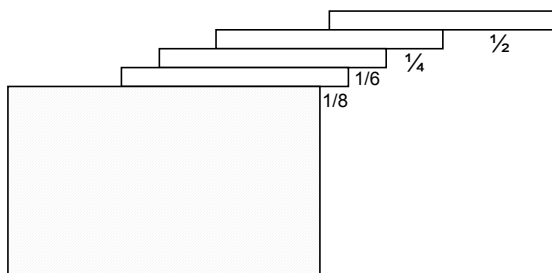


图3.3  $n$ 张牌，其最长可伸出桌子边缘的长度为  
 $1/2 + 1/4 + \dots + 1/(2 \cdot n)$

**样例输入:**

1  
2  
3  
30

**样例输出:**

```
12345678901234567
# Cards Overhang
1      0.500
2      0.750
3      0.917
30     1.997
```

**注意：**在样例输出第一行中的数字是用来帮助你按照正确的格式输出的，不是你的程序所应该输出来的

**分析:**

在本题中，没有标明测试数据个数的数据，所以不是第 1 种输入情形；也没有标志着输入结束的数据，所以不是第 2 种情形；显然更不是第 4 种情形；因此，输入数据是一直到文件尾的，是第 3 种输入情形。

这道题目实际上就是求数列和： $1/2 + 1/4 + \dots + 1/(2 \cdot n)$ ，而且这道题对输出格式要求很严格。另外，正如 3.3.2 节提到的，对于题目中要求输出的提示信息"# Cards Overhang"，可以从题目中拷贝到程序里，这样就不容易出错。

**代码如下:**

```
#include <stdio.h>
int main( )
{
    int n, j;
    printf( "# Cards Overhang\n" );
    //输入一直到文件尾，此时读入的是文件结束标记 EOF
    while( scanf("%d", &n)!=EOF )
    {
        double len = 0.0;
        for( j=1; j<=n; j++ )    //计算长度
```

```

        len += 1.0/(double)(2*j);
    printf( "%5d", n );      //按照要求的格式进行输出
    printf( "%10.3f\n", len ); //按照要求的格式进行输出
}
return 0;
}

```

### 例 3.4 特殊的四位数(Specialized Four-Digit Numbers)

**题目来源:**

Pacific Northwest 2004

**题目描述:**

找到并输出所有的 4 位数（十进制数）中具有如下属性的数：四位数字之和等于其十六进制形式各位数字之和，也等于其十二进制形式各位数字之和。

例如，十进制数 2991，其四位数字之和  $2+9+9+1 = 21$ 。十进制数 2991 的十二进制形式为  $1893_{(12)}$ ，其各位数字之和也为 21，但是它的十六进制形式为  $BAF_{(16)}$ ，其各位数字之和等于  $11+10+15 = 36$ ，因此你的程序要舍去 2991 这个数据。

下一个数，2992，其十进制、十二进制、十六进制形式各位数字之和均为 22，因此 2992 符合要求，应该输出来。（只考虑 4 位数，2992 是第一个符合要求的数）。

**输入描述:**

本题没有输入。

**输出描述:**

你的程序要求输出 2992 及其他更大的、满足要求的四位数（要求严格按升序输出），每个数占一行（前后都没有空行），整个输出以换行符结尾。输出中没有空行。输出中的前几行如样例输出所示。

**样例输入:**

本题没有输入。

**样例输出:**

```

2992
2993
2994
2995
2996

```

**分析:**

本题没有输入，是输入的第 4 种情形。该题在求解时要用到枚举的算法思想(详见第 4 章)，即枚举所有的 4 位数(1000~9999)，判断是否满足：其十六进制、十二进制、十进制形式中各位数之和相等。

这里要特别注意进制转换的方法。如果要将一个十进制数 NUM 转换到 M 进制，其方法是：将 NUM 除以 M 取余数，直到商为 0 为止，存储得到的余数，先得到的余数为低位，后得到的余数为高位进行排列，余数 0 不能被舍去。关于进制转换，详见第 7 章。当然本题只需要得到 NUM 在 16、12、10 进制下各位和，所以只要累加余数即可。

另外，本题的程序还使用了 continue 语句：如果 NUM 的 16 进制各位和不等其 12 进制各位和，则不需再判断 10 进制各位和与 16、12 进制各位和是否相等，可以提前结束本次循环，所以要用到 continue 语句。

**代码如下:**

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

```

int main( )
{
    int NUM, temp;
    for( NUM = 1000; NUM <= 9999; NUM++ ) //枚举所有的 4 位数
    {
        int s16 = 0, s12 = 0, s10 = 0; //NUM 的 16、12、10 进制各位和
        temp = NUM;
        while( temp ) //等效于 while( temp!=0 )
        {
            s16 += temp % 16;
            temp /= 16;
        }
        temp = NUM;
        while( temp )
        {
            s12 += temp % 12;
            temp /= 12;
        }
        if( s16 != s12 ) //如果 NUM 的 16 进制各位和 不等于 12 进制各位和
            continue; //不需判断下去，提前结束本次循环
        temp = NUM;
        while( temp )
        {
            s10 += temp % 10;
            temp /= 10;
        }
        if( s16 == s10 ) printf( "%d\n", NUM );
    }
    return 0;
}

```

## 练习

注意：以下 4 道练习题中，练习 3.1~3.4 分别对应第 1~4 种输入情形，读者在做练习题可以参考本章对这 4 种输入情形的解释，通过练习加深对 ACM/ICPC 题目 4 种输入情形处理方法的理

### 3.1 二进制数(Binary Numbers, Central Europe 2001, Practice)

#### 题目描述：

给定一个正整数  $n$ ，要求输出对应的二进制数中所有数码“1”的位置。注意最低位为第 0 位。例如 13 的二进制形式为 1101，因此数码 1 的位置为：0，2，3。

#### 输入描述：

输入文件中的第 1 行为一个正整数  $d$ ，表示输入文件中测试数据的个数， $1 \leq d \leq 10$ ，接下来有  $d$  个测试数据。每个测试数据占一行，只有一个整数  $n$ ， $1 \leq n \leq 10^6$ 。

#### 输出描述：

输出包括  $d$  行，即对输入文件中的每个测试数据，输出一行。第  $i$  行， $1 \leq i \leq d$ ，以升序的顺序输出第  $i$  个测试数据中的整数的二进制形式中所有数码“1”的位置，位置之间有 1 个空格，

最后一个位置后面没有空格。

**样例输入：**

2  
13  
127

**样例输出：**

0 2 3  
0 1 2 3 4 5 6

**提示：**

- 1) 对输入的整数  $n$ ，依次用 2 去整除，用变量  $pos$  充当计数器（代表二进制的位），如果得到的余数为 1，则输出  $pos$ ，否则不输出； $pos$  的初值为 0，每次将  $n$  除以 2 后， $pos$  自增 1。
- 2) 题目要求两个位置之间有 1 个空格，最后一个位置之后没有空格。解决方法是在第 1 个位置之前不输出空格，然后在接下来的所有数码“1”的位置之前输出一个空格。

### 3.2 完数(Perfection, Mid-Atlantic USA 1996)

**题目描述：**

判断一个数是 **perfect**，**abundant**，还是 **deficient**，判断标准为：如果它的所有 **proper** 因子之和等于它本身，则这个数为 **perfect**（注意，**perfect** 数其实就是完数）；如果它的所有 **proper** 因子之和大于它本身，则这个数为 **abundant**；如果它的所有 **proper** 因子之和小于它本身，则这个数为 **deficient**。**proper** 因子的定义： $a = b \cdot c$ ，如果  $c$  不为 1，则  $b$  为  $a$  的一个 **proper** 因子， $a$ 、 $b$ 、 $c$  均为正整数。也就是说，所谓 **proper** 因子，就是除本身之外的所有因子。

**输入描述：**

输入文件中有若干个（假设为  $N$  个， $1 < N < 100$ ）正整数（这些整数都不大于 60000），最后一个数为 0，表示输入结束。

**输出描述：**

输出的第一行为字符串“PERFECTION OUTPUT”。接下来有  $N$  行，表明输入文件中的  $N$  个数是否为 **perfect**，**deficient**，或 **abundant**，格式如样例输出中所示。输出中的最后一行必须为字符串“END OF OUTPUT”。

**样例输入：**

15 6 56 60000 496 0

**样例输出：**

PERFECTION OUTPUT  
15 DEFICIENT  
6 PERFECT  
56 ABUNDANT  
60000 ABUNDANT  
496 PERFECT  
END OF OUTPUT

### 3.3 求三角形外接圆周长(The Circumference of the Circle, University of Ulm Local Contest 1996)

**题目描述：**

给定平面上**不共线**的三个点的坐标，求这三个点所确定的三角形外接圆的周长。

**输入描述：**

输入文件包含一个或多个测试数据。每个测试数据占一行，为 6 个浮点数  $x_1, y_1, x_2, y_2, x_3, y_3$ ，代表 3 个点的坐标。由这三个点确定的三角形外接圆周长不超过 1,000,000。输入数据一直到文件尾。

### 输出描述:

对输入文件中的每个测试数据, 输出占一行, 为一个浮点数, 表示所求得的外接圆周长, 保留小数点后 2 位有效数字。 $\pi$  的值可以用近似值 3.141592653589793。

### 样例输入:

```
0.0 -0.5 0.5 0.0 0.0 0.5
0.0 0.0 0.0 1.0 1.0 1.0
5.0 5.0 5.0 7.0 4.0 6.0
0.0 0.0 -1.0 7.0 7.0 7.0
50.0 50.0 50.0 70.0 40.0 60.0
```

### 样例输出:

```
3.14
4.44
6.28
31.42
62.83
```

## 3.4 根据公式计算 e(u Calculate e, Greater New York 2000)

### 题目描述:

计算  $e$  的一个简单公式是: 
$$e = \sum_{i=0}^n \frac{1}{i!}$$

其中  $n$  趋于无穷大。当  $n$  较小的值时, 得到的近似值也接近于  $e$  的实际值。

### 输入描述:

本题没有输入。

### 输出描述:

输出  $n$  取值从 0 到 9 时, 根据上述公式计算出的  $e$  的近似值。输出的格式请参照样例输出, 在样例输出中, 给出了  $n$  取 0~4 时的输出。

### 样例输入:

本题没有输入。

### 样例输出:

```
n e
- -
0 1
1 2
2 2.5
3 2.666666667
4 2.708333333
```



## 第4章 枚举

**枚举**(enumeration, 又称为**穷举**), 是一种很朴素的解题思想。当需要求解的问题存在大量的可能的答案(或中间过程), 而暂时又无法用逻辑方法排除这些可能答案中的大部分时, 就不得不采用逐一检验这些答案的策略, 这就是枚举算法的思想。

例如, 第1章例1.26在“判断 $m=199$ 是否为素数”时, 从素数的定义出发, 试图找出 $2\sim 198$ 范围内能整除 $m$ 的自然数, 如果不能找到, 则 $m$ 是素数。根据分析, 我们将范围缩小到 $2\sim 14$ ( $14$ 是小于等于 $199$ 的平方根的最大正整数)。依次判断 $2$ 能否整除 $m$ , 判断 $3$ 能否整除 $m$ , ..., 一直判断到 $14$ 都还没有找到能整除 $m$ 的自然数, 因此得出结论:  $m=199$ 是素数。这个过程实际上就是一个枚举的过程, 即枚举 $2, 3, 4, 5, \dots, 14$ 能否整除 $m$ 。

### 4.1 枚举的基本思路

本节首先通过一个简单的例子介绍枚举的基本思路, 然后再结合一道ACM/ICPC竞赛题目总结采用枚举方法解题时需要注意的问题。

**例4.1** 求 $x^2 + y^2 = 2000$ 的正整数解。

**分析:**

$x$ 和 $y$ 都是正整数, 因此 $x$ 和 $y$ 的取值范围只能是:  $1, 2, \dots, 44$ , 其中 $44$ 是小于等于 $\sqrt{2000}$ 的最大正整数。对于在这个范围内的所有 $(x, y)$ 组合, 都去判断一下。也就是枚举所有的 $(x, y)$ 组合, 判断是否满足 $x^2 + y^2 = 2000$ , 如果满足, 则是一组解。当 $x$ 取 $1$ 时, 考虑 $y$ 取 $1, 2, \dots, 44$ ; 然后当 $x$ 取 $2$ 时, 又考虑 $y$ 取 $1, 2, \dots, 44$ ; ...; 最后当 $x$ 取 $44$ 时, 又考虑 $y$ 取 $1, 2, \dots, 44$ 。整个过程如图4.1所示。在实现时要用到2重循环, 从算法思想的角度看, 这个过程就是枚举, 即枚举所有的 $(x, y)$ 组合。

**代码如下:**

```
#include <stdio.h>
#include <math.h>
int main( )
{
    int x, y;
    int m = sqrt(2000);    //循环变量 x 和 y 的终值
    for( x=1; x<=m; x++ ) //x 从 1 枚举到 m
    {
        for( y=1; y<=m; y++ ) //y 也从 1 枚举到 m
        {
            if( x*x + y*y == 2000 ) //注意, 判断相等必须用两个等于号: “==”。
                printf( "2000=%d*%d+%d*%d\n", x, x, y, y );
        }
    }
    return 0;
}
```

程序的输出如下:

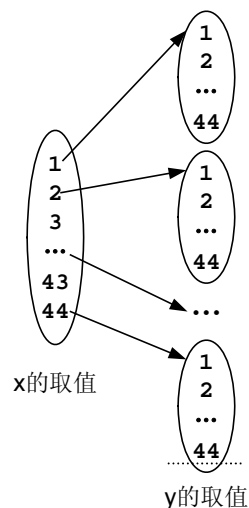


图4.1 枚举策略

```
2000=8*8+44*44
2000=20*20+40*40
2000=40*40+20*20
2000=44*44+8*8
```

从程序的输出结果可以看出, (8, 44)这一组解和(44, 8)这一组解实际上只是交换了  $x$  和  $y$ 。如果认为这是同一组解, 那么方程就只有两组解: (4, 44)和(20, 40), 该怎样修改程序呢?

注意到, 以上两组解有个特点:  $y \geq x$ , 因此我们在枚举时, 只要保证  $y \geq x$ , 则(40, 20)和(44, 8)这两组解都不会被枚举出来, 程序的输出就满足要求。因此只需将上述代码中的二重循环修改成如下的代码:

```
for( x=1; x<=m; x++ ) //x 从 1 枚举到 m
{
    //y 从 x 开始枚举, 保证 y>=x, 这样枚举出来的结果就没有重复的
    for( y=x; y<=m; y++ )
    {
        if( x*x + y*y == 2000 )
            printf( "2000=%d*%d+%d*%d\n", x, x, y, y );
    }
}
```

程序的输出如下:

```
2000=8*8+44*44
2000=20*20+40*40
```

从本题的分析可以看出, 在采用枚举方法时, 要分析题目的要求, 避免输出重复的解。

#### 例 4.2 切换状态(Switch)

**题目来源:**

Zhejiang University 2003 Summer Camp Qualification Contest

**题目描述:**

有  $N$  盏灯, 排成一排。给定每盏灯的初始状态(开或关), 你的任务是计算至少要切换多少盏灯的状态(将开着的灯关掉, 或将关掉的灯开起来), 才能使得这  $N$  盏灯开和关交替出现。

**输入描述:**

输入文件中包含多个测试数据, 每个测试数据占一行。首先是一个整数  $N$ ,  $1 \leq N \leq 10000$ , 然后是  $N$  个整数, 表示这  $N$  盏灯的初始状态, 1 表示开着的, 0 表示关着的。测试数据一直到文件尾。

**输出描述:**

对每个测试数据, 输出占一行, 表示至少需要切换状态的灯的数目。

**样例输入:**

```
9 1 0 0 1 1 1 0 1 0
3 1 0 1
```

**样例输出:**

```
3
0
```

**分析:**

本题可以采取不同的枚举思路求解。

第一种枚举思路。 $N$ 盏灯, 每盏灯都有两种状态: 1 和 0,  $N$ 盏灯共有  $2^N$ 种状态, 从 0 0 0 ... 0 到 1 1 1 ... 1。可以枚举这  $2^N$ 种状态, 每种状态都判断一下是否是开和关交替出现, 如果是则还要统计从初始状态转换到该状态需要切换的灯的数目。但这种枚举策略势必要花费很多时间, 因为 $N$ 最大可以取到 10000, 而  $2^{10000}$ 的数量级是  $10^{3010}$ 。

第二种思路。要使得  $N$  盏灯开和关交替出现，实际上只有两种可能：奇数位置上的灯是开着的，或者偶数位置上的灯是开着的。只要分别计算从  $N$  盏灯的初始状态出发，切换到这两种状态所需要切换灯的数目，取较小者即可。例如样例输入中的第 1 个测试数据对应的初始状态如图 4.2 所示，将这 9 盏灯切换到奇数位置上的灯是开着的，需要切换 6 盏灯；切换到偶数位置上的灯是开着的，需要切换 3 盏灯；因此至少需要切换 3 盏灯。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |

(a) 初始状态

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

(b) 奇数位置上的灯开着

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

(c) 偶数位置上的灯开着

图4.2 切换状态

第三种思路。注意到上述分析中  $3 + 6 = 9$ ，9 就是灯的数目  $N$ 。稍加分析就可以得到结论：如果将  $N$  盏灯调整成奇数位置上的灯是开着的，需要调整灯的数目为  $numo$ ，则将这  $N$  盏灯调整为偶数位置上的灯是开着的，需要调整灯的数目  $nume = N - numo$ 。因此只要判断  $numo$  是否小于  $N/2$ ，如果是则取  $numo$ ，否则取  $N - numo$ 。这种思路只要枚举一种状态即可。下面的代码采用的就是这种枚举思路。

代码如下：

```
#include <stdio.h>
int main( )
{
    int N, i; //N 表示灯的数目
    int numo; //调整成奇数位置上的灯是开着的需要调整灯的数目
    int lights[10001];
    while( scanf( "%d", &N )!=EOF )
    {
        numo = 0;
        for( i=1; i<=N; i++ ) scanf( "%d", &lights[i] ); //读入 N 盏灯的初始状态
        for( i=1; i<=N; i++ ) //将 N 盏灯的状态调整为奇数位置上为 1，偶数位置上为 0
        {
            if( i%2==1 && lights[i]==0 ) numo++; //奇数位置上为 0，需要调整
            if( i%2==0 && lights[i]==1 ) numo++; //偶数位置上为 1，需要调整
        }
        printf( "%d\n", numo<N/2 ? numo : N-numo );
    }
    return 0;
}
```

通过前面两道例题的分析，我们应该注意到，在采用枚举法解题时，重要的是：

- 1) 为保证结果正确，应做到既不重复又不遗漏。
- 2) 为减少程序运行时间，应尽量减少枚举的次数。

## 练习

### 4.1 几何问题变得简单了(Geometry Made Simple)

题目描述：

如果你有一台计算机，那么数学问题会变得很简单。例如，在直角三角形中，三条边的长度  $a$ 、 $b$  和  $c$  (其中  $c$  为斜边，如图 4.3 所示) 满足勾股定理： $a^2 + b^2 = c^2$ 。在本题中，已知两条边的长度，要求第 3 条边的长度。

#### 输入描述：

输入文件包含了多个直角三角形的数据。每个三角形的数据占一行，为 3 整数  $a$ 、 $b$  和  $c$ ，表示三角形 3 条边的长度。这 3 个整数中仅有一个整数为 -1，代表长度未知的边，而其他两个整数都为正整数。输入文件的最后一行为 3 个 0，表示输入数据结束。

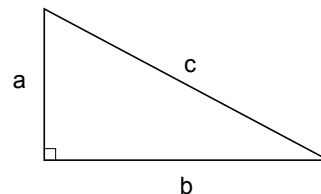


图4.3 直角三角形及其三条边

#### 输出描述：

对输入文件中的每个三角形，首先输出三角形的序号，如样例输出所示。然后，如果这 3 条边不能构成三角形，输出 "Impossible."，否则输出第 3 条边的长度，保留小数点后 3 位有效数字。每个三角形的输出之后有一个空行。

#### 样例输入：

```
-1 2 7
5 -1 3
0 0 0
```

#### 样例输出：

```
Triangle #1
a = 6.708

Triangle #2
Impossible.
```

#### 提示：

3 条边  $a$ 、 $b$  和  $c$  要么  $a$  为 -1，要么  $b$  为 -1，要么  $c$  为 -1，只要枚举这三种情况即可。

## 4.2 歌德巴赫猜想

1742 年，德国数学家哥德巴赫 (Goldbach) 提出了著名的哥德巴赫猜想 (Goldbach Conjecture)：任何一个不小于 4 的偶数都可以表示为两个素数之和。本节例 4.3 通过枚举的方法验证了任意输入的一个大于或等于 4 的偶数的确可以分解为两个素数之和；例 4.4 则是对任何一个大于或等于 4 的偶数，求满足条件的分解个数。

#### 例 4.3 验证歌德巴赫猜想。

编写程序，实现将一个不小于 4 的偶数分解成两个素数之和，并输出所有的分解形式。例如 34 有 4 种分解形式：

```
34 = 3 + 31
34 = 5 + 29
34 = 11 + 23
34 = 17 + 17
```

#### 分析：

对偶数 4，只有一种分解，即： $4 = 2 + 2$ 。

对任何一个不小于 6 的偶数  $n$ ，假设它可以表示两个数之和： $n = a + b$ ，如果  $a$  和  $b$  都是素数，则这是一种满足要求的分解形式。枚举所有可能的  $(a, b)$  组合，判断是否满足题目的要求。为了减少枚举的次数，本题可以采取如下的策略：

① 最小的素数是 2，但在本题中，从  $a = 3$  开始枚举，因为如果  $a$  的值为 2，则  $b$  的值为大于 2 的偶数，不可能是素数；

② 在枚举过程中， $a$  的值每次递增 2，而不是 1。这是因为如果每次递增 1，在枚举过程中  $a$  的值可以取到偶数，而每次递增 2，则可以跳过偶数，减少很多次枚举。

③ 另外， $a$  的值只需枚举到  $n/2$  即可，因为如果继续枚举，则枚举得到的符合要求的分解形式只不过是交换了  $a$  和  $b$  的值而已。

例如，假设  $n$  的值为 20， $a = 3$  时， $a$  是素数， $b = n - a = 17$ ， $b$  也为素数，则  $20 = 3 + 17$  是符合要求的分解形式。

下一步  $a$  的值递增 2，即  $a = 5$ ， $a$  是素数，而  $b = n - a = 15$  不是素数，不符合要求。

再下一步， $a$  的值再递增 2，即  $a = 7$ ， $a$  是素数，而  $b = n - a = 13$  也是素数，符合要求。

如此枚举到  $n > 10$  为止，如果继续枚举，得到的符合要求的分解形式，如  $20 = 13 + 7$ ，这个分解形式只是将“ $20 = 3 + 17$ ”分解形式中  $a$ 、 $b$  的值互换而已。

代码如下：

```
#include <stdio.h>
#include <math.h>
int prime( int );    //函数声明
int main( )
{
    int n;
    printf( "input n: " );
    scanf( "%d", &n ); //输入一个整数
    int a, b;
    if( n < 4 || n % 2 != 0 )
    { printf( "You should input an even integer no less than 4." ); return 1; }
    else if( n == 4 )
    { printf( "4 = 2 + 2\n" ); return 0; }
    for( a = 3; a <= n / 2; a = a + 2 ) //从 a=3 开始枚举，每次递增 2，跳过偶数
    {
        if( prime(a) ) //如果 a 为素数，再判断 b 是否为素数
        {
            b = n - a;
            if( prime(b) ) //b 也为素数，找到一个满足条件的分解
                printf( "%d = %d + %d\n", n, a, b );
        }
    }
    return 0;
}

int prime( int m ) //判断 m 是否为素数，如果为素数，返回 1，否则返回 0。
{
    int i, k = sqrt(m);
    for( i = 2; i <= k; i++ )
    {
        if( m % i == 0 ) //如果 i 能整除 m，提前退出循环
            break;
    }
    if( i > k ) return 1; //m 为素数
    else return 0; //m 为合数
}
```

```
}

```

程序运行示例如下：

```
input n: 40✓

```

```
40 = 3 + 37

```

```
40 = 11 + 29

```

```
40 = 17 + 23

```

**思考 4.1：** 对于一个大于或等于 5 的奇数，有的可以表示成两个素数之和，有的则不能。试编写程序，实现：输入一个大于或等于 5 的奇数，判断是否能分解成两个素数之和。

#### 例 4.4 歌德巴赫猜想(Goldbach's Conjecture)

**题目来源：**

Asia 1998, Tokyo (Japan)

**题目描述：**

歌德巴赫猜想：对任何一个不小于 4 的偶数，总是可以分解成一个素数对之和，即： $n = p_1 + p_2$ ，其中  $p_1$  和  $p_2$  都是素数。

这个猜想至今都还没有被证明是正确的，或者是错误的。没有人知道这个猜想到底是否正确。然而，对于一个给定的偶数，我们可以去找这样的素数对。本题的任务是编写一个程序，实现：对于一个给定的偶数，输出满足条件的素数对的个数。

注意，在本题中，对两个素数  $p_1$  和  $p_2$ ， $(p_1, p_2)$ 和 $(p_2, p_1)$ 是同一个素数对。

**输入描述：**

输入文件包含多个测试数据，每个测试数据占一行，为一个整数，并且假定这个整数是偶数，且不小于 4，小于  $2^{15}$ 。输入文件的最后一行为 0，表示输入结束。

**输出描述：**

对输入文件中的每个偶数(除最后的 0 外)，输出满足条件的素数对的个数。

**样例输入：**

```
6

```

```
10

```

```
12

```

```
0

```

**样例输出：**

```
1

```

```
2

```

```
1

```

**分析：**

例 4.3 实现了对输入的任何一个小于 4 的偶数，枚举并输出满足歌德巴赫猜想的分解形式。如果在枚举过程中进行计数，即可实现统计满足条件的素数对个数。代码如下：

```
#include <stdio.h>

```

```
#include <math.h>

```

```
int prime( int p ) //判断 m 是否为素数，如果为素数，返回 1，否则返回 0。

```

```
{

```

```
    int i, k = sqrt(p);

```

```
    for( i=2; i<=k; i++ )

```

```
    {

```

```
        if( p%i==0 ) break; //如果 i 能整除 m，提前退出循环

```

```
    }

```

```
    if( i>k ) return 1; //m 为素数

```

```
    else return 0; //m 为合数

```

```
}

```

```

int main( )
{
    int m;
    while( 1 )
    {
        scanf( "%d", &m );    //输入一个整数
        if( !m ) break;
        if( m==4 ){ printf( "1\n" ); continue; }
        int a, b;
        int count = 0;    //满足条件的素数对个数
        for( a=3; a<=m/2; a=a+2 ) //从 a=3 开始枚举，每次递增 2，跳过偶数
        {
            if( prime(a) ) //如果 a 为素数，再判断 b 是否为素数
            {
                b = m - a;
                if( prime(b) ) count++;
            }
        }
        printf( "%d\n", count );
    }
    return 0;
}

```

但是，因为对每个偶数 $m$ ，需要枚举近 $m/2$ 个组合，而输入文件中每个偶数 $m$ 的取值最大可达到 $2^{15} = 32768$ ，所以，如果输入文件中测试数据较多，上述方法可能会超时。更好的方法是按以下3个步骤进行（其中第2步最关键，也正是这一步很好地体现了枚举的思想）：

① 先采用第1章练习1.50介绍的筛选法求出2~32768之间的所有素数，保存在数组Prime中；32768以内的素数，共有3512个(可以通过编程统计)。

② 然后枚举所有不同的素数对(Prime[i], Prime[k])，其中Prime[i] < Prime[k]，如果其和sum不超过32768，则count[sum]自增1，即对sum，找到一种分解形式。这里假设有一数组count，count[i]表示整数i(包括奇偶数)的满足条件的素数对个数。请注意，由于素数2是偶数，所以对某些奇数，如25，也存在满足条件的素数对，如 $25 = 2 + 23$ 。

③ 对输入每个偶数m，输出求得的素数对个数count[m]。

需要说明的是：这种方法前两个步骤花费时间比较多，后一个步骤花费的时间相对少得多，所以如果测试数据比较少，则花费的时间不一定比前面的方法所花费的时间少；但数据越多，每个偶数m越大，越能体现这种方法的高效率。

代码如下：

```

#include <stdio.h>
#include <math.h>
#define MAX 32770    //2^15 = 32768 < 32770(数组长度定义得稍微大一些)
int Natures[MAX]; //初始时存放≥2的自然数
int Prime[3513]; //存储32768以内的素数，共有3512个
int count[MAX]; //count[i]为整数i(包括奇偶数)的满足条件的素数对个数

int main( )
{
    int i, j, k; //循环变量
    int m;    //输入的偶数

```

```

//筛选法求出 32768 以内的所有素数，依从小到大的顺序存放在 Prime 数组中
for( i=0; i<MAX; i++ ) Natures[i] = i+2;
for( i=0; i<MAX; i++ )
{
    if( Natures[i] )
    {
        for( k=i+1; k<MAX; k++ ) //所有能被 Natures[i]整数的数，都是合数
        {
            if( Natures[k] && Natures[k]%Natures[i]==0 )
                Natures[k] = 0; //Natures[k]为合数
        }
    }
}
for( i=0, j=0; i<MAX; i++ ) //将 Natures 数组中剩下的素数保存到 Prime 数组中
{
    if( Natures[i] )
    {
        Prime[j] = Natures[i]; j++;
    }
}
//枚举所有不同的素数对(Prime[i], Prime[k]),如果其和 sum 不超过 MAX
//则 count[sum]自增 1，即对 sum，找到一种分解形式
int sum;
for( i=0; i<j; i++ )
{
    for( k=i; k<j; k++ )
    {
        sum = Prime[i] + Prime[k];
        if( sum<MAX ) count[sum]++;
    }
}
while( scanf("%d",&m) && m ) //对输入的偶数 m，输出求得的素数对个数
    printf( "%d\n", count[m] );
return 0;
}

```

## 练习

### 4.2 歌德巴赫猜想 2(Goldbach's Conjecture)

#### 题目描述:

本题的任务是对小于 1,000,000 的偶数验证歌德巴赫猜想。

#### 输入描述:

输入文件包含一个或多个测试数据，每个测试数据为一个整数  $n$ ， $6 \leq n < 1000000$ 。 $n = 0$  表示输入结束。

#### 输出描述:

对每个测试数据  $n$ ，输出一行，格式为： $n = a + b$ ，其中  $a$  和  $b$  均为素数。数和运算符之间用一个空格隔开。如果存在多对素数满足要求，则选择差值( $b - a$ )最大的那对。如果不存在



这样的素数对，则输出"Goldbach's conjecture is wrong."。

**样例输入：**

8  
20  
42  
0

**样例输出：**

8 = 3 + 5  
20 = 3 + 17  
42 = 5 + 37

**提示：**

只需输出满足条件的第 1 对素数即可。

### 4.3 其他竞赛题目解析

本节再通过几道竞赛题目，分析枚举的思想及其实现方法。

#### 例 4.5 绑定正多边形(Bounding Box)

**题目来源：**

University of Waterloo Local Contest 2001.09.22

**题目描述：**

考古学家发现文物位于正多边形的顶点。沙漠里的移动沙丘使得挖掘工作十分艰难，所以一旦发现了正多边形的三个顶点，就必须用保护性的建筑物来覆盖整个正多边形。

**输入描述：**

输入文件包含多个测试数据，每个测试数据描述了一个正多边形。描述信息起始于一个整数  $n$ ， $n \leq 50$ ，即顶点的个数，接下来是三对实数，给出了这个多边形的三个顶点的  $x$  坐标和  $y$  坐标。每对实数用空格隔开，每对实数占一行。当  $n$  等于 0 时，表示输入结束，这个测试数据不需处理。

**输出描述：**

要求输出能够覆盖多边形所有顶点的、面积最小的矩形的面积，这个矩形的边界是平行  $x$  轴和  $y$  轴的。

**样例输入：**

4  
10.00000 0.00000  
0.00000 -10.00000  
-10.00000 0.00000  
6  
22.23086 0.42320  
-4.87328 11.92822  
1.76914 27.57680  
23  
156.71567 -13.63236  
139.03195 -22.04236  
137.96925 -11.70517  
0

**样例输出：**

Polygon 1: 400.000  
Polygon 2: 1056.172  
Polygon 3: 397.673

**分析：**

已知正多边形  $n$  个顶点中 3 个顶点(这 3 个顶点未必相邻)的坐标, 就可以确定正多边形的外接圆, 从而可以确定这  $n$  个顶点的坐标。枚举这  $n$  个顶点, 记录其横坐标的最小值、最大值, 纵坐标的最小值、最大值, 这四个值就是能够覆盖多边形所有顶点的、面积最小的矩形的边界。

本题的枚举策略比较简单, 关键在于: ① 求出三角形的外接圆圆心坐标; ② 根据向量旋转公式求其他所有顶点的坐标。

已知三角形的三个顶点坐标为  $(xx1, yy1)$ 、 $(xx2, yy2)$  和  $(xx3, yy3)$ , 要求圆心坐标  $(x, y)$ , 如图 4.4(c) 所示, 方法如下。由圆心  $O$  到 3 个顶点  $A$ 、 $B$ 、 $C$  距离相等, 可得两个方程:

$$\begin{cases} (xx - xx1)^2 + (yy - yy1)^2 = (xx - xx2)^2 + (yy - yy2)^2 \\ (xx - xx1)^2 + (yy - yy1)^2 = (xx - xx3)^2 + (yy - yy3)^2 \end{cases}$$

以上两个方程化简得:

$$\begin{cases} 2 \cdot (xx1 - xx2) \cdot x + 2 \cdot (yy1 - yy2) \cdot y = xx1^2 + yy1^2 - xx2^2 - yy2^2 \\ 2 \cdot (xx1 - xx3) \cdot x + 2 \cdot (yy1 - yy3) \cdot y = xx1^2 + yy1^2 - xx3^2 - yy3^2 \end{cases}$$

联立求解上述两个方程, 即可求得圆心坐标。下面代码中的 `Circle_center` 函数实现了这个求解过程。

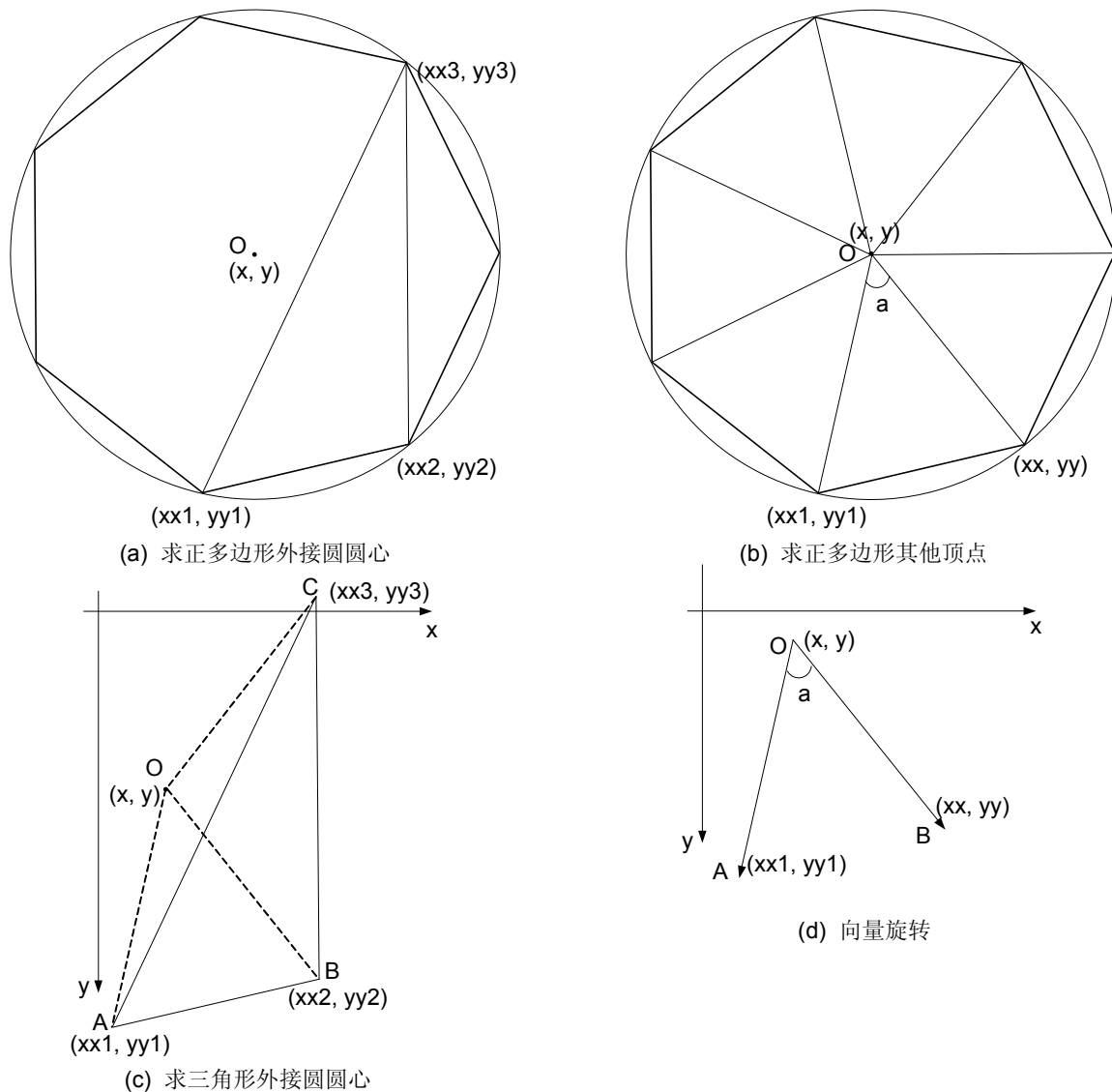


图4.4 求正多边形顶点坐标

根据已求得的圆心坐标，求正多边形其他顶点坐标的方法有很多，适合编程实现的是利用向量旋转公式，如图 4.4(d)所示。向量 AO 逆时针旋转  $\alpha$  角度，为向量 BO，则向量旋转公式为：

$$\begin{pmatrix} xx - x \\ yy - y \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \sin \alpha \end{pmatrix} \times \begin{pmatrix} xx1 - x \\ yy1 - y \end{pmatrix}$$

即：

$$xx = (xx1 - x) \times \cos \alpha - (yy1 - y) \times \sin \alpha + x$$

$$yy = (xx1 - x) \times \sin \alpha + (yy1 - y) \times \sin \alpha + y$$

对正  $n$  边形来说， $\alpha = 2 \times \pi / n$ ，依次将向量 AO 旋转  $\alpha$ 、 $2\alpha$ 、...、 $(n-1)\alpha$ ，就可求得其他所有顶点的坐标。在求解每个顶点坐标的过程当中，记录这  $n$  个顶点中  $x$  坐标的最大值  $\max\_x$ 、最小值  $\min\_x$ ，以及  $y$  坐标的最大值  $\max\_y$ 、最小值  $\min\_y$ 。则所求矩形面积为：

$$(\max\_x - \min\_x) * (\max\_y - \min\_y)。$$

代码如下：

```
#include <stdio.h>
#include <math.h>
#define INFINITY 100000000 //无穷大
int n; //正多边形顶点个数
double xx1, yy1, xx2, yy2, xx3, yy3; //输入的 3 个顶点坐标
double x, y; //根据给定的 3 个顶点求得的圆心的坐标
double min_x, max_x; //n 个顶点 x 坐标的最小值、最大值
double min_y, max_y; //n 个顶点 y 坐标的最小值、最大值

void Circle_center( ) //求正多边形外接圆圆心坐标
{
    double A1 = 2*(xx1-xx2);
    double A2 = 2*(xx1-xx3);
    double B1 = 2*(yy1-yy2);
    double B2 = 2*(yy1-yy3);

    double C1 = xx1*xx1+yy1*yy1-xx2*xx2-yy2*yy2;
    double C2 = xx1*xx1+yy1*yy1-xx3*xx3-yy3*yy3;

    double Jacobi = A1*B2-A2*B1;
    x = (C1*B2-C2*B1)/Jacobi;
    y = (A1*C2-A2*C1)/Jacobi;
}

void Get_rectangle( ) //求 n 个顶点 x 坐标的最大、最小值，y 坐标的最大、最小值
{
    min_x = INFINITY; max_x = -INFINITY;
    min_y = INFINITY; max_y = -INFINITY;

    double angle = 2*3.1415926535897932384626433832795/n;
    double xx, yy;
    for( int i=0; i<n; i++ )
    {
        xx = cos(i*angle)*(xx1-x)-sin(i*angle)*(yy1-y) + x;
        yy = sin(i*angle)*(xx1-x)+cos(i*angle)*(yy1-y) + y;
```

```

        if( xx<min_x )    min_x = xx;
        if( xx>max_x )    max_x = xx;
        if( yy<min_y )    min_y = yy;
        if( yy>max_y )    max_y = yy;
    }
}
int main( )
{
    int number = 1;
    while( scanf("%d", &n)!=EOF ) //读入正多边形的顶点数
    {
        if( n==0 ) break;
        scanf( "%lf%lf%lf%lf%lf%lf", &xx1, &yy1, &xx2, &yy2, &xx3, &yy3 ); //顶点坐标
        Circle_center( ); //求正多边形外接圆圆心坐标
        Get_rectangle( ); //求覆盖多边形 n 个顶点的矩形
        printf( "Polygon %d: %.3f\n", number, ( max_x - min_x ) * ( max_y - min_y ) );
        number++;
    }
    return 0;
}

```

#### 例 4.6 假银币(Counterfeit Dollar)

题目来源:

East Central North America 1998

题目描述:

Sally 有 12 枚银币，然而只有 11 枚是真的，有一枚是假的，它的颜色和大小跟真的银币是一样的，无法分辨。但是假银币的重量跟真银币的重量不一样，然而 Sally 并不知道假银币到底是比真银币重还是比真银币轻。

幸亏 Sally 的一个朋友借给她一台很精确的天平。她的朋友允许她称三次，从而找出假银币。例如，如果 Sally 在天平的两边各放一枚银币，天平是平衡的，则 Sally 就知道这两块银币是真的。进一步，如果 Sally 将其中一块真银币和第三枚银币放到天平上，而天平不平衡，则 Sally 知道第 3 枚银币是假银币，并且可以得知假银币比真银币轻还是重：如果假银币所在的一侧是下沉的，则它比真银币重，否则比真银币轻。

仔细地选择称重方法，Sally 能够确保 3 次称重就能找出假银币。

输入描述:

输入文件的第 1 行为一个整数  $n(n>0)$ ，代表接下来测试数据的数目。每个测试数据占三行，每一行代表一次称重。Sally 已经将 12 枚银币标为字母 A~L。每一次称重用两个字符串和一个单词“up”、“down”或“even”表示。第一个字符串代表天平左边的银币，第 2 个字符串代表天平右边的银币。Sally 总是在天平的两边放同样多的银币。最后面的单词告诉此次称重右边的一侧是上浮、下沉还是跟左边平衡。

输出描述:

对每个测试数据，输出必须表明哪个字母对应的银币是假银币，并且告知假银币比真银币重还是轻。输出格式如样例输出所示。输入数据保证每个测试数据的解是唯一的。

样例输入:

样例输出:

```

2
ABCD EFGH even
ABCI EFJK up
ABIJ EFGH even
ABCDEF GHIJKL up
ABHLEF GDIJKC down
CD HA even

```

K is the counterfeit coin and it is light.  
L is the counterfeit coin and it is light.

分析:

12 枚银币，标号为 A~L，只有 1 枚是假的，因此可以枚举这 12 枚银币：分别假设 A~L 为假银币，如果某枚银币是假银币且使得给定的 3 次称重是正确的，则该银币就是假银币，并且不需再判断下去了，因为题目提到“输入数据保证每个测试数据的解是唯一的”。

例如对样例输入中的第 1 个测试数据：

假设 A 为假银币：第 1 次称重是错误的；

假设 B 为假银币：第 1 次称重是错误的；

...

假设 H 为假银币：第 1 次称重是错误的；

假设 I 为假银币：第 1 次称重是正确的，第 2 次称重是正确的，且假银币比真银币重，第 3 次称重是错误的。

假设 J 为假银币：第 1 次称重是正确的，第 2 次称重是正确的，且假银币比真银币轻，第 3 次称重是错误的。

假设 K 为假银币：第 1 次称重是正确的，第 2 次称重是正确的，且假银币比真银币轻，第 3 次称重也是正确的。

从而得出结论：K 为假银币，且比真银币轻。

采用这种思路进行枚举时，要注意一种情形：如果某枚银币是假银币且使得给定的 3 次称重都是正确的，但在这 3 次称重中得到假银币比真银币轻或重的结论不一致，则该银币也不可能是假银币。程序在枚举时必须排除这种情形。例如，对样例输入中的第 2 个测试数据：

假设 A 为假银币：第 1 次称重是正确的，且假银币比真银币重，第 2 次称重是正确的，且假银币比真银币轻(已经矛盾了)，第 3 次称重是错误的；

假设 B 为假银币：第 1 次称重是正确的，且假银币比真银币重，第 2 次称重是正确的，且假银币比真银币轻(已经矛盾了)，第 3 次称重是正确的；

...

L 为假银币：第 1 次称重是正确的，且假银币比真银币轻，第 2 次称重是正确的，且假银币比真银币轻(前后一致)，第 3 次称重是正确的。

从而得出结论：L 为假银币，且比真银币轻。

以下程序中的临时变量 t 就是为了避免前后两次得出假银币轻重不一致而设置的变量。

代码如下：

```

#include <stdio.h>
#include <string.h>
char left[3][7], right[3][7]; //天平左边和右边的银币
char result[3][6]; //每次称重的结果
int weight; //1 表示假银币重，0 表示假银币轻，2 表示暂时还没得出结论
int find( char c, char* pc ) //在字符串 pc 中查找字符 c
{
    for( ; *pc; pc++ )
        if(*pc==c) return 1;
}

```

```

    return 0;
}
//如果 c 为假银币，判断这次称重是否正确，1 为正确，0 为错误，并判断假银币或轻或重
int judge( char c, char* p1, char* p2, char* p3 )
{
    if( find(c,p1) )//在左边字符串中找到字符 c
    {
        if( !strcmp(p3,"even") ) return 0; //称重结果为 even 则是错误的
        else if( !strcmp(p3,"up") ) weight = 1; //右边 up，则假银币重
        else weight = 0;
    }
    else if( find(c,p2) )//在右边字符串中找到字符 c
    {
        if( !strcmp(p3,"even") ) return 0; //称重结果为 even 则是错误的
        else if( !strcmp(p3,"up") ) weight = 0; //右边 up，则假银币轻
        else weight = 1;
    }
    else //在左边和右边字符串中都没有找到字符 c,称重结果不为 even 则是错误的
    { if( strcmp(p3,"even") ) return 0; }
    return 1; //如果 c 为假银币，此次称重是正确的
}

int main( )
{
    int n, i; //测试数据的个数，及循环变量
    char coin, feit; //代表每枚银币的字符，及最终找到的假银币
    scanf( "%d", &n );
    while( n-- ) //处理 n 个测试数据
    {
        for( i=0; i<3; i++ ) //读入 3 次称重
            scanf( "%s%s%s", left[i], right[i], result[i] );
        for( coin='A'; coin<='L'; coin++ ) //枚举 A~L 为假银币的情形
        {
            int t = -1; //防止前后两次得到 coin 轻重不一的临时变量
            weight = 2;
            //如果 coin 为假银币，是否符合 3 次称重
            for( i=0; i<3; i++ )
            {
                if( !judge(coin, left[i], right[i], result[i]) ) break;
                else
                {
                    if( t== -1 && weight!=2 ) t=weight;
                    else if( weight!=2 && t!=weight ) break;
                    //前后两次得到的假银币轻重不一样，也是错误的。
                }
            }
            if( i>=3 ) //如果 coin 为假银币符合 3 次称重，则已经找到假银币
            { feit = coin; break; }
        }
        printf( "%c is the counterfeit coin and it is ", coin );
        if( weight==1 ) printf( "heavy.\n" );
    }
}

```

```

        else printf("light.\n");
    }
    return 0;
}

```

**思考 4.2:** 在本题中, 如果某次称重的结果为“even”, 则左右盘中的银币均为真银币。如果能先排除这些银币, 则能减少一些枚举次数。读者不妨试着按照这种思路改写上述代码。

#### 例 4.7 自我数(Self Numbers)

**题目来源:**

Mid-Central USA 1998

**题目描述:**

1949 年, 印度数学家 D.R.Kaprekar 发现了一类叫做自我数(self number)的数。对于任一正整数  $n$ , 定义  $d(n)$  为  $n$  加上  $n$  的每一位数字得到的总和。

例如,  $d(75) = 75 + 7 + 5 = 87$ 。

取任意正整数  $n$  作为出发点, 你可以建立一个无穷的正整数序列  $n, d(n), d(d(n)), d(d(d(n))), \dots$ 。

例如, 如果你从 33 开始, 下一个数字就是  $33 + 3 + 3 = 39$ , 再下一个是  $39 + 3 + 9 = 51$ , 再下一个是  $51 + 5 + 1 = 57, \dots$ 。如此便产生一个整数数列:

33, 39, 51, 57, 69, 84, 96, 111, 114, 120, 123, 129, 141, ……

数字  $n$  被叫做整数  $d(n)$  的生成器。在如上的数列中, 33 是 39 的生成器, 39 是 51 的生成器, 51 是 57 的生成器, 等等。

有些数字有多于一个生成器, 如 101 有两个生成器, 91 和 100。而一个没有生成器的数字则称做自我数(self number)。100 以内的自我数共有 13 个: 1, 3, 5, 7, 9, 20, 31, 42, 53, 64, 75, 86, 和 97。

**输入描述:**

此题无输入。

**输出描述:**

输出所有小于或等于 1000000 的真的自我数, 以升序排列, 并且每个数占一行。

**样例输入:**

此题无输入。

**样例输出:**

```

1
3
5
7
9
... <— 这里有许多自我数
9949
9960
...

```

**分析:**

最容易想到的思路是: 枚举 1~1000000 之间的每个数  $n$  产生的  $d(n)$ , 用一个 bool 型的数组 **self** 记下来(如果 **self**[ $i$ ] 的值为 0, 则  $i$  为自我数, 否则如果 **self**[ $i$ ] 的值为 1 则  $i$  不是自我数; 初始时, **self**[ $i$ ] 为 0)。

具体方法为: 从  $n=1$  开始, 因为 **self**[1] 为 0, 即 1 是自我数, 所以输出 1, 接着产生  $d(1)=2$ ,

则将 `slef[2]` 的值设置为 1；然后因为 `self[2]` 为 1，即 2 不是自我数，所以不会输出，接着产生 `d(2)=4`，则 `slef[4]=1`；然后因为 `self[3]` 为 0，即 3 是自我数，所以要输出 3，接着产生 `d(3)=6`，则 `slef[6]=1`；...；一直到 `n=1000000` 为止。

现在的问题是，对某个数  $n$ ，如果产生了  $d(n)$ ，要不要继续产生  $d(d(n))$ ， $d(d(d(n)))$ ，...？答案是不需要的。因为对  $n$  来说，产生了  $d(n)$ ；则在后续的某次循环，会对  $n'=d(n)$ ，产生  $d(n')$ ；以及对  $n''=d(n')$ ，产生  $d(n'')$ ，...。

代码如下：

```
#include <stdio.h>
int main( )
{
    bool self[1000001] = { 0 }; //表示 1~1000000 是否为自我数,self[i]为 0 表示 i 是自我数
    int i;    //循环变量，表示 1,2,...,1000000 之间的每个数
    int sum; //累加
    int temp; //用来取出 i 各位上数字的临时变量
    for( i=1; i<=1000000; i++ )
    {
        if( !self[i] ) printf( "%d\n", i );
        temp = i; sum = i;
        while( temp ) //累加 i 各位数字和
        {
            sum += temp%10;
            if( sum>1000000 ) break;
            temp /=10 ;
        }
        if( sum<=1000000 ) self[sum] = 1;
    }
    return 0;
}
```

**例 4.8** 各位数码全为 1 的数(Ones)

题目来源：

University of Waterloo Local Contest 2001.06.02

题目描述：

给定任一整数  $n$ ， $0 < n \leq 10000$ ， $n$  不能被 2 整除，也不能被 5 整除，求一个位数最小的十进制数、每位数码都为 1，且能被  $n$  整除，输出其位数。

样例输入：

3  
7  
9901

样例输出：

3  
6  
12

分析：

题目中提到“ $n$  不能被 2 整除，也不能被 5 整除”，这一点保证本题有解。该题可以采用的枚举思路是：依次(枚举)判断 1, 11, 111, 1111, 11111, ... 能不能被  $n$  整除。现在存在的问题是，因为  $0 < n \leq 10000$ ，直接判断的话需要枚举的数会超出 `int` 整数的取值范围。例如，对样例输入数据中的  $n = 9901$ ，求得的满足要求的数位数有 12 位，已经超出了 `int` 整数的取值范围。

这里需要利用数论里的同余理论：



$(a+b)\%n = (a\%n+b\%n)\%n$ , 含义是:  $a+b$  对  $n$  取余数, 等效于  $a\%n$  再加上  $b\%n$ , 这个结果可能大于  $n$ , 所以要再取余数。

$(a*b)\%n = (a\%n*b)\%n$ , 含义是:  $a*b$  对  $n$  取余数, 等效于  $a$  对  $n$  取余数再乘以  $b$ , 这个结果可能大于  $n$ , 所以要再取余数。公式 “ $(a*b)\%n = (a\%n)*(b\%n)\%n$ ” 也是成立的。

以  $n = 7$  为例解释上述同余理论。

位数为 1 时, 余数  $\text{remainder} = 1\%7 = 1$ , 不满足要求, 即 1 不能被 7 整除。

位数为 2 时, 本题只需要得到余数, 因为  $11 = 1*10 + 1$ , 所以:

$$\text{remainder} = 11\%7 = (1*10 + 1)\%7 = ((1\%7)*10 + 1)\%7 = 4,$$

不满足要求, 即 11 不能被 7 整除。

上面这个式子可能还不是很理解, 再过渡一步就好理解了。位数为 3 时, 要求  $111\%7$ , 而前面已经求得  $11\%7$  的值为 4。所以:

$$\text{remainder} = 111\%7 = (11*10 + 1)\%7 = ((11\%7)*10 + 1)\%7 = (4*10 + 1)\%7 = 6,$$

不满足要求, 即 111 不能被 7 整除。

...

采取这样的思路, 对任意  $0 < n \leq 10000$ , 保证参与取余运算的整数都不会太大, 不会超出 `int` 型数据类型的取值范围。

另外, 再考虑一个特殊值, 当  $n$  为 1 时, 直接输出 1, 即可。

代码如下:

```
#include <stdio.h>
int main( )
{
    int n;
    while( scanf("%d", &n)!=EOF )
    {
        if( n==1 ) { printf( "1\n" ); continue; }
        int digitnum = 1; //数的位数
        //remainder 为求得的余数, 最小的, 每位数码都为 1 的十进制数是 1,
        //对 n 的余数也是 1, 所以 remainder 的初值为 1
        int remainder = 1;
        while( remainder!=0 )
        {
            digitnum++;
            remainder = (remainder*10+1)%n;
        }
        printf( "%d\n", digitnum );
    }
    return 0;
}
```

## 练习

### 4.3 围住多边形的边(Frame Polygonal Line)

#### 题目描述:

读入整数对的序列, 每对整数代表了二维平面上一个点的笛卡儿坐标, 第 1 个数是  $x$  坐标, 第 2 个数是  $y$  坐标。这个整数对序列代表了多边形的边。你的任务是画一个矩形, 把多边形的边都围起来, 并且矩形的周长最小。矩形的边分别平行于  $x$  轴和  $y$  轴。

**输入描述:**

输入文件包含了多个测试数据。每个测试数据中给出了一串点的坐标。每个点的x坐标和y坐标占一行，x和y的绝对值小于 $2^{31}$ 。测试数据的最后一行为0 0，代表这个测试数据的结束。注意(0, 0)不会作为任何一条边的顶点。空的多边形行代表输入的结束(即输入最后两行均为0 0，第1行的0 0代表最后一个测试数据的结束，第2行的0 0表示这个多边形是空的，代表输入结束)。

**输出描述:**

对每个测试数据，输出一行，为两个整数对，分别代表求得的周长最小的矩形的左下角顶点和右上角顶点的坐标。这四个整数用空格隔开。

**样例输入:**

```
12 56
23 56
13 10
0 0
12 34
0 0
0 0
```

**样例输出:**

```
12 10 23 56
12 34 12 34
```

**提示:**

枚举所有点的横坐标和纵坐标，取横坐标的最小、最大值，纵坐标的最小、最大值，即为所求矩形的边界。本题的关键是要正确区分输入数据中表示每个测试数据结束的“0 0”和表示所有输入数据结束的“0 0”。

**4.4 假币(False Coin)****题目描述:**

Gold Bar 银行得到可靠消息，得知上次他们收到的一堆共 N 枚硬币中，有一枚是假币。假币的重量跟真币不一样，真币的重量是一样的。他们有一台很简单的天平，这台天平可以测出左盘中的重量比右盘中的重量轻、重还是一样。

为了找出假币，银行把 N 枚硬币标上 1~N 的整数，这样每个整数唯一地确定了一枚硬币的序号。然后他们把相同数目的硬币放在天平的左右盘进行称重，每次称重左右盘中硬币的序号以及称重的结果都详细地记录下来。

你的任务是编写程序，根据称重的结果找出假币的序号。

**输入描述:**

输入文件包含多个测试数据。输入文件的第 1 行为一个整数 T，代表输入文件中测试数据的数目。然后是一个空行，之后是 T 个测试数据。各测试数据之间有一个空行。每个测试数据的第 1 行包含两个整数 N 和 K，用空格隔开，N 代表硬币的数目， $2 \leq N \leq 100$ ，K 代表称重的次数， $1 \leq K \leq 100$ 。接下来的 2K 行描述了这 K 次称重，连续的两行描述了每次称重。这两行的格式为：第 1 行首先是整数  $P_i$ ， $1 \leq P_i \leq N / 2$ ，表示此次称重左右盘硬币的数目，然后是放在左盘中的  $P_i$  硬币的序号，以及放在右盘中  $P_i$  个硬币的序号，所有的数值用空格隔开；第 2 行是一个符号，为 '<'， '>' 或 '='，含义是：

'<': 左盘中的重量轻于右盘中的重量。

'>': 左盘中的重量重于右盘中的重量。

'=': 两盘中的重量相等。

**输出描述:**

对每个测试数据，输出假币的序号，如果根据给定的称重无法判断出假币，则输出 0。两个测试数据的输出之间有空行。

**样例输入：**

```
2

5 3
2 1 2 3 4
<
1 1 4
=
1 2 5
=

6 4
3 1 2 3 4 5 6
<
1 1 2
=
2 1 3 4 5
<
2 4 5 2 6
>
```

**样例输出：**

```
3

0
```

**提示：**

可枚举第 1~N 枚硬币为假币的情形，如果只有一枚符合三次称重，则找到假币，如果有多枚硬币符合这 3 次称重，则无法判断。但 N 的值可以取到 200，所以这不是一种好方法。更好的方法是：1) 如果天平平衡，则左右秤盘中的硬币被标记为真币，且不再改变；2) 天平不平衡时，轻盘中各币被标记“轻”一次，重盘中各币被标记“重”一次；3) 然后扫描所有硬币，凡是被标记“轻”或“重”的次数与天平不平衡次数相等的钱币被重点怀疑。只有一个，则其必定为假，一个以上，则无法判断；4) 如果 K 次称量中天平没有不平衡过，则没有被标记真硬币的钱币被怀疑，只有一个，就是假币，一个以上，也是无法判断。

#### 4.5 积木(Blocks)

**题目描述：**

Donald 想给他的小侄子送礼物。Donald 是一个传统的人，他给他的小侄子选择了一套积木。这套积木共 N 个，每个积木都是一个立方体，长宽高都是 1 英寸。Donald 想把这些积木放到一个长方体里，用牛皮纸包装起来拖运，请问，Donald 至少需要多大的牛皮纸？

**输入描述：**

输入文件的第 1 行为一个整数 C，代表测试数据的数目。每个测试数据占一行，为一个正整数 N，表示需要包装的积木数目。N 不超过 1000。

**输出描述：**

对每个测试数据，输出占一行，为包装这 N 个积木需要牛皮纸的最小面积，单位为平方英寸。

**样例输入：**

```
3
```

**样例输出：**

```
30
```

9  
26  
100

82  
130

提示:

题目的意思是要使得  $N$  个积木“堆满”整个长方体，不能有“空隙”。

设  $m$  为小于等于  $N$  的三次方根的最大整数。假设长方体的长宽高分别为  $a$ 、 $b$ 、 $c$ ；枚举长方体的长度  $a$ ， $a$  的取值从 1 到  $m$ (含  $m$ )；对  $a$  的每个取值，枚举长方体的宽度  $b$ ， $b$  的取值从  $a$  开始，最大不能使得  $a \times b \times b > N$ ；对  $a$  和  $b$  的每一对取值，如果  $N \%(a \times b)$  不为 0，则跳过，否则  $c = N/(a \times b)$ ，从而求长方体的表面积。在枚举过程取最小的。

$N = 26$  时枚举过程如图 4.5 所示。 $a$  取值为 1： $b$  取值为 1 时， $c$  为 26，此时表面积为 106； $b$  取值为 2 时， $c$  为 13，此时表面积为 82。按照上述方法枚举时， $a$ 、 $b$ 、 $c$  的其他取值都不能使得 26 块积木堆满一个长方体，因此最小面积为 82。

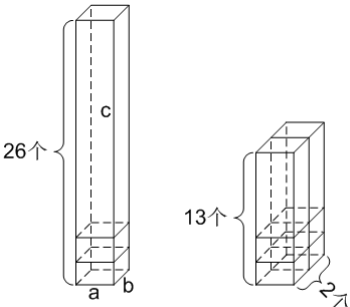


图4.5 积木

4.6 关灯游戏增强版(Extended Lights Out)

题目描述:

5 行 6 列按钮组成的矩阵，每个按钮下面有一盏灯。当按下一个按钮，该按钮和相邻 4 个按钮(上、下、左、右)的灯状态变反(如果是开着的，则关闭；如果是关闭的，则开起)。例如，在图 4.6(a)中，如果作了“X”标记的按钮按下后，则各灯的状态如图 4.6(b)所示。(在该图中，有阴影的表示灯是开着的)

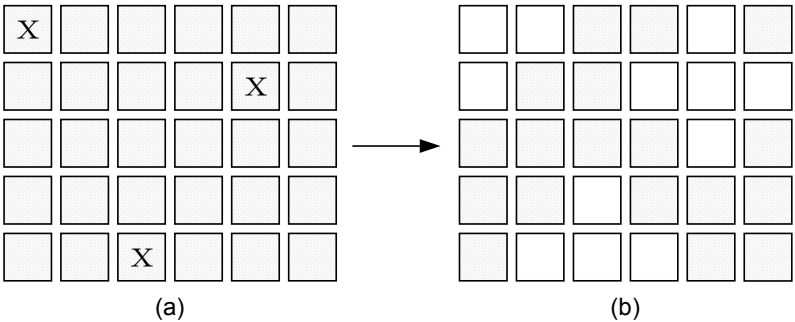


图4.6 关灯游戏增强版

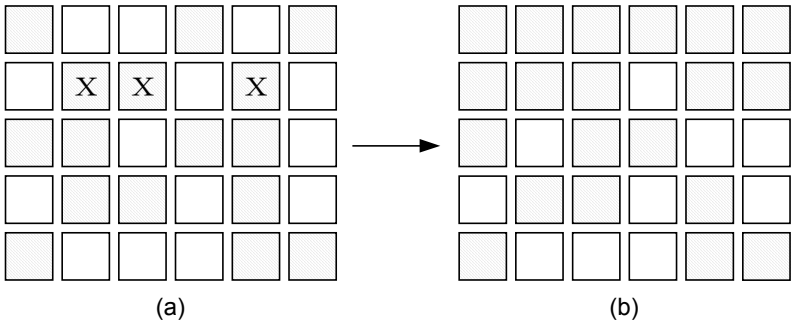


图4.7 关灯游戏增强版：按下一个按钮会取消另一个按钮按下的效果

游戏的目的是，从任意的初始状态出发，按下某些按钮使得所有灯都关闭。编写程序实现这一目的。

注意，按下一个按钮可能会取消另一个按钮按下的效果。如图 4.7 所示，按下第 2 行第 3 列和第 5 列的按钮后，第 2 行第 4 列的按钮的灯，由关变成开，然后又由开变成关。

另外请注意：

1. 按钮按下的顺序不会影响最终的效果。
2. 如果一个按钮按下两次，则第2次按下的效果只是取消了第1次按钮按下的效果，没有意义，所有没有哪个按钮需要按下2次。
3. 如图4.6所示，要使得第1行的灯全关闭，只需要按下第2行中对应的按钮即可，重复这一过程，可以使得前面4行的灯全部关闭。同理，通过按下第2~6列的按钮，可以使得1~5列灯全部关闭。

#### 输入描述：

输入文件的第1行为一个正整数  $n$ ，表示测试数据的个数。每个测试数据占5行，每行有6个整数，这些整数用空格隔开，取值为0或1，0表示初始时灯是关闭的，1表示初始时灯是开着的。

#### 输出描述：

对每个测试数据，首先输出一行："PUZZLE #m"，其中  $m$  为测试数据的序号。然后输出5行，每行为6个整数，用空格隔开，取值也为0或1。这里的0和1含义跟上面的含义不一样，1表示该按钮要按下，0表示不按。

#### 样例输入：

```
2
0 1 1 0 1 0
1 0 0 1 1 1
0 0 1 0 0 1
1 0 0 1 0 1
0 1 1 1 0 0
0 0 1 0 1 0
1 0 1 0 1 1
0 0 1 0 1 1
1 0 1 1 0 0
0 1 0 1 0 0
```

#### 样例输出：

```
PUZZLE #1
1 0 1 0 0 1
1 1 0 1 0 1
0 0 1 0 1 1
1 0 0 1 0 0
0 1 0 0 0 0
PUZZLE #2
1 0 0 1 1 1
1 1 0 0 0 0
0 0 0 1 0 0
1 1 0 1 0 1
1 0 1 1 0 1
```

#### 提示：

- 1) 首先对任意一初始状态，解是唯一的。
- 2) 要使得第1行灯全部关闭，可以通过按下第2行相应的按钮来实现，因此依次按下2~5行的按钮，可以使得前面4行的灯全部关闭，但这时第5行可能还有些灯是开着的。所以这种方法行不通，原因是第1行的按钮没有按下。
- 3) 第1行6盏灯，按下与否一共有  $2^6 = 64$  种可能。按下第1行后，为了使得第1行的灯全部关闭，第2行各按钮的按下与否就确定下来了；同样为了使得第2行的灯全部关闭，第3行各按钮的按下与否也就确定下来了；一直到第5行，其按法及各灯的状态也确定下来了。
- 4) 枚举第1行6盏灯的64种按法，当某种按法使得第5行的灯全部关闭，则找到解了。因为矩阵为  $5 \times 6$  的，规模很小，所以尽管需要枚举很多种情况，但也不会超时。

## 第 5 章 模拟

现实中有些问题难以找到公式或规律来求解，只能按照一定的步骤不停的“模拟”下去，最后才能得到答案。对于这样的问题，用计算机来求解是十分合适的，只要让计算机模拟人在解决此问题时的行为即可。这种求解问题的思想，可以称为“**模拟**”。模拟也是求解竞赛题目时经常采用的方法。本章介绍模拟算法的思想，并着重通过竞赛题目讲解在应用模拟方法时需要注意的问题。

### 5.1 模拟的基本思路

本节通过一道简单的竞赛题目总结可以采用模拟方法求解的题目所具备的特点，以及模拟方法的实现。

#### 例 5.1 醉酒的狱卒(The Drunk Jailer)

##### 题目来源：

Greater New York 2002

##### 题目描述：

某个监狱有一排、共  $n$  间牢房，一间挨一间。每间牢房关着一名囚犯，每间牢房的门刚开始时都是关着的。

有一天晚上，狱卒厌烦了看守工作，决定玩一个游戏。游戏的第 1 轮，他喝了一杯酒，然后沿着监狱，把所有牢房的门挨个挨个打开；游戏的第 2 轮，他又喝了一杯酒，然后沿着监狱，把编号为偶数的牢房的门关上；游戏的第 3 轮，他又喝了一杯酒，然后沿着监狱，对编号为 3 的倍数的牢房，如果牢房的门开着，则关上，否则打开；...；狱卒重复游戏  $n$  轮。游戏结束后，他喝下最后一杯酒，然后醉倒了。

这时，囚犯才意识到他们牢房的门可能是开着的，而且狱卒醉倒了，所以他们越狱了。  
给定牢房的数目，求越狱囚犯的人数。

##### 输入描述：

输入文件的第 1 行为一个正整数，表示测试数据的个数。每个测试数据占一行，为一个整数  $n$ ， $5 \leq n \leq 100$ ，表示牢房的数目。

##### 输出描述：

对每个测试数据所表示的牢房数目  $n$ ，输出越狱的囚犯人数。

##### 样例输入：

2  
5  
100

##### 样例输出：

2  
10

##### 分析：

在本题中， $n$  轮游戏过后，哪些牢房的门是开着的，并无规律可循。但这个游戏的规则和过程都很简单：游戏有  $n$  轮，第  $j$  轮游戏是将编号为  $j$  的倍数的牢房状态变反，原来是开着的，则关上，原来是关着的，则打开， $j = 1, 2, 3, \dots, n$ 。如图 5.1 所示。这些规则和过程用程序能较容易地实现，所以适合采用“模拟”的思路求解。

具体实现时可以定义一个一维数组，每个元素表示对应牢房的状态，初始为 1，表示牢房

门是锁着的。如图 5.1 所示。模拟  $n$  轮游戏过程：第  $j$  轮时，改变牢房编号为  $j$  的倍数的牢房状态，为 0 则改为 1，为 1 则改为 0。 $n$  轮游戏过后，统计状态为 0 的牢房数即可。

|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | n   |
|-------|---|---|---|---|---|---|---|---|---|-----|-----|
| 初始    | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... | 1   |
| 第1轮游戏 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | ... |
| 第2轮游戏 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | ... | ... |
| 第3轮游戏 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | ... | ... |

图5.1 醉酒的狱卒

代码如下：

```
#include <stdio.h>
int main( )
{
    int kase, n;          //测试数据的个数，及牢房的个数
    int prison[101];     //n 个牢房(编号从 1~n)的状态,1 表示锁着的
    int i, j;             //循环变量
    scanf( "%d", &kase );
    for( i=0; i<kase; i++ )
    {
        scanf( "%d", &n );
        for( j=1; j<=n; j++ ) prison[j] = 1; //初始时各牢房都是锁着的
        for( j=1; j<=n; j++ ) //游戏进行 n 轮
        {
            int tmp = j;
            while( tmp<=n )
            {
                prison[tmp] = (prison[tmp]==1) ? 0 : 1;
                tmp += j;
            }
        }
        int num = 0; //游戏结束后,牢房门开着的牢房的数目
        for( j=1; j<=n; j++ )
            if( !prison[j] ) num++;
        printf( "%d\n", num );
    }
    return 0;
}
```

通过这道题目的分析，我们可以看出，能够采用模拟方法求解的题目大多带有游戏性质。求解问题的关键是理解游戏的规则和过程，在用程序实现时用适当的数据(形式或结构)表示题目的状态，然后按照游戏规则模拟游戏过程。

## 5.2 模拟约瑟夫环

约瑟夫环问题的版本很多，也有很多典故。例 5.2 模拟了出列游戏，即约瑟夫环；例 5.3

及两道练习题分别从不同的角度研究约瑟夫环问题。

**例 5.2** 出列游戏：n 个人围成一圈，第 1 个人从 1 开始报数，报数报到 m 的人出列；然后又从下一个人从 1 开始报数；重复 n-1 轮游戏，每轮游戏淘汰 1 个人，最后剩下的人就是胜利者。模拟该游戏，输出依次出列的位置及最后的胜利者。

如图 5.2 所示。当  $n = 8$ ,  $m = 4$  时，图(a)~(g)演示了 7 轮游戏过程，依次出列的位置是：4 8 5 2 1 3 7，最后的胜利者是 6 号。图中方框里的数字表示这 8 个人的序号，空白的方框表示已出列的位置，方框旁边的数字表示报数过程。

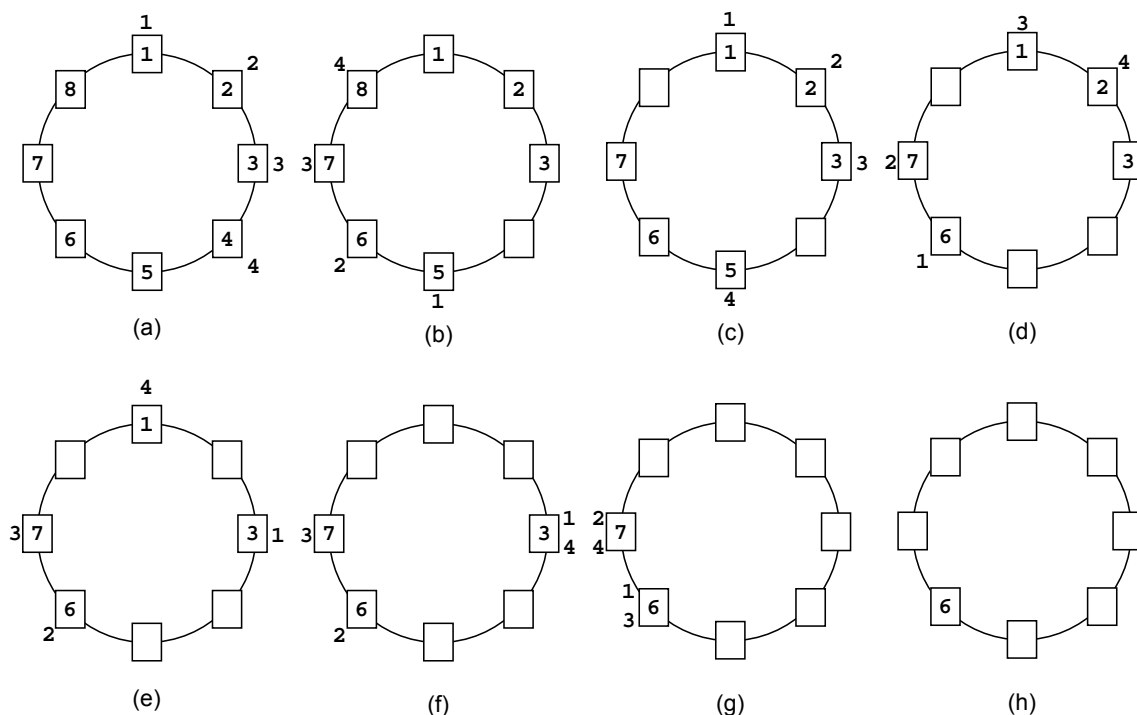


图5.2 模拟约瑟夫环

### 分析：

以  $n = 8$ ,  $m = 4$  分析该出列问题。8 个人参加该游戏，需要进行 7 轮，因为每轮淘汰一个位置出列。在程序模拟出列问题时，只需要模拟 7 轮游戏过程即可，用循环变量  $r$  来控制。

要表示 8 个人的序号，可以用一维数组  $a$  来存储 8 个位置上的号码。为了符合人们的习惯，只使用  $a[1] \sim a[8]$ ，因此数组长度为 9。

该出列游戏过程中依次出列的位置可以用图 5.3 来表示。图中 3 个变量  $i$ 、 $j$ 、 $r$  的含义分别为：

变量  $r$ ：用来表示游戏是第几轮，并且是通过该变量来控制游戏结束的。

变量  $i$ ：标明每次报数是由哪个位置上的人报出来的(注意要跳过已经出列的位置)。

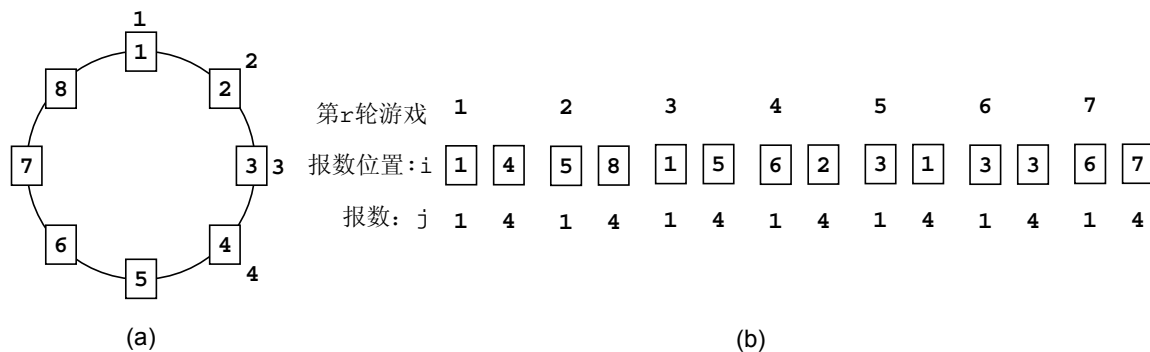
变量  $j$ ：实现报数，从 1 报数到 4，再变成 1，...

看似很简单的出列问题，在模拟时要注意以下 3 个问题：

- 1) 模拟报数过程，从 1 开始报数，达到 4 后(对应位置要出列)，又从 1 开始报数。因此需要对 4 进行取模运算。变量  $j$  用来记录报数过程报出来的数，每次继续报数本来应该是  $j=(j+1)\%4$ ，但是  $(j+1)\%4$  的范围是 0~3，我们希望  $j$  取到 1~4，所以正确的式子是： $j=(j+1-1)\%4+1$ （详细分析请参考例 1.5），即： $j=j\%4+1$ 。
- 2) 需要记录每一个报数是由哪个人报出来的，变量  $i$  来表示这个人的序号。同样每次继续报数应该在  $i$  对 8 取余后加 1，即： $i=i\%8+1$ 。



- 3) 在报数过程中, 要跳过已经出列的位置。实现方法是: 初始时, 数组 **a** 的每个元素的值为它的下标; 每出列一个位置, 将该元素的值置为 0; 在报数过程中, 如果某个位置对应的数组元素值为 0, 则跳过该位置(*i* 自增 1, *j* 保持不变)。7 轮游戏过后, 数组元素的值不为 0 的位置就是最终的胜利者。

图5.3  $n=8, m=4$ 时依次出列的位置

代码如下:

```
#include <stdio.h>
int main( )
{
    const int n = 8;
    const int m = 4;
    int a[9] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
    int r = 1; //第 r 轮
    int i; //i 指向所有的位置(跳过已经出列的位置)
    int j; //j 累加到 4 然后重新累加
    printf( "the failures are: \n" );

    //本来应该是 j=(j+1)%m,但是(j+1)%m 的范围是 0~3
    //我们希望 j 取到 1~4, 正确的式子为: j=j%m+1
    for( i=1, j=1; r<=7; i=i%n+1, j=j%m+1 )
    {
        while( a[i]==0 ) { i = i%n+1; } //跳过已经出列的
        if( j%m==0 )
        {
            printf( "a[%d]=%d\n", i, a[i] );
            a[i] = 0;
            j = 0;
            r = r+1;
        }
    }
    for( i=0; i<n; i++ )
    {
        if( a[i+1]!=0 ) printf( "the last winner is : %d\n", a[i+1] );
    }
    return 0;
}
```

该程序的输出结果如下:

the failures are:

```

a[4]=4
a[8]=8
a[5]=5
a[2]=2
a[1]=1
a[3]=3
a[7]=7
the last winner is : 6

```

### 例 5.3 网络拥堵解决方案(Eeny Meeny Moo)

#### 题目来源:

University of Ulm Local Contest 1996

#### 题目描述:

你肯定经历过很多人同时使用网络，网络变得很慢很慢。为了彻底解决这个问题，Ulm 大学决定采取突发事件处理方案：在网络负荷高峰期，将公平地、系统地切断某些城市的网络连接。德国的城市被随机地标上  $1 \sim n$  的序号。比如 Freiburg 城市的序号为 1，Ulm 城市的序号为 2，Karlsruhe 城市的序号为 3 等等，这些序号顺序纯粹是随机的。然后随机地选择一个数  $m$ 。首先切断第 1 个城市的网络连接，然后间隔  $m$  个序号，切断对应的城市，如果超出范围，则取模，并且忽略已经被切断网络连接的城市。例如，如果  $n=17$ ， $m=5$ ，被切断网络连接的城市依次为：[1,6,11,16,5,12,2,9,17,10,4,15,14,3,8,13,7]。

本题的目的是，希望最后被切断网络连接的城市是 Ulm。对于给定的  $n$  值， $m$  的值必须很仔细地选择，使得 2 号城市(即 Ulm 城市的序号)是最后被选中切断网络连接的城市。 $m$  值应该如何选？

你的任务是编写程序，读入  $n$  的值，求  $m$ ，使得 Ulm 城市最后被选择切断网络连接。

#### 输入描述:

输入文件包含多个测试数据。每个测试数据占一行，为一个整数  $n$ ， $3 \leq n < 150$ ，代表该国城市的个数。如果  $n$  的值为 0，则表示输入结束。

#### 输出描述:

对输入文件中的每个测试数据，输出求得的  $m$  值。

#### 样例输入:

```

8
9
0

```

#### 样例输出:

```

11
2

```

#### 分析:

这道题也是模拟约瑟夫环问题，只不过不是求最后的胜利者，而是给定  $n$ ，要使得最后的胜利者为 2，求  $m$ 。

与例 5.2 不同的是，这里的约瑟夫环问题首先淘汰的是编号为 1 的城市，然后是编号为  $m+1$  的城市，…。例如，样例输入中第 1 个测试数据， $n=8$ ，选择  $m$  为 11 时，如图 5.4 所示，依次被切断网络的城市为：1、5、3、4、8、6、7，最终剩下的城市是 2 号城市，因此正确的输出是 11。

本题的思路是借用例 5.2 的方法，定义函数 Joseph 实现  $m$  和  $n$  取任意值的约瑟夫环问题。在主函数中读入城市个数，从 1 开始枚举  $m$ ，直到某个  $m$  能使得该约瑟夫环问题的最后胜利者为 2 号城市，这个  $m$  值就是题目要求的结果。

注意： $m$  的值不一定小于  $n$ ，正如样例输入/输出中的第 1 个测试数据所示。

**思考 5.1：** 对给定的  $n$ ， $m$  的值是不是唯一的？

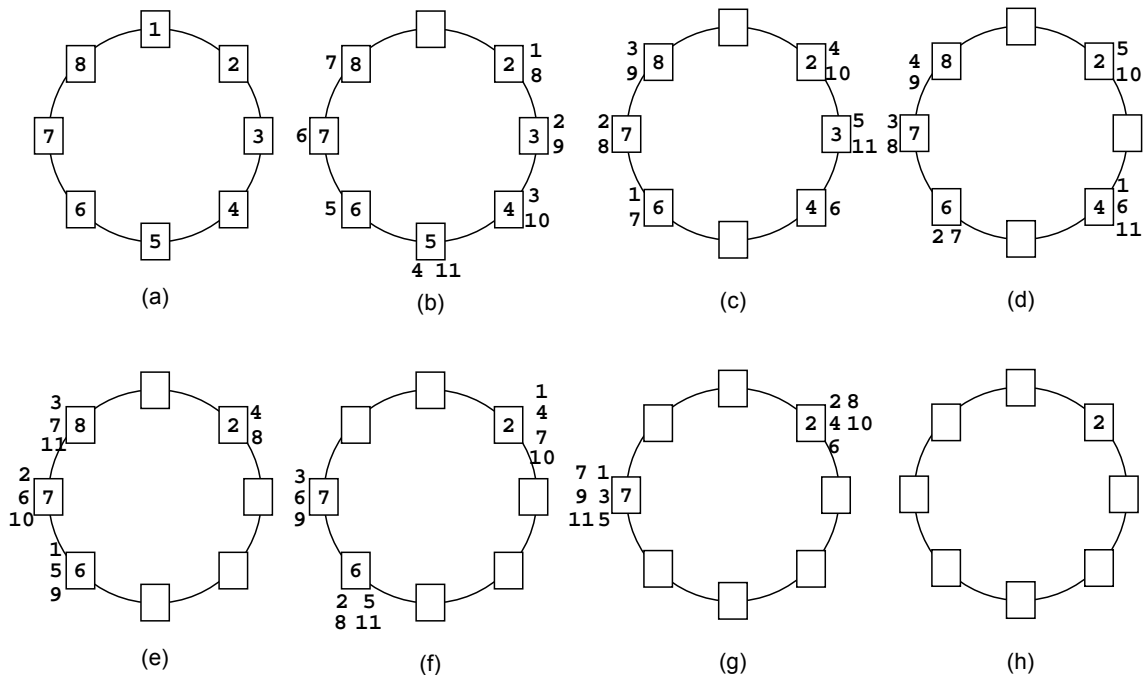


图5.4 另一个约瑟夫环( $m=8$ ,  $n=11$ )

代码如下：

```
#include <stdio.h>
#include <string.h>
int cities;    //读入的城市个数
int circle[160]; //城市的编号，第 i 个城市的编号为 i，某城市被淘汰后，对应的元素置为 0
int temp[160];  //临时
bool Joseph( int n, int m ) //选择 m 时是否能使得 2 号城市最后被切断网络
{
    int r = 1;
    int i, j;
    circle[1] = 0; //第 1 个城市首先被淘汰
    for( i=2, j=1; r<=n-2; i=i%n+1, j=j%m+1 ) //剩余 n-2 轮游戏
    {
        while( circle[i]==0 ) { i = i%n+1; } //跳过已经被切断的
        if( j%m==0 )
        {
            if( i==2 ) return false; //如果将要被切断的城市是 2 号城市，提前结束
            circle[i] = 0;
            j = 0;
            r = r+1;
        }
    }
    if( circle[2]!=0 ) return true;
    else return false;
}

int main( )
```

```

{
    int i;
    while( scanf("%d", &cities) )
    {
        if( cities==0 ) break;
        for( i=0; i<=cities; i++ )
            circle[i] = i;
        memcpy( temp, circle, sizeof(circle) );
        int m;
        for( m=1; ; m++ )    //不停地选择 m，直到某个 m 满足题目要求
        {
            memcpy( circle, temp, sizeof(circle) );
            if( Joseph(cities, m) )
                break;
        }
        printf( "%d\n", m );
    }
    return 0;
}

```

## 练习

### 5.1 约瑟夫环问题(Joseph)

#### 题目描述:

约瑟夫问题是大家所熟知的。如果你不知道这个问题的话，这里可以简短地描述：给定  $n$  个人编号依次为 1, 2, ...,  $n$ ，站成一圈，循环报数，报数到  $m$  的人将依次被处决，而且只有最后剩下的一个人可以得救。约瑟夫可以聪明地为自己选择最后剩下的那个位置，以使自己得救，因此才有机会告诉我们这件事。例如， $n = 6$ ,  $m = 5$  时，依次被处决的人是：5, 4, 6, 3, 2，只有第 1 个人可以被得救。

假设有  $k$  个好人和  $k$  个坏人，而且在这个圆圈里，前  $k$  个人是好人，而后  $k$  个人是坏人。你必须选取最小的  $m$  值，使得所有的坏人先被处决掉。

#### 输入描述:

输入文件包含若干行，每行都是一个数， $k$ ， $0 < k < 14$ 。输入文件的最后一行为 0，表示输入结束。

#### 输出描述:

对输入文件中的每个  $k$  值，输出对应的  $m$  值。

#### 样例输入:

3  
4  
0

#### 样例输出:

5  
30

### 5.2 另一个约瑟夫环问题(Yet Another Josephus Problem)

#### 题目描述:

$n$  个人围成一圈玩约瑟夫游戏，约瑟夫本人的序号为  $p$ 。游戏的玩法是第 1 个人从 1 开始报数，报数为  $m$  的人被淘汰掉，下一个人又从 1 开始报数，如此直到剩下一个人为止，这个人

就是最终的胜利者。如图 5.5 所示,  $n=8$ ,  $m=4$ , 假设约瑟夫本人是处在位置 1 的, 则依次淘汰 4、8、5、2 后, 接下来要淘汰的是约瑟夫本人。如果约瑟夫不想被淘汰, 他有一次选择机会, 他选择 6 号代替他, 从而这次淘汰的是 6 号(注意, 此后是从 6 号的下一个位置, 即 7 号开始报数, 而不是 1 号的下一个位置)。继续游戏。游戏过后最后剩下的是约瑟夫本人。

已知  $n$ 、 $m$  和  $p$ , 问约瑟夫应该选择哪个人代替他被淘汰, 才能使得他是最后的胜利者?

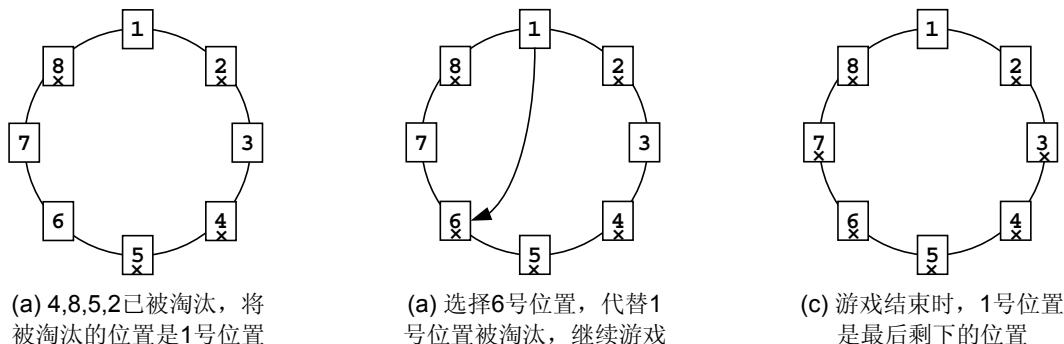


图5.5 另一个约瑟夫环问题

#### 输入描述:

输入文件包含多个(但不超过 100 个)测试数据。每个测试数据占一行, 包含 3 个整数:  $n$ 、 $m$  和  $p$ 。  $n$  表示圆圈中人的个数,  $1 \leq n \leq 1000$ 。  $m$  表示每轮报数, 报数为  $m$  的人被淘汰掉。  $1 \leq m \leq 1000000$ 。  $p$  表示约瑟夫最初在圆圈中的位置序号, 位置序号是从 1 开始标记的,  $1 \leq p \leq n$ 。输入最后一行为 3 个 0, 表示输入结束。

#### 输出描述:

对输入文件中的每个测试数据, 输出用来替换约瑟夫的那个人的序号。如果不必选择某个人来替换约瑟夫(即位置  $p$  本来就是最后剩下的位置), 则输出约瑟夫自己的序号。

#### 样例输入:

```
8 4 1
1000 1 1
0 0 0
```

#### 样例输出:

```
6
2
```

## 5.3 游戏的模拟

ACM/ICPC 很多题目取材于一些经典游戏, 通过对这些游戏的规则进行简化来构造题目。本节分析 4 道这种类型的题目。

### 例 5.4 三子棋游戏(Tic Tac Toe)

#### 题目来源:

University of Waterloo Local Contest 2002.09.21

#### 题目描述:

Tic Tac Toe 是一个小孩玩的游戏, 在一个  $3 \times 3$  的棋盘中进行。游戏有两个玩家。其中一个玩家(用字母字符 “X” 表示)先走棋, 在一个没有被占用的网格位置中放置一个 X, 然后另一个玩家(用字母字符 “O” 表示), 在一个没有被占用的网格中放置一个 O。

这两个玩家交替地放置 X 和 O, 直到棋盘的网格都被占用了, 或者某个玩家的棋子占据了整条线(水平、垂直或者对角线)。

游戏开始时棋盘是空的，用 3 行 3 列共 9 个字符 “.” 表示。如果 X 玩家走棋，则在相应位置上放置 X；如果是 O 玩家走棋，则在相应位置上放置 O。下面的棋盘表明从游戏开始直到 X 玩家最后赢得比赛的一系列走棋过程。

```
...   X..   X.O   X.O   X.O   X.O   X.O   X.O
...   ...   ...   ...   .O.   .O.   OO.   OO.
...   ...   ...   ..X   ..X   X.X   X.X   XXX
```

你的任务是读入棋盘状态，问可不可能是一个有效的三子棋棋盘，也就是说是否存在一系列走棋，能到达该棋盘状态。

#### 输入描述：

输入的第一行是整数 N，表示测试数据的个数，接下来有  $4 \times N - 1$  行，表示 N 个棋盘格局，每两个棋盘格局之间用空行隔开。

#### 输出描述：

对每个棋盘格局，如果是一个有效的三子棋格局，则输出 yes，否则输出 no。

#### 样例输入：

```
2
X.O
OO.
XXX
```

```
O.X
XX.
OOO
```

#### 样例输出：

```
yes
no
```

#### 分析：

假设读入的棋盘格局中字符 “X” 和字符 “O” 的数目分别为 xcount 和 ocount。另外，如果棋盘中某行、某列、主对角线或次对角线都为某玩家的字符，则该玩家赢得了游戏。以下情形是不合法的棋盘格局：

- 1) ocount > xcount，因为玩家 X 总是先走棋；
- 2) xcount > ocount + 1，玩家 X 顶多比玩家 O 多走一步棋；
- 3) 玩家 X 和玩家 O 都赢得了游戏；
- 4) 玩家 O 赢得了游戏，但 xcount 不等于 ocount；
- 5) 玩家 X 赢得了游戏，但 xcount 等于 ocount。

注意以上第 4 和第 5 种情形，如果是玩家 O 赢得了游戏，则合法的情形是玩家 O 和玩家 X 走棋的步数一样，而如果是玩家 X 赢得了游戏，则合法的情形是玩家 X 走棋步数比和玩家 O 走棋步数多 1 步。因此，样例输入中的第 2 个棋盘格局不是一个合法的格局。

将棋盘格局读入到一个二维字符数组中，然后通过遍历该二维数组，统计字符 “X” 和字符 “O” 的数目，以及判断玩家 X 或玩家 O 是否赢得了游戏，再按照上述规则判断即可。

#### 代码如下：

```
#include <stdio.h>
int N;           //测试数据的个数
char g[3][4];    //读入的棋盘格局
int xcount, ocount; //棋盘字符 X 和字符 O 的数目

//判断棋盘中某行、某列、主对角线或次对角线是否等于字符 c
//如果存在返回 1，否则返回 0
```

```

int win( char c )
{
    int i, j;
    for( i=0; i<3; i++ )
    {
        for( j=0; j<3 && g[i][j]==c; j++ ); //判断行
        if( j==3 ) return 1;
        for( j=0; j<3 && g[j][i]==c; j++ ); //判断列
        if( j==3 ) return 1;
    }
    for( i=0; i<3 && g[i][i]==c; i++ ); //判断主对角线
    if( i==3 ) return 1;
    for( i=0; i<3 && g[i][2-i]==c; i++ ); //判断次对角线
    if( i==3 ) return 1;
    return 0;
}

int main( )
{
    int i, j; //循环变量
    scanf( "%d", &N );
    while( N-- )
    {
        scanf( " %s %s %s", g[0], g[1], g[2] );
        xcount = ocount = 0;
        for( i=0; i<3; i++ ) //统计棋盘中字符 X 和字符 O 的个数
        {
            for( j=0; j<3; j++ )
            {
                if( g[i][j] == 'X' ) xcount++;
                if( g[i][j] == 'O' ) ocount++;
            }
        }
        if ( //判断是否是一个合法的棋盘格局
            ocount > xcount || xcount > ocount + 1
            || win('X') && win('O')
            || win('O') && xcount != ocount
            || win('X') && xcount == ocount
        )
        {
            printf( "no\n" ); continue;
        }
        printf( "yes\n" );
    }
    return 0;
}

```

### 例 5.5 扫雷游戏(Mine Sweeper)

题目来源:

University of Waterloo Local Contest 1999.10.02

题目描述:

扫雷游戏是在  $n \times n$  的网格内进行的，其中藏有  $m$  颗地雷，这些地雷分布在不同的位置。游戏者不停地点开网格中的位置。如果点开了地雷，则引爆地雷，游戏失败。如果点开了没有地雷的位置，则显示一个  $0 \sim 8$  之内的整数，表示这个位置的 8 个相邻位置中藏有地雷的位置的数目。图 5.6 显示了某次游戏中的几个步骤:

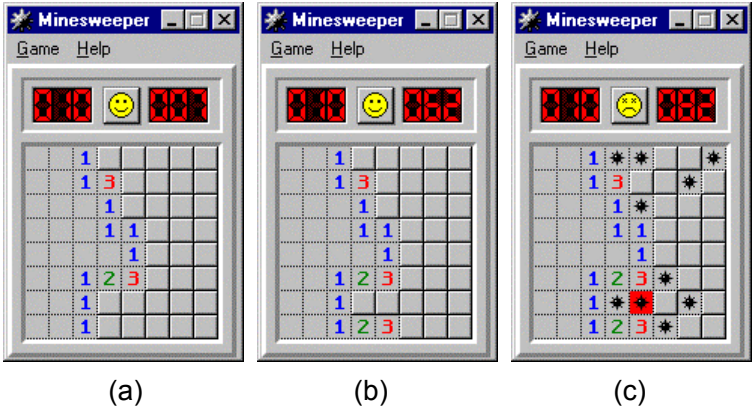


图 5.6 扫雷游戏

在图 5.6 中， $n$  为 8， $m$  为 10。图中空白的位置代表整数 0，凸起的位置表示还没点开，类似于 “\*” 号的符号代表地雷。图(a)表示点开了部分位置。从图(a)到图(b)，游戏者已经点开了 2 个位置，都没有点到地雷；但在图(c)中，游戏者就没那么幸运了，他点中了藏有地雷的位置(7,5)，游戏失败。如果游戏者将所有没有地雷的位置都点开，只有  $m$  个藏有地雷的位置没点开，则游戏成功。

你的任务是读入地雷分布图及游戏者的点击信息，输出对应的游戏地图。

输入描述:

输入文件包含了多个测试数据。每个测试数据的第 1 行为一个正整数  $n$ ， $n \leq 10$ ，表示该扫雷游戏的地图是  $n \times n$  大小的。接下来  $n$  行描绘了地雷的位置。每一行有  $n$  个字符，每个字符为 “.” 或 “\*”，其中 “.” 表示没有地雷的位置，“\*” 表示地雷的位置。接下来又是  $n$  行，每行  $n$  个字符，每个字符为 “X” 或 “.”，其中 “X” 表示已经点开的位置，“.” 表示没有点开的位置。例如，对样例输入中的测试数据，2 个  $8 \times 8$  行所描绘的地图分别对应于图 5.6 中的图(a)和(b)。

输入数据一直到文件尾。

输出描述:

对每个测试数据，输出对应的地图，每个位置都用正确的符号填充：已经点开并且没有地雷的位置用  $0 \sim 8$  之内的整数表示，注意，如果某个位置藏有地雷且被点开了，则将所有地雷的位置都用 “\*” 号表示；对没有点开的其他位置都用 “.” 表示。

每两个测试数据对应的输出之间有一个空行。

样例输入:

```
8
...*. .*
.....*
....*...
.....
.....
.....
.....*..
```

样例输出:

```
001.....
0013....
0001....
00011...
00001...
00123...
001.....
```



```

...**.*.          00123...
.....*..
XXX.....
XXXX....
XXXXX...
XXXXX...
XXXXX...
XXXXX...
XXXXX...
XXX.....
XXXXX...

```

#### 分析:

这是一道很有意思的题目，模拟的是扫雷游戏。输入的是标明地雷位置的地图，及游戏者已经点开的位置，要求输出显示给游戏者看的地图。

首先要根据输入的地图(即第一个 **n** 行所描绘的地图)，统计每个位置的 8 个相邻位置上地雷的数目，可以设计一函数来实现。对位置  $(i, j)$  来说，它的 8 个相邻位置从左上角位置开始按顺时针顺序依次为： $(i-1, j-1)$ 、 $(i-1, j)$ 、 $(i-1, j+1)$ 、 $(i, j+1)$ 、 $(i+1, j+1)$ 、 $(i+1, j)$ 、 $(i+1, j-1)$ 、 $(i, j-1)$ 。下面的代码中，函数 `ww( char s[][max], int i, int j )` 用于统计  $(i, j)$  位置周围 8 个相邻位置上的地雷数。但在本题中，并非需要统计所有位置，只需要统计已点开过且没有地雷的位置。在统计过程中，还可判断是否引爆了地雷。

然后要根据统计的结果输出显示给游戏者看的地图：如果没有引爆了地雷，则点开过的位置显示其周围 8 个位置上地雷总数，未点开的位置显示“.”；如果引爆了地雷，则所有地雷都输出“\*”号，其他位置的处理跟没有引爆地雷时的处理是一样。例如图 5.6(c)就引爆了地雷，该图对应的输出为：

```

001**.*
0013.*.
0001*..
00011..
00001..
00123*..
001**.*
00123*..

```

注意，本题要求在两个测试数据之间输出空行，言下之意就是除最后一个测试数据外，每个测试数据的输出之后有一个空行。如果在每个测试数据的输出之后都输出一个空行，得到的评判结果是格式错误。

#### 代码如下:

```

#include <stdio.h>
#include <string.h>
#define max 11

//统计(i, j)位置周围 8 个相邻位置上的地雷数
int ww( char s[][max], int i, int j )
{
    int count = 0;
    if( s[i-1][j-1]=='*' ) count++;
    if( s[i-1][j]=='*' ) count++;
    if( s[i-1][j+1]=='*' ) count++;

```

```

        if( s[i][j+1]=='*' ) count++;
        if( s[i+1][j+1]=='*' ) count++;
        if( s[i+1][j]=='*' ) count++;
        if( s[i+1][j-1]=='*' ) count++;
        if( s[i][j-1]=='*' ) count++;
        return count;
    }
    int main( )
    {
        int n;
        char minepos[max][max]; //表示雷的位置, '*'号表示雷, '.'号表示不是雷
        char played[max][max]; //点开的位置用'x'表示, 没点开的位置用'.'号表示
        int mines[max][max]; //周围 8 个位置上雷的个数
        int number = 1; //测试数据的序号, 用来控制输出空行
        int i, j;
        int flag; //引爆地雷的标志变量, flag 为 0 表示没有引爆, 为 1 表示引爆

        while( scanf( "%d", &n )!=EOF )
        {
            if( number!=1 ) printf( "\n" );
            number++;

            memset( minepos, 0, sizeof(minepos) );
            memset( played, 0, sizeof(played) );
            memset( mines, 0, sizeof(mines) );

            for( i=0; i<n; i++ ) scanf( "%s", minepos[i] );
            for( i=0; i<n; i++ ) scanf( "%s", played[i] );

            flag = 0;
            //判断是不是引爆了地雷, 并统计每个位置周围 8 个位置上的雷的个数
            for( i=0; i<n; i++ )
            {
                for( j=0; j<n; j++ )
                {
                    if( played[i][j]=='x' ) //点开过
                    {
                        if( minepos[i][j]!='*' ) //不是雷
                            mines[i][j] = ww( minepos, i, j );
                        else flag = 1; //是雷, 点开了, 被炸
                    }
                    else continue;
                }
            }
            for( i=0; i<n; i++ ) //输出
            {
                for( j=0; j<n; j++ )
                {
                    if( !flag ) //没有引爆了地雷, 未点开的位置都是.号
                    {

```

```

        if( played[i][j]=='.' ) printf( "." ); //未点开, 输出 “.” 号
        else printf( "%d", mines[i][j] ); //点开了, 输出周围地雷数
    }
    else //引爆了地雷
    {
        if( minepos[i][j]=='*' ) printf( "*" ); //显示所有地雷(不管有没有点开)
        else if( played[i][j]=='.' ) printf( "." ); //未点开, 输出 “.” 号
        else printf( "%d", mines[i][j] ); //点开了, 输出周围地雷数
    }
}
printf( "\n" );
}
}
return 0;
}

```

### 例 5.6 弹球游戏(Linear Pachinko)

题目来源:

Mid-Central USA 2006

题目描述:

这道题目起源于弹球游戏。但与传统的弹球机器不同的是, 在本题中, 一个弹球游戏机器是包含一个或多个以下字符的序列: 孔("." )、地板("\_")、墙壁("|")、山峰("^")。在弹球游戏机器中, 一个墙壁(或山峰)永远不会与另一个墙壁(或山峰)相邻。

该游戏的玩法是: 在机器的上方随机地抛下弹球。如果弹球能通过孔, 那么弹球最终穿过机器; 如果弹球落到地板上方则停下来; 如果弹球落到山峰的左边, 则弹球反弹, 通过所有连续的地板直到掉到孔里去, 或者出了机器的边界, 或者撞到墙壁或山峰则停下来; 如果弹球落到山峰的右边, 结果类似; 如果弹球抛到墙壁的上方, 则分别以 50% 的概率做落到山峰左、右边一样类似的处理。

本道题要求解的是, 如果弹球随机的从机器的上方抛下(随机的意思就是从每个字符位置上垂直抛下的机会均等), 那么弹球最终能通过孔和出边界的几率是多少?

例如, 考虑如图 5.7 所示的机器, 其中的数字表明字符的位置, 并不是机器的一部分。

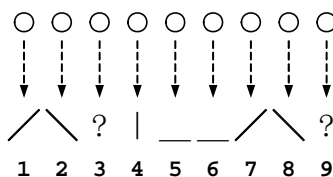


图5.7 弹球游戏

当在字符上方抛下弹球时, 弹球通过孔或出边界的几率分别为: 1=100%, 2=100%, 3=100%, 4=50%, 5=0%, 6=0%, 7=0%, 8=100%, 9=100%。因此最终对整个机器, 在机器上方随机抛下弹球, 弹球通过孔和出边界的几率为以上几率的平均值, 即为 61.111%。

输入描述:

输入文件包含一个或多个测试数据, 每个测试数据表示一个弹球游戏, 包含 1~79 个字符, 占一行。输入文件最后一行为字符"#", 表示输入文件的结束。

输出描述:

对每个弹球游戏, 精确地计算随机抛下弹球后, 弹球通过孔和出边界的几率并输出。每个

弹球游戏的输出占一行，对求得的几率(百分比)精确到整数(舍弃小数部分)。

**样例输入：**

```
/\ . | _ _ /\ .
_ . _ /\ _ | . _ _ /\ . /\ _
...
_ _
_ . /\ .
#
```

**样例输出：**

```
61
53
100
0
100
```

**分析：**

这也是一道很有趣的题目，模拟的是弹球游戏。

每个弹球游戏的字符序列中包含的字符只有有限的 5 种。对这 5 种字符的处理方法是：

- ① 对字符"."，以 100%的概率通过孔。
- ② 对字符"\_", 弹球会停下来，则通过孔或出边界的概率为 0%。
- ③ 对山峰的左边"/", 则反弹，是否通过孔或出边界或停下来，要观察左边的字符序列：如果左边第一个字符为"."，则以 100%的概率通过孔；如果为字符"|"或字符"\", 则停下来，通过孔(或出边界)的概率为 0；如果为字符"\_", 则继续判断左边的字符。如果左边的字符序列判断完毕还没通过孔或停下来，则出边界，概率为 100%。
- ④ 同样对山峰的右边 "\", 也会反弹，是否通过孔或出边界或停下来，要观察右边的字符序列，处理方法与③类似。
- ⑤ 而对字符"|", 则分别以 50%的概率做③和④的处理。

在程序中，可以用 if 结构或 switch 结构按上述分析处理字符序列中的每个字符，得到的概率和再除以字符数就是题目要求的概率。

**代码如下：**

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char ch[80]; //弹球游戏中的字符序列
    int i, j;
    while( scanf("%s", ch) )
    {
        if( ch[0]=='#' ) break;
        int prob = 0; //概率
        int len = strlen(ch); //字符序列的长度
        for( i=0; i<len; i++ )
        {
            if( ch[i]=='.' ) prob += 100;
            else if( ch[i]=='/' )
            {
                for( j=i-1; j>=0; j-- )
                {
                    if( ch[j]=='.' ) //遇到孔的位置，则通过孔
                    { prob += 100; break; }
                    else if( ch[j]=='|' || ch[j]=='\' ) break; //遇到墙壁或山峰，则停下来
                }
            }
            if( j<0 ) prob += 100; //出左边界
        }
    }
}
```

```

    }
    else if( ch[i]=='\\' )
    {
        for( j=i+1; j<len; j++ )
        {
            if( ch[j]=='.' ) //遇到孔的位置，则通过孔
            { prob += 100; break; }
            else if( ch[j]=='|' || ch[j]=='/' ) break;
        }
        if( j>=len ) prob += 100;
    }
    else if( ch[i]=='|' )
    {
        for( j=i-1; j>=0; j-- )//以 50%的几率相当于 '/'
        {
            if( ch[j]=='.' )
            { prob += 50; break; }
            else if( ch[j]=='|' || ch[j]=='\\' ) break;
        }
        if( j<0 ) prob += 50;

        for( j=i+1; j<len; j++ ) //以 50%的几率相当于 '\\'
        {
            if( ch[j]=='.' )
            { prob += 50; break; }
            else if( ch[j]=='|' || ch[j]=='/' ) break;
        }
        if( j>=len ) prob += 50; //出右边界
    }
    //else if( ch[i]=='_' ) //'_'字符不用处理
}
prob /= len;
printf( "%d\n", prob );
}
return 0;
}

```

### 例 5.7 分糖果的游戏(Candy Sharing Game)

题目来源:

Greater New York 2003

题目描述:

N 个学生围成一圈坐着，面向老师（老师位于中心）。每个学生手头上刚开始都有偶数块糖果。每轮游戏：老师一吹哨子，每个学生将他的糖果的一半分给他右边相邻的学生；N 个学生分糖果完毕后，如果某个学生手头上的糖果数为奇数，则由老师再给一块糖果凑成偶数块。当所有学生的糖果数一样，则游戏结束。要求编写程序，输出游戏进行的轮数，以及最终每个学生手头糖果的块数。

输入描述:

输入文件描述了多次游戏(即输入文件中包含多个测试数据)。每次游戏的数据都单独占一行,首先是一个整数  $N$ ,表示学生的人数,接下来是  $N$  个偶数,代表初始时  $N$  个学生手上的糖果数目(逆时针顺序排列)。输入文件最后一行为 0,表示输入结束。

#### 输出描述:

对每次游戏,输出游戏进行的轮数,以及游戏结束后每个学生手上的糖果数。

#### 样例输入:

```
6
36
2
2
2
2
2
2
4
2
4
6
8
0
```

#### 样例输出:

```
15 14
4 8
```

#### 注意:

游戏会在有限次内结束,因为(设每轮游戏过后  $N$  个学生中拥有糖果的最大数目为  $\max$ ,最小数目为  $\min$ ):

- 1)  $\max$  不会增加;
- 2)  $\min$  不会减少;
- 3) 且任何一个糖果数多于  $\min$  的学生,他的糖果数不会减少到  $\min$ ;
- 4) 如果  $\max$  和  $\min$  不等,则至少有一个学生,他的糖果数会增加,这个学生就是糖果数最少的学生。

#### 分析:

对于给定的学生人数  $N$ ,及初始时  $N$  个学生的糖果数,最终需要进行游戏的轮数及最终每个学生的糖果数是没有规律可循的,只能进行模拟。模拟游戏的每一轮,直到  $N$  个学生的糖果数一样为止。

以样例输入中的第 1 个测试数据为例解释游戏的过程,如图 5.8 所示。图(a)中圆圈里的数字表示初始时 6 个学生的糖果数,旁边的数字表示学生的序号(从 0 开始,按逆时针顺序排列)。图(b)表示第 1 轮游戏,每个人将手上糖果数的一半分给右边的学生(虚线箭头所示);如果某个学生手头上的糖果数为奇数,则由老师再给一块糖果凑成偶数块,如图(c)所示;第 1 轮游戏过后,每人手上的糖果数如图(d)所示。

具体实现时,可以用一整型数组 `candys` 存储  $N$  个学生(序号为  $0 \sim N-1$ ,如图 5.8(a)所示)的糖果数。在模拟每一轮游戏时,要先用临时变量(设为 `half0`)保存第 0 个学生糖果数的一半,然后第  $i$  个学生的糖果数 `candys[i]`,更新为: `candys[i]/2 + candys[i+1]/2`。最后一个学生,即第  $N-1$  个学生的糖果数更新为: `candys[N-1]/2 + half0`。

在判断  $N$  个学生的糖果数是否一致时,注意只要有前后两个学生的糖果数不一致的情形,就可判断这  $N$  个学生的糖果数不一致;另外第  $N-1$  个学生右边相邻的学生是第 0 个学生,所以要取模,即要判断 `candys[i]` 与 `candys[(i+1)%N]` 是否相等。

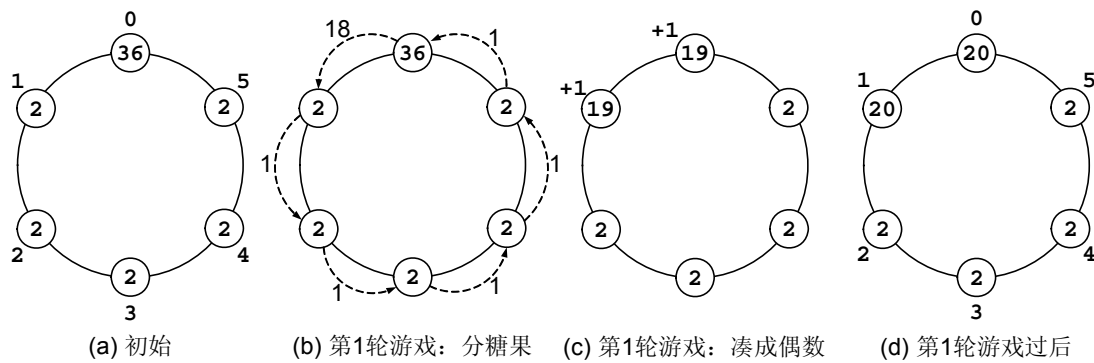


图5.8 分糖果游戏

代码如下：

```
#include <stdio.h>
int main( )
{
    int N;
    int candys[100];
    int i;
    while( scanf("%d", &N) )
    {
        if( N==0 ) break;
        for( i=0; i<N; i++ ) scanf( "%d", &candys[i] );

        int rounds = 0, sum = 0; //游戏进行的轮数，及 N 个学生糖果数的总数
        while( 1 )
        {
            //模拟每一轮游戏
            int half0 = candys[0]/2;
            for( i=0; i<N-1; i++ )
            {
                candys[i] = candys[i]/2 + candys[i+1]/2;
                if( candys[i]%2!=0 ) candys[i]++;
            }
            candys[N-1] = candys[N-1]/2 + half0;
            if( candys[N-1]%2!=0 ) candys[N-1]++;

            bool same = true; //判断 N 个学生的糖果数是否一样
            rounds++;
            for( i=0; i<N; i++ )
            {
                if( candys[i] != candys[(i+1)%N] )
                {
                    same = false; break;
                }
            }
            if( same )
            {
                for( i=0; i<N; i++ ) sum += candys[i];
                break;
            }
        }
    }
}
```

```

    }
}
printf( "%d %d\n", rounds, sum/N );
}
return 0;
}

```

## 练习

### 5.3 石头、剪刀、布(Rock, Scissors, Paper)

#### 题目描述:

Lisa 开发了一种二维网格上的游戏。初始时，网格中每个格子可能被三种生物形态之一占领：石头、剪刀和布。每天白天，水平方向或垂直方向上相邻的不同生物形态之间发生战争。在战争中，石头总是能打败剪刀，剪刀总是能打败布，布总是能打败石头。每天晚上，胜利者占领失利者的领土。

你的任务是输出  $n$  天后领土占领情形。

#### 输入描述:

输入文件第 1 行是一个整数  $t$ ，表示测试数据的数目。每个测试数据的第 1 行为 3 个整数，都不超过 100： $r$ 、 $c$  和  $n$ ， $r$  和  $c$  代表网格的行和列， $n$  代表天数。网格用  $r$  行表示，每行有  $c$  个字符。网格中的字符为 R，S 或 P，分别代表该位置为石头、剪刀和布。

#### 输出描述:

对每个测试数据，输出  $n$  天后的网格情形。每两个测试数据的输出之间有一个空行。

#### 样例输入:

```

2
3 3 1
RRR
RSR
RRR
3 4 2
RSPR
SPRS
PRSP

```

#### 样例输出:

```

RRR
RRR
RRR

RRRS
RRSP
RSPR

```

#### 提示:

注意是白天发生所有战争，并得出结果，晚上再进行领土扩张。也就是说在同一天里不能根据某些位置的战争结果继续战争。这条规则可以保证每天按任意的顺序发生战争，得到的结果是一样的。白天发生战争得到的结果，需要临时保存起来，晚上根据这个临时的结果进行领土扩张。

### 5.4 贪吃蛇游戏(The Worm Turns)

#### 题目描述:

贪吃蛇是一个很老的电脑游戏了。这个游戏有许多版本，但所有的版本都是通过操纵“蛇”在屏幕上移动，设法避免蛇碰到本身或障碍物。

我们这里将模拟一个简化的贪吃蛇游戏。游戏将在一个  $50 \times 50$  的棋盘上演示，并对棋盘中的位置进行编号。棋盘左上角位置被编号为(1,1)。蛇最初是由 20 个正方形连接而成的。这



些正方形是水平连接的或者是垂直连接的。开始时蛇水平地处在位置(25,11)到(25,30)上,蛇的头在(25,30)处。蛇可以向东部(E)、西部(W)、北部(N)或者南部(S)移动,但是自身绝不会向后移动。所以,在最初的位置时,向西(W)移动是不可能的。这样,在蛇移动改变位置时,只有两个正方形在移动,即它的头和尾。注意:蛇的头部可以移动到蛇的尾巴空出来的正方形。

题目将给出一系列的移动方向,然后蛇进行移动,直到蛇碰到本身,或者蛇跑到棋盘的边界以外,或者蛇成功地完成了所有移动。在前两种情况下,应该忽略掉碰到本身或者出界后剩余的移动。

#### 输入描述:

输入文件中包含多组测试数据。每组测试数据包括两行。第一行是标明移动的次数,为整数  $n$ ,  $n < 100$ 。当  $n=0$  时,标明输入结束。第二行包含  $n$  个字符(E、W、N 或者 S),字符和字符之间没有空格,字符序列表示移动顺序。

#### 输出描述:

对于每个测试数据都有一行输出。输出可能是以下三种情况:

The worm ran into itself on move  $m$ .

The worm ran off the board on move  $m$ .

The worm successfully made all  $m$  moves.

这里的  $m$  是程序求得的移动步数,并且一次移动为 1 步。

例如,下面样例输入/输出中第 2 个测试数据所描述的贪吃蛇游戏过程如图 5.9 所示。贪吃蛇的初始状态如图(a)所示,在经过前面 8 步(SSSWWNEN)移动后,到达图(b)所示的状态,这时第 9 步向北(N)移动一步将碰到贪吃蛇本身。

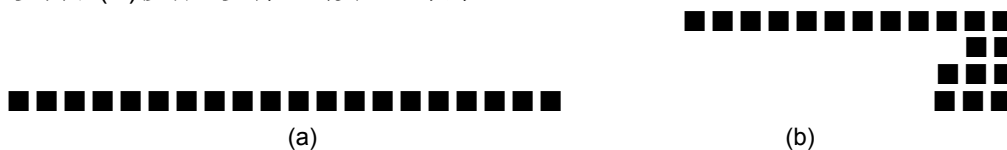


图 5.9 贪吃蛇游戏

#### 样例输入:

```
18
NWWWWWWWWWWWSSESSWS
20
SSSWWNENNNNNWWWWSSSS
30
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
13
SWWWWWWWWWWNEE
0
```

#### 样例输出:

```
The worm successfully made all 18 moves.
The worm ran into itself on move 9.
The worm ran off the board on move 21.
The worm successfully made all 13 moves.
```

### 5.5 纸牌游戏(Undercut)

#### 题目描述:

Undercut 是一种纸牌游戏,两个游戏者,每人有 5 张牌,点数从 1 到 5。每轮游戏,每个游戏者选择一张牌,并出示。如果两张牌大小相等,两人都没有得分。否则有两种情形:这两

张牌大小相差刚好为 1(这种情形称为 **undercut**)，或者两张牌点数相差超过 1。对后一种情形，点数大的游戏者得分为点数大的牌的点数。前一种情形，点数小的游戏者得分为两张牌点数和，这种情形有一个例外，当两张牌的点数为 1 和 2 时，这时点数小的游戏者得分为 6，而不是 3。每轮游戏过后，出示的牌放回手里，继续下一轮游戏。

例如，如果游戏有 5 轮，A 游戏者按顺序出牌为 5, 3, 1, 3, 5, B 游戏者按顺序出牌为 3, 3, 3, 3, 4，则每轮的得分情况为：A 得 5 分；没有得分；B 得 3 分；没有得分；B 得 9 分。最终的得分：A 为 5 分，B 为 12 分。

在本题中，给出两个游戏者出的牌，计算他们的得分。

#### 输入描述：

输入文件中有多个测试数据。每个测试数据代表一次游戏。每次游戏的第 1 行为一个整数  $n$ ， $n \leq 20$ ，表示该轮游戏两个游戏者出牌的数目。如果  $n = 0$  则代表输入结束。接下来 2 行每一行都包含  $n$  个整数，范围在  $[1, 5]$ ，代表每轮游戏两个游戏者出示的牌。第 1 行代表 A 游戏者，第 2 行代表 B 游戏者。

#### 输出描述：

对输入文件中的每个测试数据，输出一行，格式为：

A has a points. B has b points.

其中  $a$  和  $b$  的值需要你来计算。两个测试数据的输出行之间有一个空行。

#### 样例输入：

```
5
5 3 1 3 5
3 3 3 3 4
4
2 3 1 1
1 5 5 5
0
```

#### 样例输出：

```
A has 5 points. B has 12 points.

A has 0 points. B has 21 points.
```

## 5.4 其他模拟题目解析

本节再分析两道采用模拟思想求解的题目。

### 例 5.8 爬动的蠕虫(Climbing Worm)

#### 题目来源：

East Central North America 2002

#### 题目描述：

一只 1 英寸长的蠕虫在一口深为  $n$  英寸的井的底部。每分钟蠕虫可以向上爬  $u$  英寸，但必须休息 1 分钟才能接着往上爬。在休息的过程中，蠕虫又下滑了  $d$  英寸。上爬和下滑重复进行。蠕虫需要多长时间才能爬出井？不足一分钟按一分钟计，并且假定只要在某次上爬过程中蠕虫的头部到达了井的顶部，那么蠕虫就完成任务了。初始时，蠕虫是趴在井底的(即高度为 0)。

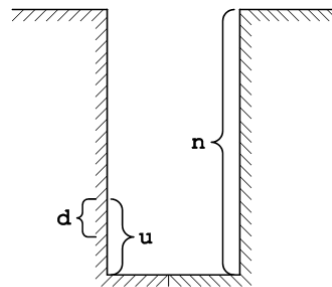


图5.10 爬动的蠕虫

#### 输入描述：

输入文件包含多个测试数据。每个测试数据占一行，为 3 个正整数  $n$ ,  $u$ ,  $d$ ，其中  $n$  是井

的深度,  $u$  是蠕虫每分钟上爬的距离,  $d$  是蠕虫在休息的过程中下滑的距离。假定  $d < u$ ,  $n < 100$ 。  
 $n=0$  表示输入数据结束。

#### 输出描述:

对输入文件中的每个测试数据, 输出一个整数, 表示蠕虫爬出井所需的时间(分钟)。

#### 样例输入:

```
10 2 1
20 3 1
0 0 0
```

#### 样例输出:

```
17
19
```

#### 分析:

题目的意思可以用图 5.10 来表示。蠕虫爬动的过程和判断蠕虫是否爬出井的依据都是很简单, 但到底需要多少分钟才能爬出井是不知道的, 本题适合用“模拟”思路求解。

在本题中, 整个模拟过程是通过一个永真循环实现的。在永真循环里, 先是上爬一分钟, 蠕虫的高度要加上  $u$ , 然后判断是否达到或超过了井的高度, 如果是则退出循环; 如果不是则要下滑  $d$  距离。也就是说, 执行一次循环, 实际上分别上爬了一分钟和下滑一分钟。是否退出循环是在上爬后判断的。

#### 代码如下:

```
#include <stdio.h>
int main( )
{
    int n, u, d;    //井的深度, 蠕虫每秒钟上爬和下滑的距离
    int time, curh; //所需时间, 蠕虫当前的高度
    while( 1 )
    {
        scanf( "%d%d%d", &n, &u, &d );
        if( !n ) break;
        curh = 0, time = 0;    //当前高度及所花时间
        while( 1 )
        {
            curh += u;    //每跳一次, 上升 u 距离
            time++;
            if( curh >= n ) break;
            curh -= d;    //休息时滑下 d 距离
            time++;
        }
        printf( "%d\n", time );
    }
    return 0;
}
```

#### 例 5.9 遍历迷宫(Maze Traversal)

##### 题目来源:

University of Waterloo Local Contest 1996.09.28

##### 题目描述:

迷宫导航是人工智能领域一个常见的问题。迷宫中有走廊和墙壁, 机器人可以通过走廊, 但不能穿过墙壁。

**输入描述:**

输入数据文件包含多个测试数据。

每个测试数据的第一行是两个整数：**M N**，表示迷宫的大小，其中 **M** 表示迷宫的行数，**N** 表示迷宫的列数，这两个整数的大小都不超过 **60**。

接下来有 **M** 行，每行有 **N** 个字符，描绘了这个迷宫。其中空格字符表示走廊，星号字符表示墙壁。迷宫没有出口。

接下来一行是两个整数，表示机器人的初始位置。初始时，机器人是朝北的。

测试数据中剩余的数据表示机器人接收到的指令，用字符表示，其中可能包含空格。有效的命令字符及代表的含义为：

**R**: 顺时针旋转 **90** 度。

**L**: 逆时针旋转 **90** 度。

**F**: 往前移动一步，如果前方位置为墙壁，则不移动。

**Q**: 退出程序。每个测试数据中指令序列的最后一个字符为 **Q**，此时应输出机器人当前的位置和朝向。

输入数据文件中的测试数据一直到文件尾。

**输出描述:**

对每个测试数据，输出机器人最终的位置和朝向(**N**, **W**, **S**, 或 **E**)，表示位置的行和列的整数、及表示朝向的字符用空格隔开。

**样例输入:**

```
3 3
***
* *
***
2 2
RRFLFF FFR
FF
RFFQ
7 8
*****
* * * **
* *   *
* * ** *
* * * *
*   * **
*****
2 4
RRFLFF FFR
FF
RFFQ
```

**说明:**

本题的求解要用到“第 5 章 字符及字符串的处理”的知识，读者可等到学完第 5 章后再来阅读本题。

**分析:**

本题模拟的是机器人在迷宫中根据指令进行移动或者旋转。本题的规则比较简单，在模拟

**样例输出:**

```
2 2 W
5 6 W
```

是要注意以下 3 个问题。

首先要存储迷宫，需要用二维数组，但数组中的下标是从 0 开始计数的，而题目中表示位置的行和列是从 1 开始计数的，所以需要转换。

其次，机器人旋转时，如果一直顺时针旋转，从机器人的朝向依次为北、东、南、西、北、…。在程序中，可以用整数 *d* 表示机器人的朝向，取值 0~3 分别表示北、东、南、西 4 个朝向。顺时针旋转(*d* 的值加 1)和逆时针旋转(*d* 的值减 1，或加 3)都需要取模。

最后，机器人向前移动时要判断前面的位置是否为空格，只有为空格才可以向前移动。在程序实现时，可以用两个数组 *rm* 和 *cm*，表示分别表示机器人在 4 个方向(依次为北、东、南、西)上往前移动一步后位置(行和列)的增量，这样机器人往 *d* 方向移动一步后，其行列位置增量分别为 *rm[d]*和 *cm[d]*。

代码如下：

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

char maze[80][80];    //存储迷宫
//rm 和 cm 分别表示机器人在 4 个方向(依次为北、东、南、西)上往前移动一步后
//位置(行和列)的增量
int rm[4] = { -1, 0, 1, 0 };
int cm[4] = { 0, 1, 0, -1 };
char direction[ ] = "NESW";//表示 4 个朝向的字符

int main( )
{
    int M, N; //迷宫的大小
    int i, j;  //循环变量
    int rpos, cpos, d; //机器人当前的位置及当前的朝向
    char command; //机器人接收到的指令
    while( scanf( "%d%d", &M, &N )!=EOF )
    {
        for( i=0; i<M; i++ )//读入迷宫
        {
            getchar( ); //跳过上行的回车换行键
            for( j=0; j<N; j++ )
                maze[i][j] = getchar( );
        }
        scanf( "%d%d", &rpos, &cpos );//读入机器人的初始位置
        rpos--; cpos--;
        d = 0; //初始时机器人是朝北的
        while( ( command = getchar( ) )>=0 ) //处理接收到的指令
        {
            if( command=='F' ) //往前移动一步
            {
                if( maze[ rpos+rm[d] ][ cpos+cm[d] ] == ' ' )
                { rpos += rm[d]; cpos += cm[d]; }
            }
            else if( command=='R' ) //顺时针旋转
                d = (d+1)%4;
            else if( command=='L' ) //逆时针旋转
```

```

        d = (d+3)%4;
    else if( command=='Q' )    //退出
    { printf( "%d %d %c\n", rpos+1, cpos+1, direction[d] ); break; }
    else { }
    }
}
return 0;
}

```

## 练习

### 5.6 货币兑换(Currency Exchange)

#### 题目描述:

当 Issac 在某个国家旅游的时候, 例如法国, 他要把美元兑换成法郎。汇率是一个实数, 当用美元乘以这个数时, 就得到法郎。举个例子, 当用美元兑换法郎的汇率为 4.81724 时, 10 美元能兑换 48.1724 法郎。当然, 你只能得到小数点后两位那么多的法郎, 所以要小数点后两位要四舍五入。( .005 以上要入为 .01 )。所有货币金额都要四舍五入。

有时候 Issac 的旅程跨越多个国家, 于是他要把货币兑换来兑换去。当他回家的时候, 他又要换回美元。这使他想到, 他在这些兑换过程中可能损失了或者赚到了美元。现在就要你算他到底是赚到了还是损失了。你永远要从美元开始, 以美元结束。

#### 输入描述:

输入文件中有多组测试数据! 输入数据的第一行为 N, 表示测试数据的组数。然后是一空行。

接下来就是 N 组测试数据。每两组测试数据之间有一空行。

每组测试数据的前 5 行是 5 个国家之间的汇率, 标号为 1 到 5。第 i 行表示第 i 个国家与 5 个国家的汇率。第 i 行第 j 列表示第 i 个国家与第 j 个国家的汇率。当然, 自己兑换自己汇率就是 1。第一个国家就是美国。接下来有若干行, 每一行表示 Issac 的一次旅程, 每行的格式为:

n c1 c2 ... cn m

当  $1 \leq n \leq 10$  和 c1, ..., cn 是从 2 到 5 的整数时, 他们表示 Issac 的行程。

当 n = 0 时表示输入结束, 这一行没有多余数字。

那么他的旅程就是 1 -> c1 -> c2 -> ... -> cn -> 1。

实数 m 表示一开始他带的美元。

#### 输出描述:

对应到输入文件中的 N 组测试数据, 输出也有 N 组。每组测试数据中的每次旅程对应一个输出, 表示当他回到家的时候他拥有的美元金额, 精确到小数点后两位。

每两组测试数据对应的输出块之间有一个空行。

#### 样例输入:

```

1

1 1.57556 1.10521 0.691426 7.25005
0.634602 1 0.701196 0.43856 4.59847
0.904750 1.42647 1 0.625627 6.55957
1.44616 2.28059 1.59840 1 10.4843
0.137931 0.217555 0.152449 0.0953772 1
3 2 4 5 20.00

```

#### 样例输出:

```

19.98
99.99
120.01

```

```

1 3 100.00
6 2 3 4 2 4 3 120.03
0

```

### 5.7 古怪的钟(Weird Clock)

#### 题目描述:

有一只很古怪的钟，它只有分针，刻度从 0 到 59。只有到往它的盒子里投一些特制的硬币后，它的分针才会走动。你可以选择不同的硬币。然而一旦你选择某一种硬币后，就不能用其他硬币了。每种硬币有无限多枚。每种硬币都对应到一个数目  $d(1 \leq d \leq 1000)$ ，表示当你投下这种硬币后，时钟的分针将从当前时间开始顺时针走当前时间的  $d$  倍刻度。例如，如果当前时间是 45， $d = 2$ ，则分针顺时针走 90 分钟，然后分针指向的刻度是 15。

给定初始时间  $s$ ， $1 \leq s \leq 59$ ，以及硬币的类型  $d$ ，编写程序求至少需要投多少枚这样的硬币才能使得分针指回到 0 刻度。

#### 输入描述:

输入文件包含多个测试数据，每个测试数据占一行，为两个正整数  $s$  和  $d$ 。输入文件的最后一行为两个 0，代表输入结束。

#### 输出描述:

对输入文件中的每个测试数据，输出最少的硬币数目。如果不能使得分针指回到 0 刻度，则输出 "Impossible"。

#### 样例输入:

```

30 1
59 59
59 58
0 0

```

#### 样例输出:

```

1
1
Impossible

```

#### 提示:

在用程序实现时，可以定义一状态数组 `state[60]`，`state[i]=1` 表示投若干枚给定的硬币后可以到达时刻  $i$ ；初始时，`state[i]` 为 0。对给定的  $s$  和  $d$ ，模拟投第 1 枚硬币、第 2 枚硬币，...；如果在投币过程中，重复到达了某一时刻，说明存在一个环，后面到达的时刻又会以前到达过的时刻，则输出 Impossible。当到达时刻 0 时，设置 `state[0]` 的值为 1，并输出当前所投的硬币数。

### 5.8 金币(Gold Coins)

#### 题目描述:

国王赏给他的武士金币。第一天，武士得到 1 块金币；接下来的 2 天（也就是第二天和第三天），每天得到 2 块金币；接下来的 3 天（第四天、第五天、第六天），每天得到 3 块金币；接下来的 4 天（第七、八、九、十天），每天得到 4 块金币。如此无限进行下去：接下来的  $N$  天里，每天得到  $N$  块金币，接下来的  $N+1$  天里，每天得到  $N+1$  块金币。

你的任务是给定第几天，要求武士从第 1 天到该天获得的金币总数。

#### 输入描述:

输入文件包含多组测试数据。

输入文件的第一行为一个整数  $N$ ，表示输入文件中有  $N$  组测试数据。接下来是一个空行，然后是  $N$  组测试数据。每组测试数据至少 1 行，至多 21 行，每行（除了最后一行）代表一个测试数据，为一个整数  $d$ ， $1 \leq d \leq 10000$ ，表示第几天；最后一行是 0，表示该组测试数据结

束。每两测试数据之间有一个空行。

**输出描述:**

对每组测试数据中的每个测试数据，输出一行，包括从输入文件中得到的整数，然后是空格，接着是武士获得的金币的块数。每两组测试数据的输出内容之间有空行。

**样例输入:**

2  
  
10  
6  
15  
100  
0  
  
10000  
1000  
22  
0

**样例输出:**

10 30  
6 14  
15 55  
100 945  
  
10000 942820  
1000 29820  
22 98

**提示:**

根据题目意思，在连续的N天里，武士共获得了 $N^2$ 枚金币。所以本题实际上是求数列和： $1^2 + 2^2 + 3^2 + \dots + N^2$ ，其中N是满足： $1 + 2 + 3 + \dots + N \leq d$ 的最大整数；并且如果d的值使得连续的N天后还有s天剩余，则还要额外加上这s天所获得的金币，这s天，每天将获得 $N + 1$ 枚金币。



## 第 6 章 字符及字符串处理

本章将集中讲解字符及字符串的处理。涉及到的知识和应用问题包括：字符转换与编码、回文的判断与处理、子串的处理、字符串匹配等。

### 6.1 字符转换与编码问题

字符转换和编码通常都是基于字符的 ASCII 编码值进行的。这些题目通常比较简单，可以作为字符及字符串处理的入门练习题。

#### 6.1.1 字符转换

所谓字符转换就是将字符按照某种规律转换成对应的字符。例如在例 2.15 中，把小写字母字符转换成字母表后面第 4 个字符，形成环状序列，即 'w'、'x'、'y'、'z' 分别变成 'a'、'b'、'c'、'd'。这种简单的转换可以实现简单的加密方法，如例 6.1。

**例 6.1** 曾经最难的题目(the hardest problem ever)

**题目来源：**

South Central USA 2002

**题目描述：**

凯撒大帝生活在充满危险和阴谋的时代。凯撒大帝面临最严峻的形势就是如何生存下去。为了生存，他决定设计一种密码。这种密码设计得是如此难以置信的合理，以至于不知道它的原理就无法破译。

你是凯撒军队的一个小队长。你的工作就是对凯撒发布的密文进行解码，然后告诉军队的司令官。加密的规则其实很简单：对原文中的每个字母，转换成字母表后面第 5 个字母，如原文中的字符为字母 A，则密文中对应的字符为 F。因为你的工作是解码，所有要将密文翻译成原文。

**ciphertext(密文):** A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

**plaintext(原文):** V W X Y Z A B C D E F G H I J K L M N O P Q R S T U

加密时，只有字母字符才按照上述规则进行加密。任何非字母字符保持不变，而且所有字母字符均为大写字母。

**输入描述：**

输入文件(非空)最多包含 100 组数据。每组数据为下面的格式，每组数据之间没有空行，所有的字符为大写。

每组数据由 3 行组成：

1. 首行为字符串"START";
2. 第 2 行为密文，包含的字符个数大于等于 1，小于等于 200，表示凯撒发布的密文；
3. 第 3 行为字符串"END".

最后一组数据后有"ENDOFINPUT"，表示输入结束。

**输出描述：**

对每组数据，输出一行，为你解密出来的原文。

**样例输入：**

```
START
NS BFW, JASYSX TK NRUTWYFSHJ FWJ YMJ WJXZQY TK YWNANFQ HFZXJX
END
START
IFSLJW PSTBX KZQQ BJQQ YMFY HFJXFW NX RTWJ IFSLJWTZX YMFS MJ
END
ENDOFINPUT
```

样例输出：

```
IN WAR, EVENTS OF IMPORTANCE ARE THE RESULT OF TRIVIAL CAUSES
DANGER KNOWS FULL WELL THAT CAESAR IS MORE DANGEROUS THAN HE
```

分析：

本题针对的大写字母，实现的转换是类似于例 2.15 的逆转换，即把每个字母转换成字母表前 5 个字母，形成环状序列。'F'转换成'A'，'G'转换成'B'，...，'Z'转换成'U'，'A'、'B'、'C'、'D'、'E'分别转换成'V'、'W'、'X'、'Y'、'Z'。如图 6.1 所示。

具体转换的方法见例 2.15 的分析，本题中转换的式子是：

$\text{Cipher}[i] = (\text{Cipher}[i] - 5 - 65) \% 26 + 65;$

或：

$\text{Cipher}[i] = (\text{Cipher}[i] + 21 - 65) \% 26 + 65;$

本题还要特别注意输入数据的格式，每组数据占 3 行，但只有中间一行是需要处理的，在读入数据时要跳过第 1 行和第 3 行。另外，输入数据结束是以“ENDOFINPUT”为标志的。这些都是程序中需要特别注意的地方。

代码如下：

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char Cipher[210]; //存储读入的每行字符串，每行字符串长度小于等于 200
    while( gets(Cipher) != NULL ) //一直读到文件尾
    {
        //读入"ENDOFINPUT"，处理结束
        if( strcmp(Cipher, "ENDOFINPUT")==0 ) break;
        gets( Cipher ); //读入的是密文
        int i = 0;
        while( Cipher[i]!='\0' ) //字符转换
        {
            if( Cipher[i]>='A' && Cipher[i]<='Z' )
                Cipher[i] = (Cipher[i] + 21 - 65)%26 + 65;
            i++;
        }
        puts( Cipher ); //输出原文
        gets( Cipher ); //读入"END"
    }
    return 0;
}
```

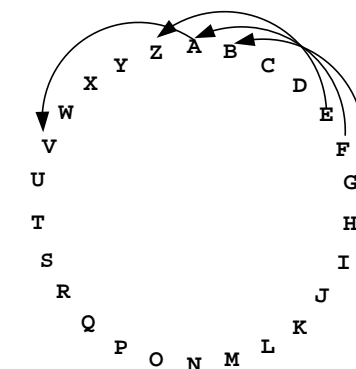


图6.1 将字母转换成前面第5个字母

例 6.2 打字纠错(WERTYU)

**题目来源:**

University of Waterloo Local Contest 2001.01.27

**题目描述:**

一种常见的打字键入错误是将键盘上的键位错按成它右侧相邻的按键,如图 6.2 所示。比如想键入“Q”却误按成“W”,想键入“J”却被误按成“K”。你的任务是对上述错误的打字方式进行正确地“解码”。



图6.2 键盘

**输入描述:**

输入文件包含若干行,每行可以包含数字,空格,以及除“A”、“Z”、“Q”外的大写字母,还有除单引号“'”外的标点符号。除此之外,也不会错按到 Tab、BackSpace、Control 等标记了单词的按键。

**输出描述:**

你的任务是将每个字母或标点符号用它左边的符号替换,输入中的空格原样输出(即空格不会键入错误)。

**样例输入:**

```
O S, GOMR YPFSU/
234567890-=WERTYUIOP[]
```

**样例输出:**

```
I AM FINE TODAY.
1234567890-QWERTYUIOP[
```

**分析:**

这道题比较简单,有两种处理方法。

第一种方法是用 **switch** 或 **if** 结构判断读取的字符,然后顺序输出在键盘上位于它前面的那个键值,有多少种输入字符就有多少个分支。由于输入的字符有很多种,所以有很多分支,比较繁琐。

第二种方法就是把所有可能误输入的字符按在键盘上的先后顺序存放到一个大的数组里。然后对输入字符串中的每个字符,在数组里进行查找,如果查找到则输出数组中左边的字符;否则(没有查找到该字符)原样输出。很明显,这是一种更好的方法。但要注意,右斜杆字符'必须用转义字符表示,即表示成“\\”。

**代码如下:**

```
#include <stdio.h>
char key[50] = "1234567890-=QWERTYUIOP[]\ASDFGHJKL;'ZXCVBNM,./";
int main( )
{
    int i,c;
    while ( (c = getchar())!=EOF )
    {
        //在字符数组 key 中查找与读入字符相等的字符
        for( i=1; key[i] && key[i]!=c; i++ )
            ;
        if( key[i] ) putchar(key[i-1]); //查找到, 输出左边的字符
        else putchar(c); //没有查找到, 原样输出
    }
    return 0;
}
```

### 6.1.2 字符编码

所谓字符编码就是将字符串中的每个字符按照编码规则换成一个数字或一串数字，或者将字符串中具有某种规律的子串转换成一串数字或其他字符等等。例如在例 2.4 中，将字母 A 编码成数值 1，...，将 Z 编码成 26；读入一个单词，最终求得整个单词的编码值。以下再举两道例题。

#### 例 6.3 Soundex 编码(Soundex)

题目来源：

University of Waterloo Local Contest 1999.09.25

题目描述：

Soundex 编码方法根据单词的拼写将单词进行分组，使得同一组的单词发音很接近。例如，“can”与“khawn”，“con”与“gone”在 Soundex 编码下是相同的。

Soundex 编码方法将每个单词转换成一串数字，每个数字代表一个字母。规则如下：

1 代表 B, F, P 或 V；

2 代表 C, G, J, K, Q, S, X 或 Z；

3 代表 D 或 T；

4 代表 L；

5 代表 M 或 N；

6 代表 R。

而字母 A, E, I, O, U, H, W 和 Y 不用任何数字编码，并且相邻的、具有相同编码值的字母只用一个对应的数字代表。具有相同 Soundex 编码值的单词被认为是相同的单词。

输入描述：

输入文件中的每行为一个单词，单词中的字母都是大写，每个单词长度不超过 20 个字母。

输出描述：

对输入文件中的每个单词，输出一行，为该单词的 Soundex 编码。

样例输入：

KHAWN  
PFISTER  
BOBBY

样例输出：

25  
1236  
11

分析：

样例输入/输出可以帮助我们分析题目的意思。比如样例输入中的第一个单词“KHAWN”，它的 Soundex 编码值之所以是“25”，是因为第一个字母“K”的编码值为“2”，而接下来的三个字母“H”、“A”和“W”都没有编码值，最后一个字母“N”的编码值为“5”。样例输入中的最后一个单词“BOBBY”，它的 Soundex 编码值之所以是“11”，是因为第一个字母“B”的编码值为“1”，第 2 个字母“O”没有编码值，之后两个字母“B”相邻，只编码成一个“1”，最后一个字母“Y”没有编码值。

从上面的分析可以看出，题目中提到的“相邻的、具有相同编码值的字母只用一个对应的数字代表”，如果具有相同编码值的字母之间间隔了若干个没有编码值的字母，则要单独编码。例如 BB 编码成“1”，“BV”也编码成“1”，而“BOB”编码成“11”。没有理解这一点，程序的处理就是错误的。

本题在处理时可以把 26 个字符的编码值(数字字符)按顺序存放到一个字符数组中，对没有编码值的字符，用“\*”号表示。然后对字符串中的每个字符，输出其对应的数字；如果后一个

字母的编码值跟前一个字母的编码值一样，则后一个字母的编码值不输出。

代码如下：

```
#include <stdio.h>
#include <string.h>
int main( )
{
    //26 个字母(对应到数组元素，下标为 0~25)对应的 Soundex 编码值
    /*号表示没有对应的编码值
    char change[27]="*123*12**22455*12623*1*2*2";
    char input[21];    //读入的单词
    int i;
    while( scanf("%s",input)!=EOF )
    {
        int len; //单词长度
        char temp; //单词中每个字母对应的编码值
        char prev = '0'; //前一个字母的编码值
        len = strlen(input);
        for( i=0; i<len; i++ )
        {
            temp = change[input[i]-'A'];
            if( temp=="*" )//第 i 个字母没有编码值
            {
                prev = '0'; continue;
            }
            if( temp==prev ) continue; //第 i 个字母的编码值同前一个字母的编码值
            printf("%c",temp);
            prev = temp;
        }
        printf("\n");
    }
    return 0;
}
```

以上程序中，变量 `prev` 的作用很关键。为了实现“相邻的、具有相同编码值的字母只用一个对应的数字代表”，需要记住前一个字母的编码值，如果当前字母的编码值和前一个字母的编码值一样，则后一个字母的编码值不输出。

#### 例 6.4 圆括号编码(Parencodings)

题目来源：

Asia 2001, Tehran (Iran)

题目描述：

令  $S=s_1 s_2 \dots s_n$  是一个正则(well-formed)的圆括号串。 $S$  可以编码成 2 种不同的形式：

- 1) 编码成一个整型序列  $P=p_1 p_2 \dots p_n$ ,  $p_i$  代表在  $S$  序列中第  $i$  个右圆括号前的左圆括号数量。(记为  $P$ -序列)
- 2) 编码成一个整型序列  $W=w_1 w_2 \dots w_n$ , 对每一个右圆括号  $a$ , 编码成一个整数  $w_i$ , 表示从与之匹配的左圆括号开始到  $a$  之间的右圆括号的数目(包括  $a$  本身)。(记为  $W$ -序列)

下面是一个例子：



的圆括号为左圆括号'('且 **left** 的值刚好为 1, 则这个左圆括号就是与第 *i* 个右圆括号匹配的左圆括号, 扫描结束。记录这个扫描过程当中扫描到的右圆括号数, 这个值就是第 *i* 个右圆括号的编码值。

以前面分析中得到的原始圆括号串为例加以解释, 圆括号下方的数字表示该圆括号在圆括号串中的序号(从 0 开始计起)。

```
( ( ( ( ) ( ) ) ) )
0 1 2 3 4 5 6 7 8 9
```

以第 7 个圆括号为例, 它是一个右圆括号')', 为匹配当前扫描到的左右圆括号所需的左圆括号数 **left** = 1。

往左遍历, 第 6 个圆括号为右圆括号')', 则 **left** 加 1 为 2。

第 5 个圆括号为左圆括号'(', **left** 减 1 为 1。

第 4 个圆括号为右圆括号')', 则 **left** 加 1 为 2。

第 3 个圆括号为左圆括号'(', **left** 减 1 为 1。

第 2 个圆括号为左圆括号'(', 且此时 **left** 的值为 1, 扫描结束。这个左圆括号'('就是第 7 个圆括号(它是右圆括号')')匹配的左圆括号。在这个扫描过程中扫描到的右圆括号的个数为 3, 所以编码值为 3。

对原始圆括号串“((( ( ) ( ) ) ) )”中的每个右圆括号都按上述方法处理, 得到最终的 W-序列为: 1 1 3 4 5。

代码如下:

```
#include <stdio.h>
#include <string.h>
int main( )
{
    int num[21] = {0}, result[21] = {0}; //读入的 P-序列, 以及转换后的 W-序列
    char parentheses[41] = {0}; //读入的 P-序列对应的正则圆括号串
    int t, n; //测试数据的个数, 以及测试数据所表示的 P-序列中整数的个数
    int i, j, k; //循环变量
    scanf( "%d", &t );
    for( i=0; i<t; i++ ) //处理 t 个测试数据
    {
        scanf( "%d", &n );
        for( j=0; j<n; j++ ) //读入第 i 个测试数据中的 n 个整数
            scanf( "%d", &num[j] );
        int temp = num[0];
        int temp1 = 0;
        for( j=0; j<n; j++ ) //将读入的 P-序列转换成对应的圆括号串
        {
            for( k=temp1; k<temp; k++ )
                parentheses[k] = '(';
            parentheses[k] = ')';
            temp1 = k+1;
            temp = num[j+1] - num[j] + temp1;
        }
        int left; //为匹配当前扫描到的左右圆括号所需的左圆括号数
        int count; /*对每个右圆括号 a, 从与之匹配的左圆括号开始
                    到 a 之间的右圆括号的数目*/
        int m=0; //将第 m 个右圆括号编码成 result[m]
```

```

for( j=0; j<strlen(parentheses); j++ )//将圆括号串转换成 W-序列
{
    if( parentheses[j]=='(' )//碰到 "(" 时开始处理
    {
        count = 1; left = 1;
        for( k=j-1; k>0; k-- )    //遍历 "(" 之前的括号情况
        {
            //当前扫描到的为 "(", 且仅需一个 "(" 时跳出, 这个 "(" 即为所需
            if( parentheses[k]=='(' && left==1 ) break;
            //当扫描到一个 "(" 时, 即可以匹配掉一个 ")", 所以 left--
            if( parentheses[k]=='(' ) left--;
            //当扫描到一个 ")" 时, 所需左圆括号数 left++
            if( parentheses[k]==')' )
            { left++; count++; }
        }
        result[m++] = count;
    }
}
for( j=0; j<n-1; j++ ) printf( "%d ", result[j] ); //输出前 n-1 个数
printf( "%d\n", result[j] );    //输出最后一个数
memset( parentheses, 0, sizeof(parentheses) );
}
return 0;
}

```

## 练习

### 6.1 字符减一(IBM Minus One)

#### 题目描述:

输入一行大写字母, 输出每个字母(在字母表里)随后的字母, 如果是 Z 则变为 A。

#### 输入描述:

输入文件的第 1 行为一个整数 n, 表示接下来字符串的个数。接下来的 n 行中, 每行为一个字符串, 包含大写字母字符(最多 50 个)。

#### 输出描述:

对输入文件中的每个字符串, 首先输出字符串的序号, 如样例输出中所示。然后输出对应字符串: 对输入字符串中的每个字符, 用字母表中随后的字母替换, 对字母 Z, 用 A 替换。

在每个测试数据的输出之后, 输出一个空行。

#### 样例输入:

```

2
HAL
SWERC

```

#### 样例输出:

```

String #1
IBM

String #2
TXFSD

```

### 6.2 置换加密法(Substitution Cypher)

#### 题目描述:



置换加密算法是最简单的加密算法，其原理是将一个字母表中的字符替换成另一个字母表中的字符。这种形式的加密方法，已经存在 2000 多年的历史了。

#### 输入描述：

输入文件中第一行是原文用的字母表，第 2 行是密文用的字母表。接下来有若干行字符，每一行是待加密的原文，每一行都不超过 64 个字符。

#### 输出描述：

输出的第 1 行是密文用的字母表，第 2 行是原文用的字母表。接下来的每一行是将输入文件中对应行加密后得到的密文字符串，原文字母表中没有的字符不用替换。

#### 样例输入：

```
abcdefghijklmnopqrstuvwxyz
zyxwvutsrqponmlkjihgfedcba
Shar's Birthday:
The birthday is October 6th, but the party will be Saturday,
October 5. It's my 24th birthday and the first one in some
```

#### 样例输出：

```
zyxwvutsrqponmlkjihgfedcba
abcdefghijklmnopqrstuvwxyz
Sszi'h Brigswzb:
Tsv yrigswzb rh Oxyglyvi 6gs, yfg gsv kzigb droo yv Szgfiwzb,
Oxyglyvi 5. Ig'h nb 24gs yrigswzb zmw gsv urihg lmv rm hlnv
```

### 6.3 Quicksum 校验和(Quicksum)

#### 题目描述：

校验和是一种算法，这种算法扫描数据包并返回一个值。这种算法的思想是当数据包被改变了，校验和同样要改变，因此校验和通常用来检查传输错误，用来确认传输内容的正确性。

在本题中，你需要实现一种校验和算法，称为 Quicksum。一个 Quicksum 数据包只允许包含大写字母和空格，并且起始字符和终止字符都是大写字母。除此之外，空格和大写字母允许以任何的组合方式出现，包括连续的空格。

校验和 Quicksum 是数据包中所有字符在数据包中的位置和它的值的乘积的累加和。空格的值为 0，其他大写字母的值为其在字母表中的位置，即 A=1, B=2, ..., Z=26。

下面是 Quicksum 数据包"ACM" and "MID CENTRAL"的校验和计算方法：

ACM:  $1*1 + 2*3 + 3*13 = 46$

MID CENTRAL:  $1*13 + 2*9 + 3*4 + 4*0 + 5*3 + 6*5 + 7*14 + 8*20 + 9*18 + 10*1 + 11*12 = 650$

#### 输入描述：

输入文件包括一个或多个数据包，输入文件的最后一行为符号#，表示输入文件的结束。每个数据包占一行，每个数据包不会以空格开头或结尾，每个数据包包含 1~255 个字符。

#### 输出描述：

对每个数据包，输出一行，为它的校验和。

#### 样例输入：

```
ACM
MID CENTRAL
```

#### 样例输出：

```
46
650
```

REGIONAL PROGRAMMING CONTEST 4690  
#

## 6.4 字符宽度编码(Run Length Encoding)

### 题目描述:

你的任务是编写一个程序实现简单的字符宽度编码方法。规则如下：

任何 2~9 个相同字符构成的序列编码成 2 个字符：第 1 个字符是序列的长度，用数字字符 2~9 表示，第 2 个字符为这一串相同字符序列中的字符。超过 9 个相同字符构成的序列编码方法是先编码前面 9 个字符，然后再编码剩余的字符。

任何不包含连续相同字符的序列编码成：先是字符"1"，然后是字符序列本身，最后还是字符"1"。如果字符"1"是序列中的字符，则对每个"1"用两个字符"1"替换。

例如，字符串"12142"，编码后为"111211421"。这是因为这个字符串没有连续相同的字符，则编码后前后都是字符 1，中间是字符串本身，而字符串本身又包含了两个"1"，对每个"1"，用两个"1"替换。

输入描述:

输入文件包含若干行，每行的字符都是大小写字母字符、数字字符、空格或标点符号，没有其他字符。

**输出描述:**

对输入文件中每行进行字符宽度编码，并输出。

样例输入：

AAAAABCCCC  
12344  
AAAAAAAAAAAAAAAAAAAA

样例输出：

6A1B14C  
11123124  
9A7A

## 6.5 摩尔斯编码(P.MTHBGWB)

### 题目描述:

摩尔斯编码采用变长的点号“.”和短划线“-”序列来代表字符。实际编码时，电文中的字符用空格隔开。表 6.1 是摩尔斯编码中各字符对应的编码。

表 6.1 摩尔斯编码表

| 字符 | 编码      | 字符 | 编码      | 字符 | 编码      | 字符 | 编码      |
|----|---------|----|---------|----|---------|----|---------|
| A  | . -     | H  | . . . . | O  | - - -   | V  | . . . - |
| B  | - . . . | I  | . .     | P  | . - - . | W  | . - -   |
| C  | - . - . | J  | . - - - | Q  | - - . - | X  | - . . - |
| D  | - . .   | K  | - . -   | R  | . - . - | Y  | - . - - |
| E  | . . . . | L  | . - . . | S  | . . .   | Z  | - - . . |
| F  | . . - . | M  | - -     | T  | -       |    |         |
| G  | - - .   | N  | - .     | U  | . . -   |    |         |

注意，在上表中点号和短划线有 4 个组合没有采用。在本题中，将这四种组合分配给以下的字符：

下划线: ..--      点号: ---.  
逗号: .-.-      问号: ----

因此，电文“ACM\_GREATER\_NY\_REGION”被编码为：.-.-.-. -- ..---. .-.. -  
- .-..- : ..-- -. : -.-- :..-- :.: .- :.. --- -.。

Ohaver 基于摩尔斯编码提出了一种加密方法。这种方法的思路是去掉字符间的空格，并且在编码后给出每个字符编码的长度。例如电文“ACM”编码成“.-. --. --242”。

Ohaver 的加密(解密也是一样的)方法分为 3 个步骤:

1. 将原文转换成摩尔斯编码，去掉字符间的空格，然后把每个字符长度的信息添加在后面；
2. 将表示各字符长度的字符串反转；
3. 按照反转后的各字符长度，解释点号和短划线序列，得到密文。

例如，假设密文为“AKADTOF IBOETATUK IJN”，解密步骤如下：

1. 将密文转换成摩尔斯编码，去掉字符间的空格，添加各字符长度组成的字符串，得到

`" .-.-.-.--..--...-.---.-.-.-.-.-.232313442431  
121334242";`

2. 将字符长度字符串反转，得到“242433121134244313232”；

3. 对字符长度字符串反转后的编码字符串, 用摩尔斯编码解释该字符串, 得到原文为“ACM GREATER NY REGION”。

本题的目的是实现 Ohaver 的解密算法。

输入描述:

输入文件中包含多个测试数据。输入文件的第 1 行为一个整数  $n$ ，表示测试数据的个数。每个测试数据占一行，为一个用 **Ohaver** 加密算法加密后的密文。每个密文中允许出现的符号为：大写字母，下划线，逗号，点号和问号。密文长度不超过 100 个字符。

**输出描述:**

对输入文件中的每个密文，首先输出密文的序号，然后是冒号，空格，最后是解码后的原文。

样例输入：

2  
AKADTOF\_IBOETATUK\_IJN  
?EJHUT.TSMYGW?EJHOT

样例输出：

1: ACM\_GREATER\_NY\_REGION  
2: TO BE OR NOT TO BE?

## 6.2 回文的判断与处理

所谓回文(**palindrome**)字符串,就是从左向右读和从右向左读结果相同的字符串。回文的判断与处理经常出现在 **ACM/ICPC** 题目中。例 6.5 实现了回文的判断;例 6.6 实现了回文的构造:对于不是回文的字符串,通过在其后添加最少的字符,使其成为回文;例 6.7 是回文字符串和镜像字符串的判断。

**例 6.5** 编写程序，实现回文的判断。

判断回文的方法很简单，假设字符串长度为  $n$ ，只需依次判断字符串中第  $i$  个字符与第  $n-1-i$  个字符是否相等即可， $i=0,1,2,3,\dots,n/2$ 。

代码如下：

```
#include <stdio.h>
#include <string.h>
int huiwen( char *s )    //判断回文的函数,返回 1 表示 s 是回文,返回 0 表示 s 不是回文
{
    char *p1, *p2;
    int i, t = 1;
    p1 = s;    //p1 指向 s 中第 0 个字符
    p2 = s + strlen(s) - 1;    //p2 指向 s 中最后一个字符
    for( i=0; i<=strlen(s)/2; i++ )
```

```

    {
        if( *p1 != *p2 )
        { t = 0; break; } //对应字符不相等，提前结束判断
        p1++; p2--;
    }
    return t;
}
void main( )
{
    char str[80];
    printf( "Input a string : " );
    scanf( "%s", str ); //输入字符串
    if( huiwen(str) ) printf( "%s is a palindrome!\n", str );
    else printf( "%s is not a palindrome!\n", str );
}

```

该程序的运行示例如下：

Input a string : abcba✓

abcba is a palindrome!

#### 例 6.6 构造回文

##### 题目描述：

如果一个字符串不是回文，则可以在其后面添加一些字符，使其变成一个回文。本题的目的是，给定一个字符串 **a**，输出长度最小的字符串 **x**，**x** 添加在 **a** 的后面，并且 **ax** 为回文。

##### 输入描述：

输入文件包含多个测试数据。每个测试数据为一个字符串。字符串中只包含小写字母字符，长度不超过 100 个字符。

##### 输出描述：

对输入文件中的每个字符串 **a**，如果该字符串为回文，则输出"**a is a palindrome!**"，**a** 为输入的字符串。如果 **a** 不是回文，则输出字符串 **x**，**x** 是添加在 **a** 后面并使 **ax** 为回文的最短字符串。

##### 样例输入：

abcba

abcdc

##### 样例输出：

abcba is a palindrome!

ba

##### 分析：

设字符串 **A** 的长度为 **n**，如果 **A** 不是回文，则要构造回文，最多需要添加 **n-1** 个字符(取字符串中的前 **n-1** 个字符，按相反的顺序添加在字符串后面)。

本题要求最少需要添加多少个字符，则依次考察以下子串 **A<sub>i</sub>**：从字符串的第 **i** 个字符开始，一直到最后一个字符所组成的子串，**i = 0, 1, …, n-1**。只要第一个子串 **A<sub>i</sub>** 为回文，则需要添加的最少字符就是第 **i** 个字符前的所有字符，顺序刚好相反。

例如，对样例输入中的字符串“abcdc”，判断如下：

**i = 0** 时，子串 **A<sub>i</sub>** 为“abcdc”，不是回文；

**i = 1** 时，子串 **A<sub>i</sub>** 为“bcdc”，不是回文；

**i = 2** 时，子串 **A<sub>i</sub>** 为“cdc”，是回文，因此需要添加的长度最小字符串是“ba”，即第 2 个字符前的所有字符以相反的顺序组成的字符串；

并且不需再判断下去了。

代码如下:

```
#include <stdio.h>
#include <string.h>
//判断 s 字符串中从第 start 个字符开始共 n 个字符所组成的子串是否为回文
int judge( char *s, int start, int n )
{
    int i;
    for( i = 0; i < n/2; i++ )
    {
        if( s[start+i] != s[start+n-i-1] ) return 0;
    }
    return 1; //回文
}
int main( )
{
    char str[101];
    int i, j;    //循环变量
    int len, d;
    while( gets(str) )
    {
        len = strlen(str);
        for( i=0; i<len; i++ )
        {
            d = judge(str, i, len-i);
            //从第 0 个字符到最后 1 个字符组成的子串(就是字符串本身)为回文
            if( i==0 && d )
            {
                printf( "%s is a palindrome!\n", str );
                break;
            }
            if( d )    //从第 i 个字符到最后 1 个字符组成的子串为回文
            {
                //添加的字符为第 i 个字符前的所有字符, 顺序刚好相反
                for( j=i-1; j>=0; j--) putchar(str[j]);
                putchar('\n');
                break;
            }
        }
    }
    return 0;
}
```

### 例 6.7 镜像回文(Palindromes)

题目来源:

South Central USA 1995

题目描述:

一个规则的回文是一串由字符或数字组成的字符串, 从前往后读与从后往前读完全一样。例如, 字符串 "ABCDEDCBA" 就是一个回文, 因为从前往后读与从后往前读都是

"ABCDEDCBA"。

所谓镜像字符串，就是将字符串中的每个字符转换成它的相反字符(如果该字符存在相反字符的话)后，得到的字符串从后往前读，跟原来的字符串一样。例如，各字符的相反字符如表 6.2 所示，则"3AIAE"就是一个镜像字符串，因为"A"和"I"是它们本身的相反字符，并且"3"和"E"互为相反字符。字符串"3AIAE"中，各字符转换成其相反字符后，变成"EAIA3"，这个字符串从后往前读，就是原来的字符串。

镜像回文就是同时满足回文字符串和镜像字符串条件的字符串。例如，"ATOYOTA"就是一个镜像回文，因为这个字符串从后往前读跟原来的字符串是一样的，并且将字符串中的每个字符用它的相反字符替换，得到的字符串为"ATOYOTA"，该字符串从后往前读，跟原来的字符串也一样。当然，在这个字符串里，字符"A"，"T"，"O"和"Y"的相反字符都是它们本身。

在本题中，字符串中允许出现的有效字符和它们对应的相反字符如表 6.2 所示。

表 6.2 镜像字符串中允许出现的字符及对应的相反字符

| 有效字符 | 相反字符 | 有效字符 | 相反字符 | 有效字符 | 相反字符 |
|------|------|------|------|------|------|
| A    | A    | M    | M    | Y    | Y    |
| B    |      | N    |      | Z    | 5    |
| C    |      | O    | O    | 1    | 1    |
| D    |      | P    |      | 2    | S    |
| E    | 3    | Q    |      | 3    | E    |
| F    |      | R    |      | 4    |      |
| G    |      | S    | 2    | 5    | Z    |
| H    | H    | T    | T    | 6    |      |
| I    | I    | U    | U    | 7    |      |
| J    | L    | V    | V    | 8    | 8    |
| K    |      | W    | W    | 9    |      |
| L    | J    | X    | X    |      |      |

注意：数字'0'和字符'O'被认为是同一个字符，因此只有字符'O'才是有效的字符。

#### 输入描述：

输入文件中包含多个字符串，每行一个，每个字符串包含 1~20 个有效字符，每个字符串中都不包含无效的字符。输入数据一直到文件尾。

#### 输出描述：

对输入文件中的每个字符串，首先从第 1 列开始输出字符串本身，然后根据情况输出以下字符串中的一个：

" -- is not a palindrome." 如果这个字符串既不是回文，也不是镜像字符串。

" -- is a regular palindrome." 如果这个字符串是回文，但不是镜像字符串。

" -- is a mirrored string." 如果这个字符串不是回文，但它是镜像字符串。

" -- is a mirrored palindrome." 如果这个字符串既是回文，也是镜像字符串。

注意：请严格按照要求输出空格和字符"-"，如上述描述及样例输出中那样。另外，每个输出之后有一个空行。

#### 样例输入：

NOTAPALINDROME  
ISAPALINILAPASI  
2A3MEAS  
ATOYOTA

#### 样例输出：

NOTAPALINDROME -- is not a palindrome.  
ISAPALINILAPASI -- is a regular palindrome.  
2A3MEAS -- is a mirrored string.

ATOYOTA -- is a mirrored palindrome.

分析:

这道题其实很简单,前面例 6.5 和例 6.6 已经实现了回文字符串的判断。对镜像字符串的判断,也很简单,假设字符串长度为  $n$ ,只需依次判断字符串中第  $i$  个字符与第  $n-1-i$  个字符的相反字符是否相等即可,  $i=0,1,2,3,\dots,n/2$ 。如果同时满足回文字符串和镜像字符串,则是镜像回文。

代码如下:

```
#include <stdio.h>
#include <string.h>
char charset[36] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ123456789"; //字符集
char mirrors[36] = "A 3 HIL JM O 2TUVWXY51SE Z 8 "; //各字符对应的镜像字符
char instring[80]; //读入的字符串
char messages[4][30] = { //输出的信息
    "-- is not a palindrome.", //不是回文,也不是镜像字符串
    "-- is a regular palindrome.", //回文
    "-- is a mirrored string.", //镜像字符串
    "-- is a mirrored palindrome." }; //回文, 且是镜像字符串
char get_mirror( char ch ) //获得字符 ch 的镜像字符
{
    int i;
    for( i=0; ; i++ )
    {
        if( charset[i]==ch )
            return( mirrors[i] );
    }
}
int check_string( void ) //判断字符串: 回文、镜像字符串、镜像回文等
{
    int i, mirror, palin;
    mirror = 2; palin = 1; //mirror 表示镜像字符串, palin 表示回文字符串
    for( i = 0; i < strlen(instring); i++ )
    {
        if( instring[i] != get_mirror(instring[strlen(instring) - (i+1)]) ) //判断镜像
            mirror = 0;
        if( instring[i] != instring[strlen(instring) - (i+1)] ) //判断回文
            palin = 0;
    }
    return(palin + mirror);
}
int main(void)
{
    while( gets(instring)!=NULL )
    {
        printf( "%s%s\n\n", instring, messages[check_string()] );
    }
    return 0;
}
```

在上述代码中，`check_string` 函数的返回值为 0、1、2 或 3，含义分别表示输入的字符串为：不是回文也不是镜像字符串、回文、镜像字符串、回文且是镜像字符串。在 `main` 函数中，根据 `check_string` 函数的返回值输出二维字符数组 `messages` 中对应的信息。

## 练习

### 6.6 添加后缀构成回文(Suffidromes)

#### 题目描述：

给定两个由小写字母字符组成的字符串 `a` 和 `b`，输出长度最小的字符串 `x`，`x` 由小写字母字符组成，且满足 `ax` 和 `bx` 中有且仅有一个是回文。

#### 输入描述：

输入文件中包含多个测试数据，每个测试数据为两个字符串 `a` 和 `b`，每个字符串单独占一行，每个字符串包含 0~1000 个小写字符。

#### 输出描述：

对每个测试数据，输出占一行，为求得的字符串 `x`。如果多个 `x` 满足题中的条件，输出字母序最前一个。如果不存在满足条件的 `x`，则输出 "No Solution."。

#### 样例输入：

```
abab
ababab
abc
def
```

#### 样例输出：

```
Baba
ba
```

**提示：**回文的构造要用到例 6.6 的方法。

## 6.3 子串处理

字符串中任意个连续的字符组成的字符序列称为该字符串的**子串**。有的时候，从字符串中抽取不连续的字符所组成的字符序列，也可以看成是字符串的子串。下面例 6.8 是连续字符组成的子串，例 6.9 是不连续字符组成的子串。

需要说明的是，子串处理中的问题大多都属于子串匹配的问题，其中涉及到的算法(如 KMP 算法)比较复杂，超出了本教材的难度。本节例题和练习题的求解不需要采用这些算法。

### 例 6.8 字符串的幂(Power Strings)

#### 题目来源：

University of Waterloo Local Contest 2002.07.01

#### 题目描述：

给定两个字符串 `a` 和 `b`，定义 `a*b` 为两个字符串的连接。例如，设 `a` 为字符串 "abc"，`b` 为字符串 "def"，则 `a*b = "abcdef"`。如果将字符串的连接理解为乘法，则字符串的非负整数次幂递归地定义为： $a^0 = ""$  (空串)， $a^{(n+1)} = a*(a^n)$ 。

#### 输入描述：

输入文件包含多个测试数据，每个测试数据占一行，为一个字符串 `s`，`s` 中的字符都是可显示的。`s` 的长度至少为 1，最多不超过 1,000,000 个字符。输入文件最后一行为字符 ".", 代表



输入结束。

**输出描述:**

对输入文件中的每个字符串  $s$ ，输出满足以下条件的最大整数  $n$ :  $s = a^n$ ， $a$  为某个字符串。

**样例输入:**

Abcd  
aaaa  
ababab  
.

**样例输出:**

1  
4  
3

**分析:**

题目中虽然没有直接要求  $a$  为  $s$  的子串，但如果  $a$  不是  $s$  中由连续字符组成的子串，则不可能存在整数  $n$ ，使得  $s = a^n$ 。另外，对任意字符串  $s$ ，满足条件的子串  $a$  及整数  $n$  总是存在的，因为如果  $a$  为字符串  $s$  本身，则  $s = a^1$  总是成立的。

本题的求解思路是：依次判断字符串  $s$  是否能分成  $i$  个等分， $i$  从 2 开始计起并递增；如果  $s$  能分成  $i$  个等分，但某些等分不相同（如图 6.3(a)），则  $i$  递增 1，再判断是否能等分；如果每等分都相同，则再对第 1 等分按照上述思路进行细分，如果不能再细分成更小的、相等的等分，则要求的  $n$  就是此时  $i$  的值，如图(b)所示；如果对第 1 等分能再细分成更小的、相等的等分，则再进行细分，如图(c)所示。在图(c)中，求得的  $n = 9$ 。

由于字符串的长度最长可达 1,000,000 个字符，所以上述细分过程必须快速地结束，下面的代码采取的思路是：如果字符串  $s$  能细分成  $i$  等分，且这  $i$  等分都相同，则在对第 1 等分再细分时，从判断能否细分成  $i$  等分开始判断能否细分成  $i$ 、 $i+1$ 、...等分。如图(d)所示， $s$  分成 5 等分且各等分都相等，则对 1 等分不需要判断是否能分成 2~4 等分，这是因为如果该等分能细分成 2~4 等分、且各等分相同，则在前面对整个字符串的细分过程就能判断出这种情形。又如  $s$  为 "ababababababababab"，分成 2 等分时，不相等；分成 3 等分后，各等分均相等，此时对第 1 等分 "ababab"，尽管能分成 2 等分，但这 2 等分肯定不相同，所以可以从判断能否分成 3 等分开始判断。

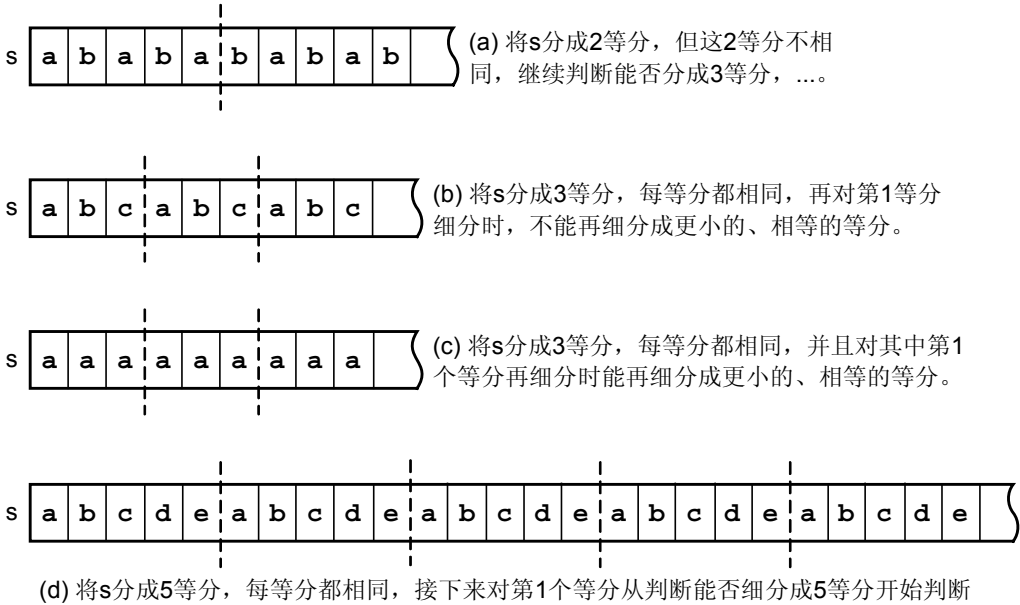


图 6.3 在  $s$  中查找满足条件的子串

注：本题的解题过程包含了分治的思想，将  $s$  字符串逐渐细分成更小的字符串，关于分治

法思想的简单介绍，可参考 9.4 节。

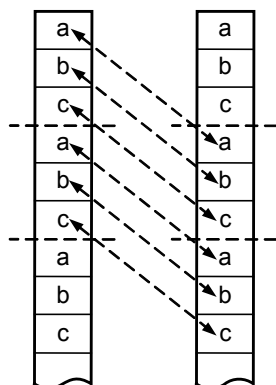


图6.4 判断字符串s的i个等分是否都相同

下面的代码在判断字符串 *s* 的 *i* 个等分是否都相同时采取的思路可以用图 6.4 来描述：每个等分长度为  $m/i$ ，依次判断第 0 个与第  $m/i$  个字符是否相同，第 1 个与第  $m/i+1$  个字符是否相同，…。直至所有的字符对都相同，则这 *i* 个等分都相同。

代码如下：

```
#include <stdio.h>
#include <string.h>
char s[2000002]; //读入的字符串 s
int main( )
{
    int i, j;    //循环变量
    int m;      //最终将 s 分成 n 等分时每等分的长度，最终求得的  $n = \text{strlen}(s)/m$ 
    int L;      //L 初始为 s 的长度，如果 s 能分成 i 个等分，则 L 的值缩小到  $L/i$ 
    while( gets(s) )
    {
        if( strcmp(s, ".") == 0 ) break;    //输入结束
        m = L = strlen(s);
        for( i=2; i<=L; i++ )
        {
            while( L%i == 0 ) //s 能分成 i 个等分，每等分长度为  $m/i$ 
            {
                L /= i;
                for( j=0; j<m-m/i; j++ ) //判断 i 个等分是否都相同
                {
                    if( s[j] != s[j+m/i] ) break;
                }
                if( j == m-m/i ) //上述 for 循环正常结束，即这 i 个等分都相同
                    m /= i;
            }
        }
        printf( "%d\n", strlen(s)/m );
    }
    return 0;
}
```

**例 6.9** 字符串包含问题(All in All)

**题目来源:**

University of Ulm Local Contest 2002

**题目描述:**

给定两个字符串  $s$  和  $t$ , 判断  $s$  是否是  $t$  的子串, 也就是说, 是否能通过从  $t$  中去掉一些字符, 使得剩余的字符构成的字符串是  $s$ 。

**输入描述:**

输入文件包含多个测试数据, 每个测试数据占一行, 为两个字符串  $s$  和  $t$ , 这两个字符串是由大小写字母字符构成的, 两个字符串之间用空格隔开。输入数据一直到文件尾。

**输出描述:**

对输入文件中的每个测试数据, 判断  $s$  是否为  $t$  的子串。

**样例输入:**

```
person compression
VERDI vivaVittorioEmanueleReDiItalia
```

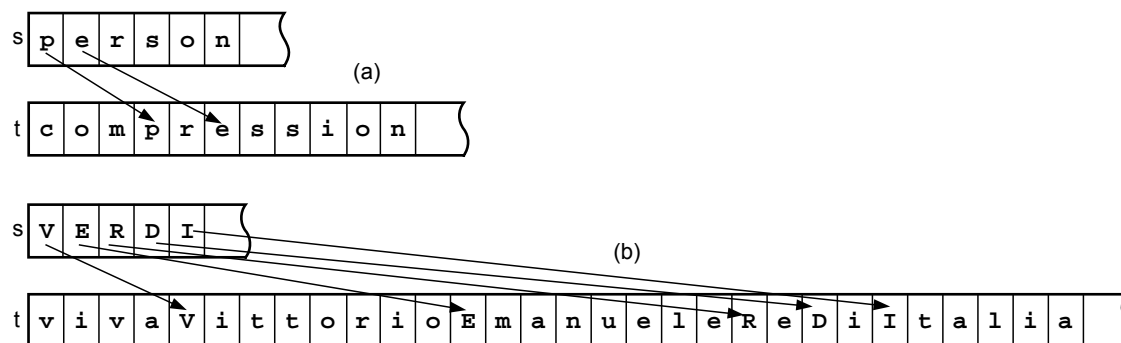
**样例输出:**

```
No
Yes
```

**分析:**

本题的思路是: 对字符串  $s$  的第 0 个字符  $s[0]$ , 在字符串  $t$  中进行查找, 假设查找到, 其第一次出现的位置为  $t_0$ ; 在字符串  $t$  的  $t_0$  下一个位置继续查找  $s[1]$ , 假设查找到, 其(第一次出现的)位置为  $t_1$ ; 在字符串  $t$  的  $t_1$  下一个位置继续查找  $s[2]$ , ...。如果对  $s$  中的每个字符, 都查找到, 则  $s$  是  $t$  的“子串”; 否则如果  $s$  中后面某些字符在  $t$  中没有找到对应的字符, 则  $s$  不是  $t$  的“子串”。

例如, 对样例输入中的第 1 个测试数据, 按照上述方式在字符串  $t$  中查找到字符串  $s$  中的前两个字符  $s[0]$  和  $s[1]$  后, 后面的 4 个字符没能在  $t$  中查找到, 如图 6.5(a) 所示, 所以  $s$  不是  $t$  的“子串”。相反, 第 2 个测试数据中, 对  $s$  中的每个字符, 都在  $t$  中查找到, 如图 6.5(b) 所示, 所以  $s$  是  $t$  的“子串”。

图6.5 依次在 $t$ 中查找 $s$ 中的每个字符**代码如下:**

```
#include <stdio.h>
#include <string.h>
char s[1000000]; //读入的字符串 s
char t[1000000]; //读入的字符串 t
int main( )
{
    long ls, lt;    //字符串 s 和 t 的长度
    long ps, pt;    //字符串 s 和 t 的查找位置
```

```

while( scanf("%s%s", s, t) != EOF )
{
    ls = strlen(s);
    lt = strlen(t);
    for( ps = pt = 0; ps < ls && pt < lt; pt ++ )
    {
        if( s[ps] == t[pt]) ps ++;
    }
    if( ps < ls ) puts("No");    //s 字符串中某些字符在 t 中没有找到对应的字符
    else puts("Yes");
}
return 0;
}

```

## 练习

### 6.7 粗心的 Tony(Careless Tony)

#### 题目描述:

Tony 是一个粗心的打字员,这不,他又犯错误了。更糟糕的是,光标键坏了,所以他只能用退格键回到出错的地方,纠正以后,对之后的正确字符又得重新输入了(因为为了回到出错的字符,这些字符都删除了)。

现在请你帮助他找出至少需要多长时间才能纠正错误。

#### 输入描述:

输入文件的第一行为一个整数  $N$ , 表示测试数据的个数。接下来有  $N$  个测试数据。

每个测试数据占 3 行:

第 1 行为一个正整数  $t(\leq 100)$ , 表示 Tony 删除或者输入一个字符所花的时间。

第 2 行为正确的文本内容。

第 3 行为 Tony 输入的文本内容。

注意: 文本只包含可读的字符, 每行文本的字符数不超过 80 个。

#### 输出描述:

对每个测试数据, 输出一行, 为 Tony 纠正错误所花的最少时间。

#### 样例输入:

```

2
1
WishingBone
WashingBone
1
Oops
Oooops

```

#### 样例输出:

```

20
6

```

**提示:** 从第 0 个字符开始比较正确的字符串与 Tony 输入的字符串, 找到首次不匹配位置, 则要删除和重新输入的字符串为两字符串在首次不匹配位置之后的子串。

### 6.8 令人惊讶的字符串(Surprising Strings)

#### 题目描述:

字符串  $S$  由字母字符组成, 它的“ $D$ -对字符串”为  $S$  中相隔  $D$  个位置的两个字符组成的有序对。如果  $S$  所有的“ $D$ -对字符串”都不相同, 则称  $S$  是“ $D$ -唯一的”。如果  $S$  对所有可

能的 D 值，都是“D—唯一的”，则称 S 是一个“令人惊讶的字符串”。

例如，考虑字符串"ZGBG"，它的“0—对字符串”为"ZG"、“GB”和"BG"，由于这三个字符串都不相同，因此"ZGBG"是“0—唯一”的。同样，字符串"ZGBG"的“1—对字符串”为"ZB"和"GG"，并且由于这两个字符串不同，所有"ZGBG"是“1—唯一”的。最后，字符串"ZGBG"的“2—对字符串”只有一个，就是"ZG"，因此"ZGBG"也是“2—唯一”的。

因此"ZGBG"是一个“令人惊讶的字符串”。

注意："ZG"既是"ZGBG"的“0—对字符串”，也是"ZGBG"的“2—对字符串”，这是不相关的，因为 0 和 2 是不同的距离。

#### 输入描述：

输入文件中包含了若干个非空字符串，由大写字母字符组成，长度最长为 79 个字符。每个字符串占一行。输入文件的最后一行为“\*”字符，代表输入结束。

#### 输出描述：

对输入文件中的每个字符串，判断是否为“令人惊讶的字符串”，并输出。

#### 样例输入：

```
ZGBG
X
AAB
AABA
BCBABCC
*
```

#### 样例输出：

```
ZGBG is surprising.
X is surprising.
AAB is surprising.
AABA is surprising.
BCBABCC is NOT surprising.
```

## 6.4 其他竞赛题目解析

本节再分析 3 道字符及字符串处理方面的题目。

### 例 6.10 数字字符

#### 题目描述：

最早的计算机使用点阵来显示数字和文字，例如，我们把数字 0123456789 分别用以下的图案表示：

```
*** * *** ** * * *** ** * * ***
* * * * * * * * * * * * * *
* * * *** ** ** ** ** ** * *** **
* * * * * * * * * * * * *
*** * *** ** * *** ** * *** **
0 1 2 3 4 5 6 7 8 9
```

本题要求把输入的数字变成以上的图案进行输出。

#### 输入描述：

输入数据可能包含多行，每行是一个正整数 n，(0<n≤99999)。

#### 输出描述：

对输入每个正整数，输出相应的图案数字。每个数字间仅用一列空格隔开，最后一位数字之后没有空格。

#### 样例输入：

#### 样例输出：

```

1234          *  ***  ***  *  *
99999         *    *    *  *  *
              *  ***  ***  ***
              *  *      *    *
              *  ***  ***    *
            ***  ***  ***  ***  ***
            *  *  *  *  *  *  *  *
            ***  ***  ***  ***  ***
              *    *    *    *    *
            ***  ***  ***  ***  ***
    
```

### 分析:

本题考察的是字符数组的使用。把 0~9 共十个数字的字符形式存放在一个三维字符数组 digit[10][5][4] 中。第 1 维“10”代表 10 个数字，第 2 维“5”代表每个数字的字符形式有 5 行，第 3 维“4”代表每行有 3 个字符，最后一个字符为字符串结束标志。

在读入整数时，采用字符形式读入更为方便，因为这种方式获取整数的总位数(即字符串的长度)、以及取得整数的每位(即字符数组中的每个字符)都是很方便的。对每个整数都输出 5 行，第 i 行为整数中每个数字的第 i 行组成。

另外，本题要求每个数字之间有一个空列，而在最后一个字符后没有空列，在输出时要特别注意。

### 代码如下:

```

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

char digit[10][5][4] = { //存储数字 0~9 的字符形式
    { "****", " *", " **", " ***", " ****", //0
      ,
      ,
      ,
      ,
      , //1
    { " *", " **", " ***", " ****", " *****",
      ,
      ,
      ,
      ,
      , //2
    { "****", " **", " ****", " ****", " ****",
      ,
      ,
      ,
      ,
      , //3
    { "****", " **", " ****", " ****", " ****",
      ,
      ,
      ,
      ,
      , //4
    { "****", " **", " ****", " ****", " ****",
      ,
      ,
      ,
      ,
      , //5
    { "****", " **", " ****", " ****", " ****",
      ,
      ,
      ,
      ,
      , //6
    { "****", " **", " ****", " ****", " ****",
      ,
      ,
      ,
      ,
      , //7
    { "****", " **", " ****", " ****", " ****",
      ,
      ,
      ,
      ,
      , //8
    { "****", " **", " ****", " ****", " ****", //9
  };

int main( )
{
    int i, j; //循环变量
    char ch[6]; //以字符形式读入整数
    while( scanf("%s", ch) != EOF )
    {
        int len = strlen(ch);
        for( i=0; i<5; i++ ) //输出 5 行
        {
            for( j=0; j<len; j++ ) //输出该整数每位数字的每行
            {
    
```

```

        printf( "%s", digit[ch[j]-'0'][i] );
        if( j<len-1 ) printf( " " );
    }
    printf( "\n" );
}
}
return 0;
}

```

### 例 6.11 英语数字翻译(English-Number Translator)

题目来源:

Czech Technical University Open 2004

题目描述:

在本题中, 你的任务是将英文单词表示的整数翻译成阿拉伯数字形式。整数的范围是从 -999,999,999 到 +999,999,999。以下是整数中可能出现的所有英文单词:

negative, zero, one, two, three, four, five, six, seven, eight,  
 nine, ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen,  
 seventeen, eighteen, nineteen, twenty, thirty, forty, fifty, sixty,  
 seventy, eighty, ninety, hundred, thousand, million

输入描述:

输入包含多个测试数据, 每个测试数据占一行, 为一串英文单词所表示的整数。注意: 负数最前面的单词是 "negative"; 当能用单词 "thousand" 表示时, 就不会用 "hundred" 来表示。如 1500 表示成 "one thousand five hundred", 而不是 "fifteen hundred"。输入以空行结束。

输出描述:

对输入文件中的每个测试数据, 输出一行, 为对应的整数。

样例输入:

```

six
negative seven hundred twenty nine
one million one hundred one
eight hundred fourteen thousand twenty two
(表示输入结束的空行)

```

样例输出:

```

6
-729
1000101
814022

```

分析:

这是一道很有意思的题目。将英文整数中可能出现的所有英文单词(共 31 个)存储到一个二维字符数组中(详见下面的代码): 当  $i$  取值为  $0 \sim 20$  时, 第  $i$  个单词所表示的数值为  $i$ ; 当  $i$  取值为  $21 \sim 27$  时, 第  $i$  个单词所表示的数值为  $(i-18)*10$ 。"hundred", "thousand", "million" 这 3 个单词的处理很特别。注意这 3 个单词不会出现的英文整数的最前面。

"hundred": 它会使前一个单词(注意是一个单词, 因为根据题目的意思, 测试数据中不可能出现 twenty one hundred 这种英文数字)所表示的数值扩大 100 倍。例如, seven hundred twenty nine, 第一个英文单词所表示的数值是 7, 第 2 个单词为 hundred, 它会使 7 扩大 100 倍, 即变成 700。

"thousand": 它会使前面若干个单词所表示的数值扩大 1000 倍。例如, eight hundred fourteen thousand twenty two, 其中 eight hundred fourteen 表示的数值为 814, 紧接着是 thousand, 它会使 814 扩大 1000 倍, 即变成 814000。

"million": 它会使前面若干个单词所表示的数值扩大 1000000 倍。例如, twenty one million

所表示的数值将是 21000000。

代码如下：

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

char *string[] = {
    "zero", "one", "two", "three", "four", "five", "six",
    "seven", "eight", "nine", "ten", "eleven", "twelve",
    "thirteen", "fourteen", "fifteen", "sixteen", "seventeen",
    "eighteen", "nineteen", "twenty", "thirty", "forty", "fifty",
    "sixty", "seventy", "eighty", "ninety", "hundred",
    "thousand", "million"
};

int main( )
{
    int i;
    int sign; //当第 1 个单词为"negative"时，表示负数，sign = -1
    long temp; //前面若干个英文单词累积所表示的数值
    long sum; //整个英文数字所表示的数值
    char str[9] = {'\0'}, ch; //读入的每个单词，及单词中的字符
    while( scanf( "%c",&ch ) != EOF ) //输入字符串
    {
        if( ch==10 ) break; //要加上这一行，题目中有一句话“输入以空行结束”
        i = 0;
        str[i++] = ch;
        while( scanf("%c",&ch) , ch!=' ' && ch!='\n' ) //读入第一个单词
            str[i++] = ch;
        str[i] = '\0';
        sum = temp = 0;
        sign = 1;
        if( 0 == strcmp(str,"negative") ) sign = -1;
        else
        {
            for( i = 0; i < 28; i++ )
            {
                if( 0 == strcmp(str,string[i]) )
                {
                    if(i <= 20) temp += i;
                    else if(i < 28) temp += (i-18) * 10;
                }
            }
        }
        while(ch != '\n')
        {
            i = 0;
            while( scanf("%c",&ch) , ch!=' ' && ch!='\n' ) //继续读入其他单词
                str[i++] = ch;
            str[i] = '\0';
        }
    }
}
```



```

    for(i = 0; i < 31; i++)
    {
        if( 0 == strcmp(str, string[i]) )
        {
            if(i <= 20) temp += i;
            else if(i < 28) temp += (i-18) * 10;
            else if(i == 28) temp *= 100; //hundred
            else if(i == 29) //thousand
            { temp *= 1000; sum += temp; temp = 0; }
            else if(i == 30) //million
            { sum += temp; sum *= 1000000; temp = 0; }
            break;
        }
    }
    sum += temp;
    printf( "%ld\n", sum*sign );
}
return 0;
}

```

### 例 6.12 单词的 anagrammatic 距离(Anagrammatic Distance)

题目来源:

Southeastern Europe 2005

题目描述:

两个单词互为 **anagrams**，意思是一个单词的字母经过重新排序后可以形成另一个单词。例如，单词“occurs”和“succor”互为 **anagrams**。然而“dear”和“dared”不是互为 **anagrams**，因为字母 d 在“dared”中出现了 2 次，而在“dear”中只出现了 1 次。英语中最著名的 **anagrams** 是“dog”和“god”。

两个单词的 **anagrammatic** 距离为 **N**，意思是至少要在两个单词中一共去掉 **N** 个字母，才能使两个单词中剩下的部分是互为 **anagrams** 的。例如，给定两个单词：“sleep”和“leap”，我们需要至少去掉 3 个字母—从单词“sleep”中去掉 2 个字母、从单词“leap”中去掉 1 个字母—这样两个单词中剩下的部分互为 **anagrams**(每个单词都只剩下 **lep**)。对于单词“dog”和“cat”，由于两个单词没有相同的字母，所以他们的 **anagrammatic** 距离两个单词字母个数的总和，因为要去掉所有的字母才能是他们“互为 **anagrams**”。(注意，一个单词总是和它本身互为 **anagrams** 的。)

你的任务是编写一个程序，计算两个给定单词的 **anagrammatic** 距离。

输入描述:

输入文件的第一行为一个正整数 **N** (小于 60,000)，标明输入文件中测试数据的个数。每个测试数据包含两个单词，有可能为空，每个单词占一行（所以有可能为空行）。因此输入文件中第一行之后还有 **2N** 行。

尽管单词可能为空的，但它们肯定是很简单的单词—单词中的字母都是小写的，而且都是英文字母表中 26 个字母中的一个(abcdefghijklmnopqrstuvwxyz)。最长的单词是：pneumonoultramicroscopicsilicovolcanoconiosis。(45 个字母)

输出描述:

对每个测试数据，输出两个单词的 **anagrammatic** 距离，按照样例输出中的格式进行输出。

**样例输入：**

```
4
crocus
succor
dares
seared
empty
```

```
smell
lemon
```

**分析：**

在求两个单词 **a** 和 **b** 的 **anagrammatic** 距离时，去掉这两个单词的某些字母后，单词 **a** 中剩余字母**任意重排顺序**，看是否是单词 **b** 的剩余字母组成的单词。因此，实际上只需对单词 **a** 中的每个字母 **x**，扫描单词 **b**，如果能找到相同的字母，在字母 **x** 可以保留，否则不保留。但要注意，由于在单词中同一个字母可能出现多次，所以需要用数组记录每个字母是否保留。

例如样例数据中的第 4 个测试数据，“smell”和“lemon”，对“smell”中第 1 个“l”字母，在“lemon”中查找到对应的字母“l”，所以这个字母要保留；而对“smell”中第 2 个“l”字母，由于“lemon”剩下的字母中不存字母“l”，所以“smell”中第 2 个“l”字母不能保留。

最终求得的单词 **a** 和 **b** 的 **anagrammatic** 距离 = **a** 的长度 + **b** 的长度 - 2×单词 **a** 中保留的字母数。

另外，正如样例数据中第 3 个测试数据所示，单词可能为空，所以要能正确地读入测试数据中的单词。

**代码如下：**

```
#include <string>
#include <stdio.h>
char a[50], b[50]; //读入的两个单词
int len_a, len_b; //两个单词的长度
bool flag_a[50], flag_b[50]; //表示 a,b 两个单词中每个字母是否保留的标志

int main( )
{
    int N; //测试数据的个数
    int i, j, k; //循环变量
    scanf( "%d\n", &N );
    for( k = 1; k <= N; k++ )
    {
        int count; //a,b 单词中保留的字母数
        gets( a ); gets( b ); //读入两个单词
        len_a = strlen( a ); memset( flag_a, 0, sizeof(flag_a) );
        len_b = strlen( b ); memset( flag_b, 0, sizeof(flag_b) );
        for( i = count = 0; i < len_a; i++ ) //扫描单词 a 的每个字母
        {
            for( j = 0; j < len_b; j++ ) //对 a 的每个字母,扫描 b,看是否能找到相同字母
            {
                if( !flag_a[i] && !flag_b[j] && a[i]==b[j] )
```

```

        {
            flag_a[i] = flag_b[j] = 1;    //保留 a[i]和 b[j]
            count++;
        }
    }
    printf( "Case #%%d:  %%d\\n", k, len_a + len_b - 2*count );
}
return 0;
}

```

### 练习

#### 6.9 LC 显示器(LC-Display)

##### 题目描述:

你的一个朋友刚买了一台新电脑。之前，他用过的最好的电脑只是便携式计算器。现在，你的朋友看着他的新电脑，他很失望，因为他很喜欢他的计算器的 LC 显示器。因此你决定给你的朋友编写一个程序，模拟 LC 显示器来显示数字。

##### 输入描述:

输入文件包含多行，每一行为需要显示的数。每一行中有两个整数  $s$  和  $n$ ， $1 \leq s \leq 10$ ， $0 \leq n \leq 99\,999\,999$ 。 $n$  是要显示的数值， $s$  是显示的大小。

输入文件的最后一行为两个 0，这一行不需处理。

##### 输出描述:

以 LC 显示器方式输出输入文件中的数，用符号 “-” 表示水平的线段，用符号 “|” 表示垂直的线段。数值中的每个数字占  $s+2$  列、 $2s+3$  行。在输出时，对每两个个数字之间的空白区域，要确保用空格填满，对最后一位数字之后的空白区域，不能输出空格。每两个数字之间仅有一个空列。(样例输出中给出了 0~9 每个数字的输出格式)

每个数值之后输出一个空行。

##### 样例输入:

```

2 12345
3 67890
0 0

```

##### 样例输出:

```

      --  --  --
      |  |  |  |  |
      |  |  |  |  |
      --  --  --
      |  |  |  |  |
      |  |  |  |  |
      --  --  --

      ---  ---  ---  ---  ---
      |  |  |  |  |  |  |
      |  |  |  |  |  |  |
      ---  ---  ---
      |  |  |  |  |  |
      |  |  |  |  |  |
      ---  ---  ---  ---

```

## 6.10 单词逆序(Word Reversal)

### 题目描述:

对一组单词，输出每个单词的逆序，并且不改变这些单词的顺序。

### 输入描述:

输入文件的第 1 行为一个整数  $N$ ，然后是一个空行，接下来是  $N$  组数据，每组数据之间有一个空行。每组数据的格式为：首先第一行为一个整数  $K$ ，代表这组数据中有  $K$  个测试数据，每个测试数据为一行，包含若干个单词，这些单词用空格隔开，每个单词仅由大小写字母字符组成。

### 输出描述:

对输入文件中的每组数据，相应有一组输出，每组输出之间有一个空行。

对每组数据中的每个测试数据，输出一行，为逆序后的各个单词。

### 样例输入:

```
1
3
I am happy today
To be or not to be
I want to win the practice contest
```

### 样例输出:

```
I ma yppah yadot
oT eb ro ton ot eb
I tnaw ot niw eht ecitcarp tsetnoc
```

## 6.11 多项式表示问题(Polynomial Showdown)

### 题目描述:

给定多项式的系数，要求输出多项式的可读格式，并去掉多余的字符，多项式中自变量的幂的次数为 8 到 0。例如，假设给定的系数为 0, 0, 0, 1, 22, -333, 0, 1 和 -1，则输出的多项式为： $x^5 + 22x^4 - 333x^3 + x - 1$ 。

在表示多项式时要遵守以下规则：

- 1) 多项式的各项必须按幂的次数由高到低的顺序排列。
- 2) 指数用符号“^”来表示。
- 3) 常数项仅用常数来表示，不需要乘以  $x^0$ 。
- 4) 只有系数非 0 的项才需要表示出来。如果所有项的系数都为 0，则要输出常数项，即 0，尽管这一项的系数也为 0。
- 5) 二元运算符“+”和“-”左右两边各有一个空格符号，除此之外，表达式中没有多余的空格符号。
- 6) 如果多项式的第 1 项系数为正，则系数前面没有正号；如果第 1 项的系数为负数，则在系数前有符号，例如： $-7x^2 + 30x + 66$ 。
- 7) 对系数为负数的项，除非该项是第 1 项，否则该项的系数应该表示成减去对应的正数项，也就是说，不能输出“ $x^2 + -3x$ ”，而应该输出“ $x^2 - 3x$ ”。
- 8) 常数 1 和 -1 只能出现在常数项，也就是说，不能输出“ $-1x^3 + 1x^2 + 3x^1 - 1$ ”，而应该输出“ $-x^3 + x^2 + 3x - 1$ ”。

### 输入描述:

输入文件中包含若干个测试数据，每个测试数据占一行，为多项式的 9 个系数，用空格隔开，每个系数的绝对值不超过 1000。

**输出描述:**

对输入文件每个测试数据所给出的 9 个系数，输出一行，为对应的多项式。

**样例输入:**

```
0 0 0 1 22 -333 0 1 -1
0 0 0 0 0 -55 5 0
```

**样例输出:**

```
x^5 + 22x^4 - 333x^3 + x - 1
-55x^2 + 5x
```

## 第三篇 程序设计方法与在线实践（提高篇）

本篇是第二篇的提高，介绍比较复杂的算法思想和应用问题，包括第 7~9 章，各章的内容安排如下。

第 7 章介绍高精度计算。高精度计算是字符及字符串处理知识的具体应用（有些题目采用整型数组处理更方便）。本章介绍高精度计算涉及到的基础知识：进制转换、用数组实现算术运算，从而引出高精度计算的基本思路；最后通过经典竞赛题目讲解高精度数的基本运算以及其他一些问题的实现方法。

第 8 章介绍递归算法思想及其应用。递归思想是一种重要的算法思想，通过递归函数这种函数形式来实现。本章从求阶乘、Fibonacci 数列的递归求解等例题出发引入递归算法思想，再过渡到递归思想在实际竞赛题目中的应用。特别地，本章还介绍了求解问题的一种通用方法：搜索，并以排列组合问题的求解来阐述搜索方法在求解这一类问题中的应用。

第 9 章介绍排序及检索。对数据进行排序是数据处理中经常要用到的操作，本章介绍常用的、比较简单的排序算法的思想及其应用。另外，对一组已经排好顺序的数据进行检索也是经常要用到的操作，本章因此介绍了二分法的思想，以及二分法在检索中的应用。

### 程序实践提示

本篇介绍的程序设计方法和应用问题比较难，建议读者通过验证例题→理解思考题→完成练习题等这些实践来理解这些程序设计方法及其应用，对例题和练习题一定要提交到 OJ 网站上来验证算法的正确性。

---

## 第7章 高精度计算

在 32 位机器里，有符号整数(int)的取值范围是-2147483648~+2147483647，无符号整数(unsigned int)的取值范围是 0~4294967295。超过这个范围的数据可以用浮点型(double)来表示，如 50 的阶乘。但用浮点数来表示整数通常不便于整数的运算，比如整数除法跟浮点数除法含义不一样，浮点数也无法实现取余运算等等。另外，超过浮点数取值范围的数据，比如一个 1000 位的整数，无法用常规方法来处理。这些精度很高的数据通常称为**高精度数**，或称为**大数**。高精度数的运算只能用本章介绍的高精度数计算方法来处理。

### 7.1 基础知识

本章的基础知识涉及到进制转换和用字符型数组(或整型数组)进行算术运算。在数值型数据的处理中经常要涉及到进制转换；另外有些问题无法通过直接运算来求解，如统计加法中进位的次数等，这就需要用字符型数组(或整型数组)来实现算术运算。

#### 7.1.1 进制转换

所谓进位计数制(简称进制)，是指用一组固定的符号和统一的规则来表示数值的方法，按进位的方法进行计数。一种进位计数制包含以下 3 个要素：

- 1) 数码：计数使用的符号；
- 2) 基数：使用数码的个数；
- 3) 位权：数码在不同位上的权值。

例如日常生活中使用的十进制，它使用的数码是 0,1,2,3,4,5,6,7,8,9 共十个；基数就是“十”(10)；位权：个位是 1 ( $10^0$ )，十位是 10 ( $10^1$ )，百位是 100 ( $10^2$ ) 等。

在计算机中进行运算采用的是二进制，它使用的数码只有 0 和 1；基数就是“二”(10)；第*i*位的位权是  $2^i$ ， $i=0,1,2,\dots$ 。

除了以上介绍的两种进制外，常用的还有八进制和十六进制，在 ACM/ICPC 题目中可能还有其他进位计数制。各种进制之间的转换主要有两种形式：将**其他进制的数转换成十进制**；将**十进制数转换成其他进制**。

对于第一种转换，规则很简单，只需“按权值展开”即可。例如： $(1101.11)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 8 + 4 + 0 + 1 + 0.5 + 0.25 = (13.75)_{10}$ 。

对于第二种转换，以十进制转换成二进制为例讲解。方法是：对整数部分，除以 2 取余数，注意先得到的余数放在低位，后得到的余数放在高位，余数 0 不能舍去；对小数部分，乘以 2 取整数，注意先得到的整数放在高位，后得到的整数放在低位，整数 0 不能舍去。例如，将十进制数 29.375 转换成二进制的过程如图 7.1 所示。因此， $(29.375)_{10} = (11101.011)_2$ 。

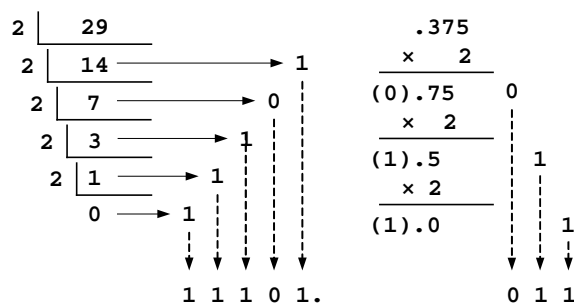


图7.1 十进制数转换成二进制



将十进制转换成其他任何一种进制，其原理与将十进制转换成二进制的原理是一样的。进制转换过程中经常需要灵活使用整除和取余运算，具体情形详见例 7.1。

### 例 7.1 回文数(Palindrom Numbers)

**题目来源：**

South Africa 2001

**题目描述：**

我们称一个数是一个回文数，当且仅当它从左往右读和从右往左读起来都是一样的。比如 75457 就是回文数。

当然，这种性质取决于这个数是在什么进制下。比如 17 在十进制下不是一个回文数，但在二进制下是一个回文数（10001）。

这道题的目的是验证给定的一组数分别在 2 进制~16 进制下是否是回文数。

**输入描述：**

输入文件包含了若干个整数，每个整数  $n$  都是在十进制下给出的，每个整数占一行。 $0 < n < 50000$ 。输入文件以 0 表示结束。

**输出描述：**

当该整数在某些进制下是回文数，则输出 "Number i is palindrom in basis "，分别列出这些基数，其中  $i$  是给定的整数。如果该整数在 2~16 进制下都不是回文数，则输出 "Number i is not palindrom"。

**样例输入：**

17  
19  
0

**样例输出：**

Number 17 is palindrom in basis 2 4 16  
Number 19 is not a palindrom

**分析：**

对读入的每个十进制数  $number$ ，依次判断  $number$  在 2~16 进制下是否为回文数并输出，如果都不是则输出 "Number i is not palindrom"。

在判断十进制数  $number$  在  $basis$  进制下是否为回文数时，首先要将十进制数  $number$  转换成  $basis$  进制：将  $number$  除以  $basis$  取余数。存储余数时要注意以下两个问题：

- 1) 十进制以上的进制中的数码除了 0~9 外，还有字母，例如十六进制的数码为 0~9，以及 A、B、C、D、E、F。那么是否需要将得到的余数以字符形式存放呢？
- 2) 进制转换时，得到的余数排列顺序是：先得到的余数位于低位，后得到的余数位于高位，是否有必要严格按照这个顺序（即与余数产生顺序相反的顺序）存储得到的余数？

对于第一个问题，答案是不需要，更方便的做法是在取余数时把得到的余数以整数形式存放在一个整型数组里。例如，在判断十进制数 2847 在 15 进制下是否为回文数时，依次得到的余数是：12、9 和 12。其中第 1 和第 3 个余数在 15 进制下为字符 C。但在本题中，并不需要得到真正的 15 进制数，只需要判断各位数码中的某些位是否相等，所以按整数存储是可以的。

对于第二个问题，答案也是不需要的。因为如果一个数是回文数，则各位逆序后仍然是回文数，因此在取余时可以按先后顺序存放在整型数组里，然后判断数组中的数是否构成回文数。

另外，本题的输出也很特别：如果  $number$  在某些进制下是回文数，则输出这些进制；否则，如果  $number$  在 2~16 进制下都不是回文数，则是另外一种输出。所以需要设置一个状态变量  $IsPal$ ，如果  $IsPal$  为 false，则表示  $number$  在 2~16 进制下都不是回文数；初始时  $IsPal$  为 false，如果首次判断出  $number$  在某进制下是回文数，则将  $IsPal$  的值置为 true，并开始输

出进制信息。最后 `IsPal` 的值如果仍为 `false`，才会输出 "Number i is not palindrom"。

代码如下：

```
#include <stdio.h>
//判断 number 在 basis 进制下是否位回文数，如果是，返回 true
bool IsPalindrom( int number, int basis )
{
    //number 最大为 50000，转换成 2 进制不超过 16 位
    int a[16]; //将十进制数 number 转换成 basis 进制数得到的每一位
    int i = 0, j = 0, k; //循环变量
    int Len; //转换到 basis 进制后的位数
    while( number )
    {
        a[i++] = number % basis;
        number /= basis;
    }
    Len = i;
    k = Len/2;
    while( j < k ) //判断转换后的数是否为回文数
    {
        if( a[j] != a[Len-1-j] ) return false;
        j++;
    }
    return true;
}

int main( )
{
    int number; //读入的每个数
    bool IsPal; //如果 IsPal 为 false，则 number 在 2~16 进制下都不是回文数
    while( scanf("%d", &number) )
    {
        if( !number ) break;
        IsPal = false;
        for( int i = 2; i <= 16; i++ )
        {
            if( IsPalindrom(number, i) )
            {
                if( !IsPal )
                {
                    IsPal = true;
                    printf( "Number %d is palindrom in basis", number );
                }
                printf( " %d", i );
            }
        }
        if( !IsPal ) printf( "Number %d is not a palindrom", number );
        printf( "\n" );
    }
    return 0;
}
```

### 7.1.2 用字符型数组或整型数组实现算术运算

对两个整数直接相加，通常只能得到最终的结果。例如，假设有两个 `int` 型变量 `a` 和 `b`，它们的值分别是 7543 和 976210，计算 “`a+b`” 只能得到最终的结果，即 983753。

如果要得到运算过程，或者每一位的运算结果，比如进位，那么就需要把整数的每一位存储到数组里，每个数组元素相当于整数中的某一位，然后按数组元素中的值进行每一位的运算。如图 7.2 所示。

|   |   |   |   |   |   |   |  |  |  |
|---|---|---|---|---|---|---|--|--|--|
| a | 7 | 5 | 4 | 3 |   |   |  |  |  |
| b | 9 | 7 | 6 | 2 | 1 | 0 |  |  |  |

图7.2 用数组存储整数的每一位

在将整数的每一位存储到数组时，可以选择整型数组，也可以选择字符型数组。到底选择整型数组还是字符型数组，应视题目而定。但采用字符型数组存储时在以下三个方面要比用整型数组存储方便得多：

- ① 如果用字符数组存储，则整数的总位数就是字符数组中所存储字符串的长度，而用整型数组存储时要得到整数的总位数要稍微麻烦一些。
- ② 输入时，用字符数组读入整数很方便；而如果用整型数组存储整数的每一位，则要将读入的整数的每一位先取出来再存储到整型数组中。另外，如果整数很大，超过了 `int` 型数据的表示范围，则只能采用字符数组读入。
- ③ 如果整数是保存在字符数组中，那么在输出时也很方便。

下面例 7.2 就是用字符数组存储整数，以便统计加法运算过程中进位的次数。

不过，对于用字符数组读入整数这种方式，初学者往往难以理解，现举例解释。对于整数 “7543”，如果采取以下这种方式读入：

```
int a; scanf("%d", &a); //将整数读入到整型变量中
```

则整数 “7543” 以二进制形式存储到整型变量 `a` 所占的 4 个字节中。如果采取以下方式读入：

```
char str[10]; scanf("%s", str); //用字符数组读入整数
```

则是将每位数字 “7”、“5”、“4”、“3” 以数字字符形式读入并存储到字符数组 `str` 中。当然，要得到每位数字字符对应的数值，以及整个字符数组所表示的数值，要进行一定的转换，详见例 7.2 及后面的例题。

#### 例 7.2 初等算术(Primary Arithmetic)

**题目来源：**

University of Waterloo Local Contest 2000.09.23

**题目描述：**

小学生在学多位数的加法时，是将两个加数右对齐，然后从右往左一位一位地加。多位数的加法经常会有进位。如果对齐的两位相加结果大于或等于十就给左边一位进一。对小学生来说，进位的判断是比较难的。你的任务是：给定两个加数，统计进位的次数，从而帮助老师评估加法的难度。

**输入描述：**

输入文件中的每一行为两个无符号整数，少于 10 位。最后一行位两个 0，表示输入结束。

**输出描述：**

对输入文件每一行(最后一行除外)的两个加数，计算它们进行加法运算时进位的次数并输出。具体输出格式详见样例输出。

样例输入：

123 456  
555 555  
123 594  
0 0

样例输出：

No carry operation.  
3 carry operations.  
1 carry operation.

分析：

正如前面分析的那样，本题可以采用字符数组来存储读入的两个加数。对两个加数进行加法运算时，要注意以下两点：

- 1) 在进行加法时，要得到每个数字字符对应的数值，方法是将每个数字字符减去数字字符“0”；
- 2) 从两个加数的最低位开始按位求和，如果和大于 9，则会向前一位进位。要注意某一个加数的每一位都运算完毕，但另一个加数还有若干位没有运算完毕的情形。如图 7.2 所示， $999586 + 798$ ，这两个加数分为有 6 位和 3 位数。当第 2 个加数的最低 3 位数都运算完毕时，还会向前面进位，这时第 1 个加数还有 3 位没有运算完毕，由于进位的存在，这 3 位在运算时都还会产生进位。

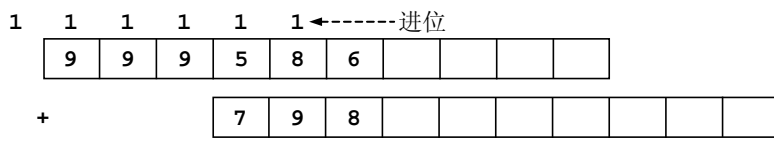


图7.3 用字符数组实现算术运算

代码如下：

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char add1[11], add2[11]; //读入的两个加数
    while( scanf("%s%s", add1, add2) )
    {
        if( !strcmp(add1,"0") && !strcmp(add2,"0") ) break;
        int carry = 0; //进位次数
        int i1 = strlen(add1) - 1;
        int i2 = strlen(add2) - 1;
        int C = 0; //进位
        while( i1>=0 && i2>=0 ) //从两个加数的右边开始对每位进行加法运算
        {
            if( add1[i1]-'0'+add2[i2]-'0'+C>9 )
            {
                carry++; C = 1;
            }
            else C = 0;
            i1--;
            i2--;
        }
        while( i1>=0 ) //如果第 1 个加数还有若干位没有运算完
        {
            if( add1[i1]-'0'+C>9 )
```

```

        {
            carry++; C = 1;
        }
        else C = 0;
        i1--;
    }
    while( i2>=0 )    //如果第 1 个加数还有若干位没有运算完
    {
        if( add2[i2]-'0'+C>9 )
        {
            carry++; C = 1;
        }
        else C = 0;
        i2--;
    }
    if( carry>1 ) printf( "%d carry operations.\n", carry );
    else if( carry==1 ) printf( "%d carry operation.\n", carry );
    else printf( "No carry operation.\n" );
}
return 0;
}

```

注意，本题中告诉了读入的无符号整数少于 10 位，因此可以用 `unsigned int` 变量(其取值范围是 0~4294967295)来保存读入的整数，并将读入的整数取出各位存放到整型数组中，再按整型数组进行运算，统计进位的次数。读者不妨试试。

另外，本题并不需要存储加法运算的结果，如果需要存储，通常需要将两个加数逆序后再进行运算，详见 7.2.1 节。

### 7.1.3 高精度计算的基本思路

高精度计算的**基本思路**是：用数组存储参与运算的数的每一位，在运算时以数组元素所表示的位为单位进行运算。可以采用字符数组、也可以采用整数数组存储参与运算的数，到底采用字符数组还是整数数组更方便，应视具体题目而定。如下面的例 7.3。

#### 例 7.3 skew 二进制(Skew Binary)

题目来源：

Mid-Central USA 1997

题目描述：

在十进制里，第 $k$ 位数字的权值是  $10^k$ 。(每位数字的顺序是从右到左的，最低位，也就是最右边的位，称为第 0 位)。例如：

$$\begin{aligned}
 81307_{(10)} &= 8 * 10^4 + 1 * 10^3 + 3 * 10^2 + 0 * 10^1 + 7 * 10^0 \\
 &= 80000 + 1000 + 300 + 0 + 7 = 81307.
 \end{aligned}$$

而在二进制里，第 $k$ 位的权值为  $2^k$ 。例如：

$$\begin{aligned}
 10011_{(2)} &= 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 \\
 &= 16 + 0 + 0 + 2 + 1 = 19.
 \end{aligned}$$

在skew二进制里，第 $k$ 位的权值为  $2^{(k+1)} - 1$ ，skew二进制的数码为 0 和 1，最低的非 0 位可以取 2。例如：

$$\begin{aligned}
 10120_{(skew2)} &= 1 * (2^5 - 1) + 0 * (2^4 - 1) + 1 * (2^3 - 1) \\
 &\quad + 2 * (2^2 - 1) + 0 * (2^1 - 1)
 \end{aligned}$$

$= 31 + 0 + 7 + 6 + 0 = 44.$

skew 二进制的前 10 个数为 0, 1, 2, 10, 11, 12, 20, 100, 101 和 102。

#### 输入描述:

输入文件包含若干行, 每行为一个整数  $n$ 。  $n = 0$  代表输入结束。除此之外,  $n$  是 skew 二进制下的一个非负整数。

#### 输出描述:

对输入文件中的每个 skew 二进制数, 输出对应的十进制数。输入文件中  $n$  最大值对应到十进制为  $2^{31} - 1 = 2147483647$ 。

#### 样例输入:

```
10120
20000000000000000000000000000000
10
10000000000000000000000000000000
0
```

#### 样例输出:

```
44
2147483646
3
2147483647
```

#### 分析:

很明显, 对输入文件中的 skew 二进制数, 不能采用整数形式(int)读入, 必须采用字符数组。那么需要定义多长的字符数组呢? 题目中提到“输入文件中的 skew 二进制数最大值对应到十进制为  $2^{31} - 1 = 2147483647$ ”, 正如样例输入数据所示, 十进制数 2147483647 对应的 skew 二进制数为: 10000000000000000000000000000000, 因此存储输入文件中的 skew 二进制数可以采用长度为 40 的字符数组。

在把 skew 二进制数转换成十进制时, 只需把每位按权值展开求和即可。在本题中, 读者就可以发现采用字符数组存储高精度数, 要求高精度数的总位数及取出每位上的数码都是很方便的。

#### 代码如下:

```
#include <stdio.h>
#include <string.h>
#include <math.h>
int main( )
{
    char str[40]; //读入的每个 skew 二进制数, 用字符数组存放
    while( scanf( "%s", str )!=EOF )
    {
        int len = strlen(str); //高精度数的总位数就是字符串的长度
        int num = 0; //对应的十进制数
        if( len==1 && str[0]=='0' ) break;
        for( int i=len-1; i>=0; i-- )
            num += (str[i]-'0')*( pow(2,len-i) - 1 ); //高精度数的每位: str[i]-'0'
        printf( "%d\n", num );
    }
    return 0;
}
```

## 练习

### 7.1 设计计算器(Basically Speaking)

**题目描述:**

某个公司最近邀请你设计一个计算器。作为计算机科学家，你建议将这个计算器设计得灵巧一点，使得它能在各种进制之间进行转换。公司认为这是一个很好的想法，并要求你拿出实现进制转换的算法原型。公司经理告诉你，该计算器有以下特征：

- a) 它的显示器有 7 位；
- b) 它的按键除了数字 0 到 9 外，还有大写字母 A 到 F。
- c) 它支持 2~16 进制。

**输入描述:**

输入文件中的每一行要实现一次进制转换。每一行有 3 个数，第一个数是 A 进制下的一个整数，第 2 个数就是 A，第 3 个数是 B，要实现的是将第一个数从 A 进制转换到 B 进制下。这 3 个数的两边可能有一个或多个空格。输入数据一直到文件尾。

**输出描述:**

实现输入文件中的每次进制转换，转换后的数右对齐到 7 位显示器。如果转换后的数的位数太多了，在 7 位显示器中显示不下，则输出"ERROR"，也是右对齐到 7 位显示器。

**样例输入:**

```
1111000 2 16
2102101 3 15
 12312 4 2
1234567 10 16
```

**样例输出:**

```
    78
   7CA
  ERROR
 12D687
```

**7.2 进制转换(Number Base Conversion)****题目描述:**

编写程序，实现将一个数从一种进制转换到另一种进制。在这些进制中，可以出现的数码有 62 个：{ 0-9, A-Z, a-z }。

**输入描述:**

输入文件的第 1 行为一个正整数 N，表示测试数据的个数。接下来有 N 行，每行的格式为：输入数据的进制(用十进制表示)，输出数据的进制(用十进制表示)，最后一个是用输入数据的进制所表示的数。输入/输出数据的进制范围是 2~62，也就是说 A~Z 相当于十进制中的 10~35，a~z 相当于十进制中的 36~61。

**输出描述:**

对每个测试数据，程序输出 3 行。第 1 行为输入数据的进制，空格，然后是在该进制下的输入数据；第 2 行为输出数据的进制，空格，然后是在该进制下的输出数据；第 3 行为空行。

**样例输入:**

```
2
62 2 abcdefghiz
10 16 1234567890123456789012345678901234567890
```

**样例输出:**

```
62 abcdefghiz
2 110111100000100010111110010010110011111001001100011010010001

10 1234567890123456789012345678901234567890
16 3A0C92075C0DBF3B8ACBC5F96CE3F0AD2
```

### 7.3 Wacmian 数(Wacmian Numbers)

#### 题目描述:

在假设的 Wacmahara 无人沙漠里，一个非普通人组成的部落被发现了。Wacmians 的每个手上除一个大拇指外仅有两个手指。他们发明了自己的数字系统。该系统中使用的数字和用来表示数字的符号都很奇特，但是人类学家已经能够以用下面的方法描述它们：

% — 0

) — 1

~ — 2

@ — 3

? — 4

\ — 5

\$ — -1 (没错，他们甚至有负数)

如你所愿，他们的数字系统是 6 进制的，每位上的数值达到 6 就向该位的左边进位，如下面的例子：

)@% 表示： $1 \cdot 6^2 + 3 \cdot 6 + 0 = 36 + 18 + 0 = 54$

?\$~~ 表示： $4 \cdot 6^3 + (-1) \cdot 6^2 + 2 \cdot 6 + 2 = 864 - 36 + 12 + 2 = 842$

\$~~ 表示： $(-1) \cdot 6^2 + 2 \cdot 6 + 2 = -36 + 12 + 2 = -22$

你的任务是吧 Wacmian 数字解释成标准的 10 进制数字。

#### 输入描述:

输入文件包括若干个 Wacmian 数，每行一个。每个数由 1 至 10 位 Wacmian 数字字符组成。输入文件最后一行为“#”字符，表示输入结束。

#### 输出描述:

对输入文件中的每个 Wacmian 数，输出一行，为对应的十进制数。

#### 样例输入:

)@%

?\$~~

\$~~

#

#### 样例输出:

54

842

-22

## 7.2 高精度数的基本运算

本节以几道竞赛题目为例讲解高精度数的加法、乘法和除法运算的实现方法。

### 7.2.1 高精度数的加法

#### 例 7.4 整数探究(Integer Inquiry)

#### 题目来源:

Central Europe 2000

#### 题目描述:

十进制大数的加法运算。

#### 输入描述:



输入文件的第 1 行为一个整数  $N$ ，表示输入文件中接下来有  $N$  组数据。每组数据最多包含 100 行。每一行由一个非常长的十进制整数组成，这个整数的长度不会超过 100 个字符而且只包含数字，每组数据的最后一行为 0，表示这组数据结束。

每组数据之间有一个空行。

#### 输出描述:

对输入文件中的每组数据，输出它们的和。每两组数据的输出之间有一个空行。

#### 样例输入:

1

99999278961257987  
126792340765189  
998954329065419876  
432906541  
23  
0

#### 样例输出:

1099080400800349616

#### 分析:

首先，题目中提到，整数的长度不会超过 100 位，所以这些整数只能采用字符数组读入。但在对每位进行求和时，可以采用字符形式，也可以采用整数形式。本题用整数形式处理更方便：对读入的字符数组，以**逆序**的方式将各字符转换成对应的数值存放到整数数组(整数数组中剩余元素的值为 0)，然后再以整数方式求和，最后将求和的结果以相反的顺序输出各位。例如样例输入中的那组数据，逆序转换后每个大数对应到一个整数数组，数组元素表示大数的各位，如图 7.4 所示。注意第 0 位表示整数的最低位。

进位-->

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |     |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-----|
| 2 | 3 | 2 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 2 | 1 | 2 | 1 | 1 | 1 | 0 |   |     |
| 7 | 8 | 9 | 7 | 5 | 2 | 1 | 6 | 9 | 8 | 7 | 2 | 9 | 9 | 9 | 9 | 9 | 0 | 0 | 0 | ... |
| 9 | 8 | 1 | 5 | 6 | 7 | 0 | 4 | 3 | 2 | 9 | 7 | 6 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | ... |
| 6 | 7 | 8 | 9 | 1 | 4 | 5 | 6 | 0 | 9 | 2 | 3 | 4 | 5 | 9 | 8 | 9 | 9 | 0 | 0 | ... |
| 1 | 4 | 5 | 6 | 0 | 9 | 2 | 3 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| 3 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| + |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |     |
| 6 | 1 | 6 | 9 | 4 | 3 | 0 | 0 | 8 | 0 | 0 | 4 | 0 | 8 | 0 | 9 | 9 | 0 | 1 | 0 | ... |

图7.4 大数的加法

在求和时，从各整数数组的第 0 个元素开始累加，并计算进位。在本题中，求和过程中要注意以下两点：

- 1) 计算每位和时，得到的进位可能大于 1，如图 7.4 所示。
- 2) 累加各大数得到的和，其位数可能会比参与运算的大数的位数还要多。稍加分析即可得出结论，如果参与求和运算的大数最大长度为  $\text{maxlen}$ ，因为参加求和运算的大数个数不超过 100 个，所以求和结果长度不超过  $\text{maxlen}+2$ 。因此求和时可以一直求和到  $\text{maxlen}+2$  位，然后去掉后面的 0，再以相反的顺序输出各位整数即可。如图 7.4 所示，这组数据求和的结果逆序后为：1099080400800349616。

#### 代码如下:

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

```

#include <string.h>
int main( )
{
    char buffer[200]; //存储(以字符形式)读入的每个整数
    int array[200][200]; //逆序后的大数(每位是整数形式)
    int answer[200]; //及求得和
    int i, j, k; //循环变量
    int num_integers; //读入整数的个数
    int len, maxlen; //每个整数的长度, 及这些整数的最大长度
    int sum, carry, digit; //每位求和运算后得到的总和, 进位, 及该位的结果

    int N; //测试数据的个数
    scanf( "%d", &N );
    for( k=1; k<=N; k++ )
    {
        maxlen = -1;
        memset( array, 0, sizeof(array) );
        memset( answer, 0, sizeof(answer) );
        for( num_integers = 0; num_integers < 200; num_integers++ )
        {
            gets( buffer );
            if( strcmp(buffer, "0") == 0 ) break;
            len = strlen(buffer);
            if( len>maxlen ) maxlen = len;
            for( i = 0; i < len; i++ ) //逆序存放大数的每位(整数形式)
                array[num_integers][i] = buffer[len - 1 - i] - '0';
        }
        carry = 0;
        for( i = 0; i < maxlen+2; i++ ) //对这些整数的每位进行求和
        {
            sum = carry;
            for( j = 0; j < num_integers; j++ )
                sum += array[j][i];
            digit = sum % 10;
            carry = sum / 10;
            answer[i] = digit;
        }
        for( i = maxlen+2; i >= 0; i-- ) //统计求和结果的位数
        {
            if( answer[i] != 0 ) break;
        }
        while( i >= 0 ) printf( "%d", answer[i--] ); //逆序输出求和的结果
        printf( "\n" );
        if( k<N ) printf( "\n" ); //两个输出块之间有一个空行
    }
    return 0;
}

```

### 7.2.2 高精度数的乘法

联想初等数学里乘法的运算过程。如图 7.5(a)所示。该运算过程有如下特点：

- 1) 多位数的乘法是转换成 1 位数的乘法及加法来实现的，即把第二个乘数的每位数乘以第一个乘数，把得到的中间结果累加起来。
- 2) 第二个乘数的每位数进行乘法运算得到的中间结果，是与第二个乘数参与运算的位右对齐的。如图(a)所示，第二个程序的第 2 位为“7”，参与乘法运算得到的中间结果“8638”是和“7”对齐的。

在用程序实现乘法运算过程时，都要特别注意以上两个特点。

另外，在初等数学里，乘法运算得到的每个中间结果都是处理了进位的：在中间结果里，一出现进位马上累加到高一位，如图(a)中的中间结果“11106”是已经处理了进位的结果。

但是，为方便程序实现，对中间结果的进位处理更方便的做法是等全部中间结果运算完后，再统一处理。如图(b)所示，每个中间结果，“6 12 18 24”、“7 14 21 28”、“8 16 24 32”、“9 18 27 36”都是没有处理进位的，都是第二个乘数的每位乘以第一个乘数每位的原始乘积。等这些中间结果累加后，再一位一位的处理进位。

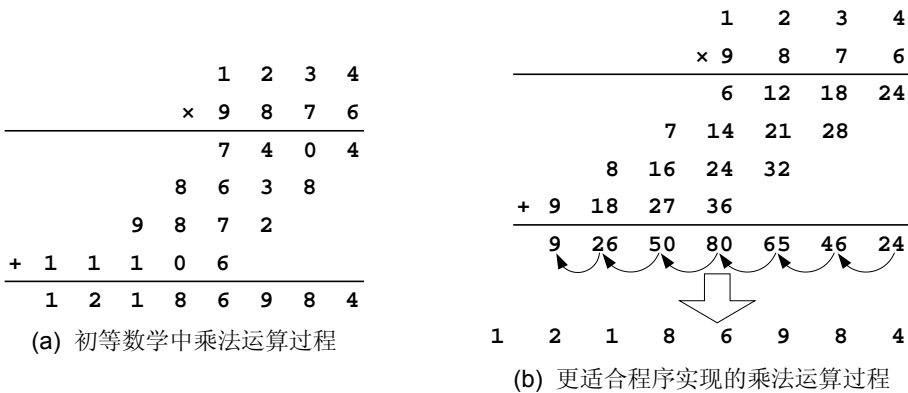


图7.5 大数的乘法

例 7.5 高精度数的乘法

题目描述：

给定两个位数不超过 100 位的正整数，求它们的乘积。

输入描述：

输入文件中包含多个测试数据。每个测试数据占两行，分别为一个正整数，每个正整数的位数不超过 100 位。输入数据一直到文件尾。

输出描述：

对输入文件中的每个测试数据，输出其中两个正整数的乘积。

样例输入：

981567  
32976201  
123456789  
987654321  
123456789987654321  
987654321123456789

样例输出：

32368350686967  
121932631112635269  
121932632103337905662094193112635269

分析：

两个长度不超过 100 位的正整数必须用字符数组 a 和 b 来读入，其乘积不超过 200 位。大整数的乘法运算过程可分为以下几个步骤：

- ① 对读入的字符形式的大整数，把其各位上的数值以整数形式取出来，以相反的顺序存放到一个整型数组里。如图 7.6 所示。
- ② 把第二个乘数中的每位乘以第一个乘数，把得到的中间结果累加起来，注意对齐方式，以及累加每位运算的中间结果时不进位。
- ③ 把累加的中间结果，由低位向高位进位。再把得到的最终结果按相反的顺序转换成字符串输出。

整个过程如图 7.6 所示，图中的标号①、②、③对应到上述 3 个步骤。

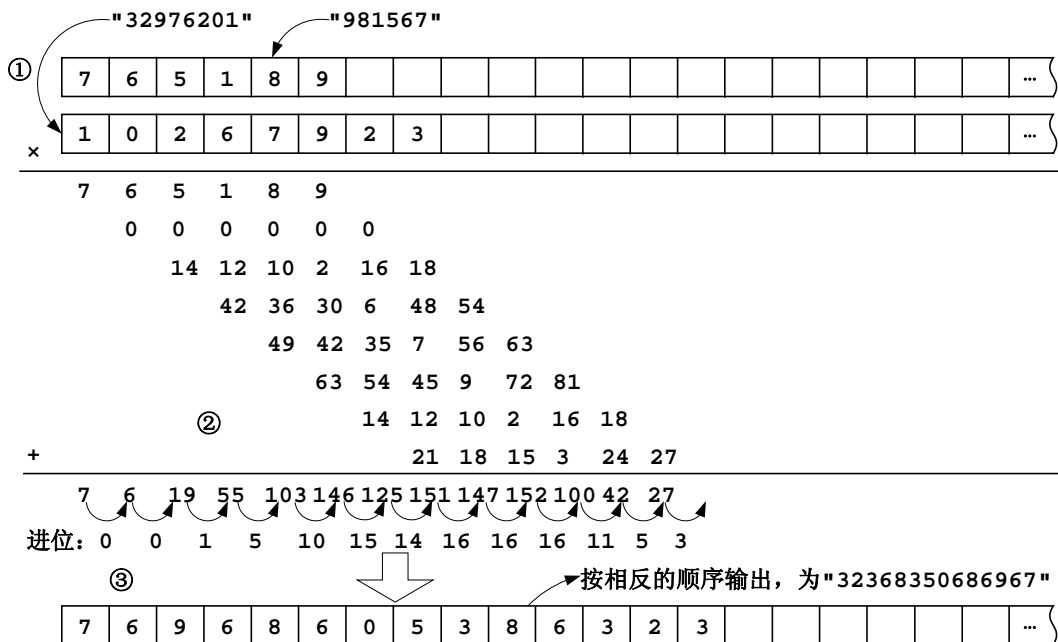


图7.6 大整数的乘法运算过程的实现

代码如下：

```
#include <stdio.h>
#include <string.h>
char a[101], b[101]; //输入的两个正整数(字符形式)
int len_a, len_b; //输入的正整数长度
int ai[101], bi[101]; //输入的两个正整数(以整数形式存储每一位)
int temp[202]; //每一位乘法的中间结果
char product[201]; //乘积
void reverse( char s[ ], int si[ ] ) //以逆序顺序将大数中的各位数存放到整型数组 si
{
    int len = strlen(s);
    for( int i=0; i<len; i++ )
        si[len-1-i] = s[i]-'0';
}
int main( )
{
    int i, j;
    while( scanf( "%s", a ) != EOF )
    {
        scanf( "%s", b );
        len_a = strlen(a); len_b = strlen(b);
        reverse(a, ai); reverse(b, bi);
```

```

memset( temp, 0, sizeof(temp) );
memset( product, 0, sizeof(product) );
for( i=0; i<len_b; i++ ) //用大整数 b 的每位去乘大整数 a
{
    int start = i; //得到的中间结果跟大整数 b 中的位对齐
    for( j=0; j<len_a; j++ )
    {
        temp[start++] += ai[j]*bi[j];
    }
}
for( i=0; i<202; i++ ) //低位向高位进位
{
    if( temp[i]>9 )
    {
        temp[i+1] += temp[i]/10;
        temp[i] = temp[i]%10;
    }
}
for( i=201; i>=0; i-- ) //求乘积的长度
{ if( temp[i] ) break; }
int lenp = i+1; //乘积的长度
for( i=0; i<lenp; i++ ) //将乘积各位转换成字符形式
    product[lenp-1-i] = temp[i]+'0';
product[lenp] = 0; //串结束符标志
printf( "%s\n", product );
}
return 0;
}

```

### 7.2.3 高精度数的除法

高精度数的除法比较复杂，本节仅通过一道例题介绍除数为 1 位数的除法运算。首先联想初等数学里除法的运算过程(除数只有 1 位数)。图 7.7 演示了“351 除以 8”的运算过程。

$$\begin{array}{r}
 0 \quad 4 \quad 3 . 8 \quad 7 \quad 5 \\
 8 \overline{) 3 \quad 5 \quad 1} \\
 \underline{3 \quad 5 \quad 1} \phantom{0} \\
 3 \quad 1 \phantom{0} \\
 \underline{7 \quad 0} \phantom{0} \\
 6 \quad 0 \phantom{0} \\
 \underline{4 \quad 0} \phantom{0} \\
 0
 \end{array}$$

图7.7 除法运算过程(除数只有1位数)

具体过程为：

- 1) 先将被除数“351”最高位除以 8，得到的商为 0，余数为 3。
- 2) 把余数和被除数“351”的次高位组合，为“35”，除以 8，得到的商为 4，余数为 3。
- 3) 把余数和被除数“351”的最后一位组合，为“31”，除以 8，得到的商为 3，余数为 7，不为 0，所以还没有除尽，要补 0，图中所有补 0 均用斜体标明。补 0 前的商为整数部分，补 0 后的商为小数部分。
- 4) 补 0 后为“70”，除以 8，得到的商为 8，余数为 6，再补 0。

5) 补 0 后为“60”，除以 8，得到的商为 7，余数为 4，再补 0。

6) 补 0 后为“40”，除以 8，得到的商为 5，余数为 0。

至此，整个除法运算完毕，得到的商为 43.875。

### 例 7.6 八进制小数(Octal Fractions)

题目来源：

South Africa 2001

题目描述：

八进制下的小数可以精确地转换成十进制的小数。例如，八进制里的 0.75 转换成十进制，结果为 0.963125 ( $7/8 + 5/64$ )。小数点右边有  $n$  位的八进制小数，转换成十进制后，小数点右边不超过  $3n$  位。

编写程序将  $[0,1]$  内的八进制小数转换成对应的十进制小数。

输入描述：

输入文件包含若干行，每行是一个八进制小数。每个八进制小数的形式为：0.d<sub>1</sub>d<sub>2</sub>d<sub>3</sub> ... d<sub>k</sub>，其中 d<sub>i</sub> 为八进制数字(0..7)，k 没有限制。

输出描述：

对输入文件中的每个八进制小数，按照如下的格式输出：

0.d<sub>1</sub>d<sub>2</sub>d<sub>3</sub> ... d<sub>k</sub> [8] = 0.D<sub>1</sub>D<sub>2</sub>D<sub>3</sub> ... D<sub>m</sub> [10]

等号左边就是输入文件中的八进制小数，右边是对应的十进制小数，末尾没有 0，也就是说 D<sub>m</sub> 不为 0。

样例输入：

0.75  
0.123  
0.0001  
0.01234567

样例输出：

0.75 [8] = 0.953125 [10]  
0.123 [8] = 0.162109375 [10]  
0.0001 [8] = 0.000244140625 [10]  
0.01234567 [8] = 0.020408093929290771484375 [10]

分析：

八进制小数转换成十进制小数，其原理本来是按权值展开，小数点后第 1 位的权值为  $8^{-1} = 0.125$ ，第 2 位的权值为  $8^{-2} = 0.015625$ 。因此  $0.75 [8] = 7 \times 0.125 + 5 \times 0.015625 = 0.953125$ 。但在本题中，如果按照这种思路去求解，不容易实现。

更好的方法是转换成除法运算，小数点后第 1 位的权值为  $8^{-1}$ ，相当于除以 8；第 2 位的权值为  $8^{-2}$ ，相当于除以两次，...。具体过程为：循环除 8，即从八进制小数的最后一位开始除以 8，把得到的结果加到前一位，再除以 8；...；一直到小数点后第 1 位为止。假设读入的八进制为 0.d<sub>1</sub>d<sub>2</sub>d<sub>3</sub> ... d<sub>n</sub>，转换后的十进制为 0.D<sub>1</sub>D<sub>2</sub>D<sub>3</sub> ... D<sub>m</sub>，则循环除 8 的公式为：

$$0.D_1D_2D_3 \dots D_m = (d_1 + (d_2 + (d_3 + \dots (d_{n-1} + d_n/8)/8 \dots)/8)/8)/8$$

例如，对八进制小数 0.75 [8]，其循环除 8 的运算过程为：(7 + 5/8)/8。

又如对八进制小数 0.123 [8]，其循环除 8 的运算过程为：(1 + (2 + 3/8)/8)/8。

以 0.123 [8] 为例讲解具体实现过程，如图 7.8 所示。注意得到的十进制小数都不保留前面的 0 及小数点。

(1) 先读入最后一位 num = 3，按照图 7.7 所示的方法进行除法运算，其中补 0 (在图 7.8 中补 0 也用斜体标明) 相当于乘以 10。即反复将 num 乘以 10，再除以 8，记录其商，直到余数为 0 为止，得到的结果是 375，如图(a)所示，这表示 0.375。这实际上是 0.3 [8] 对应的十进制小数。

(2) 接下来读入的八进制位 num = 2，这时要求的是 2.375/8，转换成求 2375/8，其运算

过程如图(b)所示。得到的结果为 296875，这表示 0.296875。这实际上是 0.23 [8]对应的十进制小数。

(3) 接下来读入的八进制位  $\text{num} = 1$ ，这时要求的是  $1.296875/8$ ，转换成求  $1296875/8$ ，其运算过程如图(c)所示。得到的结果为 162109375，这表示 0.162109375。这实际上是 0.123 [8]对应的十进制小数。

具体实现时每位的除 8 运算要分成两个过程：① 除 8 得到整数部分；② 对前面得到的余数补 0 继续除 8 直到余数为 0。当然最低位的除 8 运算只有第②个过程。

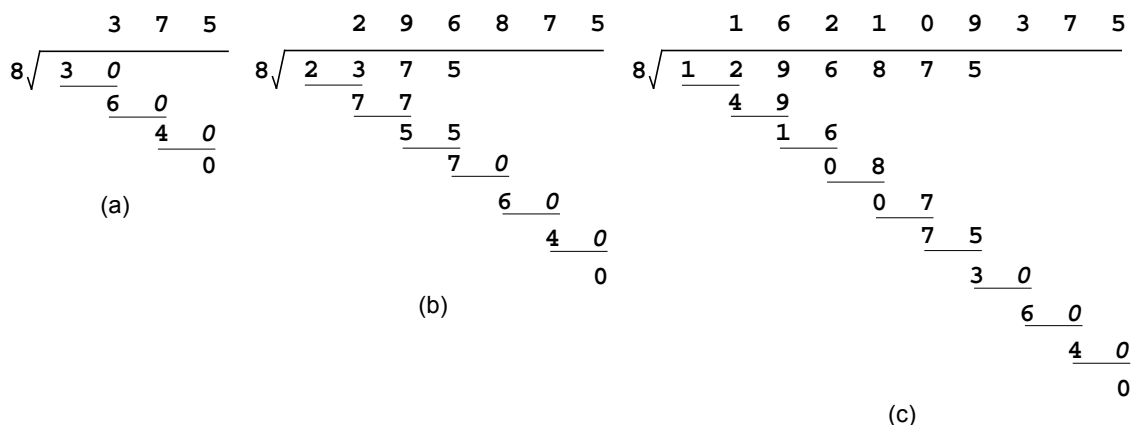


图7.8 八进制小数转换成十进制小数(不保留小数点及前面的0)

图 7.9 演示了八进制小数“0.123”转换成十进制小数的实现过程。其中，src 字符数组存储读入的八进制小数，即“0.123”；dest 字符数组存储转换后的十进制小数(去掉小数点及前面的 0)。具体实现过程为：

首先读入最低位，“3”，补 0 除 8 直到余数为 0，得到的商为“375”，存储在 dest 数组中，如图(b)所示。

然后读入八进制位“2”，第①个过程：将“2”与 dest 数组中的“3”组合后为“23”，除 8 后商为“2”，余数为“7”；将余数“7”与 dest 数组中的“7”组合后为“77”，除 8 后商为 2，余数为“5”；将余数“5”与 dest 数组中的“5”组合后为“55”，除 8 后商为 6，余数为“7”，此时 dest 数组的值为“296”，如图(c)所示。第②个过程：将前面得到的余数“7”补 0 除 8 直到余数为 0，得到的商为“875”，添加在 dest 数组后面，如(d)所示。

最后，读入八进制位“1”，其处理过程与八进制位“2”的处理类似。

src数组

|   |   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0 | . | 1 | 2 | 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

(a) 读入的八进制小数

dest数组

|   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 3 | 7 | 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

(b) 最低位“3”除8的结果

dest数组

|   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 2 | 9 | 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

(c) “2375”除8的结果(整数部分)

dest数组

|   |   |   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 2 | 9 | 6 | 8 | 7 | 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

(d) 把余数7除8的结果添在后面

dest数组

|   |   |   |   |   |   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|
| 1 | 6 | 2 | 1 | 0 | 9 | 3 | 7 | 5 |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|

(e) 最终结果

图7.9 八进制小数转换成十进制小数的实现过程

代码如下：

```

#include <stdio.h>
#include <string.h>
const int MAX_LENGTH = 20;
int main( )
{
    char src[MAX_LENGTH]; //读进来的八进制小数(字符形式)
    int i, j; //循环变量
    while( scanf("%s",src) != EOF ) //设读入的八进制小数为 0.d1d2d3...dn
    {
        char dest[MAX_LENGTH] = {'0'}; //存放转化后的十进制数（无 0.）
        int num; //读取出来的每个八进制位(整数形式)
        int index = 0; //前一个八进制位除 8 后 dest 数组中的位数
        int len = 0; //当前这个八进制位除 8 后 dest 数组中的位数
        int temp; //当前这个八进制位与前一位运算结果的每一位组合得到的值
        for( i = strlen(src)-1; i > 1; i-- )
        {
            num = src[i]-'0'; //取第 i 位上的八进制数字
            for(j=0; j < index; j++)//d1~dn-1 的处理
            {
                temp = num*10 + dest[j]-'0';
                dest[j] = temp/8+'0';
                num = temp%8;
            }
            while( num ) //d1~dn 的处理(余数的处理：补 0 再除直到商为 0 为止)
            {
                num *= 10;
                dest[len++] = num/8+'0';
                num %= 8;
            }
            index = len;
        }
        dest[len] = '\0';//串结束符标志
        printf("%s [8] = 0.%s [10]\n", src, dest); //输出
    }
    return 0;
}

```

## 练习

### 7.4 火星上的加法(Martian Addition)

#### 题目描述：

在 22 世纪，科学家发现在火星上居住着智能程度很高的居民。火星星人很喜欢做算术题。每年他们都要举行算术比赛。比赛的任务是计算两个 100 位数的和，谁用的时间最少，谁将是胜利者。今年他们邀请地球上的人类也参加这项赛事。

作为地球的代表，你被送到火星上向火星星人展示人类的智力。幸运的是，你把你的手提电脑带上了，这帮了你的大忙。现在剩下的问题是编写程序计算两个 100 位数的和。在编程之前要告诉你的是，火星星人用的是 20 进制，因为他们有 20 个手指。

#### 输入描述：



输入文件中有多对 20 进制的数，每个数占一行。20 进制采用的数码是 0~9，以及小写字母 a~j，小写字母分别代表十进制中的 10~19。

输入文件中的每个数位数不超过 100 位。

#### 输出描述:

对每对 20 进制的数，输出占一行，为这两个数的和。

#### 样例输入:

```
1234567890
abcdefghij
99999jjjjj
9999900001
```

#### 样例输出:

```
bdfi02467j
iiiij00000
```

### 7.5 总和(Total Amount)

#### 题目描述:

给定一组标准格式的货币金额，计算其总和。标准格式为：

1. 每个金额以符号“\$”开头；
2. 仅当金额小于 1 时，金额有前导 0；
3. 每个金额小数点后有两位数；
4. 金额小数点前的各位，以 3 位一组进行分组，并且以逗号分隔开，最前面的一组可能只有 1 位或 2 位。

#### 输入描述:

输入文件包含多个测试数据。每个测试数据的第 1 行为一个整数 N， $1 \leq N \leq 10000$ ，表示该测试数据中金额的个数。接下来的 N 行表示 N 个金额。所有的金额，包括最终求得的总和范围是 \$0.00~\$20,000,000.00(含)。最后一行，N 的值为 0，表示输入数据结束。

#### 输出描述:

对输入文件中的每个测试数据，输出其总和。

#### 样例输入:

```
2
$1,234,567.89
$9,876,543.21
3
$0.01
$0.10
$1.00
0
```

#### 样例输出:

```
$11,111,111.10
$1.11
```

### 7.6 循环数(Round and Round We Go)

#### 问题描述:

所谓循环数，就是一个具有 n 位的整数，如果用 1~n 之间的任何一个数去乘以它，得到的 n 个整数，每个整数的各位数字的顺序是原来这个整数各位数字的“循环”。也就是说，如果将最后一位连到第 1 位，构成一个环状，这样，得到的这 n 个整数跟原来的整数就是一样的，尽管它们可能是从不同的位置开始的。

例如，142857 就是循环数，用 1~6 去乘以它，得到的 6 个整数为：

```
142857 × 1 = 142857
142857 × 2 = 285714
```

$142857 \times 3 = 428571$

$142857 \times 4 = 571428$

$142857 \times 5 = 714285$

$142857 \times 6 = 857142$

这 6 个整数如果将最后一位连到第 1 位, 则是同一个数。

编写程序, 判断一个数是否为循环数。

#### 输入描述:

输入文件包含若干行, 每行是一个整数, 整数的位数范围是 2 到 60。注意, 整数的前导 0 不能忽略, 它们被认为是整数的一部分。因此, “01” 是一个 2 位数, 而 “1” 是一个 1 位数。

#### 输出描述:

对输入文件中的每个整数, 输出一行, 判断该整数是否为循环数。输出格式如样例输出所示。

#### 样例输入:

142857

142856

01

0588235294117647

#### 样例输出:

142857 is cyclic

142856 is not cyclic

01 is not cyclic

0588235294117647 is cyclic

### 7.7 余数(Basic Remains)

#### 题目描述:

给定一个基数 B, 和 B 进制下的两个非负整数 P 和 M, 求 P 对 M 的余数, 求得的余数也是 B 进制下的数。

#### 输入描述:

输入文件包含多个测试数据。每个测试数据占一行, 包含 3 个无符号整数。第 1 个数为 B, 十进制数, 范围在 2~10; 第 2 个数, P, 为 B 进制下的数, 最多包含 1000 位, 每位都是 0~B-1 之内的数码; 第 3 个数, M, 为 B 进制下的数, 最多包含 9 位。输入数据最后一行为 0, 表示输入结束。

#### 输出描述:

对每个测试数据, 输出一行, 为在 B 进制下求得的 P 对 M 取余的结果。

#### 样例输入:

2 1100 101

10 123456789123456789123456789 1000

0

#### 样例输出:

10

789

## 7.3 其他高精度题目解析

本节再讲解一些需要按照大数方法来处理的题目。

### 7.3.1 数列问题

很多数列的增长速度都是很快的, 比如在第 1 章 1.8 节就提到 Fibonacci 数列的增长速度是很快的, 第 48 个数(4807526976)就超出了无符号整数所表示的范围(0~4294967295)。所以要求这些数列的某一项, 有时需要采用大数来处理, 如例 7.7。

**例 7.7 Fibonacci 数(Fibonacci Numbers)****题目来源:**

University of Waterloo Local Contest 1996.10.05

**题目描述:**

Fibonacci 数列的第 1、2 项都为 1，此后每一项都是前两项之和。即： $f(1) = 1, f(2) = 1, f(n) = f(n-1) + f(n-2), n > 2$ 。

给定一个数  $N$ ，输出 Fibonacci 数列中第  $N$  个数。 $N$  的大小保证得到的第  $N$  个 Fibonacci 数的位数不超过 1000 位。

**输入描述:**

输入文件包含多个测试数据，每个测试数据占一行，为一个正整数  $N$ 。

**输出描述:**

对输入文件中的每个整数  $N$ ，输出 Fibonacci 数列中的第  $N$  项。

**样例输入:**

40  
100

**样例输出:**

102334155  
354224848179261915075

**分析:**

在例 1.29、例 1.38 及例 1.41 中介绍了 Fibonacci 数列的递推方法。本题采用大数方法进行处理，设表示 Fibonacci 数列连续 3 项的字符数组分别为  $n1$ 、 $n2$  和  $n3$ ，则递推过程如下：

```
add( n1, n2, n3 );    //由第 1 项和第 2 项递推到第 3 项
strcpy( n1, n2 );    //推导完，第 2 项变成第 1 项
strcpy( n2, n3 );    //推导完，第 3 项变成第 2 项
```

以下代码在表示 Fibonacci 数列中各项及求和时都是以逆序方式表示的，在输出时以相反的顺序输出递推的结果。例如，Fibonacci 数列中第 38、39 项分别是 39088169 和 63245986，由这两项递推出第 40 项。在程序中字符数组  $n1$  和  $n2$  的内容分别为 "96188093" 和 "68954236"，将这两个字符数组所表示的大数相加，得到的结果是 "551433201"，再以相反的顺序输出各数组元素，得到的就是第 40 项。

由于 Fibonacci 数列中各项是递推出来的，并非任意数值，所以求和过程可以简化。设  $n1$  的长度为  $len$ ，在求和时只需计算  $n1$  和  $n2$  前  $len$  位各位和，再判断两种特殊情况：①  $n2$  比  $n1$  多 1 位；②  $n1$  加上  $n2$  后最高位还有进位。

**代码如下:**

```
#include <stdio.h>
#include <string.h>
#define MAXSIZE 1001

char n1[MAXSIZE], n2[MAXSIZE], n3[MAXSIZE]; //表示两个大数及其和的字符数组
void set1( char *p )    //重新设置字符数组 n1 和 n2
{
    memset( p, 0, MAXSIZE );
    p[0] = '1';
}

//求两个大数和，这两个大数存储在字符数组 n1 和 n2 中，结果保存在字符数组 n3 中
void add( char *n1, char *n2, char *n3 )
```

```

{
    memset( n3, 0, MAXSIZE );
    int len = strlen(n1);
    int i;
    for( i=0; i<len; i++ )
    {
        n3[i] = n1[i]-'0' + n2[i]-'0' + n3[i];
        if( n3[i] >= 10 )
        {
            n3[i] = n3[i] - 10 + '0';
            n3[i+1] = 1; //进位
        }
        else n3[i] += '0';
    }
    if( n2[i]!=0 ) n3[i] = n2[i] + n3[i]; //第 2 个数还有 1 位没有运算
    else if( n3[i]!=0 ) n3[i] += '0'; //n1 加上 n2 后最高位还有进位
}
void prn(char *p) //输出字符数组 p 中的大数
{
    int i = strlen(p)-1;
    for( ; i>=0; i-- ) //输出大数的每一位数字
        printf( "%c", p[i] );
    printf( "\n" );
}
int main( )
{
    int i, N; //循环变量及输入的正整数 N
    while( scanf("%d",&N)!=EOF )
    {
        if( N==1 || N==2 )
        { printf( "1\n" ); continue; }
        set1( n1 );
        set1( n2 );
        for( i=3; i<=N; i++ ) //每次递推一个数
        {
            add( n1, n2, n3 );
            strcpy( n1, n2 );
            strcpy( n2, n3 );
        }
        prn( n3 );
    }
    return 0;
}

```

### 7.3.2 其他题目

有些题目本身没有告诉数据的范围，无法判断是否要按高精度处理，如例 7.8。对于这种题目，可以先按常规的数据形式(如整数)来处理。如果正确的程序提交得到的结果是 **Wrong Answer**，说明输入文件中的数据有可能超出了整数的表示范围，则可以试着按照大数来处理，即用字符数组存储读入的数据。

**例 7.8 Niven 数(Niven Numbers)****题目来源:**

East Central North America 1999, Practice

**题目描述:**

如果一个数,其各位和能整除它本身,则这个数称为 **Niven 数**。例如,十进制下的整数 111 就是一个 **Niven 数**,因为,其各位和为 3,3 能整除 111。对其他进制下的数,我们也可以定义 **Niven 数**。如果在  $b$  进制下,某个数的各位和能整除它本身,则在  $b$  进制下这个数就称为 **Niven 数**。

给定基数  $b(2 \leq b \leq 10)$ , 和一个数,判断这个数在  $b$  进制下是否为 **Niven 数**。

**输入描述:**

输入文件包含多组数据。输入文件的第 1 行为一个整数  $N$ ,表示输入文件中接下来有  $N$  组数据。每组数据有若干行,每一行首先是基数  $b$ ,然后是一串数字,代表  $b$  进制下的一个整数,这个整数没有前导 0。每组数据的最后一行为 0,表示这组数据结束。

每组数据之间有一个空行。

**输出描述:**

对输入文件中的每组数据的每一行,如果该整数在  $b$  进制下是 **Niven 数**,输出 **yes**,否则输出 **no**。每两组数据的输出之间有一个空行。

**样例输入:**

```
1
10 111
10 123
2 110
8 2314
0
```

**样例输出:**

```
yes
no
yes
no
```

**分析:**

题目中并没有告诉读入的( $b$  进制下)整数的范围是多少,所以可以先按整数方法进行处理:以十进制方式读入  $b$  进制下的整数,先求其各位数之和(十进制),然后将  $b$  进制的整数转换成十进制,最后取余。提交得到的结果如果是 **Wrong Answer**,说明测试数据超出了整数的范围,必须按大数处理。

其实,由于基数  $b$  最小可以取到 2,11 位二进制数,如 10000000000,上述方法在读入时是采用十进制方式读入,实际上就已经超出了整数的取值范围,因此可以进一步判断出必须按大数进行处理。

该题大数处理方法如下。

对  $b$  进制下的整数,采用字符数组读入。由于题目中未明确给定  $b$  进制数的最大长度,可以先将数组长度设为 50,如果提交时溢出再往上加。累加该整数的各位数之和时,只需将字符数组中各数字字符对应的数值累加即可,得到的和 **sumdigit** 是十进制的。

接下来需要将  $b$  进制数转换成十进制数,然后再对 **sumdigit** 取余数 **remain**。其实余数可以在转换成十进制的过程中得到,方法是利用同余理论,详见例 4.8 中的分析。

设  $b$  进制数为  $B_n B_{n-1} B_{n-2} \dots B_0$ , 则该  $b$  进制数转换成十进制数的过程可以用下式表示:

$$(((B_n * b + B_{n-1}) * b + B_{n-2}) * b + \dots) * b + B_0.$$

则取余数 **remain** (初始为 0) 的过程可以表示为:

```

remain = ( remain + Bn ) % sumdigit
remain = ( remain*b + Bn-1 ) % sumdigit
remain = ( remain*b + Bn-2 ) % sumdigit
...

```

代码如下:

```

#include <stdio.h>
#include <string.h>
int main( )
{
    int b;          //b 进制
    char s[100];    //保存输入的数
    int N;          //有 n 组数
    int j, k;       //循环变量
    scanf( "%d", &N );
    for( j = 0; j < N; j++ )
    {
        while( scanf("%d", &b) )
        {
            if( b==0 ) break;
            scanf( "%s", s );
            int len = strlen(s);
            int sumdigit = 0; //各位数之和
            for( k = 0; k < len; k++ )
                sumdigit += s[k] - '0';

            int remain = 0; //remain 是该数在 10 进制下的值除以 sumdigit 后的余数
            for( k = 0; k < len; k++ ) //利用同余理论求余数
                remain = (remain * b + s[k] - '0') % sumdigit;

            if( remain == 0 ) printf( "yes\n" );
            else printf( "no\n" );
        }
        if( j != N-1 ) printf( "\n" );
    }
    return 0;
}

```

## 练习

### 7.8 有多少个 Fibonacci 数(How Many Fibs?)

#### 题目描述:

Fibonacci 数的定义为:  $f(1) = 1, f(2) = 1, f(n) = f(n-1) + f(n-2), n > 2$ 。

给定两个整数  $a$  和  $b$ , 计算在区间 $[a,b]$ 之间有多少个 Fibonacci 数。

#### 输入描述:

输入文件中包含多个测试数据。每个测试数据占一行, 为两个非负整数 $a$ 和 $b$ 。当 $a = b = 0$ 时表示输入结束。否则 $a$ 和 $b$ 的值满足:  $a \leq b \leq 10^{100}$ 。 $a$ 和 $b$ 都没有多余的前导 0。

#### 输出描述:

对输入文件中的每个测试数据，输出一行，为 $[a,b]$ 区间中 Fibonacci 数  $F_i$  的个数(即  $a \leq F_i \leq b$ )。

**样例输入：**

```
10 100
1234567890 9876543210
0 0
```

**样例输出：**

```
5
4
```

## 7.9 颠倒数的和(Adding Reversed Numbers)

**题目描述：**

M 国古代喜剧家喜欢喜剧胜于悲剧。不幸的是，大部分这个时期的戏剧都是悲剧。因此喜剧家们决定将其中一些悲剧改编成喜剧。显然，这个工作是很困难的，因为要保持戏剧的基本意义完整，尽管戏剧中所有的事物要变为相反的。例如悲剧中的数，它必须被转换成它的颠倒形式，使得在喜剧中能被接受。

颠倒数是由阿拉伯数字组成的，但其数字的顺序颠倒了。即原来的第一位数字在颠倒数中变为最后一位数字。例如，在悲剧中主人公有 1245 个草莓，而现在他有 5421 个。注意所有前导的 0 被省略。也就是，如果一个数以 0 结尾，在颠倒数中 0 将丢失。(如 1200 颠倒后得到 21)。这样颠倒数将不可能有后导的 0。

喜剧家们需要对颠倒数进行运算。你的任务是编写一个程序，将两个颠倒数相加，并输出它们的颠倒和。当然，结果不是唯一的，因为每一个颠倒数都可能是多个数的颠倒形式，例如，颠倒数 21 可以是 12、120 或 1200 等等数的颠倒。因此，我们必须假定，颠倒是没有 0 丢失的，这样颠倒数 21，在颠倒前是 12。

**输入描述：**

输入文件中包含 N 组测试数据。输入文件的第 1 行为正整数 N。接下来是 N 个测试数据。每个测试数据占一行，为两个正整数，用空格隔开。这就是要求和的两个颠倒数。

**输出描述：**

对输入文件中的每组数据，输出一行，为一个整数，表示两个颠倒数的和(注意它们的和也是颠倒数)，忽略所有的前导 0。

**样例输入：**

```
3
24 1
4358 754
305 794
```

**样例输出：**

```
34
1998
1
```

**注意：**

题目没有明确告诉参与运算的颠倒数位数最多有多少位，可以先按正常的整数来处理：对两个整数颠倒，求和后再颠倒。如果正确的程序提交得到的结果是 Wrong Answer，说明评判系统中输入文件中的数据超出了整数的表示范围，则必须按照大数来处理，即必须用字符数组存储读入的数据。

## 7.10 数基(Digital Roots)

**题目描述：**

一个正整数的数基是通过求其各位数字之和得到的。如果各位数字之和是 1 位数，则该和就是这个正整数的数基。如果各位数字之和不止 1 位数，则再求这个和的各位数字之和，重复这个过程，直到各位数字之和为 1 位数。最终得到的结果就是该整数的数基。

例如，对正整数 **24**，它的各位数字之和为 **6**，是 **1** 位数，因此 **24** 的数基就是 **6**。而对正整数 **39**，其各位数字之和为 **12**，为 **2** 位数，因此还要对 **12** 求其各位数字之和，最终得到的数基是 **3**。

**输入描述：**

输入文件中包含若干个正整数，每个占一行。输入文件最后一行为 **0**，代表输入结束。

**输出描述：**

对输入文件中的每个正整数，输出其数基。

**样例输入：**

24  
39  
0

**样例输出：**

6  
3



## 第 8 章 递归与搜索

递归是一种重要的算法思想。递归既可以实现递推过程，也可以实现求解诸多问题的通用思路—搜索。本章首先介绍递归的基本思想及递归函数的执行过程，然后介绍递归思想在竞赛题目中的应用；本章还介绍了通过递归函数实现搜索的方法及搜索方法在竞赛题目中的应用，以及用递归方法求解排列组合问题。

### 8.1 递归的基本思想

#### 8.1.1 什么是递归

在数学上，求  $n$  的阶乘，有两种表示方法：

$$\textcircled{1} \quad n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

$$\textcircled{2} \quad n! = n \times (n-1)!$$

这两种表示方法实际上对应到两种不同的算法思想。

在第①种表示方法中，求  $n!$  要反复把 1、2、3、...、 $(n-2)$ 、 $(n-1)$ 、 $n$  累乘起来，是循环的思想，要用循环结构来实现，代码如下：

```
int n, F = 1;
scanf( "%d", &n );
for( i=1; i<=n; i++ ) F = F*i;
printf( "%d 的阶乘为%d", n, F );
```

在第②种表示方法中，求  $n!$  时需要用到 $(n-1)!$ 。如果有一个函数 **Factorial** 能实现求  $n$  的阶乘，其原型为：`int Factorial( int n );`，则该函数在求  $n!$  时要使用到表达式： $n * \text{Factorial}(n-1)$ ，`Factorial(n-1)`表示调用 `Factorial( )`函数去求 $(n-1)!$ 。具体代码见例 8.1。

**例 8.1** 递归求阶乘。

```
#include <stdio.h>
int Factorial( int n )
{
    if( n<0 ) return -1;
    else if( n==0 || n==1 ) return 1;
    else return n*Factorial(n-1); //递归调用 Factorial 函数
}
int main( )
{
    int A;
    scanf( "%d", &A );
    printf( "%d!=%d\n", A, Factorial(A) );
    return 0;
}
```

该程序的运行示例如下：

4↵

4!=24

在例 8.1 中，`Factorial( )`函数有一个特点，它在执行过程中又调用了 `Factorial( )`函数，这种函数称为**递归函数**。

具体来说，在执行一个函数过程中，又直接或间接地调用该函数本身，如图 8.1 所示，这种函数调用称为**递归调用**；包含递归调用的函数称为**递归函数**。

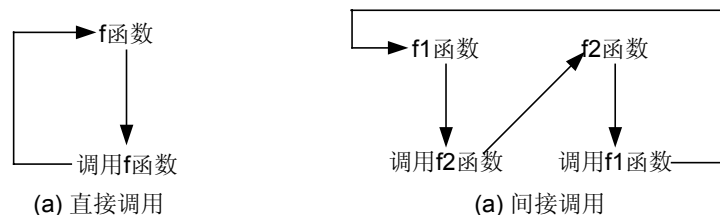


图8.1 直接调用函数本身与间接调用函数本身

例 8.1 就是直接调用函数本身的例子。假设要求  $3!$ ，其完整的执行过程如图 8.2 所示，具体过程为：

- ① 执行 main 函数的开头部分；
- ② 当执行到 Factorial 函数调用 “Factorial(3)” 时，流程转而去执行 Factorial(3)函数，并将实参 3 传递给形参 n；
- ③ 执行 Factorial(3)函数的开头部分；
- ④ 当执行到递归调用 Factorial(n-1)函数时，此时  $n-1=2$ ，所以要转而去执行 Factorial(2)函数；
- ⑤ 执行 Factorial(2)函数的开头部分；
- ⑥ 当执行到递归调用 Factorial(n-1)函数时，此时  $n-1=1$ ，所以要转而去执行 Factorial(1)函数；
- ⑦ 执行 Factorial(1)函数，此时因为形参 n 的值为 1，所以将执行 return 语句，从而返回 1，不再递归调用下去，因此，Factorial(1)函数执行完毕，返回到上一层，即返回 Factorial(2)函数中；
- ⑧ 执行 Factorial(2)函数中的 return 语句，求得表达式的值为 2，并将其返回到 Factorial(3)函数中；
- ⑨ 执行 Factorial(3)函数中的 return 语句，求得表达式的值为 6，并将其返回到 main 函数中；
- ⑩ 返回到 main 函数中后，函数调用 “Factorial(3)” 执行完毕，求得  $3!$  为 6，继续执行 main 函数的剩余部分直到整个程序执行完毕。

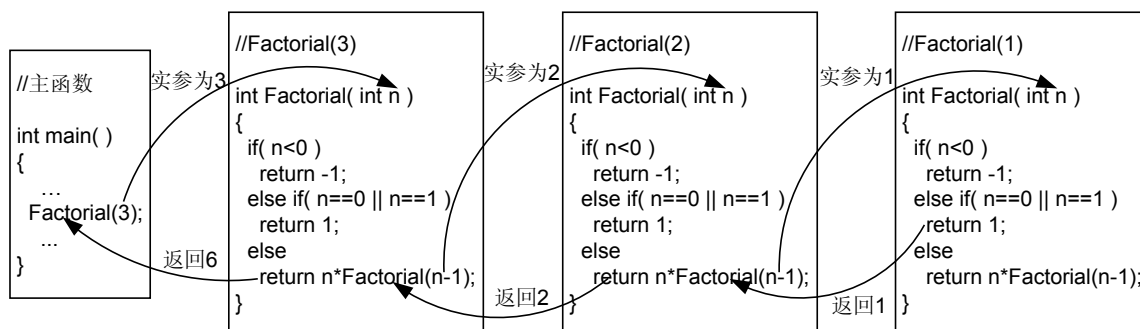


图8.2 Fractorial(3)的执行过程

在例 8.1 中，求  $n!$  转换成求  $(n-1)!$ ，而  $(n-1)!$  又可以转换成求  $(n-2)!$ ，...，一直到  $1! = 1$ 。如果问题的求解可以转换成规模更小(如例 8.1)的、或者更趋向于求出解(如例 8.8 中的搜索算法)的同类子问题的求解，并且从这些子问题的解可以构造出原问题的解。这种求解问题的思想称为**递归思想**。

递归思想需要用递归函数来实现。接下来再列举几个例子，详细分析几道用递归思想求解

的例题。

### 8.1.2 例题解析及递归函数设计

**例 8.2** 猴子吃桃问题。猴子第 1 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 2 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另加一个。到第 10 天早上想再吃时，就只剩下一个桃子了。求第 1 天共摘了多少个桃子。

**分析：**

假设  $A_i$  为第  $i$  天吃完后剩下的桃子的个数， $A_0$  表示第一天共摘下的桃子，本题要求的是  $A_0$ 。有以下递推式子：

$A_0 = 2 \times (A_1 + 1)$      $A_1$ : 第 1 天吃完后剩下的桃子数

$A_1 = 2 \times (A_2 + 1)$      $A_2$ : 第 2 天吃完后剩下的桃子数

.....

$A_8 = 2 \times (A_9 + 1)$      $A_9$ : 第 9 天吃完后剩下的桃子数

$A_9 = 1$

以上递推过程可分别用非递归思想(循环结构)和递归思想实现。

**用循环结构实现：**

如果  $x_1$ ,  $x_2$  表示前后两天吃完后剩下的桃子数，则有递推关系： $x_1 = (x_2 + 1) \times 2$ 。从第 9 天剩下 1 个桃子，反复递推 9 次，则可求第 1 天共摘下的桃子数。这里包含了反复的思想，可以用循环结构来实现，代码如下：

```
#include <stdio.h>
int main( )
{
    int day, x1, x2;
    day = 9;
    x2 = 1;
    while( day>0 )
    {
        x1 = (x2+1)*2; // 第 1 天的桃子数是第 2 天桃子数加 1 后的 2 倍
        x2 = x1;
        day--;
    }
    printf( "total=%d\n", x1 );
    return 0;
}
```

**用递归思想实现：**

前面所述的递推关系也可以采用下面的方式描述。假设第  $n$  天吃完后剩下的桃子数为  $A(n)$ ，第  $n+1$  天吃完后剩下的桃子数为  $A(n+1)$ ，则存在的推关系： $A(n) = (A(n+1) + 1) \times 2$ 。这种递推关系可以用递归函数实现，代码如下：

```
#include <stdio.h>
int A( int n )
{
    if( n>=9 ) return 1;
    else return( 2*( A(n+1)+1 ) );
}
```

```
int main( )
{
    printf( "total=%d\n", A(0) );
    return 0;
}
```

以上 2 个程序的输出结果均为：

total=1534

**例 8.3** 采用递归思想递推 Fibonacci 数列中的每一项，并输出前 20 项的值。

在例 1.29、例 1.38 及例 1.41 中分别采用不同的方法递推出 Fibonacci 数列各项。Fibonacci 数列各项的递推关系式是： $F(n) = F(n-1) + F(n-2)$ ，这种递推关系式很适合用递归函数实现。代码如下：

```
#include <stdio.h>
int Fibonacci( int n );
void main( )
{
    int i;
    int f[20] = { 0 };
    for( i=0; i<20; i++ )    //调用递归函数求每项
        f[i] = Fibonacci( i );
    for( i=0; i<20; i++ )
    {
        if( i%10==0 && i!=0 ) printf( "\n" ); //每行输出 10 个数据
        printf( "%6d", f[i] );    //每个数据占 6 个字符的宽度
    }
    printf( "\n" );
}
int Fibonacci( int n )
{
    if( n==0 || n==1 ) return 1;
    else return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

该程序的输出结果为：

```
1      1      2      3      5      8      13      21      34      55
89     144     233     377     610     987     1597     2584     4181     6765
```

**例 8.4** 输入两个正整数，求其最大公约数。

数论中有一个求最大公约数的算法称为**辗转相除法**，又称欧几里德(Euclid)算法。其基本思想及执行过程为(设  $m$  为两正整数中较大者， $n$  为较小者)：

- (1) 令  $u = m$ ， $v = n$ ；
- (2) 取  $u$  对  $v$  的余数，即  $r = u \% v$ ，如果  $r$  的值为 0，则此时  $v$  的值就是  $m$  和  $n$  的最大公约数，否则执行第(3)步；
- (3)  $u = v$ ， $v = r$ ，即  $u$  的值为  $v$  的值，而  $v$  的值为余数  $r$ 。并转向第(2)步。

整个算法的流程图如图 8.3 所示。

例如，假设输入的两个正整数为 18 和 33，则  $m = 33$ ， $n = 18$ ，用辗转相除法求最大公约数的过程如图 8.4 所示。

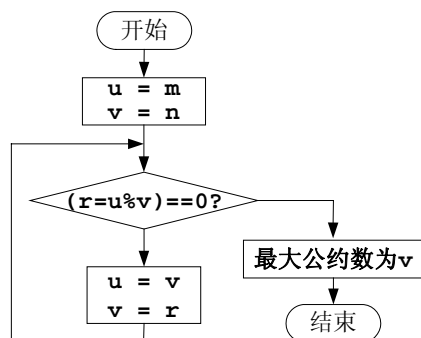


图8.3 辗转相除法的流程图

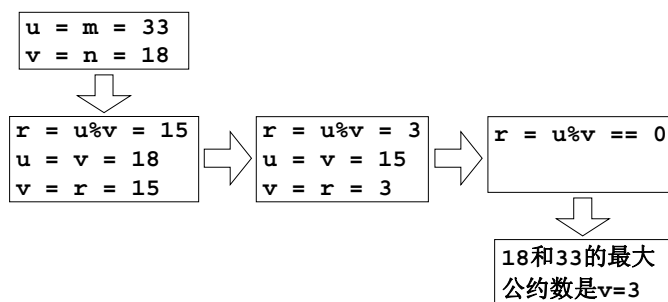


图8.4 辗转相除法求两个整数的最大公约数

辗转相除法可分别用非递归思想(循环结构)和递归思想实现。

用循环结构实现：

图 8.3 所示的辗转相除法流程图本身就包含循环结构，因此可以用循环结构实现。代码如下：

```

#include <stdio.h>
int gcd( int u, int v )    //求 u 和 v 的最大公约数
{
    int r;
    while( (r=u%v)!=0 )
    {
        u = v;
        v = r;
    }
    return(v);
}

int main( )
{
    int m, n, t;
    printf( "Please input two positive integers : " );
    scanf( "%d%d", &m, &n );
    if( m<n ) //交换 m 和 n, 使得 m 为二者较大者
    {
        t = m; m = n; n = t;
    };
    printf( "the great common divisor of these two integers is %d.\n", gcd(m,n) );
    return 0;
}
  
```

该程序的运行示例如下：

```

Please input two positive integers : 18 33 ✓
the great common divisor of these two integers is 3.
  
```

**思考 8.1：**在上述代码的主函数中，如果  $m < n$ ，但不交换  $m$  和  $n$ ，直接调用 `gcd` 函数求解，能否求得最大公约数？

用递归思想实现：

辗转相除法的思想也可以采用递归方法实现。其递归思想是：在求最大公约数过程中，如果  $u$  对  $v$  取余的结果为 0，则最大公约数就是  $v$ ；否则递归求  $v$  和  $u \% v$  的最大公约数。因此，上述代码中的 `gcd` 函数可改写成：

```
int gcd( int u, int v )    //求 u 和 v 的最大公约数
{
    if( u%v==0 ) return v;
    else return gcd(v, u%v);
}
```

在使用上述递归函数 gcd 求 “gcd(33,18)” 时，要递归调用 “gcd(18,15)”；在执行 “gcd(18,15)” 时又递归调用 “gcd(15,3)”；而在执行 “gcd(15,3)” 时，因为 15%3 的结果为 0，所以最终求得的最大公约数为 3。

**例 8.5** 经典的 Hanoi(汉诺塔)问题。

问题描述：有一个汉诺塔，塔内有 A，B，C 三个柱子。如图 8.5 所示。起初，A 柱上有  $n$  个盘子，依次由大至小、从下往上堆放，要求将它们全部移到 C 柱上；在移动过程中可以利用 B 柱，但每次只能移动一个盘子，且必须使三个柱子上始终保持大盘在下，小盘在上的状态。要求编程输出移动的步骤。

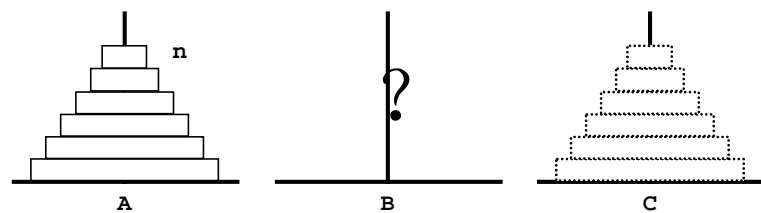


图8.5 汉诺塔问题

**分析：**

先分析盘子数量很少的情形。比如，当  $n = 2$  时，只有 2 个盘子，这个问题就变得相当的简单，只需要 3 步就可以完成整个移动操作。

A  $\rightarrow$  B (表示将 A 柱最上面的一个盘子移到 B 柱上)

A  $\rightarrow$  C (将 A 柱上此时最上面的一个盘子移到 C 柱上)

B  $\rightarrow$  C (将 B 柱上的盘子移到 C 柱上)

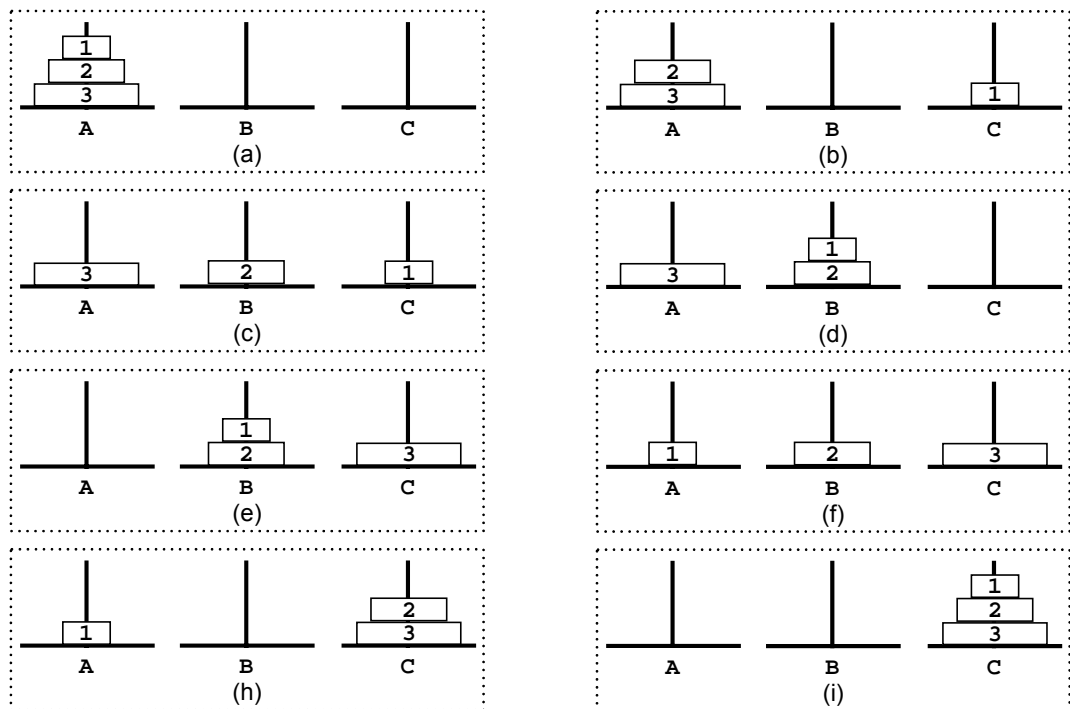


图8.6 汉诺塔问题( $n=3$ )

又如，移动 3 个盘子的情况，需要 7 步：

A → C  
A → B  
C → B  
A → C  
B → A  
B → C  
A → C

整个过程如图 8.6(b)~(i)所示。

现在的问题是，当 A 柱上有  $n$  个盘子时，至少需要移动多少步？现按如下思路进行思考。

将  $n$  个盘子从 A 柱移到 C 柱可以分解为以下 3 个步骤：

(1) 将 A 柱上  $n-1$  个盘子借助 C 柱先移到 B 柱上，如图 8.7(b)所示；

(2) 将 A 柱上剩下的 1 个盘子移到 C 柱上，如图 8.7(c)所示；

(3) 将 B 柱上的  $n-1$  个盘子借助 A 柱移到 C 柱上，如图 8.7(d)所示。

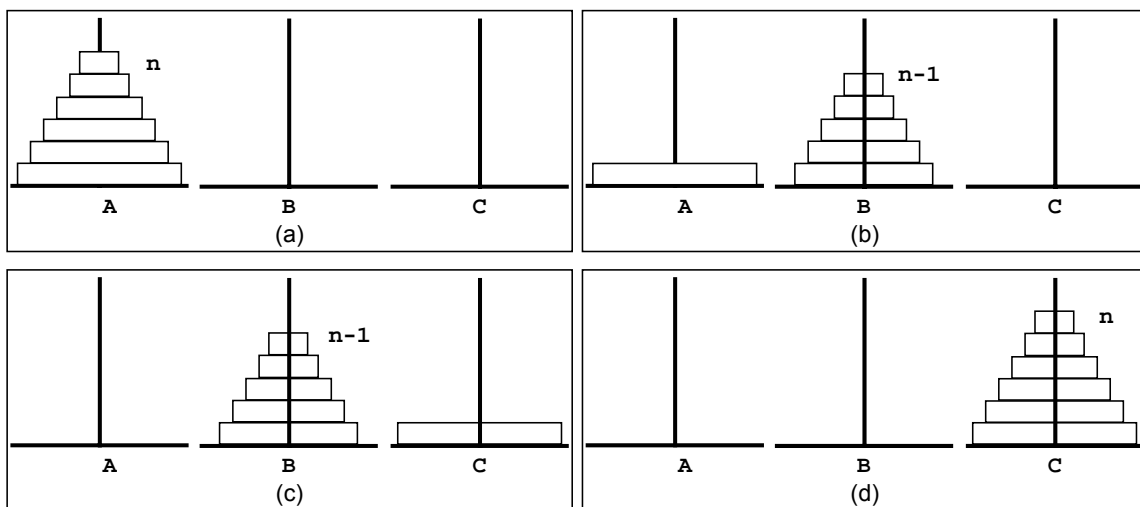


图8.7 汉诺塔问题分解

而  $n-1$  个盘子的移动又可以分解为两次  $n-2$  个盘子的移动和一次 1 个盘子的移动。依此类推。

设移动  $n$  个盘子至少需要  $A(n)$  步，则存在递推式子： $A(n) = 2 \times A(n-1) + 1$ 。这个递推式子结束条件是：当  $n = 1$  时，只有一个盘子，只需一次移动即可。因此移动  $n$  个盘子至少需要：

$$A(n) = 2^n - 1$$

步。

例如，当  $n = 3$  时，至少需要移动 7 次。

#### 实现方法：

上面 3 个步骤中的第(1)步和第(3)步，都是将  $n-1$  个盘子从一个柱子移到另一个柱子上，采用的方法是相同的，只是柱子的名字不同而已。为使之一般化，把“移动  $n$  个盘子”的任务描述成：将  $n$  个盘子从 **one** 柱子上借助 **two** 柱子移动到 **three** 柱子上。这样，以上 3 个步骤可以表示为：

第(1)步：将  $n-1$  个盘子从 **one** 柱子上借助 **three** 柱子移动到 **two** 柱子上。

第(2)步：将 **one** 柱子上的盘子(只有一个)移动到 **three** 柱子上。

第(3)步：将  $n-1$  个盘子从 **two** 柱子上借助 **one** 柱子移动到 **three** 柱子上。

$n$  个盘子的移动分解成两次  $n-1$  个盘子的移动和一次 1 个盘子的移动，可以用递归函数实现。因此，在程序实现时，设计以下 2 个函数：

1) hanoi 函数: 函数调用 `hanoi( n, one, two, three )` 实现将  $n$  个盘子从 `one` 柱子上借助 `two` 柱子移到 `three` 柱子的过程;

2) move 函数: 函数调用 `move( x, y )` 实现把 1 个盘子从 `x` 柱子上移到 `y` 柱子上的过程。`x` 和 `y` 代表 A、B、C 柱子之一, 根据不同情况分别以 A、B、C 代入。

上述两个函数中, 真正实现移动盘子的是 `move` 函数。很明显, 在 `hanoi` 函数中要调用 `move` 函数。

代码如下:

```
#include <stdio.h>
int main( )
{
    void hanoi( int m, char one, char two, char three );    //函数声明
    int m;    //盘子个数
    printf( "input the number of diskess: " );
    scanf( "%d", &m );
    printf( "The steps of moving %d disks:\n", m );
    hanoi( m, 'A', 'B', 'C' ); /*调用 hanoi 函数实现将 m 个盘子从 A 柱移动到
                                C 柱(借助 B 柱)*/
    return 0;
}
//将 n 个盘子从 one 柱移到 three 柱(借助 two 柱)
void hanoi( int n, char one, char two, char three )
{
    void move( char x, char y );    //函数声明
    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 )//将 1 个盘子从 x 柱移动到 y 柱
{
    printf( "%c-->%c\n", x, y );
}
```

该程序的运行示例如下:

input the number of diskess: 3✓

The steps of moving 3 disks:

A-->C

A-->B

C-->B

A-->C

B-->A

B-->C

A-->C

当  $m = 3$  时, `hanoi( m, 'A', 'B', 'C' )` 的执行过程如图 8.8 所示, 在图中, 实现盘子移动的 `move` 函数调用用斜体标明。

在执行函数 `hanoi( 3, 'A', 'B', 'C' )` 时, 参数 `one`、`two`、`three` 分别为 'A'、'B'、'C'。该函数



执行过程按先后顺序又分解为以下 3 步：

- (1) 递归调用 `hanoi( n-1, one, three, two )`，即 `hanoi( 2, 'A', 'C', 'B' )`；
- (2) 调用 `move( one, three )`，即 `move( 'A', 'C' )`，将 A 柱上的盘子移动到 C 柱上；
- (3) 递归调用 `hanoi( n-1, two, one, three )`，即 `hanoi( 2, 'B', 'A', 'C' )`。

其中第(1)步最终分解成 3 次移动，依次为 `move( 'A', 'C' )`、`move( 'A', 'B' )`和 `move( 'C', 'B' )`；第(3)步也最终分解成 3 次移动，依次为 `move( 'B', 'A' )`、`move( 'B', 'C' )`和 `move( 'A', 'C' )`。分析的结果和上述运行示例的输出结果是吻合的。

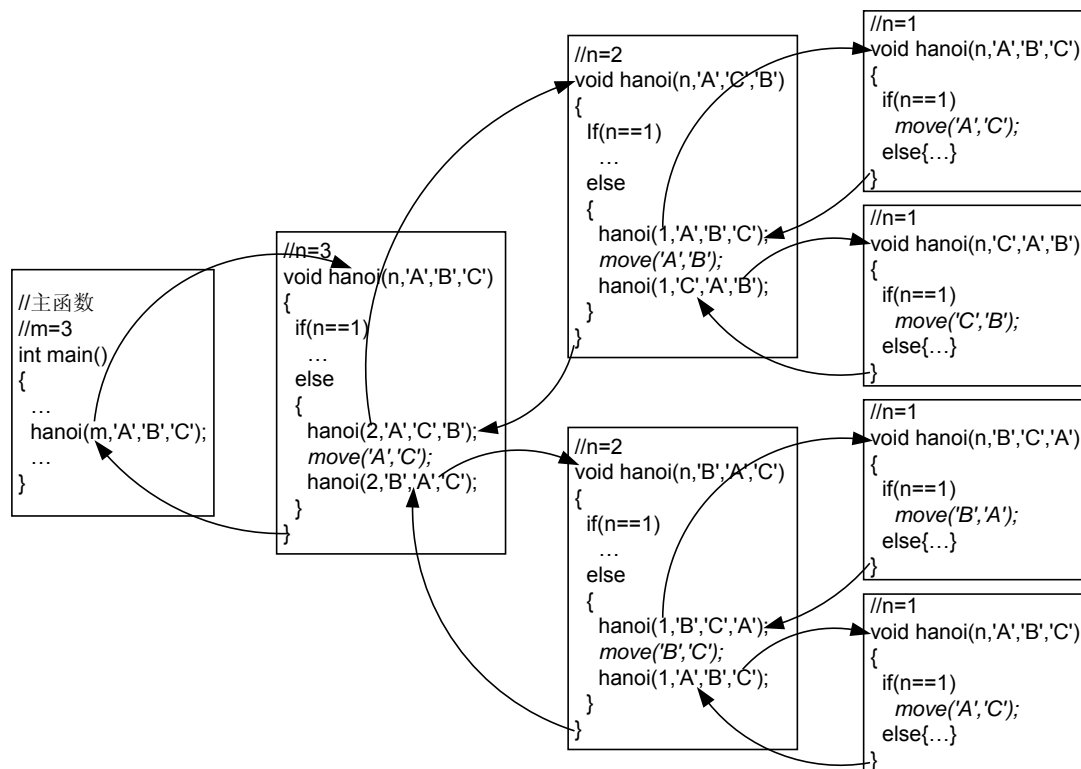


图8.8 Hanoi(3,'A','B','C')的执行过程

### 8.1.3 递归存在的问题

递归思想很好理解，特别是对于一些可以用递推式子表示的问题，用递归思想求解是一种很自然的思路。然而，使用递归的代价是十分巨大的：它会消耗大量的内存!!! 函数调用时需要用到堆栈，而堆栈的资源是十分有限的。在例 8.3 中，使用递归思想求 Fibonacci 数列的第 n 项，而求 `Fibonacci(20)`，递归调用 `Fibonacci()` 函数就高达 21891 次!!! 这一点可以用下面的代码验证。

```

#include <stdio.h>
int count = 0; //Fibonacci 函数递归调用次数
int Fibonacci( int n )
{
    count++;
    if( n==0 || n==1 ) return 1;
    else return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
void main()
{
    Fibonacci( 20 );
    printf( "count=%d\n", count );
}

```

}

## 练习

- 8.1 有 5 个人坐在一起，问第 5 个人多少岁？他说比第 4 个人大两岁。问第 4 个人的岁数，他说比第 3 个人大两岁。问第 3 个人，又说比第 2 个人大两岁。问第 2 个人，说比第 1 个人大两岁。最后问第 1 个人，他说是 10 岁。请问第 5 个人多少岁？要求用递归思想求解。
- 8.2 根据递推式子  $C(m,n) = C(m-1,n) + C(m-1,n-1)$  求组合数  $C(m,n)$ 。注意递推的终止条件是  $C(m,1) = m$ ；以及一些  $m$  和  $n$  取值的一些特殊情况，如  $m < 0$  或  $n < 0$  或  $m < n$  时  $C(m,n)$  值为 0， $m$  和  $n$  相等时  $C(m,n)=1$  等。
- 8.3 例 1.28 实现了迭代法求平方根  $\sqrt{a}$ ，迭代公式为：

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

本题要求用递归思想求  $x_{n+1}$ ，并输出求  $\sqrt{2}$  时的 10 次递推结果，即输出  $x_1 \sim x_{10}$ 。初始时  $x_0$  可任取，比如取 50。

- 8.4 核反应堆中有  $\alpha$  和  $\beta$  两种粒子。每秒钟内一个  $\alpha$  粒子可以产生 3 个  $\beta$  粒子，而一个  $\beta$  粒子可以产生 1 个  $\alpha$  粒子和 2 个  $\beta$  粒子。若在  $t=0$  时刻反应堆中只有 1 个  $\alpha$  粒子，求  $t$  时刻反应堆中分别有多少个  $\alpha$  粒子和  $\beta$  粒子。
- 8.5 在数学上有一个著名的“阿克曼 (Ackermann) 函数”，它是二元函数，其定义式为：  
 $ACK(M,N)$ ， $M \geq 0$ ， $N \geq 0$ ，且：  
 (1)  $ACK(M, 0) = ACK(M-1, 0)$  ( $M > 0$ )  
 (2)  $ACK(M, N) = ACK(M-1, ACK(M, N-1))$  ( $M > 0, N > 0$ )  
 初始条件： $ACK(0, N) = 1 + N$  ( $N \geq 0$ )  
 在本题中，请试着用递归方法求解  $ACK(M,N)$ 。注意阿克曼函数递归调用次数增长是非常快的，比如要求  $ACK(3,7)$ ，需要经过将近 70 万次 (693964 次) 递归调用！。所以本题采用递归方法求解，只能求解  $M$  和  $N$  很小的值。

## 8.2 递归思想在竞赛题目中的应用

下面列举几道实际竞赛题目，分析递归思想在竞赛题目中的应用。

### 例 8.6 另一个 Fibonacci 数列 (Fibonacci Again)

题目来源：

ZOJ Monthly, December 2003

题目描述：

定义另外一个 Fibonacci 数列： $F(0) = 7$ ， $F(1) = 11$ ， $F(n) = F(n-1) + F(n-2)$ ，( $n \geq 2$ )。

输入描述：

输入文件包含多行，每行为一个整数  $n$ ， $n < 1,000,000$ 。

输出描述：

对输入文件中的每个整数  $n$ ，如果  $F(n)$  能被 3 整除，输出 "yes"，否则输出 "no"。

样例输入：

样例输出：

|   |     |
|---|-----|
| 0 | no  |
| 1 | no  |
| 2 | yes |
| 3 | no  |

分析:

本章例 8.3 介绍了用递归思想求 Fibonacci 数列各项,但在本题中如果直接采用递归方法求  $F(n)$  对 3 取余得到的余数,则会超时!!! 因为 8.1.3 节提到“求 Fibonacci(20)时递归调用 Fibonacci( ) 函数高达 21891 次”,而在本题中  $n$  的值最大可以取到 1,000,000。

我们先下面的程序输出前 30 项对 3 取余的结果:

```
#include <stdio.h>
int f(int n)
{
    if(n==0) return 1;
    else if(n==1) return 2;
    else return ( f(n-1)+f(n-2) )%3;
}
int main( )
{
    for( int i=0; i<30; i++ ) printf( "%d ", f(i) );
}
```

前 30 项对 3 取余得到的余数分别为: 1 2 0 2 2 1 0 1 1 2 0 2 2 1 0 1 1 2 0 2 2 1 0 1 1 2 0 2 2 1 0 1。分析这些余数我们发现该 Fibonacci 数列各项对 3 取余得到的余数每 8 项构成循环: 1 2 0 2 2 1 0 1。如果我们把这 8 个余数存放到一个数组  $f0$ , 对输入的任何整数  $n$ , 则有:  $f(n)\%3 = f0[n\%8]$ 。按照这种方法可以很快判断  $f(n)$  是否能被 3 整除。

代码如下:

```
#include <stdio.h>
int f(int n) //求第 n 项对 3 取余得到的余数
{
    if(n==0) return 1;
    else if(n==1) return 2;
    else return ( f(n-1)+f(n-2) )%3;
}
int main( )
{
    int f0[8];
    for( int i=0; i<8; i++ ) //把前 8 项的余数保存下来
        f0[i] = f(i);
    int n;
    while( scanf( "%d", &n )!=EOF )
    {
        if( f0[n%8]==0 ) printf( "yes\n" );
        else printf( "no\n" );
    }
    return 0;
}
```

例 8.7 分形(Fractal)

题目来源:

## Asia 2004, Shanghai (Mainland China), Preliminary

### 题目描述:

分形是存在“自相似”的一个物体或一种量，从某种技术角度来说，这种“自相似”是全方位的。

盒形分形定义如下：

度数为 1 的分形很简单，为：

X

度数为 2 的分形为：

X X

X

X X

如果用  $B(n-1)$  代表度数为  $n-1$  的盒形分形，则度数为  $n$  的盒形分形可以递归地定义为：

$B(n-1)$            $B(n-1)$

$B(n-1)$

$B(n-1)$            $B(n-1)$

你的任务是输出度数为  $n$  的盒形分形。

### 输入描述:

输入文件包含多个测试数据，每个测试数据占一行，包含一个正整数  $n$ ， $n \leq 7$ 。输入文件的最后一行为 -1，代表输入结束。

### 输出描述:

对每个测试数据，用符号“X”输出盒形分形。在每个测试数据对应的输出之后输出一个短划线符号“-”，在每行的末尾不要输出任何多余的空格，否则得到的是“格式错误”的结果。

### 样例输入:

1  
2  
3  
-1

### 样例输出:

X  
-  
X X  
X  
X X  
-  
X X    X X  
X      X  
X X    X X  
X X  
X  
X X  
X X    X X  
X      X  
X X    X X  
-

### 分析:

首先注意到度数为  $n$  的盒形分形，其大小是  $3^{n-1} \times 3^{n-1}$ 。可以用字符数组来存储盒形分形中各字符。因为  $n \leq 7$ ，而  $3^6 = 729$ ，因此可以定义一字符数组 `Fractal[730][730]` 来存储度数不超过 7 的盒形分形。

其次，度数为  $n$  的盒形分形可以由以下递推式子表示：

$$B(n) = \begin{matrix} & B(n-1) & & B(n-1) \\ B(n-1) & & B(n-1) & \\ & B(n-1) & & B(n-1) \end{matrix}$$

因此，可以用一个递归函数来设置度数为 $n$ 的盒形分形。假设需要在 $(startX, startY)$ 位置开始设置度数为 $n$ 的盒形分形，它由5个度数为 $n-1$ 的盒形分形组成，其起始位置分别为： $(startX+0, startY+0)$ 、 $(startX+2*L0, startY+0)$ 、 $(startX+L0, startY+L0)$ 、 $(startX+0, startY+2*L0)$ 和 $(startX+2*L0, startY+2*L0)$ ，其中 $L0 = 3^{n-2}$ 。该递归函数的结束条件是：当 $n = 1$ 时，即度数为1的盒形分形，只需在 $(startX, startY)$ 位置设置一个“X”字符。

另外，题目中提到“在每行的末尾不要输出任何多余的空格”，因此在字符数组 `Fractal` 每行最后一个“X”字符之后，应该设置串结束符标志`'\0'`。

代码如下：

```
#include <stdio.h>
#include <math.h>
#define MAXSCALE 730 //n 为最大值 7 时，分形的大小是  $3^6 \times 3^6$ ，而  $3^6 = 729$ 

//函数功能：从(startX,startY)位置开始设置度数为 n 的盒形分形，
//即对盒形分形中的每个 X，在字符数组 Frac 的相应位置设置字符"X"
//其中第 1 个形参为二维数组名，其第 2 维不能省略
void SetFractal( char Frac[ ][730], int startX, int startY, int n )
{
    if( n==1 ) Frac[startX][startY] = 'X';
    else
    {
        int L0 = (int)pow(3,n-2);
        SetFractal( Frac, startX+0, startY+0, n-1 );
        SetFractal( Frac, startX+2*L0, startY+0, n-1 );
        SetFractal( Frac, startX+L0, startY+L0, n-1 );
        SetFractal( Frac, startX+0, startY+2*L0, n-1 );
        SetFractal( Frac, startX+2*L0, startY+2*L0, n-1 );
    }
}

int main( )
{
    int n; //分形的大小
    int i, j; //循环变量
    char Fractal[MAXSCALE][MAXSCALE];
    while( scanf("%d", &n) )
    {
        if( n== -1 ) break;
        int measure = (int)pow(3,n-1); // "盒形"分形大小
        SetFractal( Fractal, 0, 0, n );
        for( i=0; i<measure; i++ ) //保证每行最后的'X'后是串结束符标记'\0'
        {
            int max = 0;
            for( j=0; j<measure; j++ ) //找到每行最后的'X'
            {
                if(Fractal[i][j]=='X') max = j;
            }
        }
    }
}
```

```

        for( j=0; j<max; j++ ) //非'X'的位置上为空格
        {
            if(Fractal[i][j]!='X') Fractal[i][j] = ' ';
        }
        Fractal[i][max+1] = 0; //在每行最后的'X'后添上串结束符标记'\0'
    }
    for( i=0; i<measure; i++ )
    {
        printf( "%s\n", Fractal[i] );
    }
    printf( "-\n" );
}
return 0;
}

```

## 练习

### 8.6 欧几里得游戏(Euclid's Game)

#### 题目描述:

两个玩家, **Stan** 和 **Ollie**, 玩欧几里得游戏。他们从两个自然数开始。第一个玩家, **Stan**, 从两个数的较大数中减去较小数的任意正整数倍, 只要差为非负即可。然后, 第二个玩家, **Ollie**, 对得到的两个数进行同样的操作, 然后又是 **Stan**, 等等。直至某个玩家将较大数减去较小数的某个倍数之后为 0 时为止, 此时, 游戏结束, 该玩家就是胜利者。

例如, 两个玩家从两个自然数 25 和 7 开始:

```

      25 7
Stan:  11 7
Ollie:  4 7
Stan:   4 3
Ollie:  1 3
Stan:   1 0

```

因此最终 **Stan** 赢得这次游戏。

#### 输入描述:

输入文件包含若干个测试数据, 每个测试数据占一行, 为两个正整数, 表示每次游戏时两个整数的初始值, 每次游戏都是从 **Stan** 开始。输入文件的最后一行为两个 0, 表示输入结束, 这一行不需处理。

#### 输出描述:

对输入文件中的每个测试数据, 输出一行, 为 "**Stan wins**" 或 "**Ollie wins**". 假定 **Stan** 和 **Ollie** 玩这个游戏都玩得很好, 即 **Stan** 和 **Ollie** 都想赢得比赛, 他们在走每一步时都是尽可能选择使得他们能赢得比赛的步骤。例如, 在上面的例子中, 如果 **Stan** 第一步得到 18 7 或 4 7, 则 **Stan** 不可能赢得游戏, 所以 **Stan** 必须在第一步中得到 11 7。

#### 样例输入:

```

34 12
15 24
0 0

```

#### 样例输出:

```

Stan wins
Ollie wins

```

### 8.7 幸存者游戏(Recursive Survival)

**题目描述:**

$n$  个人围成一圈, 这  $n$  个人的序号从  $1 \sim n$ 。每隔一个人淘汰一个, 直到剩下一个人为止。定义一个函数  $J(n)$ , 表示最后剩下的这个人的号码。例如,  $J(2)=1$ ,  $J(10)=5$ 。

现在的任务是计算嵌套函数  $J(J(J(\dots J(n)\dots)))$ 。

**输入描述:**

输入文件中有多个测试数据, 每个测试数据占一行, 为两个整数: 第 1 个数代表最初围成一圈的人数, 第 2 个数代表嵌套的层数。所有的整数都不超过  $2^{63} - 1$ 。

**输出描述:**

对每个测试数据, 输出计算的结果。

**样例输入:**

```
2 1
10 1
10 2
```

**样例输出:**

```
1
5
3
```

**8.8 抽签(Lot)****题目描述:**

$N$  个士兵, 站成一排, 需要选择若干个士兵去巡逻。为了选出这些士兵, 执行以下操作若干次: 如果该排中士兵多于 3 个, 则所有的士兵当中位置是偶数的, 或者所有士兵当中位置是奇数的, 将被淘汰, 重复以上步骤, 直到剩下士兵的人数为 3 或少于 3 个为止。他们将被派去巡逻。你的任务是给定  $N$  个士兵, 计算按照这种方式选择派出去巡逻的士兵人数刚好有 3 个有多少种组合方式。

注意: 如果按照上述方式选出来的士兵少于 3 个, 则这种组合方式不算。 $0 < N \leq 10\,000\,000$ 。

**输入描述:**

输入文件包含若干个测试数据, 每个测试数据占一行, 为一个整数  $N$ 。输入数据一直到文件尾。

**输出描述:**

对输入文件中的每个测试数据, 输出满足要求的组合方式的数目。

**样例输入:**

```
10
4
```

**样例输出:**

```
2
0
```

**提示:**

样例输入中  $N = 10$  时, 有 2 种组合方式。设初始时这 10 个士兵的序号是  $1 \sim 10$ 。则这 2 种组合方式是按如下挑选方式得到的: ① 先选择序号为奇数的, 即 1、3、5、7、9, 再从中挑出 1、5、9; ② 先选择序号为偶数的, 即 2、4、6、8、10, 再从中挑出 2、6、10。其他组合都是不满足题目要求的。

## 8.3 递归与搜索

### 8.3.1 搜索算法思想

有一类问题在求解时需要一定的步骤, 通常求解这些问题采取的策略是从初始位置(或步骤)

出发，**试探性**的选择可行的步骤，如果行不通则回退到上一步，再试探其他的解决步骤。

例如图 8.9 所示的迷宫问题， $\odot$ 表示迷宫的入口， $\bigcirc$ 表示迷宫的出口， $\blacksquare$ 表示墙壁，不能通过， $\square$ 表示可以通过的空白方格。只能从迷宫的入口进入，从出口出来，不能出边界，现在要找到一条从入口到出口的路径。在每个空白方格处只能移动到上、下、左、右相邻的空白方格。假设在选择可行的空白方格时按照上、右、下、左的顺时针方向。从图(b)开始，按向上的方向走了两步以后，行不通，则回退，一直回退到初始位置。然后在初始位置，选择右边的方向（图(d)），走了一步以后，还是行不通，又回退到初始位置。下方没有空白方格，所以选择左边的空白方格（图(e)），…。如此进行下去直到找到出口或者得出无解的结论。

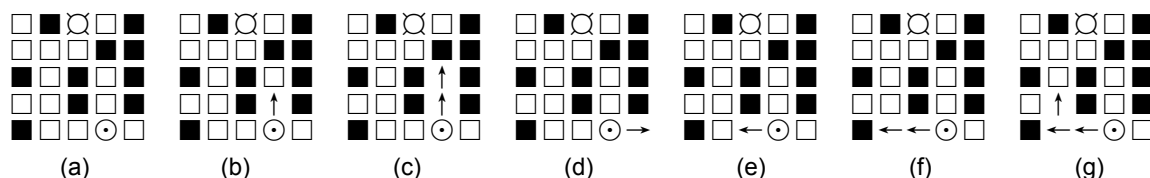


图 8.9 迷宫搜索

这种求解问题的策略称为**搜索**，在实现时需要用递归函数来实现。

### 8.3.2 递归函数的设计

利用递归思想进行搜索求解时需要注意两个问题：① 如何设计递归函数；② 如何调用递归函数进行求解。

1. 递归函数的设计主要面临以下几个问题：

- 1) 需要将什么信息传递给下一层递归调用 — 由此确定递归函数有几个参数，各参数含义是什么。
- 2) 每一层递归函数调用后会得到一个怎样的结果，这个结果是否需要返回到上一层 — 由此确定递归函数的返回值，及返回值的含义。
- 3) 在每一层递归函数的执行过程中，在什么情形下需要递归调用下一层，以及递归前该做什么准备工作，递归返回后该做什么恢复工作。
- 4) 递归函数执行到什么程度就可以不再需要递归调用下去了 — 应该在适当的时候终止递归函数的继续递归调用，也就是要确定递归的终止条件。

2. 调用递归函数进行求解：在 **main** 函数(或其他函数)中应该采取怎样的形式调用递归函数，也就是从怎样的初始状态出发进行搜索，通常也就是确定实参的值。

以上方法和思想将在下一节具体例题中阐述。

### 8.3.3 例题解析

本节并不涉及复杂的理论知识，而是通过两道实际竞赛题目讲解搜索思想及搜索策略的实现。

#### 例 8.8 骨头的诱惑(Tempter of the Bone)

题目来源：

Zhejiang Provincial Programming Contest 2004

题目描述：

一只小狗在一个古老的迷宫里找到一根骨头，当它叼起骨头时，迷宫开始颤抖，它感觉到地面开始下沉。它才明白骨头是一个陷阱，它拼命地试着逃出迷宫。

迷宫是一个  $N \times M$  大小的长方形，迷宫有一个门。刚开始门是关着的，并且这个门会在第  $T$  秒钟开启，门只会开启很短的时间（少于一秒），因此小狗必须恰好第  $T$  秒达到门的位置。每秒钟，它可以向上、下、左或右移动一步到相邻的方格中。但一旦它移动到相邻的方格，这



个方格开始下沉，而且会在下一秒消失。所以，它不能在一个方格中停留超过一秒，也不能回到经过的方格。

小狗能成功逃离吗？请你帮助他。

#### 输入描述：

输入文件包括多个测试数据。每个测试数据的第一行为三个整数： $N\ M\ T$ ，( $1 < N, M < 7$ ;  
 $0 < T < 50$ )，分别代表迷宫的长和宽，以及迷宫的门会在第  $T$  秒时刻开启。

接下来  $N$  行信息给出了迷宫的格局，每行有  $M$  个字符，这些字符可能为如下值之一：

X: 墙壁，小狗不能进入

S: 小狗所处的位置

D: 迷宫的门

. : 空的方格

输入数据以三个 0 表示输入数据结束。

#### 输出描述：

对每个测试数据，如果小狗能成功逃离，则输出"YES"，否则输出"NO"。

#### 样例输入：

3 4 5

S..

.X.X

...D

4 4 8

.X.X

..S.

....

DX.X

4 4 5

S.X.

..X.

..XD

....

0 0 0

#### 样例输出：

YES

YES

NO

#### 分析：

本题要采用搜索思想求解，本节也借助这道题目详细分析搜索策略实现方法及搜索时要注意的问题。

#### 1) 搜索策略

以样例输入中的第 1 个测试数据进行分析，如图 8.10 所示。图(a)表示测试数据及所描绘的迷宫；在图(b)中，圆圈中的数字表示某个位置的行号和列号，行号和列号均从 0 开始计起，实线箭头表示搜索前进方向，虚线箭头表示回退方向。

搜索时从小狗所在初始位置 S 出发进行搜索。每搜索到一个方格位置，对该位置的 4 个可能方向（要排除边界和墙壁）进行下一步搜索。往前走一步，要将当前方格设置成墙壁，表示当前搜索过程不能回到经过的方格。一旦前进不了，要回退，要恢复现场(将前面设置的墙壁还原成空的方格)，回到上一步时的情形。只要有一个搜索分支到达门的位置并且符合要求，则搜索过程结束。如果所有可能的分支都搜索完毕，还没找到满足题目要求的解，则该迷宫无解。

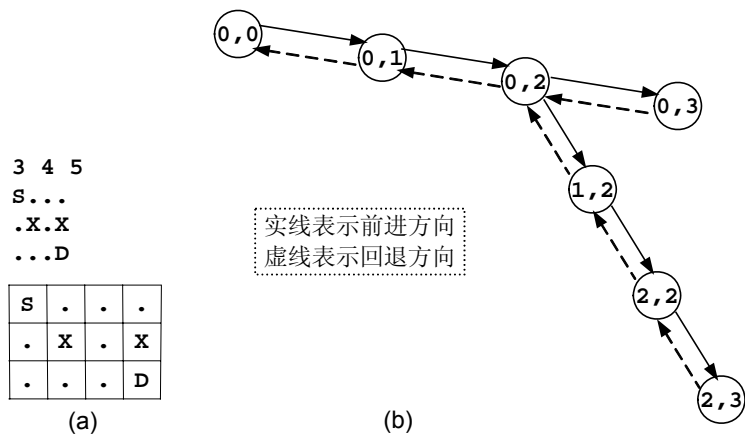


图8.10 骨头的诱惑（搜索策略）

2) 搜索实现

假设实现搜索的函数为 `dfs`，它带有 3 个参数。`dfs( si, sj, cnt )`：从 `(si, sj)` 位置出发，要求在第 `cnt` 秒到达门的位置，如果成功，搜索终止；否则继续从其相邻位置继续进行搜索。继续搜索则要递归调用 `dfs` 函数，因此 `dfs` 是一个递归函数。

成功逃离条件：`si=di, sj=dj, cnt=t`。其中 `(di,dj)` 是门的位置，在第 `t` 秒钟开启。

假设按照上、右、下、左顺时针的顺序进行搜索。则对样例输入中的第 1 个测试数据，其搜索过程及 `dfs` 函数的执行过程如图 8.11 所示。

在该测试数据中，小狗的起始位置在 `(0,0)` 处，门的位置在 `(2,3)` 处，门会在第 5 秒钟开启。在主函数中，调用 `dfs(0,0,0)` 搜索迷宫。当递归执行到某一个 `dfs` 函数 `dfs( si, sj, cnt )`，满足 `sj==2==di, sj==3==dj`，且 `cnt==5==t`，则表示能成功逃离。

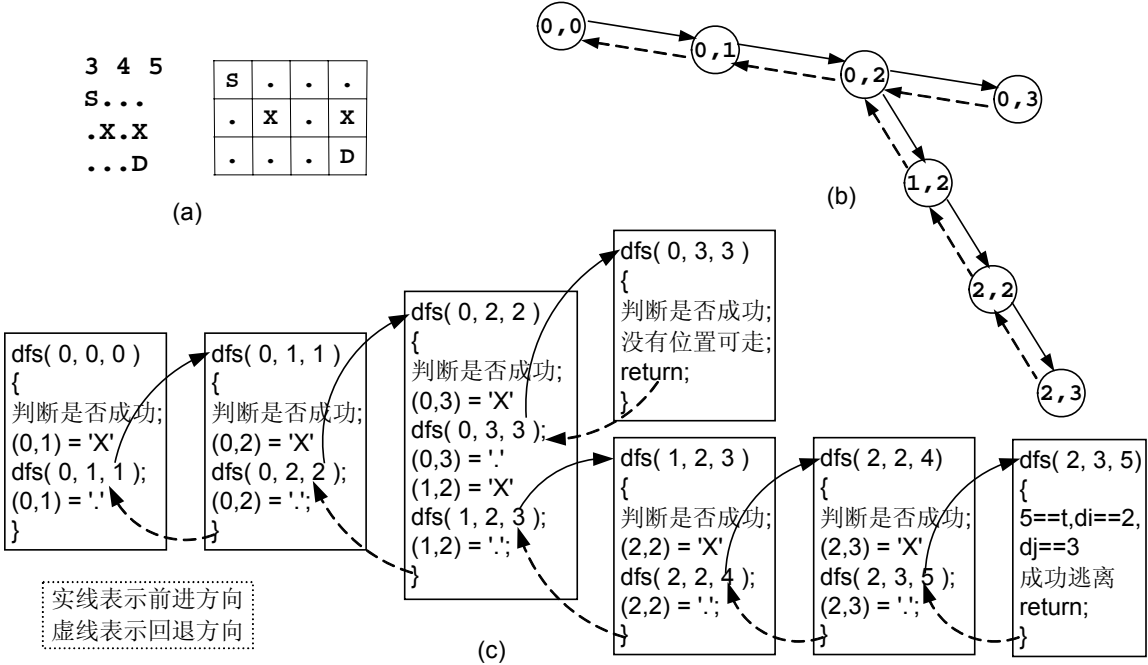


图8.11 骨头的诱惑（搜索策略的函数实现）

图 8.11(c)演示了 `dfs(0,0,0)` 的递归执行过程。在图(c)中：

`(0,1) = 'X'`：表示往前走一步，要将当前方格设置成墙壁；

`(0,1) = '.'`：表示回退过程，要恢复现场，即将 `(0,1)` 这个位置由原先设置的墙壁还原成空格。

在执行 `dfs(0,0,0)` 时，按照搜索顺序，上方是边界，不能走，所以向右走一步，即要递归调

用 `dfs(0,1,1)`。在调用 `dfs(0,1,1)`之前，将(0,1)位置置为墙壁。走到(0,1)位置后，下一步要走的位置是(0,2)，要递归调用 `dfs(0,2,2)`。在调用 `dfs(0,2,2)`之前，将(0,2)位置置为墙壁。走到(0,2)位置后，下一步要走的位置是(0,3)，要递归调用 `dfs(0,3,3)`。在调用 `dfs(0,3,3)`之前，将(0,3)位置置为墙壁。在走到(0,3)位置后，其4个相邻位置中上边、右边是边界，下边是墙壁，左边本来是空的方格，但因为在前面的搜索前进方向上已经将它设置成墙壁了，所以没有位置可走，只能回退到上一层，即 `dfs(0,3,3)`函数执行完毕，要回退到主调函数处，也就是 `dfs(0,2,2)`函数中。

回到 `dfs(0,2,2)`函数（将(0,3)位置上的墙壁还原成空的方格）处，此时处在位置(0,2)，且已经走了2秒。(0,2)位置的4个相邻位置中，还有(1,2)这个位置可以走，则从(1,2)位置继续搜索。...

按照上述搜索策略，一直搜索到(2,3)位置处，这个位置是门的位置，且刚好走了5秒。所以得出结论：能够成功逃脱。

这里要注意，搜索方向的选择是通过下面的二维数组及循环控制实现的。该二维数组表示上、右、下、左四个方向相对当前位置 `x`、`y` 坐标的增量。

```
int dir[4][2] = { {-1,0}, {0,1}, {1,0}, {0,-1} };
```

### 3) 为什么在回退过程中要恢复现场？

以样例输入中的第2个测试数据来解释这个问题。在这个测试数据中，如果加上回退过程的恢复现场操作，则不管按什么顺序(上、右、下、左顺序或者左、右、下、上顺序)进行搜索，都能成功脱离；但是去掉回退过程的恢复现场操作后，按某种搜索顺序能成功脱离，但按另外一种搜索顺序则不能成功脱离，这是错误的。图8.12~8.15以测试数据2为例分析了加上和去掉回退过程分别按两种搜索顺序进行搜索的过程和结果。

**测试数据2分析1：**回退过程有恢复现场，`dir`数组如下，即搜索方向为上、右、下、左顺序，整个搜索过程如图8.12(b)所示，`dfs`函数的执行过程如图8.12(c)所示。`dfs`函数执行的结果是能成功逃脱。

```
int dir[4][2] = { {-1,0}, {0,1}, {1,0}, {0,-1} };
```

**测试数据2分析2：**回退过程有恢复现场，`dir`数组如下，即搜索方向为左、右、下、上顺序，整个搜索过程如图8.13(b)所示，`dfs`函数的执行过程如图8.13(c)所示。`dfs`函数执行的结果是能成功逃脱。

```
int dir[4][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };
```

**测试数据2分析3：**去掉回退过程的恢复现场操作，在图8.14(c)中，“(1,3) = '1;’”等代码加上了删除线。`dir`数组如下，即搜索方向为上、右、下、左顺序，整个搜索过程如图8.14(b)所示，`dfs`函数的执行过程如图8.14(c)所示。`dfs`函数执行的结果是能成功逃脱。

```
int dir[4][2] = { {-1,0}, {0,1}, {1,0}, {0,-1} };
```

**测试数据2分析4：**去掉回退过程的恢复现场操作，在图8.15(c)中，“(0,2) = '1;’”等代码加上了删除线。`dir`数组如下，即搜索方向为左、右、下、上顺序，整个搜索过程如图8.15(b)所示，`dfs`函数的执行过程如图8.15(c)所示。`dfs`函数执行的结果是不能成功逃脱。

```
int dir[4][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };
```

为什么在回退过程中恢复现场？答案是：如果当前搜索方向行不通，该搜索过程要结束了，但并不代表其他搜索方向也行不通，所以在回退时必须还原到原来的状态，保证其他搜索过程不受影响。

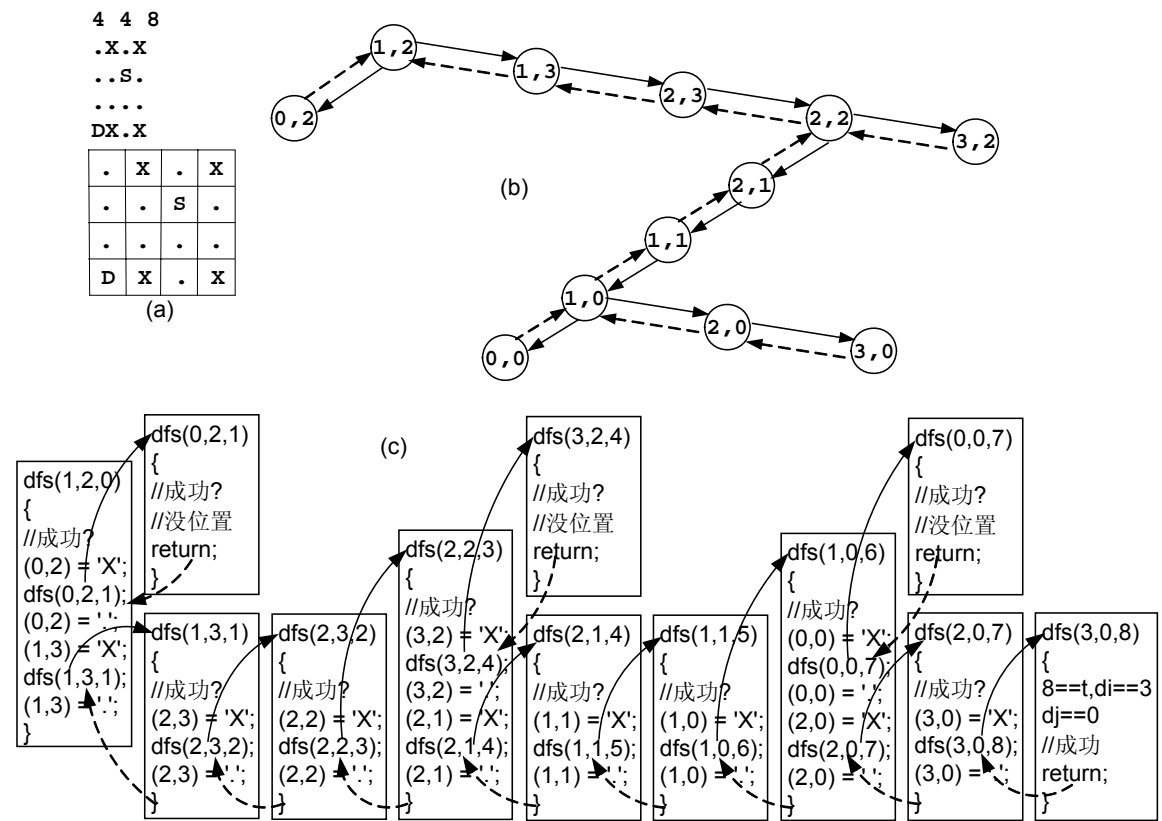


图8.12 骨头的诱惑（测试数据2分析1）

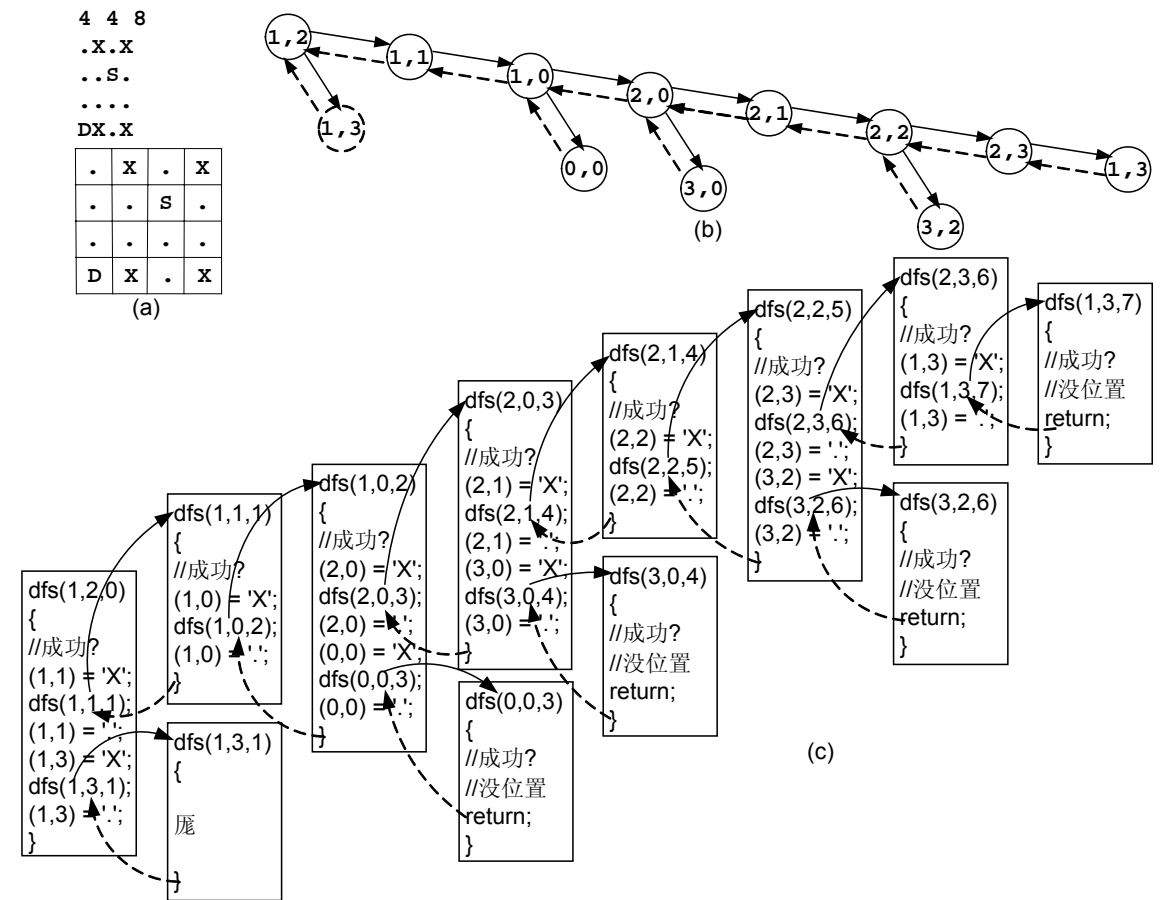


图8.13 骨头的诱惑（测试数据2分析2）

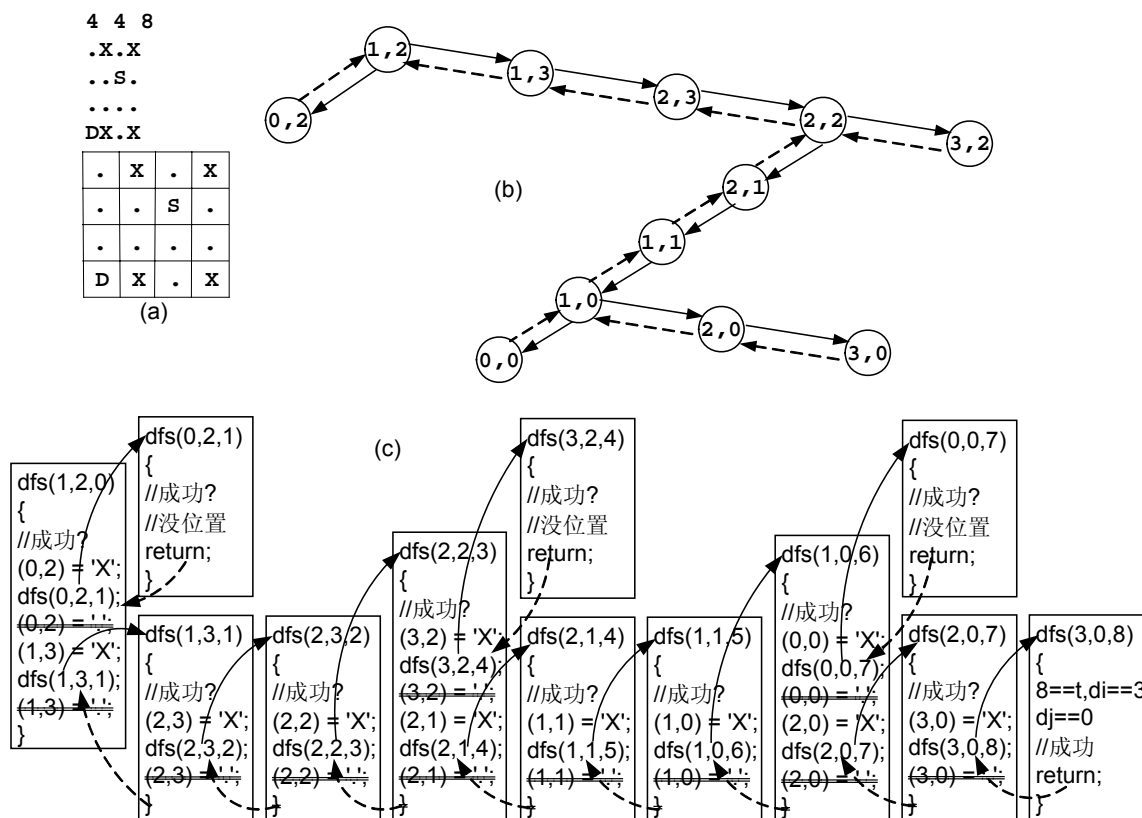


图8.14 骨头的诱惑（测试数据2分析3）

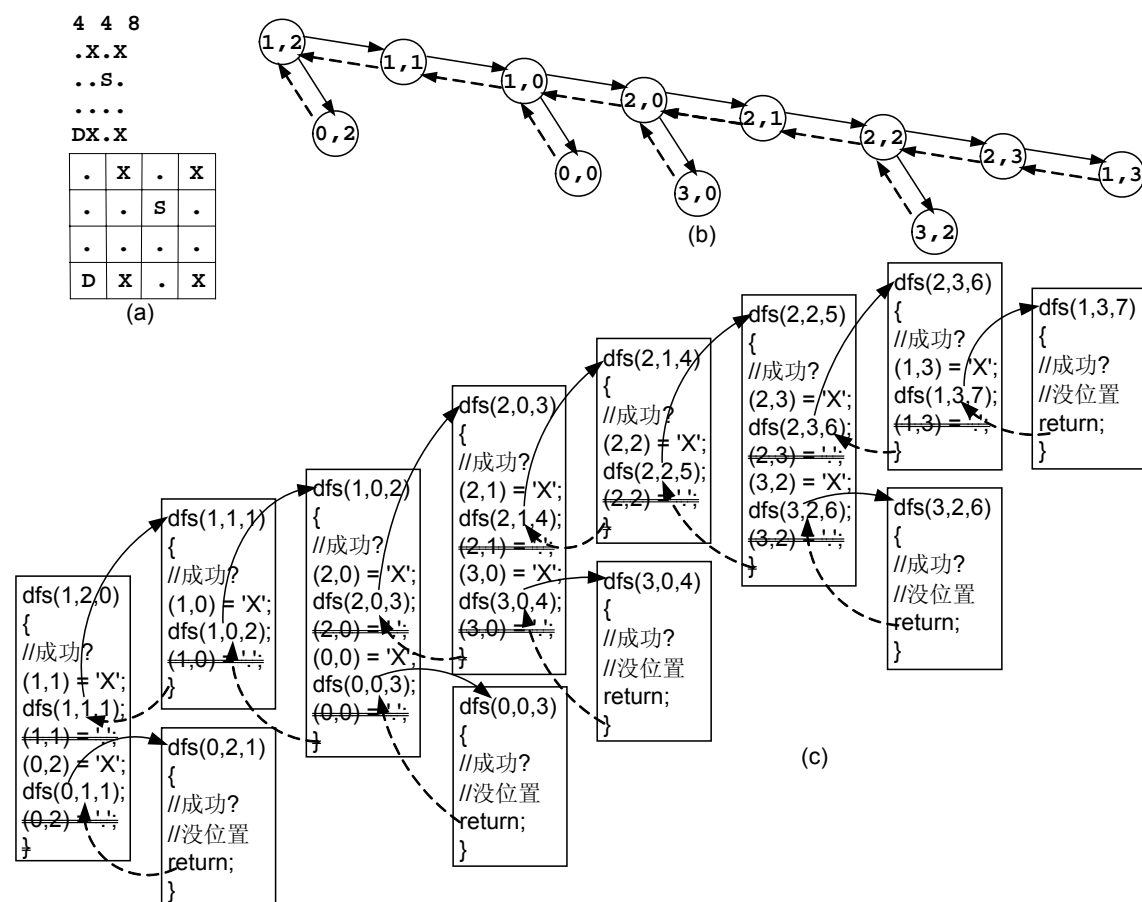


图8.15 骨头的诱惑（测试数据2分析4）

代码如下：

```
#include <stdio.h>
#include <math.h> //用到了求绝对值的函数 abs
char map[9][9]; //迷宫地图
int n, m, t; //迷宫的大小，及迷宫的门会在第 t 秒开启
int di, dj; // (di,dj): 门的位置
bool escape; //是否成功逃脱的标志，escape 为 1 表示能成功逃脱
int dir[4][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} }; //分别表示下、上、左、右四个方向
//表示起始位置为 (si,sj)，要求在第 cnt 秒达到门的位置
void dfs( int si, int sj, int cnt )
{
    int i, temp;

    if( si>n || sj>m || si<=0 || sj<=0 ) return; //边界

    if( si==di && sj==dj && cnt==t ) //成功逃脱
    {
        escape = 1; return;
    }

    //搜索过程中的剪枝
    //abs(x-ex) + abs(y - ey)表示现在所在的格子到目标格子的距离(不能走对角线)
    //t-cnt 是实际还需要的步数，将他们做差
    //如果 temp < 0 或者 temp 为奇数，那就不可能到达！
    temp = (t-cnt) - abs(si-di) - abs(sj-dj);

    if( temp<0 || temp%2 ) return;

    for( i=0; i<4; i++ )
    {
        if( map[ si+dir[i][0] ][ sj+dir[i][1] ] != 'X' )
        {
            //前进方向！将当前方格设置为墙壁 'X'
            map[ si+dir[i][0] ][ sj+dir[i][1] ] = 'X';

            dfs(si+dir[i][0], sj+dir[i][1], cnt+1); //从下一个位置继续搜索

            if(escape) return;

            map[ si+dir[i][0] ][ sj+dir[i][1] ] = '.'; //后退方向！恢复现场！
        }
    }
    return;
}

int main( )
{
    int i, j; //循环变量
    int si, sj; //小狗的起始位置
```

```

while( scanf("%d%d%d", &n, &m, &t) )
{
    if( n==0 && m==0 && t==0 ) break;    //测试数据结束

    int wall = 0;
    char temp;
    scanf( "%c", &temp ); //见下面的备注
    for( i=1; i<=n; i++ )
    {
        for( j=1; j<=m; j++ )
        {
            scanf( "%c", &map[i][j] );
            if( map[i][j]=='S' ) { si=i; sj=j; }
            else if( map[i][j]=='D' ) { di=i; dj=j; }
            else if( map[i][j]=='X' ) wall++;
        }
        scanf( "%c", &temp );
    }
    if( n*m-wall <= t ) //搜索前的剪枝
    {
        printf( "NO\n" ); continue;
    }
    escape = 0;
    map[si][sj] = 'X';
    dfs( si, sj, 0 );
    if( escape ) printf( "YES\n" ); //成功逃脱
    else printf( "NO\n" );
}
return 0;
}

```

**备注:**

1. 用 C 语言的 `scanf` 函数读入字符型数据（使用 `"%c"` 格式控制）时，会把上一行的换行符（ASCII 编码值为 10）读进来。因此在读入每一行迷宫字符前，要跳过上一行的换行符。

2. 本程序两处地方使用了剪枝，分别是搜索前的剪枝和搜索过程中的剪枝，详见程序中的注释。而所谓**剪枝**，顾名思义，就是通过某种判断，避免一些不必要的搜索过程。形象的说，就是剪去了搜索过程中的某些“枝条”，故称剪枝。有关剪枝技术的介绍，请参考相关资料。

**例 8.9 图形周长(Image Perimeters)****题目来源:**

University of Waterloo Local Contest 2001.09.22

**题目描述:**

技术员在病理实验室分析切片的数字化图象时，通过用鼠标点击切片上的对象来选择物体进行分析。研究对象的边界周长通常是个有用的量，你的任务就是计算被选择对象的周长。

数字化后的切片将被表示为矩形网格。其中，`'.'`表示空白的地方，`'X'`代表所研究对象的一部分。如图 8.16(a)是两个简单的例子。

网格内一个格子中的`'X'`代表一个完整的格子，包括它的边界，位于某个对象中(属于某个对象)。图 8.16(b)中，位于中心的`'X'`与周围 8 个方向上的`'X'`相邻。任何两个相邻的格子边重叠或

角重叠，因此任何两个相邻的格子都认为是连接的。

按照上面的法则连在一起的'X'算作一个整体。在图 8.16(a)中，网格 1 中整个矩形网格被一个物体填满了，网格 2 中有两个物体，左下方的和右上方的。

技术员总是点击含有'X'的方格，以选中包含该方格的物体。被点击方格的坐标记录下来。横纵坐标从左上角开始从 1 算起。例如，技术员可以通过点击网格 1 中的(2,2)来选中网格 1 中的物体。选中网格 2 中比较大的物体可以通过点击(2,3)，但不能点击(4,3)。

假设每个'X'方格的边长都是单位长度。因此网格 1 中的物体边长为 8(4 个边，每边为 2)。网格 2 中较大的物体周长是 18，如图 8.16(c)所示。

物体不会包括完整的空洞。因此图 8.16(d)中所示的图形是不可能的情形，而图(e)所示的图形是可能的。

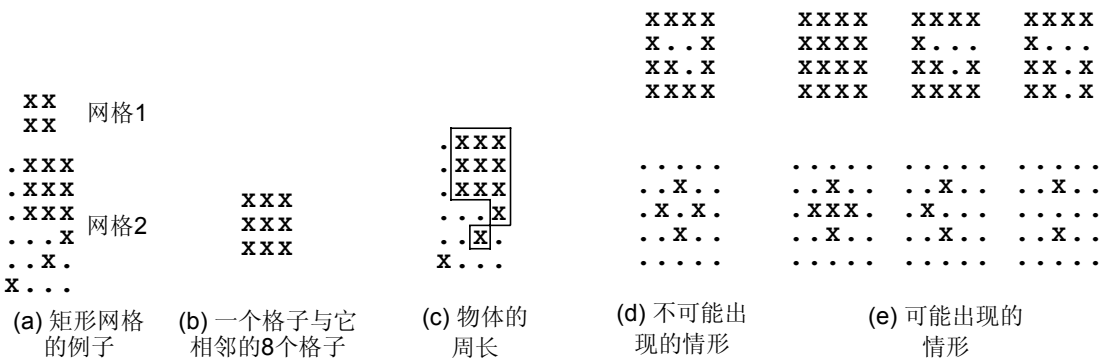


图8.16 求图形周长

**输入描述:**

输入可能包括一个或多个网格。每个网格的第一行是 4 个整数: m、n、x 和 y，分别表示该网格的行和列，以及鼠标点击的坐标。所有数据的范围在 1~20 之内。接下来有 m 行，每行有 n 个字符，描述了该网格。网格中的字符包括'.'和'X'。

输入以 4 个 0 结束，数字之间是空格，格子里面无空格。

**输出描述:**

对于输入文件每个网格，输出所选中物体的周长。

**样例输入:**

```
6 4 2 3
.XXX
.XXX
.XXX
...X
..X.
X...
5 6 1 3
.XXXX.
X...X
..XX.X
.X...X
..XXX.
0 0 0 0
```

**样例输出:**

```
18
40
```

**分析:**



样例输入中的第1个测试数据对应的示意图为图8.16(a)中的网格2，点击的位置为(2,3)，对应的数组元素为grid[1][2]，从该位置开始搜索。grid数组为保存网格的二维字符数组。

在搜索过程中要注意，对搜索到的任何一个位置(x,y)，要将该位置的字符置为除“X”和“.”字符外的其他字符，防止重复搜索(x,y)位置。

某个位置(x,y)是该物体的边界有以下八种情形，这八种情形都应使边界周长增加1：

- 1) x 为 0：左边是边界
- 2) (x-1,y)位置为字符'.': 左边的方格不属于物体
- 3) y 为 0：上边是边界
- 4) (x,y-1)位置为字符'.': 上边的方格不属于物体
- 5) x 为 m-1：右边是边界
- 6) (x+1,y)位置为字符'.': 右边的方格不属于物体
- 7) y 为 n-1：下边是边界
- 8) (x,y+1)位置为字符'.': 下边的方格不属于物体

搜索到(x,y)位置时，如果它的8个相邻位置没有超出边界，并且也是被选中物体的一部分，则要从相邻位置继续搜索。搜索的结果是整个物体的每个'X'方格都访问了一次，且仅一次。

代码如下：

```
#include <stdio.h>
char grid[20][20]; //存储网格中的字符
int m, n, count;    //网格的大小及被选中物体的边界周长
void find( int x, int y ) //从(x,y)位置开始搜索
{
    grid[x][y] = '#';    //将(x,y)位置置为其他字符，防止重复搜索
    if( y==0 ) count++;    //上边是边界
    else if( grid[x][y-1]!='.' ) count++; //上边的方格不属于物体

    if( x==m-1 ) count++;    //右边是边界
    else if( grid[x+1][y]!='.' ) count++; //右边的方格不属于物体

    if( y==n-1 ) count++; //下边是边界
    else if( grid[x][y+1]!='.' ) count++; //下边的方格不属于物体

    if( x==0 ) count++;    //左边是边界
    else if( grid[x-1][y]!='.' ) count++; //左边的方格不属于物体

    //对(x,y)的8个相邻位置，如果没有超出边界，并且也是被选中物体的一部分
    //则继续搜索
    if( x>0 && y>0 && grid[x-1][y-1]=='X' ) find( x-1, y-1 );    //左上角
    if( x>0 && grid[x-1][y]=='X' ) find( x-1, y );                //上
    if( x>0 && y<n-1 && grid[x-1][y+1]=='X' ) find( x-1, y+1 );    //右上角
    if( y<n-1 && grid[x][y+1]=='X' ) find( x, y+1 );                //右
    if( x<m-1 && y<n-1 && grid[x+1][y+1]=='X' ) find( x+1, y+1 ); //右下角
    if( x<m-1 && grid[x+1][y]=='X' ) find( x+1, y );                //下
    if( x<m-1 && y>0 && grid[x+1][y-1]=='X' ) find( x+1, y-1 );    //左下角
    if( y>0 && grid[x][y-1]=='X' ) find( x, y-1 );                //左
}
int main()
{
```

```

int i, j; //循环变量
int x, y; //点击位置
scanf( "%d %d %d %d", &m, &n, &x, &y ); //输入数据
while( m || n || x || y )
{
    getchar(); //跳过上一行的“回车键”字符
    count = 0;
    for( i=0; i<m; i++ )//输入网格中的字符
    {
        for(j=0; j<n; j++)
            scanf("%c", &grid[i][j]);
        getchar(); //跳过上一行的“回车键”字符
    }
    find( x-1, y-1 ); //输出数据中的坐标是从 1 开始计数的,所以要减 1
    printf( "%d\n", count ); //输出被选中物体的边界周长
    scanf( "%d %d %d %d", &m, &n, &x, &y );
}
return 0;
}

```

## 练习

### 8.9 礼物(Gift?!)

#### 题目描述:

在一个小山村里有一条美丽的小河。从河的左岸到右岸有  $N$  块石头排成一条直线，如下行所示：

[Left Bank] - [Rock1] - [Rock2] - [Rock3] - [Rock4] ... [Rock n] - [Right Bank]

相邻 2 块石头的距离刚好是 1 米，第 1 块石头距离左岸的距离也是 1 米，第  $N$  块石头距离右岸的距离也是 1 米。

青蛙 Frank 正准备过河，它的邻居、青蛙 Funny 过来告诉他：“Frank，儿童节快乐！我有礼物要送给你，你看到了吗？在第 5 块石头上有一个小包裹。”

Frank：“真棒！谢谢你！我过去取一下”

Funny：“等等...这个礼物只为聪明的青蛙准备的，你不能直接跳过去取包裹”

Frank：“哦？那我该怎么办？”

Funny：“你必须跳多次。第 1 跳必须从左岸到第 1 块石头上，你可以跳多次，可以向前或向后，但第  $i$  跳必须跳过  $2*i-1$  米。而且，一旦你跳上了左岸或右岸，游戏就结束了，不允许再跳了。”

Frank：“哦，这可不简单...我得想想，...，我可以试试吗？”

#### 输入描述:

输入文件包含多个测试数据，但不多于 2000 个。每个测试数据占一行，为 2 个正整数： $N$  ( $2 \leq N \leq 10^6$ ) 和  $M$  ( $1 \leq M \leq N$ )，共有  $N$  块石头，礼物位于第  $M$  块石头。 $N = 0$ ,  $M = 0$  表示输入结束。

#### 输出描述:

对每个测试数据，如果可能跳到第  $M$  块石头上，输出一行"Let me try"，否则输出"Don't make fun of me!"。

样例输入:

```
9 5
12 2
0 0
```

提示:

对样例输出的第2个测试数据, Frank 按照如下的方式跳就能取到礼物: 第1跳从左岸向前跳到第1块石头上, 距离是1米; 第2跳从第1块石头向前跳到第4块石头上, 距离是3米; 第3跳从第4块石头向前跳到第9块石头上, 距离是5米; 第4跳从第9块石头向后跳到第2块石头上, 距离是7米, 并且可以取到礼物。

样例输出:

```
Don't make fun of me!
Let me try!
```

## 8.10 火力配置网络(Fire Net)

题目描述:

假定有一个方形的城市, 街道都是直的。城市的地图是一个  $n$  行  $n$  列的方形棋盘, 每行每列代表一条街道或一堵墙。

城市中有碉堡, 每个碉堡有4个开口, 可以射击。这4个开口分别朝北、东、南和西。每个开口都有一挺机关枪朝外射击。

我们假定子弹是如此厉害, 射程可以达到任意远的距离, 也可以摧毁该方向上的碉堡。另一方面, 城市中的墙筑得是如此结实以至于可以抵挡子弹。

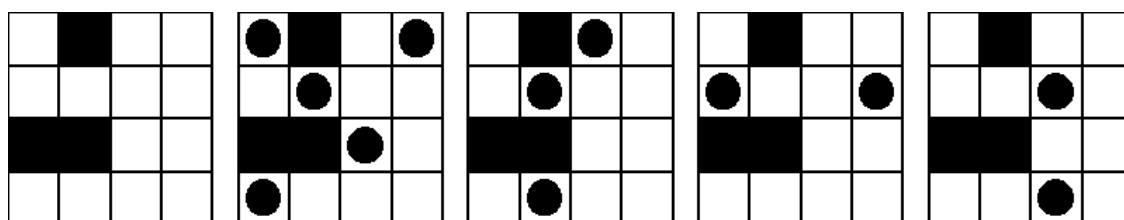


图 8.17 地图示例

本题的目标是要在城市中放置尽可能多的碉堡, 并且保证碉堡之间互相不会摧毁。碉堡放置的方案如果是合法的, 必须保证任何两个碉堡不在同一行、同一列, 除非它们之间至少有一堵墙隔开。在本题中, 我们考虑的城市都比较小, 至多  $4 \times 4$  大小。

图 8.17 给出了同一个地图的5种碉堡放置的情形。第1幅图是空的地图, 第2、3幅图是合理的放置方案, 而第4、5幅图是不合理的放置方案。在这个地图中, 最多能放置5个碉堡。第2幅图显示了放置5个碉堡的一种方法, 但也存在其他方法可以放置5个碉堡。

你的任务是编写程序, 给定地图的描述, 计算能在该地图中合理地放置碉堡的最大数目。

输入描述:

输入文件包含一个或多个地图的描述。最后一行为0, 代表输入结束。每个地图描述的第1行是一个正整数  $n$ , 表示城市的大小;  $n$  至多为4。接下来  $n$  行描述了地图, 地图中允许出现的符号及其含义如下:

' ': 代表空地;

'X': 代表墙;

输入文件中没有空格。

输出描述:

对输入文件中的每个地图描述, 输出一行, 为一个整数, 表示能在该地图中合理的放置碉堡的最大数目。

样例输入:

```
4
.X..
....
XX..
....
3
...
.XX
.XX
0
```

样例输出:

```
5
2
```

8.4 递归方法求解排列组合问题

排列与组合是组合数学中最常见的问题。例如从  $N$  个元素中取  $M$  个，计算共有多少种组合情形。对于这一类问题，根据选出来的元素的顺序是否有关，可区分为排列与组合。如果选出来的元素的顺序有关，即如果两个组合中的元素一样，但顺序不一样，这两个组合被认为是不同的组合，这是排列问题；如果顺序无关，则是组合问题。本节介绍用递归方法求解排列与组合问题。

8.4.1 排列问题

例 8.10 是全排列问题(选出所有的元素组成一个排列)，例 8.11 是一般排列问题(选出部分元素组成一个排列)。

例 8.10 素数环问题(Prime Ring Problem)

题目来源:

Asia 1996, Shanghai (Mainland China)

题目描述:

一个环上有  $n$  个圆圈，代表  $n$  个位置。现将  $1, 2, \dots, n$  共  $n$  个自然数分别放在这  $n$  个位置上，使得任意相邻两个位置上的两个数之和为素数。注意：第 1 个位置上放的数总是 1。当  $n = 6$  时，一个满足要求的素数环如图 8.18 所示。

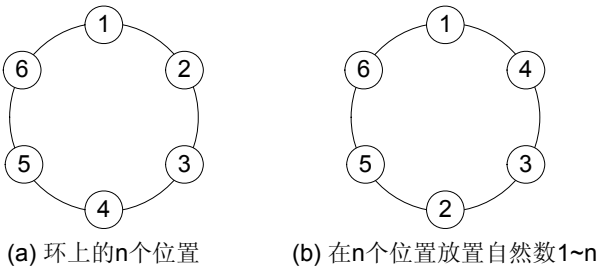


图8.18  $n=6$ 时的素数环

输入描述:

输入文件中有若干个测试数据，每个测试数据占一行，为一个整数  $n$ ， $0 < n < 20$ 。

输出描述:

输出格式如样例输出所示。每一行代表一个满足要求的放置方法，从第 1 个位置开始，按

顺时针顺序输出 1~n 位置上的自然数。按字典序输出所有满足要求的放置方法。

每个测试数据对应的输出之后有一个空行。

样例输入：

6  
8

样例输出：

6  
Case 1:  
1 4 3 2 5 6  
1 6 5 2 3 4  
  
8  
Case 2:  
1 2 3 8 5 6 7 4  
1 2 5 8 3 4 7 6  
1 4 7 6 5 8 3 2  
1 6 7 4 3 8 5 2

分析：

本题的解题思路是搜索，以  $n = 6$  加以解释，如图 8.19 所示。

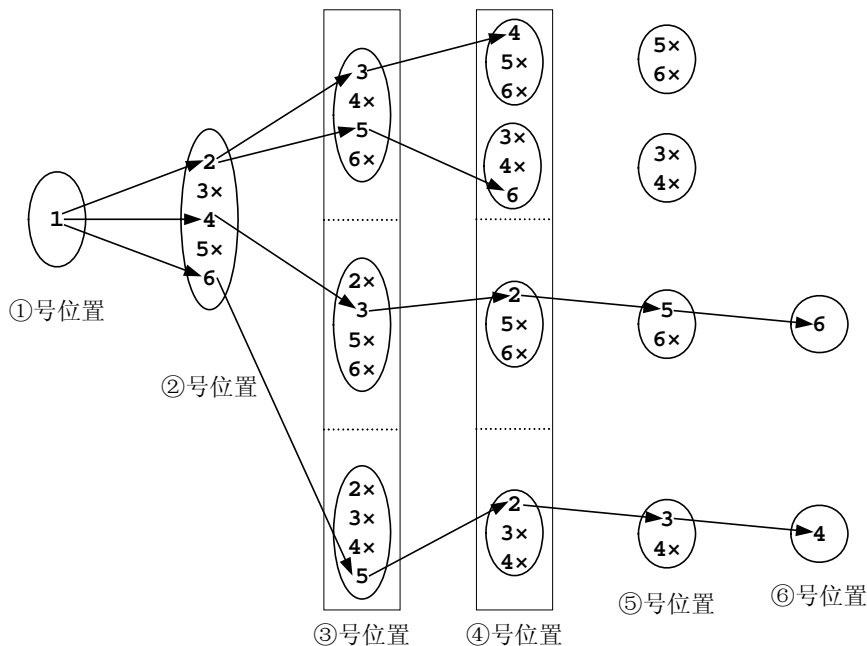


图8.19 素数环搜索策略( $n=6$ )

在①号位置上放置数 1。在②号位置上可供选择的数为 2~6，其中 3 和 5 是不可行的，对 2、4 和 6 一一试探，先试探 2。

- 1) ②号位置上放置 2 以后，③号位置上可供放置的数为 3~6，其中 4 和 6 是不可行的，对 3 和 5 也是一一试探，先试探 3。
  - a) ③号位置上放置 3 以后，④号位置上可供放置的数为 4~6，其中只有 4 是可行的，所以放置 4。这时可供⑤号位置上放置的数为 5 和 6，均不可行。所以，这些方案都不可行。
  - b) 再考虑③号位置上放置 5，④号位置上只能放置 6，此后⑤号位置上放置 3 和 4，均不可行。这些方案都排除。
- 2) 再考虑②号位置上放置 4，③号位置上只能放置 3，④号位置上只能放置 2，⑤号位置上只能放置 5，⑥号位置只能放置 6，这个方案是可行的。

...

如此搜索，直到试探所有方案为止。

本题要注意，如果输入的是奇数，则无解，直接输出空行。如果对输入的奇数，也加以搜索，则程序运行时间会增加不少。

具体实现时，因为有  $n$  个位置，要在  $n$  个位置上放置  $1\sim n$ ，而  $n$  是小于 20 的，因此可以定义两个数组 `nLoop` 和 `beUsed`，其含义如下：

`nLoop[21]`：放在  $n$  个位置上的数，`nLoop[0]` 为放置在位置 1 上的数，始终为 1。

`beUsed[21]`：使用第  $1\sim n$  个元素，每个数是否被选用的标志，`beUsed[i]` 为 1 则  $i$  已选用。

另外，为简化素数的判断，可以把 40 以内的素数存储在数组 `isPrime` 中，即：

```
int isPrime[40] = {0,0,2,3,0,5,0,7,0,0,0,11,0,13,0,0,0,17,0,19,
                  0,0,0,23,0,0,0,0,0,29,0,31,0,0,0,0,37,0,0};
```

如果 `isPrime[i]` 非 0，则  $i$  为素数，否则(`isPrime[i]` 为 0)， $i$  为和数。

搜索时，按  $1\sim n$  的顺序选择可以使用的数，则搜索得到的解的顺序就是字典序。搜索过程是通过 `search` 函数实现的，`search` 函数的原型为：

```
void search( int step );
```

参数 `step` 的含义是：已按要求放置好前 `step-1` 个数，现将要放置第 `step` 个数。

因为题目要求在第 1 个位置上总是放置 1，因此在 `main` 函数中先设置 `beUsed[1] = 1` 和 `nLoop[0] = 1`，然后调用 `search(1)` 求解。 $n=6$  时，`search` 函数执行过程如图 8.20 所示。

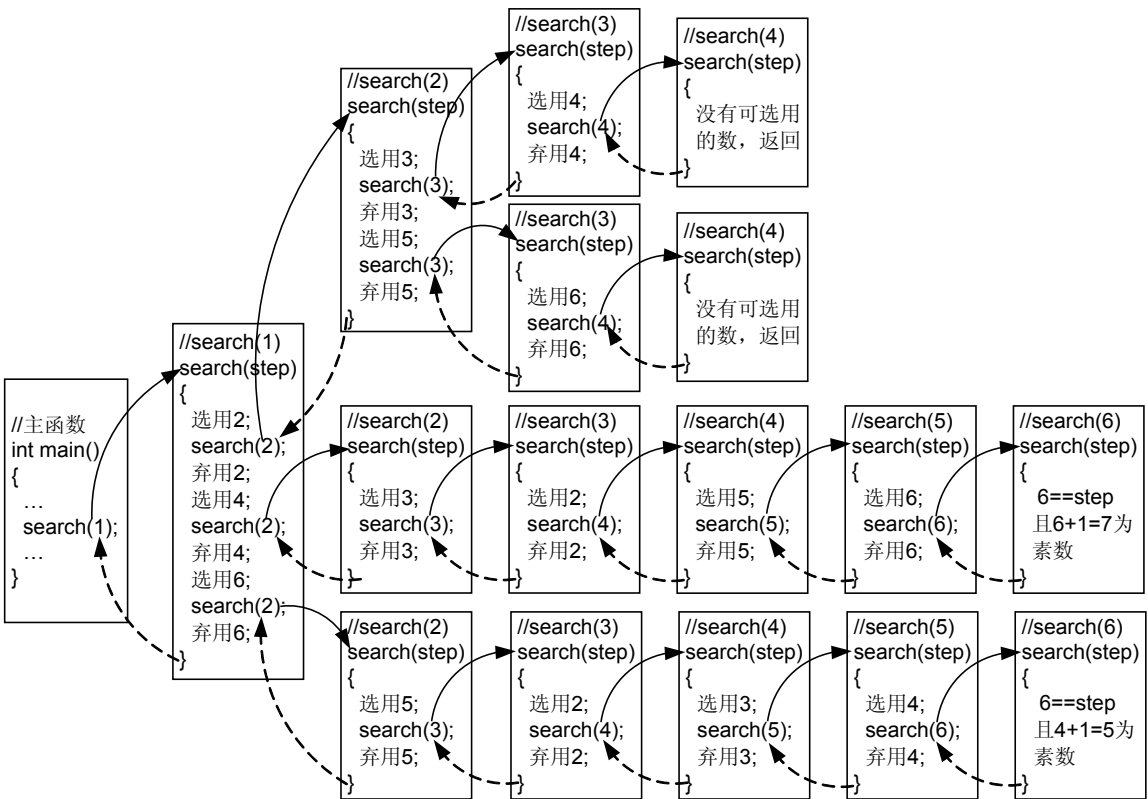


图8.20 素数环搜索策略的实现( $n=6$ )

`search(1)`的执行过程为：在执行 `search(1)`时，考虑 2 号位置上可以放置 2、4、6。以选用 4 为例加以解释，选用 4 后，递归调用 `search(2)`。

在 2 号位置上放置 4 后，3 号位置上可以选用的只有 3，所以选用 3，然后递归调用 `search(3)`。

在 3 号位置上放置 3 后，4 号位置上可以选用的只有 2，所以选用 2，然后递归调用 `search(4)`。

在 4 号位置上放置 2 后，5 号位置上可以选用的只有 5，所以选用 5，然后递归调用 `search(5)`。

在5号位置上放置5后,6号位置上可以选用的只有6,所以选用6,然后递归调用 search(6)。

在执行 search(6)时,因为6号位置上的数为6,其右边相邻位置为1号位置,已经放置1了,并且  $6+1=7$  为素数。所以,沿着这个方向搜索,找到一个可行解:1 4 3 2 5 6。

代码如下:

```
#include <stdio.h>

int n;    //输入的整数 n
int nLoop[21];    //放在 n 个位置上的数, nLoop[0]为放置在位置 1 上的数
int beUsed[21];    //1~n, 每个数是否被选用的标志, beUsed[i]为 1 则 i 已选用
//存储 40 以内的素数, isPrime[i]非 0, 则 i 为素数
int isPrime[40] = {0,0,2,3,0,5,0,7,0,0,0,11,0,13,0,0,0,17,0,19,
                  0,0,0,23,0,0,0,0,0,29,0,31,0,0,0,0,37,0,0};

int testprime( int p )    //判断 p 是否为素数
{
    return isPrime[p];
}

//参数 step 表示已按要求放置好前 step-1 个数, 现将要放置第 step 个数
void search( int step )
{
    int i;
    if( step==n ) //已放置好 n 个数
    {
        //首尾和为素数, 则这个排列满足要求
        if( testprime( nLoop[0] + nLoop[n-1] ) )
        {
            for( i=0; i<n-1; i++ )    //输出
                printf( "%d ", nLoop[i] );
            printf( "%d", nLoop[n-1] );
            printf( "\n" );
        }
        return;
    }
    for( i=1; i<=n; i++ )    //依次选用 1~n 中没有用过的数
    {
        //如果 i 没有选用, 并且选用 i 使得 i+nLoop[step-1]为素数
        //nLoop[step-1]为前一个数
        if( !beUsed[i] && testprime( i + nLoop[step-1] ) )
        {
            beUsed[i] = 1;    //选用 i
            nLoop[step] = i;
            search( step+1 );
            beUsed[i] = 0;    //回退过程, 本次弃用 i
        }
    }
}

int main( )
{
```

```

int kase = 1; //序号
while( scanf( "%d", &n )!=EOF )
{
    printf( "Case %d:\n", kase );
    kase++;
    if( n%2==0 )
    {
        beUsed[1] = 1;    //1 已使用
        nLoop[0] = 1;    //在位置 1 上放置 1
        search( 1 );
    }
    printf( "\n" );//奇数无解，直接输出空行
}
return 0;
}

```

### 例 8.11 保险箱解密高手(Safecracker)

**题目来源:**

Mid-Central USA 2002

**题目描述:**

要开Klein保险箱。保险箱的密码破解方法：要从给定的 5~12 个不同的大写字母组成的字符集中，选取 5 个，设为v, w, x, y和z，满足等式： $v - w^2 + x^3 - y^4 + z^5 = \text{target}$ 。target为一个给定的整数。在该等式中，每个字母用它在字母表中的序号替换(即A=1, B=2, ..., Z=26)。组合得到的密码为vwxyz，如果有多个满足条件的密码，则取字典序最大的，所谓字典序就是在字典中的排列顺序。

例如，给定target为 1，字符集为"ABCDEFGHIIJKL"，一个可行解为"FIECB"，这是因为： $6 - 9^2 + 5^3 - 3^4 + 2^5 = 1$ 。实际上还有其他可行解，并且最终求得解为LKEBA。

**输入描述:**

输入文件包含一行或多行，每行首先是一个正整数，表示目标值 target，不超过 12,000,000。然后是一个空格，接下来是 5~12 个不同的大写字母。输入文件最后一行，目标值 target 为 0，且字母为"END"，这标志着输入结束。

**输出描述:**

对输入文件每一行(除最后一行外)所表示的测试数据，输出满足条件的密码，如果存在多个满足条件的密码，则只输出字典序最大的；或者如果没有满足条件的密码，则输出"no solution"。具体格式如样例输出所示。

**样例输入:**

```

1 ABCDEFGHIJKL
11700519 ZAYEXIWOUVU
3072997 SOUGHT
1234567 THEQUICKFROG
0 END

```

**样例输出:**

```

LKEBA
YOXUZ
GHOST
no solution

```

**分析:**

这道题的搜索思路跟例 8.10 的搜索思路是一致的：搜索每一种组合，看是否满足题目要求，如果满足，则是一组解。



首先因为字符集中的字母都是大写字母，字母数不超过 26 个，因此可以用一整型数组 `letters` 来记录字符集中的字母。`letters[0]~letters[25]`分别对应字母 A~Z，如果 `letters[i]`为 1，则表示输入的字符集中有该字母。

在搜索时，为保证搜索到的第 1 个解就是字典序最大的解，我们可以从后面开始搜索。按 `letters[25]~letters[0]`的顺序依次选择各字母，如果 `letters[i]>0` 表示 `letters[i]`对应的字母可以选，并且如果选定 `letters[i]`，则将 `letters[i]`的值减 1(`letters[i]`的值变为 0)，这样保证不会重复选同一个字母。如果如此进行下去这种方案行不通，还得弃用 `letters[i]`，即将 `letters[i]`的值加 1(`letters[i]`的值变为 1)。

代码如下：

```
#include <stdio.h>
#include <string.h>
#include <math.h>
const int MAX_L = 26;
const int MAX_N = 5;

int target;          //输入的 target
char key[15]; //输入的 5~12 个不同的大写字母
//letters[0]~letters[25]分别对应字母 A~Z
//如果 letters[i] 为 1，则表示输入的字符集中有该字母
int letters[MAX_L];
int nums[MAX_N];      //选用的 5 个整数(输出时转换成字母输出)

//参数 depth 表示已选好前 depth-1 个数，现将要选第 depth 个数
bool FindKey(int depth)
{
    if( depth == 5 )
    {
        int sum = nums[0] - pow(nums[1],2) + pow(nums[2],3)
                    - pow(nums[3],4) + pow(nums[4],5);
        if( target == sum ) return true;
        return false;
    }
    //从后面搜索所有的字母，保证找到的第 1 个解就是字典序最大的解
    for( int i = MAX_L-1; i >= 0; i-- )
    {
        if( letters[i] > 0 )    //letters[i]对应的字母没选用
        {
            --letters[i]; //选择 letters[i]
            nums[depth] = i+1;    //在 nums 数组中存放 letters[i]字母对应的整数
            if( FindKey(depth+1) ) return true;
            ++letters[i]; //放弃 letters[i]
        }
    }
    return false;
}

int main( )
{
```

```

while( scanf( "%d%s", &target, key) )
{
    if(target == 0 && !strcmp(key,"END")) break;    //结束
    memset( letters, 0, sizeof(letters) );
    for( int i = 0; i < strlen(key); i++ ) //在 letters 中记录输入字符集中的每个字母
        ++letters[ key[i]-'A' ];
    if( FindKey(0) )    //FindKey(0)为 true 表示已搜索到解，且是字典序最大的解
    {
        for(int i = 0; i < MAX_N; ++i)
            printf( "%c", char(nums[i] + 'A' - 1) );
        printf( "\n" );
    }
    else printf( "no solution\n" );
}
return 0;
}

```

### 8.4.2 组合问题

对从  $N$  个元素中选取  $M$  个元素的组合问题，根据  $N$  个元素是否能重复选，又可分为可重复组合问题和不可重复组合问题。例 8.12 是可重复组合问题，例 8.13 是不可重复组合问题。另外，例 8.14 要选出所有的元素，这些元素的顺序无关，是全组合问题。

#### 例 8.12 方形硬币(Square Coins)

题目来源：

Asia 1999, Kyoto (Japan)

题目描述：

Silverland国家的人民用方形的硬币。不仅硬币的形状是方形的，而且硬币的面值也是平方数。硬币的面值为  $1^2, 2^2, 3^2, \dots, 17^2$ ，即 1, 4, 9, ..., 289。问要支付一定额的货币，有多少种支付方法。

例如，若要支付总额为 10 的货币，则有四种方法：

- 1) 10 个面值为 1 的货币；
- 2) 1 个面值为 4 的货币和 6 个面值为 1 的货币；
- 3) 2 个面值为 4 的货币和 2 个面值为 1 的货币；
- 4) 1 个面值为 9 的货币和 1 个面值为 1 的货币。

输入描述：

输入文件包含若干行，每行为一个整数，表示需要支付货币的定额，最后一行为 0，表示输入结束。货币的金额均为正整数并且不超过 300。

输出描述：

对输入文件中的每个货币金额，输出一个整数，表示支付该金额的方法种数。

样例输入：

```

2
10
30
0

```

样例输出：

```

1
4
27

```

**分析：**

本题对给定的货币金额，要求有多少种支付方案。这个问题类似于用天平称重，如图 8.21 所示。

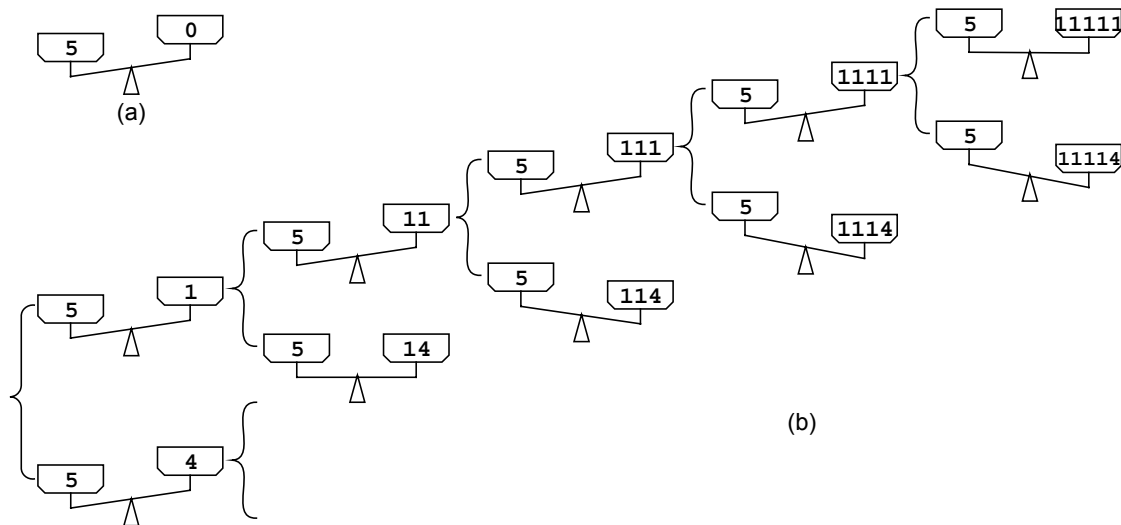


图8.21 方形硬币（搜索策略）

假设天平左边的物体重量为 5，可用的砝码重量为 1, 4, 9, 16, ...，问一共有多少种称重方案。称重时，是试探性地选择砝码，如图(b)所示。先放 1 个重量为 1 的砝码，轻了；再继续放 1 个重量为 1 的砝码，还是轻了，...，直到放了 5 个重量为 1 的砝码，天平平衡了，这是一种称重方案。注意，搜索时，如果天平平衡了，或这天平右侧超重了，则这个搜索方向不再搜索，否则搜索过程无穷尽。另外，当放了 1 个重量为 1 的砝码，再放 1 个重量为 4 的砝码，天平也平衡了，这也是一种称重方案。...。按砝码重量从小到大依次选择不同的砝码，从而统计称重方案的数目。

实现方法如下。定义一函数：

```
int build( int n, int count, int sum, int j );
```

其中各参数的含义为：**n**，需要支付的货币金额；**count**，现已求得的支付方案数；**sum**，当前选用的硬币面值总额；**j**，当前最后选的硬币是第 **j** 种硬币，面值为 **j\*j**。对输入的这个支付总额，只需调用 **build(n,0,0,0)**即可求解。

接下来，以图 8.21 所示的求重量为 5 的称重方案为例，分析本题支付货币总额为 5 时，**build** 函数的递归调用过程。如图 8.22 所示，调用 **build(5,0,0,0)**函数求解。

在执行 **build(5,0,0,0)**函数时，先选择面值为 1 的硬币，然后递归调用 **build(5,0,1,1)**函数。**build(5,0,1,1)**函数又递归调用 **build(5,0,2,1)**函数，**build(5,0,2,1)**函数又递归调用 **build(5,0,3,1)**函数，**build(5,0,3,1)**函数又递归调用 **build(5,0,4,1)**函数。而在执行 **build(5,0,4,1)**函数时，已选用 4 个面值为 1 的硬币，再选用 1 个面值为 1 的硬币，因为 **sum==n**，**build(5,0,4,1)**函数执行完毕，返回的方案数为 1。

如图 8.22 所示，返回到 **build(5,0,3,1)**函数里。这时已选用了 3 个面值为 1 的硬币，再继续选面值为 4 的硬币，超过了 5，所以不再继续搜索，返回到 **build(5,0,2,1)**函数。这时已选用了 2 个面值为 1 的硬币，再继续选面值为 4 的硬币，也超过了 5，所以不再继续搜索，返回到 **build(5,0,1,1)**函数。这时已选用了 1 个面值为 1 的硬币，再继续选面值为 4 的硬币，因为 **sum==n**，**build(5,0,1,1)**函数执行完毕，返回的方案数为 2。

返回到 **build(5,0,0,0)**函数里，这时没有选用硬币，所以继续选择面值为 4 的硬币，然后递归调用 **build(5,2,4,2)**函数。在 **build(5,2,4,2)**函数里，继续选用面值为 9 的硬币，超过了 5，所以返回到 **build(5,0,0,0)**函数里。继续选用面值为 9 的硬币，超过了 5。至此，**build(5,0,0,0)**函

数执行完毕，求得的支付方案数为 2。

本题从小到大依次选择不同的硬币，所以求得的解没有重复。

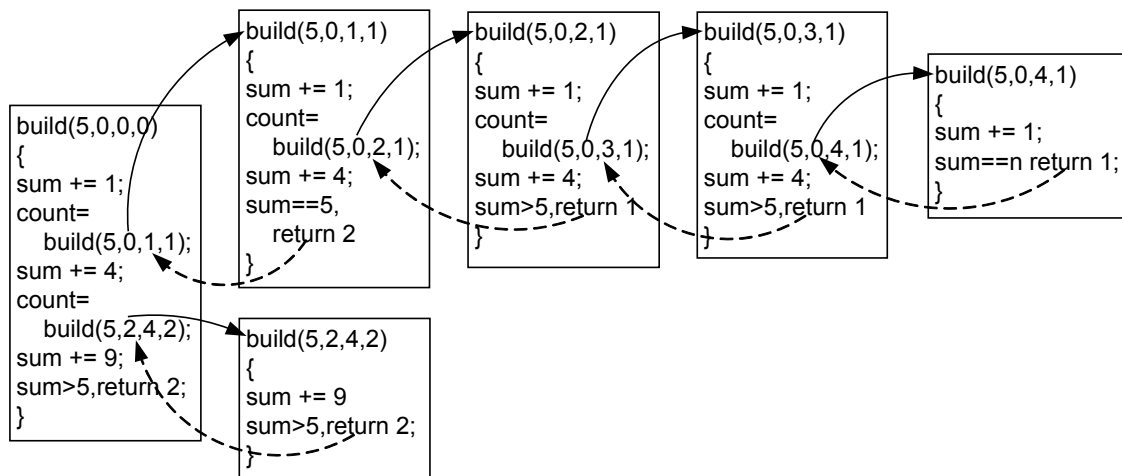


图8.22 方形硬币（搜索函数执行过程）

代码如下：

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

//n，需要支付的货币金额；count，现已求得的支付方法方案数；

//sum，当前选用的硬币面值总额；j，当前最后选的硬币是第 j 种硬币，面值为 j\*j

int build( int n, int count, int sum, int j ) //调用： build(n,0,0,1)

```
{
    int i;    //循环变量

    for( i=1; i<=17; i++ ) //搜索所有面值的硬币
    {
        //避免出现重复的，比如要支付 5 的话，如果没有这条语句
        //(1 个 1+1 个 4)这个组合将要统计 2 次，即 1+4 和 4+1
        if( i<=j ) continue;
        sum += i*i; //选用面值为 i*i 的硬币
        if( sum==n ) return ++count; //找到一种支付方案
        if( sum>n ) return count; //超出了支付总额，不再搜索
        count = build( n, count, sum, i );//没有超出则递归调用 build 函数继续搜索
        sum -= i*i; //弃用面值为 i*i 的硬币
    }
    return count;
}

int main( void )
{
    int n;    //输入的需要支付的货币金额
    int count; //求得的支付方案数
    while( 1 )
    {
        scanf("%d", &n);
        if( n==0 ) break;
        count = build( n,0,0,0 ); //搜索求解
        printf("%d\n", count);
    }
}
```

```

    }
    return 0;
}

```

### 例 8.13 求和(Sum It Up)

**题目来源:**

Mid-Central USA 1997

**题目描述:**

给定一个总和  $t$ ，以及  $n$  个整数，从这  $n$  个整数中选取若干个，使得和为  $t$ 。求所有满足这个条件的组合情况。例如，假设  $t=4$ ， $n=6$ ，这 6 个整数为：[4, 3, 2, 2, 1, 1]。这 6 个整数中，有 4 个不同的组合，满足和为 4：4, 3+1, 2+2, 2+1+1。

注意：同一个整数在一个组合中可以出现多次，只要不超过它在整数列表中出现的次数；一个整数也可以单独成为一个组合。

**输入描述:**

输入文件中包含一个或多个测试数据。每个测试数据占一行：首先是总和  $t$ ；接下来是整数  $n$ ，表示这组数中整数的个数；最后是  $n$  个整数： $x_1, \dots, x_n$ 。如果  $n=0$ ，则表示输入结束。否则， $t$  为正整数，且  $t < 1000$ ， $n$  的值满足： $1 \leq n \leq 12$ ， $x_1, \dots, x_n$  均为小于 100 的正整数。测试数据中所有数都是用空格隔开的。每个测试数据中的  $n$  个整数是以非递增的顺序排列的，允许有重复的数据。

**输出描述:**

对每个测试数据，首先输出一行，格式为“Sums of  $t$ ：”， $t$  为测试数据中的总和  $t$ 。然后输出所有满足要求的组合，每个组合占一行，如果没有满足要求的组合，则仅输出“NONE”。组合中的正整数以非递增的顺序排列。

组合以在其中出现的整数的降序排列，也就是说，首先按第一个数的降序排序，第 1 个数相同则按第 2 个数的降序排序，以此类推。在每个测试数据中，各个组合必须互不相同，不能重复输出。

**样例输入:**

```

4 6 4 3 2 2 1 1
5 3 2 1 1
400 12 50 50 50 50 50 50 25 25 25 25 25 25
0 0

```

**样例输出:**

```

Sums of 4:
4
3+1
2+2
2+1+1
Sums of 5:
NONE
Sums of 400:
50+50+50+50+50+50+25+25+25+25
50+50+50+50+50+25+25+25+25+25+25

```

**分析:**

本题的搜索策略跟例 8.12 差不多：对给定的  $n$  个数，搜索所有的组合，如果满足条件，则输出。但是要注意以下几个问题。

1. 本题要求对找到的组合以在其中出现的整数的降序排列，因此对输入的  $n$  个整数，可以先排序，按从大到小的顺序排序。排序方法详见第 9 章。

2. 本题特别提到，同一个整数在一个组合中可以出现多次，只要不超过它在整数列表中出现的次数。其实不需作特别的处理，因为如果  $n$  个整数中有相等的数，则这些数是单独读进来

的，存放在数组里，每个数要么选，要么不选，所以对相等的数，选择次数不会超过其出现的次数。

3. 本题需考虑以下一种情形，如输入为：8 3 5 3 3，表示有 3 个数 5、3 和 3，总和为 8，则有两个组合都满足要求，即“5+3”和“5+3”，本题对这种组合认为是同一个，不能重复输出。解决方法是：在搜索时，如果前后两个数相等，且前一个数没有选，则不考虑后一个数，详见代码中的注释。例如对上述输入情形，先选择第①个数 5，再选择第②个数 3，总和为 8，是符合要求的组合；再选第③个数就超出了。弃用第②个数，选择第③个数，本来这种组合也是满足条件的，但是由于第②、③个数相等，这个组合弃用了第②个数而选用第③个数，所以这个组合也不能输出。

代码如下：

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

int t, n; //输入文件中的总和 t 和整数的个数 n
int num[20]; //输入的 n 个整数
int choose[20]; //选用的整数
int FLAG; //是否找到满足要求的组合，如果 FLAG 为 0，表示没有找到
char flag[20]; //flag[i]为第 i 个数选用的标志，如果为 1，则 flag[i]已选用

int cmp( void const *a, void const *b ) //qsort 使用的比较函数
{
    return *(int *)b - *(int *)a;
}

//start, 接下来从 num 数组中的第 start 个数开始选；
//tag, 目前选用的数的和；count, 目前选用的整数个数
void search( int start, int tag, int count )
{
    int i, k; //循环变量
    if( tag==t )
    {
        FLAG = 1;
        printf( "%d", choose[0] ); //输出找到的组合
        for( k=1; k<count; k++ )
            printf( "+%d", choose[k] );
        printf( "\n" );
    }
    for( i=start; i<n; i++ ) //考虑 num 数组中第 start~n-1 个数
    {
        //前后两个数相等，前一个数没有选，则不考虑后一个数
        if( i!=0 && num[i]==num[i-1] && !flag[i-1] ) continue;
        if( tag + num[i] > t ) continue; //超出了，跳过
        choose[count] = num[i];
        flag[i] = 1; //选用 num[i]
        search( i+1, tag+num[i], count+1 );
        flag[i] = 0; //弃用 num[i]
    }
}
```

```

int main( )
{
    int i;
    while( scanf("%d%d", &t, &n) )
    {
        if( n==0 ) break; //输入结束
        FLAG = 0;
        for( i=0; i<n; i++ ) //输入 n 个整数
            scanf( "%d", &num[i] );

        qsort( num, n, sizeof(num[0]), cmp );    //对 n 个整数排序
        printf( "Sums of %d:\n", t );
        search( 0, 0, 0 );
        if( !FLAG ) printf("NONE\n");
    }
    return 0;
}

```

**例 8.14 正方形(Square)****题目来源:**

University of Waterloo Local Contest 2002.09.21

**题目描述:**

给定一些不同长度的棍子，问能不能将这些棍子头尾相连，构成一个正方形。

**输入描述:**

第一行是一个整数  $N$ ，表示测试数据的数目。每个测试数据以一个整数  $M$  开头， $4 \leq M \leq 20$ ，表示棍子的数目；接下来是  $M$  个整数，表示  $M$  根棍子的长度，这些整数的范围在 1 到 10,000 之间。

**输出描述:**

对每个测试数据，如果可以构成一个正方形，输出 **yes**，否则输出 **no**。

**样例输入:**

```

3
4 1 1 1 1
5 10 20 30 40 50
8 1 2 7 6 4 3 5 4

```

**样例输出:**

```

yes
no
yes

```

**分析:**

这道题是要求判断是否能将所给的  $M$  条木棍拼接成一个正方形。本题采用搜索求解。在搜索之前，我们应先判断问题是否一定无解，以避免不必要的搜索。

- 1) 先计算出  $M$  条木棍的总长  $sum$ ，如果  $sum$  不是 4 的倍数，显然这  $M$  条木棍不可能组成正方形。
- 2) 如果  $sum$  是 4 的倍数，记  $ave = sum/4$ ，如果  $M$  条木棍中有长度大于  $ave$  的，则这  $M$  条木棍也不可能组成正方形。

对于这两种情况，可以马上输出 **"no"**。

对于这个问题，搜索的方式有以下两种：

- 1) 搜索棍子，每次考虑将一条木棍放在一条边上。
- 2) 搜索正方形的边，每次考虑往某一条边上放木棍。

其中第二种搜索方式比第一种搜索方式高效得多。

按照第二种搜索方式进行搜索的策略为：依次构造正方形的每条边；在构造时，从没有用过的木棍中选一条放在上面，如果选用这根木棍刚好能构造这条边，则下一次继续构造下一条边；如果加上这根木棍的长度超过了  $ave$ ，则要弃用这根木棍；如果加上这根木棍长度还没达到  $ave$ ，则继续选用其他的棍子来构造这条边。一旦所有边都可以成功构造，输出"yes"，否则最后输出"no"。

对  $M$  根木棍的长度进行排序后再搜索有利于加快搜索速度。样例输入中的第 3 个测试数据，其搜索过程可以用图 8.23 来表示。图(a)表示构造正方形第 1 条边的过程，先选用 7，符号“√”表示在构造当前边时选用的木棍。选用 7 以后，还没构造好第 1 条边，所以继续选用木棍，选用 6, 5, 4, 3, 2 都会使得该边的长度超过了  $ave$ ，所以弃用，一直搜索到 1。选用 1 以后，第 1 条边移构造好。图中字符“●”表示已经被选用的木棍，字符“○”表示还未选用的木棍。

图(b)表示构造第 2 条边的过程，选用 6 和 2。图(c)表示构造第 3 条边的过程，选用 5 和 3。图(d)表示构造第 4 条边的过程，选用 4 和 4。至此 4 条边构造好，因此搜索结果标明这 8 条边能构成一个正方形。

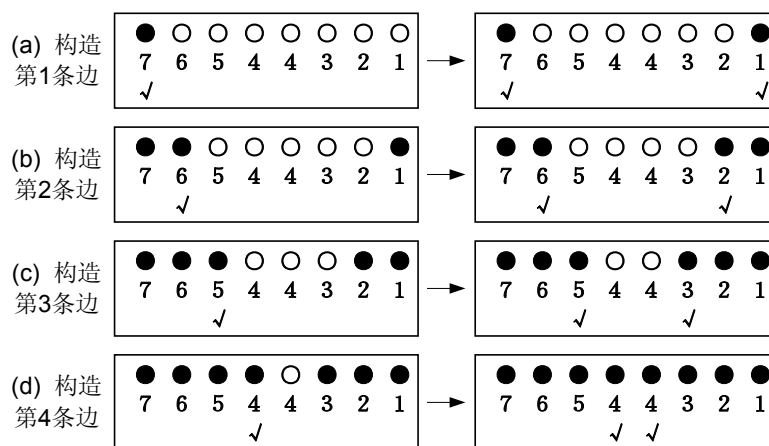


图8.23 正方形（搜索策略）

代码如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int M, side[21], ave; //棍子的数目 M, M 根棍子的长度, M 根棍子长度总和/4
int mark[21], flag; //M 根棍子选用的标志, 及是否能组成正方形的标志

int cmp(const void *a, const void *b) //排序用的比较函数
{
    return (*(int *)b - *(int *)a);
}

//st, 接下来从第 st 条边开始选; len, 当前正在构造的边的长度;
//cnt, 已构造好 cnt 条边长; index, 已经选用了 index 根棍子
void find(int st, int len, int cnt, int index) //搜索求解
{
    int i; //循环变量
    if( cnt == 4 && index == M )
    {
```



```

        flag = 1; return;
    }
    if( len == 0 ) //从 0 开始构造当前边时选用棍子
    {
        for( i = 0; i < M ; i++ )
        {
            if( !mark[i] ) break;
        }
        mark[i] = 1; //选用第 i 根棍子
        if( len + side[i] == ave ) //选用第 i 根棍子刚好构造好第 cnt 条边
            find( 0, 0, cnt+1, index + 1 ); //从 0 开始构造第 cnt+1 条边
        else find( 0, len + side[i], cnt, index + 1 );//还没构造好，继续选棍子
        mark[i] = 0; //弃用第 i 根棍子
        return;
    }
    for( i = st; i < M; i++ )
    {
        if( !mark[i] && len + side[i] <= ave )
        {
            mark[i] = 1; //选用第 i 根棍子
            if( len + side[i] == ave ) //选用第 i 根棍子刚好构造好第 cnt 条边
                find( 0, 0, cnt + 1, index + 1 ); //从 0 开始构造第 cnt+1 条边
            else find( i + 1, len + side[i], cnt, index + 1 );//还没构造好，继续选棍子
            mark[i] = 0; //弃用第 i 根棍子
            if(flag) return;
        }
    }
}

int main( )
{
    int i; //循环变量
    int N, sum; //测试数据个数，及 M 跟棍子的长度总和
    scanf( "%d", &N );
    while( N-- )
    {
        scanf( "%d", &M );
        for( sum = 0, i = 0; i < M; i++ )
        {
            scanf( "%d", &side[i] );//读入 M 根棍子的长度
            sum += side[i]; //累加 M 根棍子的长度
        }
        if( sum % 4 )
        {
            printf("no\n"); continue;
        }
        qsort( side, M, sizeof(int), cmp ); //按从大到小的顺序排序
        ave = sum / 4;
        if( side[0] > ave ) //最长棍子长度大于总和的 1/4
        {

```

```

        printf("no\n"); continue;
    }
    flag = 0;
    memset( mark, 0, sizeof(mark) );
    find( 0, 0, 0, 0 );
    if( flag ) printf("yes\n");
    else printf("no\n");
}
return 0;
}

```

## 练习

练习 8.11 是全排列问题, 练习 8.12 是一般组合问题, 练习 8.13 是不重复组合问题。

### 8.11 字母排列(Anagram)

#### 题目描述:

给定几个英文字母, 编写程序输出由这几个字母组成的所有可能的单词。比如, 给定的字母是'a'、'b'和'c', 程序要输出"abc"、"acb"、"bac"、"bca"、"cab"和"cba", 这是由这 3 个字母组合的所有可能的单词。

在给定的字母中, 有的字母可能会重复出现, 在这种情况下不同的排列可能得到相同的单词。本题需要按字母表的升序输出所有的单词, 但相同的单词只需要输出一次。

#### 输入描述:

输入数据的第 1 行是一个表示测试数据数目的正整数  $n$ 。后面有  $n$  行, 每行包含若干个大写英文字母, 并且同一个字母的大小写在本题中认为是两个不同的字母。

#### 输出描述:

对每组测试数据, 按字母表的升序输出所有可能的单词。注意, 字母表中字母的大小顺序定义为: 'A' < 'a' < 'B' < 'b' < ... < 'Z' < 'z'。

#### 样例输入:

```

1
aAb

```

#### 样例输出:

```

Aab
Aba
aAb
abA
bAa
baA

```

### 8.12 抽奖游戏(Lotto)

#### 题目来源:

University of Ulm Local Contest 1996

#### 题目描述:

在一种抽奖游戏中, 游戏者必须从集合  $S = \{ 1, 2, \dots, 49 \}$  中选取 6 个数。选取 6 个数的一种策略是: 先从  $S$  中选取一个子集  $S_1$ , 子集  $S_1$  包含  $k$  个数,  $k > 6$ , 然后再从  $S_1$  中选取 6 个数。例如, 当  $k = 8$  时, 假设选取的子集  $S_1 = \{ 1, 2, 3, 5, 8, 13, 21, 34 \}$ , 从  $S_1$  中再选 6 个数就有 28 种可能:  $[1, 2, 3, 5, 8, 13]$ ,  $[1, 2, 3, 5, 8, 21]$ ,  $[1, 2, 3, 5, 8, 34]$ ,  $[1, 2, 3, 5, 13, 21]$ , ...,  $[3, 5, 8, 13, 21, 34]$ 。

你的任务是编写程序, 读入  $k$  和子集  $S_1$ , 输出从子集  $S_1$  的  $k$  个数中选取 6 个数的所有情

形。

#### 输入描述:

输入文件包含一个或多个测试数据。每个测试数据占一行，包含若干个整数，用空格隔开。这些整数中，第 1 个数为  $k$ ， $6 < k < 13$ ，然后是  $k$  个整数，代表子集  $S_1$  中的  $k$  个数，按升序排列。 $k = 0$  代表输入结束。

#### 输出描述:

对每个测试数据，输出所有组合，每个组合占一行。组合中的数以升序排列，每个数用空格隔开。各组合以字典序排列，也就是说，先按最小的数排列，如果最小的数相同，再按次小的数排列，如此类推。如样例输出所示。各个测试数据对应的输出之间用空行隔开，最后一个测试数据的输出之后没有空行。

#### 样例输入:

```
7 1 2 3 4 5 6 7
0
```

#### 样例输出:

```
1 2 3 4 5 6
1 2 3 4 5 7
1 2 3 4 6 7
1 2 3 5 6 7
1 2 4 5 6 7
1 3 4 5 6 7
2 3 4 5 6 7
```

### 8.13 分配大理石(Dividing)

#### 题目描述:

Marsha 和 Bill 想把他们收集的大理石重新分配使得每人得到价值相等的一份。每块大理石的的价格为  $1 \sim 6$  的自然数。要求编写一个程序，判断他们是否能分到价值相等的大理石。

#### 输入描述:

输入文件中的每行代表需要按价值平均分配的大理石。每一行有 6 个非负的整数， $n_1, n_2, \dots, n_6$ ，其中  $n_i$  代表价格为  $i$  的大理石个数。

输入文件的最后一行为“0 0 0 0 0 0”表示输入结束，这一行不需处理。

#### 输出描述:

对输入文件的每个测试数据，首先输出“Collection #k:”，其中  $k$  表示测试数据的序号，然后输出“Can be divided.”或“Can't be divided.”。

每个测试数据的输出之后有一个空行。

#### 样例输入:

```
1 0 1 2 0 0
1 0 0 1 1
0 0 0 0 0 0
```

#### 样例输出:

```
Collection #1:
Can't be divided.

Collection #2:
Can be divided.
```

## 第9章 排序及检索

对数据进行排序是数据处理中经常要用到的操作。本章首先介绍3种常用的、比较简单的排序算法的思想及其实现，以及用于排序的系统函数 `qsort` 的使用方法。另外，对一组已经排好顺序的数据进行检索也是经常要用到的操作，本章因此介绍了二分法的思想，以及二分法在检索中的应用。

### 9.1 排序算法

排序算法有很多种，本节介绍其中比较简单的3种：插入法、冒泡法、简单选择法，在后续课程中还会介绍其他更复杂、也是更好的排序算法。

#### 9.1.1 插入排序法

插入排序法是一种很朴素的排序算法。其基本思路是将每个待排序的数插入到已排序序列中的合适位置。

例如，假设采用插入排序法对 49, 38, 65, 97, 76, 13, 27, 30 这8个数按从小到大的顺序排序。这8个数已经存储在数组 `a` 了，将这8个数依次插入到数组 `b` 中。图 9.1(b)~(f)演示了插入 `a[0]~a[4]` 的过程。

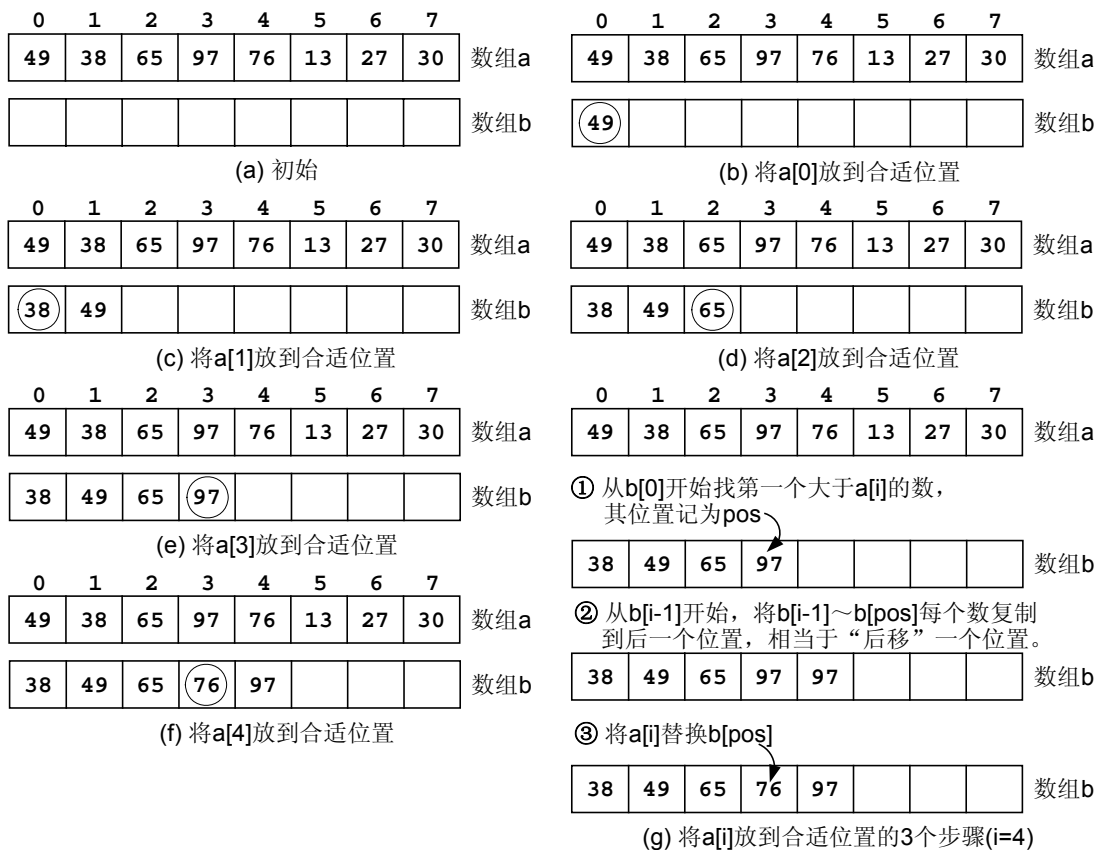


图9.1 插入排序法（演示一）

排序方法是：第  $i$  次插入的数是 `a[i]`，这时数组 `b` 中的 `b[0]~b[i-1]` 已经有序了，将 `a[i]` 插入到数组 `b` 中的合适位置；插入 `a[i]` 的过程，分为3个步骤：

- ① 从 `b[0]` 开始，在 `b[0]~b[i-1]` 范围内找第一个大于 `a[i]` 的数，其位置记为 `pos`；

- ② 从  $b[i-1]$  开始, 将  $b[i-1] \sim b[pos]$  复制到后一个位置, 相当于将这些数“后移”一个位置;
- ③ 将  $a[i]$  放置在  $b[pos]$  位置上, 替换  $b[pos]$ 。

图 9.1(g) 演示了插入  $a[4]$  的 3 个步骤。这时  $i = 4$ , 待插入的数是  $a[i] = 4$ 。数组  $b$  中  $b[0] \sim b[3]$  已经有序了, 且第一个大于  $a[4]$  的数是 97, 其位置  $pos = 3$ 。从  $b[i-1]$  开始, 将  $b[i-1] \sim b[pos]$  每个数复制到后一个位置, 腾出  $b[pos]$  这个位置。最后将  $a[4]$  放置在  $b[pos]$  位置上。

在下面的代码中, 对插入  $a[i]$  的 3 个步骤特意用边框标明了。

**例 9.1** 用插入法实现对输入的 10 个数按从小到大进行排序。

代码如下:

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

```
void main( )
```

```
{
```

```
    int a[10], b[10]; //输入的 10 个数及排序后的 10 个数
```

```
    int i, j, k; //循环变量
```

```
    int pos; //每个数的插入位置
```

```
    printf( "input 10 numbers : \n" );
```

```
    for ( i=0; i<10; i++ ) //输入
```

```
        scanf( "%d", &a[i] );
```

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

```
    for( i=0; i<10; i++ ) //将  $a[i]$  插入合适的位置
```

```
    {
```

```
        //在数组  $b$  中,  $b[0] \sim b[i-1]$  已经有序了, 给  $a[i]$  找到合适的位置
```

(1)

```
        for( j=0; j<=i-1; j++ )
```

```
        {
```

```
            if( b[j]>a[i] ) break;
```

```
        }
```

```
        pos = j; //  $a[i]$  的插入位置
```

```
        for( k=i-1; k>=pos; k-- ) //将  $b[pos] \sim b[i-1]$  后移一个位置
```

(2)

```
            b[k+1] = b[k];
```

```
        b[pos] = a[i]; //将  $a[i]$  放置在  $b[pos]$  位置上
```

(3)

```
    }
```

```
    printf( "the sorted numbers : \n" );
```

```
    for( i=0; i<10; i++ ) //输出
```

```
        printf( "%d ", b[i] );
```

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

```
}
```

该程序的执行过程如下:

input 10 numbers:

7 12 -49 38 65 97 -76 13 27 30 ✓

the sorted numbers:

-76 -49 7 12 13 27 30 38 65 97

以上第①个步骤要是改为从  $b[i-1]$  开始往前找第一个小于或等于  $a[i]$  的数, 则第①、②两个步骤可以合二为一, 即以上 3 个步骤改为以下两个步骤:

- ① 从  $b[i-1]$  往前查看每一个元素, 如果比  $a[i]$  大, 则将其“复制”到后一个位置, 相当于“后移”一个位置; 记最后一个大于  $a[i]$  的元素的位置为  $pos$ ;

② 将  $a[i]$  放置在  $b[pos]$  位置上。

如图 9.2 所示，待插入的元素为  $a[5]=13$ 。插入  $a[5]$  的第①个步骤是从  $b[4]$  开始往前查看每个元素，如果比  $a[5]$  大，则将其复制到后一个位置，如图(b)所示。最后一个大于  $a[5]$  的元素是  $a[0]$ ，即  $pos=0$ 。第②个步骤是将  $a[5]$  替换  $a[pos]$ ，如图(c)所示。

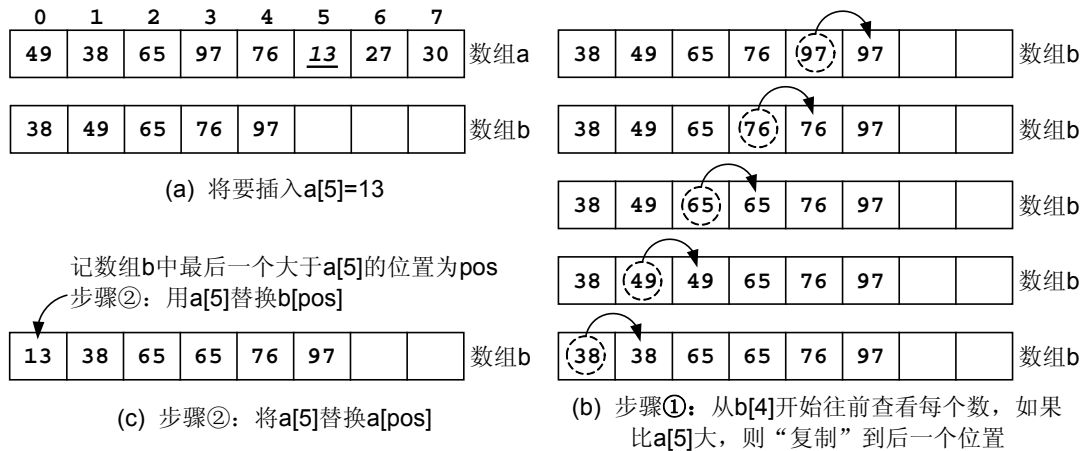


图9.2 插入排序法（演示二）

具体实现时，上述代码中的二重 for 循环需修改为：

```
for( i=0; i<10; i++ ) //在数组 b 中，b[0]~b[i-1]已经有序了，给 a[i]找到合适位置
{
    //从 b[i-1]开始查看每一个元素,如果比 a[i]大，则将其复制到后一个位置
    for( j=i-1; j>=0&& b[j]>a[i]; j-- )
    {
        b[j+1] = b[j];
    }
    b[j+1] = a[i]; //上面 for 循环结束时 j+1 的值就是 pos
}
```

### 9.1.2 冒泡法排序

“冒泡”这个词很形象地描述“冒泡法”排序的思想。在烧开水时，由于水泡中含有空气，比重比较小，所以水泡会冒上来。在“冒泡法”排序过程中，较大的数逐一“沉”到底部，而较小的数也相对“浮上来”。

假设按从小到大的顺序排序，需要排序的  $n$  个数已经存放在数组  $a$  中。冒泡法的基本思路是：

- 1) 比较第 0 个数与第 1 个数，若为逆序，即  $a[0]>a[1]$ ，则交换；然后比较第 1 个数与第 2 个数；依次类推，直至第  $n-2$  个数和第  $n-1$  个数比较为止 — 第 0 趟冒泡排序结束。第 0 趟排序的结果是最大的数被安置在最后一个元素位置上。(排序的趟数从第 0 趟开始计数主要是为了与“数组元素的下标是从 0 开始计数”一致)
- 2) 对前  $n-1$  个数进行第 1 趟冒泡排序，结果使次大的数被安置在第  $n-1$  个元素位置。
- 3) 如此重复上述过程，共经过  $n-1$  趟冒泡排序后，排序结束。

例如，假设对 9, 8, 5, 4, 2, 0 这 6 个数按从小到大的顺序排序。如图 9.3 所示。图(a)演示了第 0 趟的比较及交换过程。第 0 趟完毕，最大的数，9，已经位于最后，在下一趟，这个数不需要参与比较。图(b)演示了剩下的 5 个未排序的数所进行的第 1 趟比较及交换过程。第 1 趟完毕，此时最大的数，8，已经位于最后，在下一趟，这个数也不需要参与比较。如此共进行 5 趟比较后，6 个数已经按照从小到大的顺序排列了，如图(f)所示。

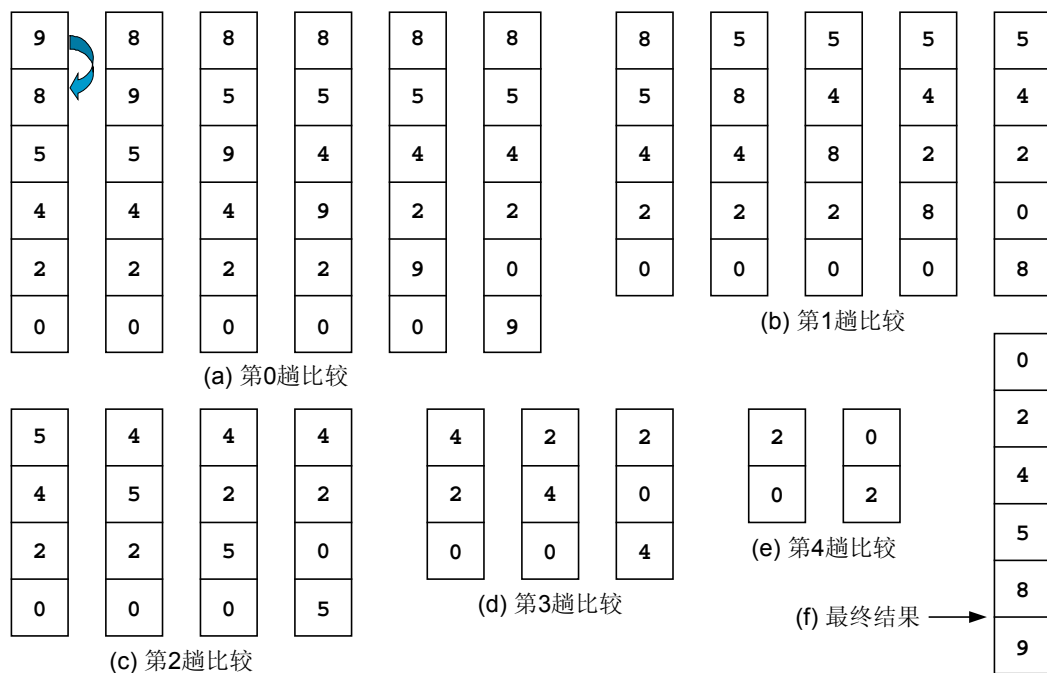


图9.3 冒泡法排序（演示一）

冒泡排序法需要用二个二重循环来实现，可以带着以下3个问题来理解其思想(有  $n$  个数，要求按照从小到大的顺序排序)：

- 1) 要进行多少趟比较？—— 要进行  $n-1$  趟比较；因此外循环循环变量  $j$  的取值是从 0 到  $n-1$  (不取等号！)。
- 2) 第  $j$  趟要比较多少次？—— 第  $j$  趟要比较  $n-1-j$  次， $j=0, 1, 2, \dots, n-2$ ；因此内循环循环变量  $i$  的取值是从 0 到  $n-1-j$  (不取等号！)。例如，第 0 趟需要进行  $n-1$  次比较。
- 3) 怎么比较？—— 比较的是相邻两个数： $a[i]$  和  $a[i+1]$ ，如果前一个数比后一个数大，马上交换。

代码如下：

```
for(j=0; j < n-1; j++) //共进行 n-1 趟比较
{
    for(i=0; i < n-1-j; i++) //在第 j 趟中要进行 n-1-j 次两两比较
    {
        if(a[i]>a[i+1]) //比较的是前后两个相邻的数 a[i] 和 a[i+1]，如逆序，则交换
        { t = a[i]; a[i] = a[i+1]; a[i+1] = t; }
    }
}
```

**例 9.2** 用冒泡法实现对输入的 10 个数按从小到大进行排序。

代码如下：

```
#include <stdio.h>
void main( )
{
    int a[10];
    int i, j; //循环变量
    int t; //用来实现交换两个数的中间变量
    printf("input 10 numbers : \n");
    for ( i=0; i<10; i++) //输入
```

```

        scanf( "%d", &a[i] );
    printf( "\n" );
    for( j=0; j<9; j++ ) //共进行 9(即 10-1)趟比较
    {
        for( i=0; i<9-j; i++ )    //在第 j 趟中要进行(9-j, 即 10-1-j)次两两比较
        {
            if( a[i]>a[i+1] )    //比较的是前后两个数 a[i]和 a[i+1]
            { t=a[i]; a[i]=a[i+1]; a[i+1]=t; }    //如果逆序, 则交换
        }
    }
    printf( "the sorted numbers : \n" );
    for( i=0; i<10; i++ )    //输出
        printf( "%d ", a[i] );
    printf( "\n" );
}

```

该程序的执行过程如下:

input 10 numbers:

7 12 -49 38 65 97 -76 13 27 30 ✓

the sorted numbers:

-76 -49 7 12 13 27 30 38 65 97

对冒泡排序法, 还有一个问题要注意:  $n$  个数的排序, 是不是一定要比较  $n-1$  趟? 答案是不一定要进行  $n-1$  趟比较。

例如对 48, 38, 65, 97, 76, 13, 27, 30 这 8 个数的排序过程如图 9.4 所示。

在图 9.4 中,  $n = 8$ , 本来需要进行  $n - 1 = 7$  趟比较和交换。但实际上在进行完前面 5 趟(第 0 趟~第 4 趟)比较和交换后, 第 5 趟比较过程中没有数据交换发生, 如图(f), 则后面的比较和交换没有必要再进行下去了。具体来说, 如果在某一趟比较过程中, 没有发现前一个数比后一个数大的情况, 即没有进行数据的交换, 那么后面的每一趟比较就不需要再进行了。

上述情形的极端是, 如果  $n$  个数已经是按从小到大的顺序排好了, 那么实际上只需要进行一趟比较(即第 0 趟), 就因为这一趟比较没有数据交换发生, 从而可以得出结论: 这  $n$  个数已经有序了。

上述思想的实现方法是:

- 1) 通过设置状态变量 **bexchange**, 为 bool 型。**bexchange** 的含义是: 当上一趟有数据交换时, 它的值为 **true**, 否则为 **false**; **bexchange** 的初始值为 **true**。
- 2) 在每趟比较之前, 即在例 9.2 的代码中, 二重 for 循环的外循环循环体最前面, 将这个状态变量置为 **false**; 然后在执行循环体过程当中如果有数据交换, 则将 **bexchange** 置为 **true**;
- 3) 比较完一趟后, 下一趟比较是否需要进行还要判断 **bchange** 是否为 **true**, 也就是要将 **bexchange** 作为循环条件之一。如果 **bexchange** 的值为 **false**, 循环条件不成立, 则表示上一趟没有进行数据交换, 后续的比较就不需再进行下去, 可以提前退出循环了; 如果 **bexchange** 的值为 **true**, 则下一趟比较还需进行。



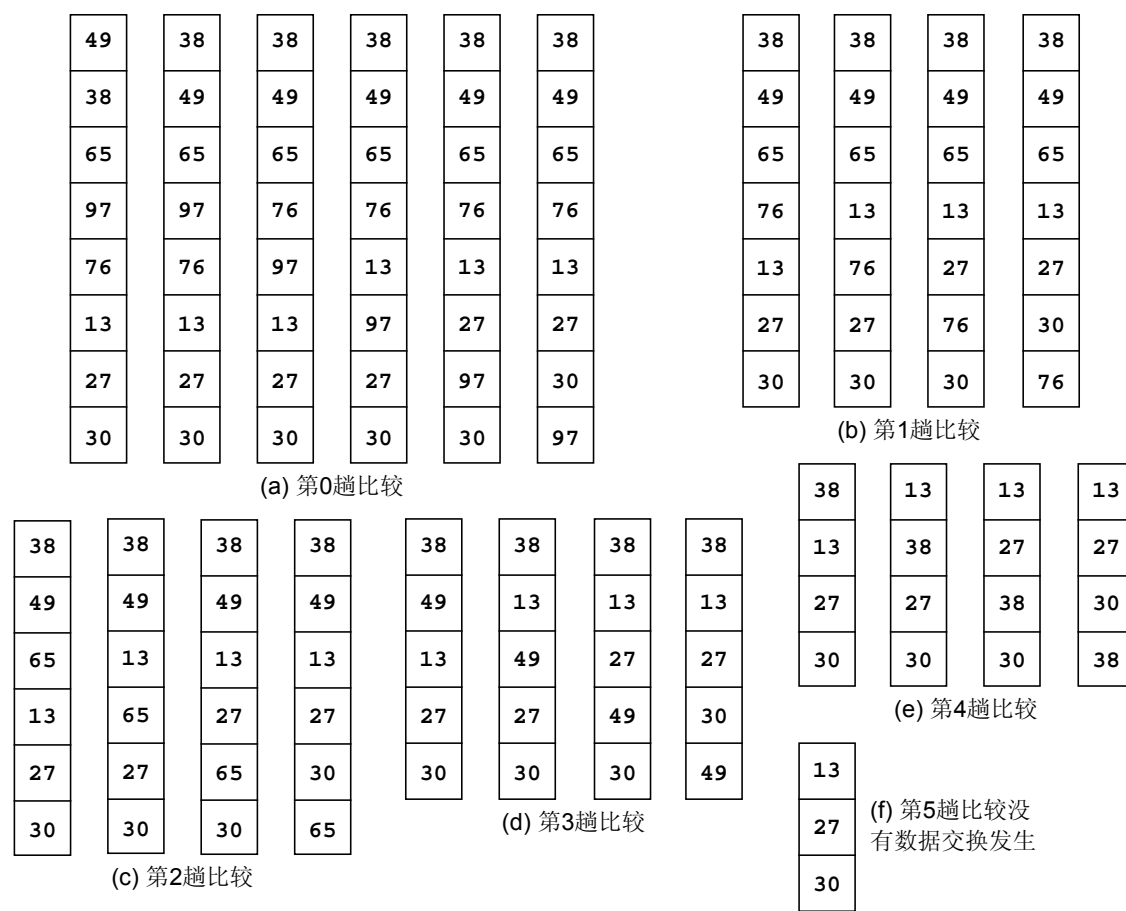


图9.4 冒泡法排序（演示二）

代码如下：

```
#include <stdio.h>
void main( )
{
    int a[10];
    int i, j;    //循环变量
    int t; //用来实现交换两个数的中间变量
    printf( "input 10 numbers : \n" );
    for ( i=0; i<10; i++ )    //输入
        scanf( "%d", &a[i] );
    printf( "\n" );
    bool bexchange = true;    //状态变量
    for( j=0; j<9 && bexchange; j++ )    //共进行 9(即 10-1)趟比较
    {
        bexchange = false;
        for( i=0; i<9-j; i++ )    //在每趟中要进行(9-j, 即 10-1-j)次两两比较
        {
            if( a[i]>a[i+1] ) //如果逆序，则交换，并且设置状态变量的值
            {
                bexchange = true;
                t=a[i]; a[i]=a[i+1]; a[i+1]=t;
            }
        }
    }
}
```

```
    }  
}  
printf( "the sorted numbers : \n" );  
for( i=0; i<10; i++ )    //输出  
    printf( "%d ", a[i] );  
printf( "\n" );  
}
```

**思考 9.1:** 例 9.2 的程序是依次将“最大”的数“沉”到最后面，如果要依次将“最小”的数“浮”上来，如图 9.5 所示，该如何实现？

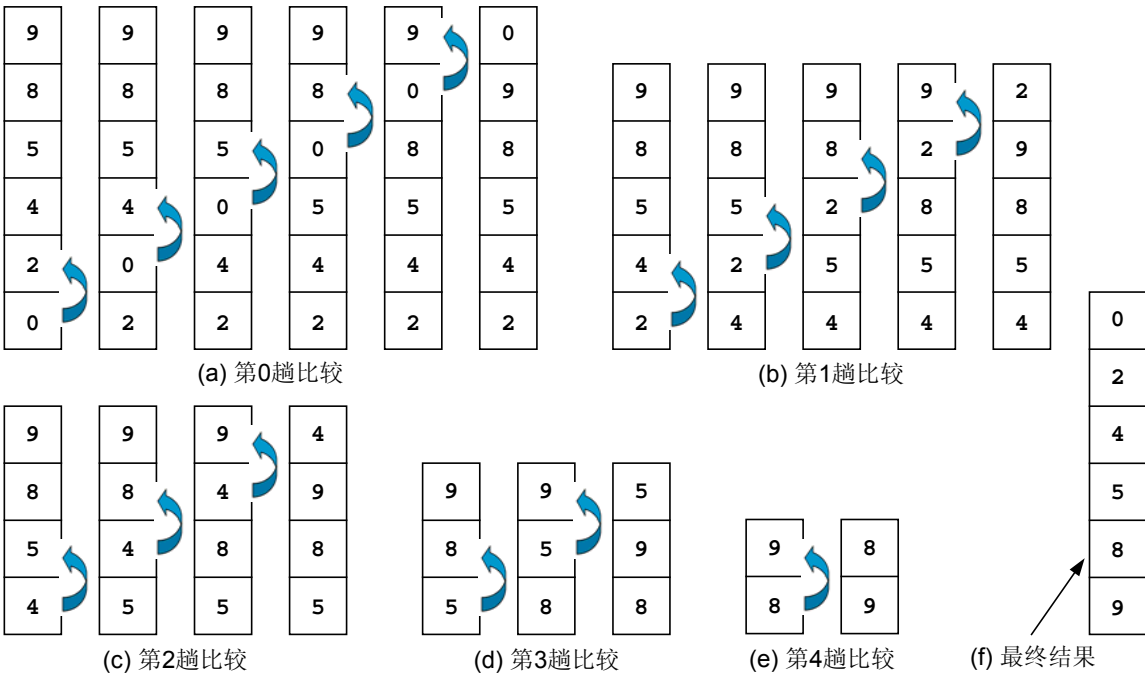


图9.5 冒泡法排序（演示三）

### 9.1.3 简单选择法排序

假设按从小到大的顺序排序，需要排序的  $n$  个数已经存放在数组  $a$  中。简单选择法的基本思路是：

- 1) 第 0 趟：通过  $n-1$  次比较，从  $n$  个数中找出最小的数，将它与第 0 个数交换。第 0 趟选择排序完毕，使得最小的数被安置在第 0 个元素位置上。
- 2) 第 1 趟：通过  $n-2$  次比较，从剩余的  $n-1$  个数中找出次小的数，将它与第 1 个数交换。第 1 趟选择排序完毕，使得次小的数被安置在第 1 个元素位置上。
- 3) 如此重复上述过程，共经过  $n-1$  趟选择交换后，排序结束。

例如，假设对 49, 38, 65, 97, 76, 13, 27, 30 这 8 个数按从小到大的顺序排序，如图 9.6 所示。这里有 3 个变量  $i$ 、 $j$ 、 $k$ ，它们的含义如下：

- $i$ ：用来表示第  $i$  趟选择的循环变量， $i = 0, 1, 2, \dots, n-2$ 。
- $k$ ：用来指向第  $i$  趟中最小的数， $k$  的值为该数的下标。第  $i$  趟中， $k$  的初始值为  $i$ 。
- $j$ ：在第  $i$  趟选择过程中，变量  $j$  用来指向  $a[i+1] \sim a[n-1]$  之间的每个数（ $j$  的值为每个数的下标），跟第  $k$  个数进行比较，以找出当前最小的数。

如图 9.6 所示。第 0 趟，首先在  $a[0] \sim a[7]$  中选择最小的数。选择方法是：假设当前最小的数就是  $a[0]$ ，其下标  $k = 0$ ；然后将  $a[1] \sim a[7]$  之间的每个数  $a[j]$  都与  $a[k]$  进行比较，如果  $a[j]$  比  $a[k]$  还小，则将  $k$  更新为  $j$ 。选择完毕，当前最小的数的下标  $k = 5$ ；然后将  $a[k]$  与  $a[0]$  交换(在

图 9.6 中, 分别用虚线圆圈标明了这两个元素)。交换完毕, 最小的数被安置在第 0 个元素位置上, 在下一趟, 这个数不需要参与选择。

第 1 趟, 在  $a[1] \sim a[7]$  中选择最小的数  $a[6]$  与  $a[i]$  即  $a[1]$  交换(同样, 在图 9.6 中, 分别用虚线圆圈标明了这两个元素)。交换完毕, 次小的数被安置在第 1 个元素位置上, 在下一趟, 这个数也不需要参与选择。

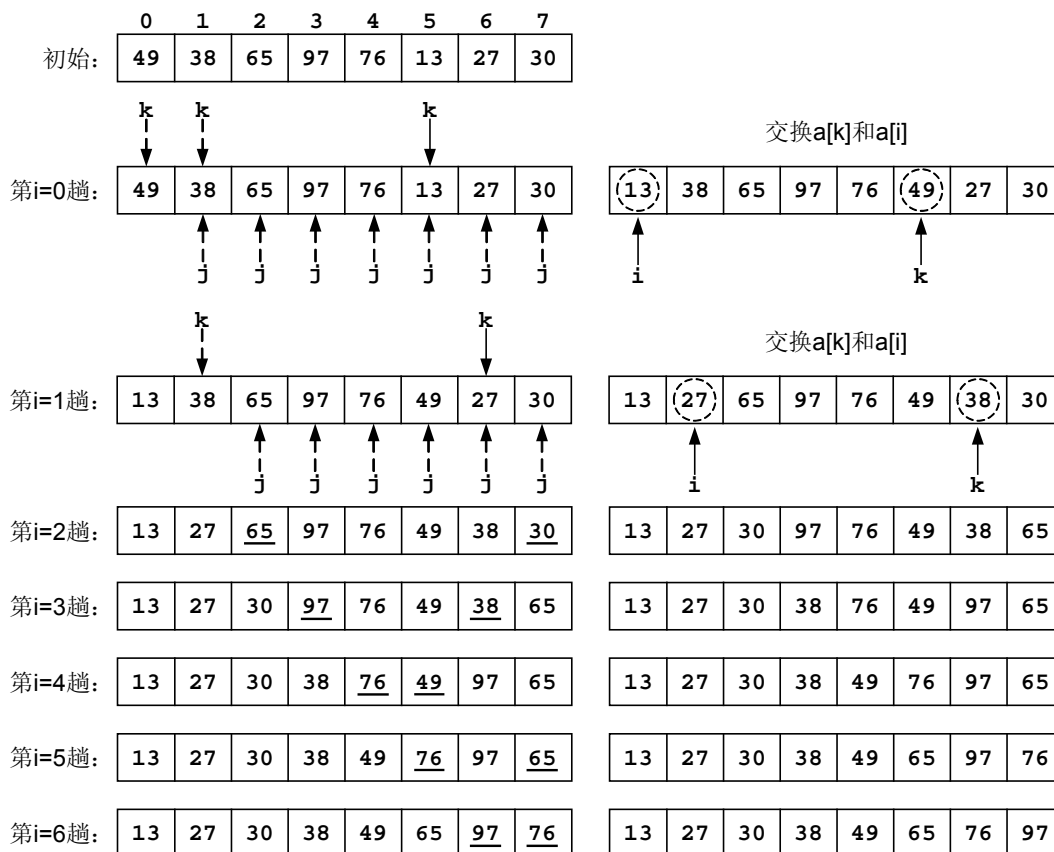


图9.6 简单选择法排序 (演示)

第 2~6 趟选择及交换过程如图 9.6 所示, 其中被交换数据的两个元素分别用下划线标明。如此共进行 7 趟选择与交换后, 这 8 个数就已经按照从小到大的顺序排好了。

简单选择排序法也需要用一个二重循环来实现, 同样可以带着以下 3 个类似问题来理解其思想(有  $n$  个数, 要求按照从小到大的顺序排序):

- 1) 要进行多少趟选择? —— 要进行  $n-1$  趟选择(第 0 趟, 第 1 趟, ..., 第  $n-2$  趟)。因此外循环的循环变量  $i$  的取值是从 0 到  $n-1$  (不取等号! )。
- 2) 在第  $i$  趟里怎么选? —— 第  $i$  趟要从  $a[i], a[i+1], \dots, a[n-1]$  中选择最小的数, 记为  $a[k]$ 。 $i=0, 1, 2, \dots, n-2$ 。首先假设  $a[i]$  就是最小的, 然后对  $a[i+1], \dots, a[n-1]$  每个数都判断一下是否比当前的最小值还要小。因此内循环的循环变量  $j$  的取值是从  $i+1$  到  $n-1$ 。
- 3) 在第  $i$  趟里怎么交换? —— 每趟选择最终交换的是  $a[i]$  和  $a[k]$ 。

代码如下:

```
int i, j, k, t; //t 是用来交换  $a[k]$  和  $a[i]$  的临时变量
for( i=0; i<n-1; i++) //共进行  $n-1$  趟选择及交换
{
    k = i; //第 i 趟中最小的数初始为  $a[i]$ 
    for( j=i+1; j<n; j++) //将  $a[i+1] \sim a[n-1]$  之间的每个数与  $a[k]$  比较
    {
        if(  $a[j] < a[k]$  ) k = j;
```

```

    }
    t = a[i]; a[i] = a[k]; a[k] = t; //交换 a[k]与 a[i]
}

```

**例 9.3** 用简单选择法实现输入的 10 个数按从小到大的顺序排序。

代码如下：

```

#include <stdio.h>
void main( )
{
    int a[10]; //存储 10 个数据的数组
    int i; //用来指示第 i 趟选择的循环变量
    int k; //用来指向第 i 趟中最小的数。第 i 趟中，k 的初始值为 i。
    int j; //在第 i 趟选择过程中，变量 j 用来指示 a[i+1]~a[n-1]之间的每个数
    int t; //用来实现交换 a[k]和 a[i]的中间变量
    printf( "input 10 numbers : \n" );
    for ( i=0; i<10; i++ ) //输入
        scanf( "%d", &a[i] );
    printf( "\n" );
    for( i=0; i<9; i++ ) //共进行 n-1 趟选择及交换
    {
        k=i; //第 i 趟中最小的数初始为 a[i]
        for( j=i+1; j<10; j++ ) //将 a[i+1]~a[n-1]之间的每个数与 a[k]比较
        {
            if( a[j]<a[k] ) k=j;
        }
        t=a[k]; a[k]=a[i]; a[i]=t; //交换 a[k]和 a[i]
    }
    printf( "the sorted numbers : \n" );
    for( i=0; i<10; i++ ) //输出
        printf( "%d ", a[i] );
    printf( "\n" );
}

```

该程序的执行过程如下：

input 10 numbers:  
7 12 -49 38 65 97 -76 13 27 30 ✓

the sorted numbers:  
-76 -49 7 12 13 27 30 38 65 97

以上程序有一个地方可以作一点改进：在第 i 趟选择完毕，如果当前最小的数就是 a[i]，即 k = i，则不需要交换 a[k]和 a[i]。即，在上述程序中，交换 a[k]和 a[i]的 3 条语句前可以加上一个条件判断。代码如下：

```

if( k!=i )
{ t=a[k]; a[k]=a[i]; a[i]=t; } //交换 a[k]和 a[i]

```

## 练习

9.1 将例 9.1、9.2 和 9.3 的程序改写成用函数实现，定义一个函数 sort，实现插入法、冒泡法或简单选择法排序。在主函数中输入 10 个数，然后调用 sort 函数实现排序，最终在 main 函数中输出排序后的 10 个数。sort 函数的原型为：

```
void sort( int a[ ], int n );
```

- 9.2 有一个已经排好序的数组(假设是从小到大的顺序), 现在要往数组中插入一个数, 要求按原来排序的规律将这个数放置在合适的位置。

提示: 假设原数组中有 10 个数, 现在要插入一个数, 则数组的长度至少为 11。

- 9.3 编写一个函数 **sort**, 实现将一个字符串中的字符按 ASCII 编码从小到大重新排序, 排序方法可采用冒泡法或简单选择法。如原串为 **viSuaL**, 排序后为 **LSaiuv**。要求在主函数中任意输入一个字符串(字符串长可任意, 只要不超过 80 个字符), 调用 **sort** 函数, 并输出排序结果。**sort** 函数的原型为:

```
void sort( char *p, int n );
```

程序运行示例如下:

enter the originl string:

3kpA!.?a5✓ (✓表示输入)

the sorted string:

!.35?Aakp

- 9.4 编写一个函数 **sort**, 实现将一个字符串中偶数位置上的字符(字符位置从 0 开始计起)按 ASCII 编码从小到大重新排序, 排序方法可采用冒泡法或简单选择法。要求在主函数中任意输入一个字符串(字符串长可任意, 只要不超过 80 个字符), 调用 **sort** 函数, 并输出排序结果。**sort** 函数的原型为:

```
void sort( char *p, int n );
```

程序运行示例如下:

enter the originl string:

TaiwanbelongstoChina✓ (✓表示输入)

the sorted string:

TaawbnheiolgntnCoisa

- 9.5 编写程序, 实现将 5 个字符串按 ASCII 编码顺序从小到大排列, 排序方法可采用冒泡法或简单选择法。(提示: 5 个字符串可以保存在一个二维字符数组中, 每一行存放一个字符串; 字符串的比较和交换分别用 **strcmp**、**strcpy** 函数实现)

程序运行示例如下:

enter 5 strings:

China✓

GERMANY✓

French✓

England✓

ENGLAND✓

the sorted strings:

China

ENGLAND

England

French

GERMANY

## 9.2 qsort 函数及其使用

9.1 节介绍了 3 种简单的排序算法, 但在实际应用中更多的是采用 **qsort** 函数进行排序。**qsort** 函数是 C/C++ 语言的库函数(包含在头文件 **stdlib.h** 中), 采用快速排序算法实现, 其效率

比 9.1 节介绍的 3 种排序算法的效率都要高得多。

### 9.2.1 qsort 函数的用法

qsort 函数是在<stdlib.h>头文件中声明的, 因此使用 qsort 函数必须包含这个头文件。qsort 函数的原型为:

```
void qsort( void *base, int num, int width,
            int ( *compare )(const void *elem1, const void *elem2 ) );
```

它有四个参数, 其含义为:

**base:** 参与排序的元素存储空间的首地址, 它是空类型指针;

**num:** 参与排序的元素个数;

**width:** 参与排序的每个元素所占字节数(宽度);

第四个参数为一个函数指针(关于函数指针, 请参考文献[1]、[3]), 这个函数需要用户自己定义, 用来实现排序时对元素之间的大小关系进行比较。compare 函数的两个参数都是空类型指针, 在实现时必须强制转换成参与排序的元素类型的指针。

如果是按从小到大的顺序(即升序)排序, compare 函数的返回值的含义为:

当第 1 个参数所指向的元素小于第 2 个参数所指向的元素, 返回值 < 0;

当第 1 个参数所指向的元素等于第 2 个参数所指向的元素, 返回值 = 0;

当第 1 个参数所指向的元素大于第 2 个参数所指向的元素, 返回值 > 0。

如果需要按从大到小的顺序(降序)排序, compare 函数的返回值具有相反的含义: 当第 1 个元素大于第 2 个元素, 则返回值 < 0; 当第 1 个元素小于第 2 个元素, 则返回值 > 0。

以下分别介绍对不同数据类型、不同排序要求时 qsort 函数的使用方法。

#### 1. 对基本数据类型的数组排序

如果数组元素是 int 型, 且按从小到大的顺序(升序)排序, compare 函数可以编写成:

```
int compare( const void *elem1 , const void *elem2 )
{
    return *(int *)elem1 - *(int *)elem2;
}
```

这样如果在 qsort 函数实现排序的过程中调用 compare 函数比较 67 和 89 这两个元素, compare 函数的返回值为-22, 即 < 0。

如果需要按从大到小的顺序(降序)排序, 只需把 compare 函数中的语句改写成:

```
return *(int *)elem2 - *(int *)elem1;
```

即可。这样 compare 函数比较 67 和 89 这两个元素, compare 函数的返回值为 22, 即 > 0。

另外, compare 函数也可以编写成(按从小到大顺序排序):

```
int compare( const void *elem1 , const void *elem2 )
{
    return ( *(int *)elem1 < *(int *)elem2 ) ? -1 :
           ( *(int *)elem1 > *(int *)elem2 ) ? 1 : 0;
}
```

compare 函数定义好以后, 就可以用下面的代码段实现一个整型数组的排序:

```
int num[100];
... //输入 100 个数组元素的值
qsort( num, 100, sizeof(num[0]), compare ); //调用 qsort 函数进行排序
```

对 char、double 等其他基本数据类型数组的排序, 只需把上述 compare 函数代码中的 int 型指针(int \*)改成其他类型指针即可。

#### 2. 对结构体一级排序

本教材没有介绍结构体，这里简单提一下。所谓结构体，就是把不同类型的数据组合成一个整体，比如一个学生的数据包括：姓名，年龄，分数。则可按如下方式声明一个 **student** 结构体：

```
struct student
{
    char name[20];    //姓名
    int age;          //年龄
    double score;     //分数
};
```

声明好 **student** 结构体以后，就可以像用 **int**、**char** 等基本数据类型一样去定义变量、数组了，如下面的例子：

```
student s1;          //定义结构体变量
student s[10];       //定义结构体数组
student ps = &s1;    //定义结构体指针，指向 s1
```

其中 **s1** 为 **student** 型变量，它包含 3 个成员：**name**、**age** 和 **score**；**s** 为 **student** 类型的数组，它有 10 个元素，每个元素都包含 3 个成员；**ps** 为 **student** 类型的指针变量，它指向 **s1**。

要引用结构体变量中的成员，需要使用**成员运算符“.”**，如下面的例子：

```
s1.age = 20; //给 s1 的 age 成员赋值为 20
strcpy( s1.name, "Wang Lin" ); //将字符串"Wang Lin"拷贝到 s1 的 name 成员
```

如果是通过结构体指针变量引用它所指向的结构体变量的成员，需要使用**指向运算符“->”**，如下面的例子：

```
ps->age = 20; //相当于：s1.age = 20;
```

所谓**对结构体一级排序**，是指对结构体中的某一个成员的大小关系排序。例如对上述的 **student** 数组 **s** 中的元素以其 **age** 成员的大小关系按从小到大的顺序(升序)排序。**compare** 函数可定义成：

```
int compare( const void *elem1 , const void *elem2 )
{
    return ((student *)elem1)->age - ((student *)elem2)->age;
}
```

**qsort** 函数调用形式为：

```
qsort( s, 10, sizeof(s[0]), compare );
```

### 3. 对结构体二级排序

所谓**对结构体二级排序**，含义是先按某个成员的大小关系排序，如果该成员大小相等，再按另一个成员的大小关系进行排序。比如上面的 **student** 数组 **s**，可以先按 **age** 成员从小到大的顺序排序，如果 **age** 成员大小相等，再按 **score** 成员从小到大的顺序排序。

**compare** 函数定义如下：

```
int compare( const void *elem1 , const void *elem2 )
{
    student *p1 = (student *)elem1;
    student *p2 = (student *)elem2;
    if( p1->age != p2->age ) return p1->age - p2->age;
    else return p1->score - p2->score;
}
```

也就是说，如果两个元素 **s1** 和 **s2** 的 **age** 成员不等，**compare** 返回的是它们 **age** 成员的大小关系；如果它们的 **age** 成员大小相等，返回的是它们的 **score** 成员的大小关系。

**qsort** 函数调用形式为：

```
qsort( s, 10, sizeof(s[0]), compare );
```

## 9.2.2 qsort 函数应用例子

### 例 9.4 快乐的蠕虫(The Happy Worm)

题目来源:

Asia 2004, Tehran (Iran), Sharif Preliminary

题目描述:

有一只快乐的蠕虫居住在一个  $m \times n$  大小的网格中。在网格的某些位置放置了  $k$  块石头。网格中的每个位置要么是空的，要么放置了一块石头。当蠕虫睡觉时，它在水平方向或垂直方向上躺着，把身体尽可能伸展开来。蠕虫的身躯既不能进入到放有石块的方格中，也不能伸出网格外。而且蠕虫的长度不会短于 2 个方格的大小。

本题的任务是给定网格，要计算蠕虫可以在多少个不同的位置躺下睡觉。

输入描述:

输入文件的第 1 行为一个整数  $t$ ， $1 \leq t \leq 11$ ，表示测试数据的个数。每个测试数据的第 1 行为 3 个整数： $m$ ， $n$  和  $k$ ， $0 \leq m, n, k \leq 200000$ ，接下来有  $k$  行，每行为两个整数，描述了一块石头的位置（行和列，最左上角位置为(1,1)）。

输出描述:

对每个测试数据，输出占一行，为一个整数，表示蠕虫可以躺着睡觉的不同位置的数目。

样例输入:

```
2
5 5 6
1 5
2 3
2 4
4 2
4 3
5 1
5 5 10
1 2
1 5
2 4
2 5
3 1
3 3
3 5
4 3
4 5
5 1
```

样例输出:

```
9
8
```

分析:

首先要理解题目的意思。题目中有两句话很关键：“当蠕虫睡觉时，它在水平方向或垂直方向上躺着，把身体尽可能伸展开来”，“而且蠕虫的长度不会短于 2 个方格的大小”。这两句话要结合起来理解。样例输入中第 1 个测试数据的网格如图 9.7(a)所示，“□”表示空的方格，“■”表示石头。如果只凭第 2 句话，则仅在第 1 列，蠕虫就可以在 3 个位置上躺着，分别是头在(1,1)、



(2,1)、(3,1)这3个位置躺着，身躯在垂直方向上向下伸展开来；但加上第1句话，则这3个位置都是一样的，因为蠕虫在第1列上躺着，它的身躯会尽可能伸展开来，占满第1列所有4个空格。

对图(a)所示的网格，蠕虫可以在9个位置上躺着，这9个位置分别是：第1列、第2列、第4列、第5列、第1行、第2行、第3行、第4行和第5行。如果把(4,2)这个位置上的石头去掉，则统计出的位置数是10个。因为在第4行(4,3)位置上石头的左边和右边都满足题目的要求。

本题测试数据中的3个值取值都很大， $0 \leq m, n, k \leq 200000$ ，如果要把整个网格用二维数组保存起来，内存使用量会超出题目的要求。即使能把整个网格保存起来，扫描这个网格需要用二重循环，时间也会超时。

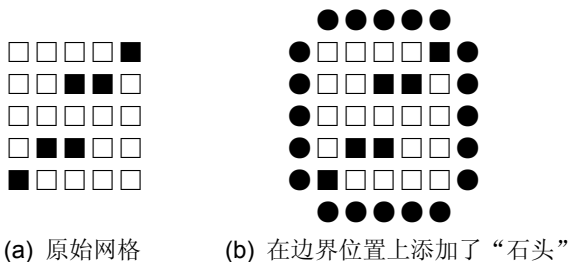


图 9.7 测试数据 1 对应的网格

本题的处理方法是，在网格的边界处“添加”一些石头，如图 9.7(b)所示，“●”表示添加的石头，只需存储输入的石头位置及添加的石头位置，然后对这些石头的位置进行如下的两种二级排序：

- 1) 先按  $x$  坐标(即行坐标)从小到大的顺序排序， $x$  坐标相同，再按  $y$  坐标(即列坐标)从小到大的顺序进行排序。排序以后，如果前后两个位置的  $x$  坐标相同(表示这两块石头在同一行)，且  $y$  坐标相差大于 2，则是蠕虫能躺着睡觉的位置。这种情形对应到蠕虫躺在水平方向上。
- 2) 先按  $y$  坐标从小到大的顺序排序， $y$  坐标相同，再按  $x$  坐标从小到大的顺序进行排序。排序以后，如果前后两个位置的  $y$  坐标相同(表示这两块石头在同一列)，且  $x$  坐标相差大于 2，则也是蠕虫能躺着睡觉的位置。这种情形对应到蠕虫躺在垂直方向上。

例如，网格中原有的石头，再加上“添加”的石头，一共 26 个。按第一种方式排序后为：(0,1)、(0,2)、(0,3)、(0,4)、(0,5)、(1,0)、(1,5)、(1,6)、(2,0)、(2,3)、(2,4)、(2,6)、(3,0)、(3,6)、(4,0)、(4,2)、(4,3)、(4,6)、(5,0)、(5,1)、(5,6)、(6,1)、(6,2)、(6,3)、(6,4)、(6,5)。扫描这 26 个位置，如果前后两个位置的  $x$  坐标相同， $y$  坐标相差  $>2$ ，就是蠕虫可以躺着睡觉的位置。例如(1,0)和(1,5)满足要求，对应到网格中第 1 行。

代码如下：

```
#include <stdio.h>
#include <stdlib.h>
struct ln
{
    int x; int y;
}s[1000000]; //存储石头的位置，(包括"添加"的石头)
int cmpx( const void *a , const void *b ) //二级排序：先比较 x，再比较 y
{
    struct ln *c = (ln *)a;
    struct ln *d = (ln *)b;
    if( c->x != d->x ) return c->x - d->x;
    return c->y - d->y;
```

```

}
int cmpy( const void *a , const void *b ) //二级排序：先比较 y，再比较 x
{
    struct ln *c = (ln *)a;
    struct ln *d = (ln *)b;
    if(c->y != d->y) return c->y - d->y;
    return c->x - d->x;
}
int main( )
{
    int kase; //输入文件中测试数据个数
    scanf( "%d", &kase );
    while( kase-- )
    {
        int m, n, k; //每个测试数据中的 3 个数
        int i, j; //循环变量
        scanf( "%d%d%d", &m, &n, &k );
        for( i=0; i<k; i++ ) //读入 k 块石头的位置
            scanf( "%d%d", &s[i].x, &s[i].y );
        for( j=1; j<=n; j++ ) //添加"垂直方向上边界的石头"
        {
            s[i].x = 0;    s[i].y = j;
            i++;
            s[i].x = m+1;  s[i].y = j;
            i++;
        }
        for( j=1; j<=m; j++ ) //添加"水平方向上边界的石头"
        {
            s[i].y = 0;    s[i].x = j;
            i++;
            s[i].y = n+1;  s[i].x = j;
            i++;
        }
        int t = 0; //蠕虫可以躺着睡觉的不同位置的数目
        qsort( s, i, sizeof(s[0]), cmpx );
        for( j=0; j<i-1; j++ )
        {
            //如果前后两个位置的 x 坐标相同，y 坐标相差超过 2
            if( s[j].x==s[j+1].x && s[j+1].y - s[j].y > 2 )
                t++;
        }
        qsort( s, i, sizeof(s[0]), cmpy );
        for( j=0; j<i-1; j++ )
        {
            //如果前后两个位置的 y 坐标相同，x 坐标相差超过 2
            if( s[j].y==s[j+1].y && s[j+1].x - s[j].x > 2 )
                t++;
        }
        printf( "%d\n", t );
    }
}

```

```

    return 0;
}

```

## 练习

### 9.6 单词重组(Word Amalgamation)

#### 题目描述:

在美国数以百万计的报纸中,有一种单词游戏称为猜词。游戏的的目标是猜谜,为了找出答案中缺少的字母,有必要对 4 个单词的字母顺序重新调整。在本题,你的任务是编写程序实现对单词中的字母顺序重新调整。

#### 输入描述:

输入文件包含 4 部分:

1. 一部字典, 包含至少 1 个单词, 至多 100 个单词, 每个单词占一行;
2. 字典后是一行字符串 “XXXXXX”, 表示字典结束;
3. 一个或多个被打乱字母顺序的“单词”, 每个单词占一行, 你必须整理这些字母的顺序;
4. 输入文件的最后一行为字符串 “XXXXXX”, 代表输入文件结束。

所有单词, 包括字典中的单词和被打乱字母顺序的单词, 都只包含小写英文字母, 并且至少包含一个字母, 至多包含 6 个字母。字典中的单词不一定是按顺序排列的, 但保证字典中的单词都是唯一的。

#### 输出描述:

对输入文件中每个被打乱字母顺序的单词 **w**, 按字母顺序输出字典中所有满足以下条件的单词的列表: 通过调整单词 **w** 中的字母顺序, 可以变成字典中的单词。列表中的每个单词占一行。如果列表为空(即单词 **w** 不能转换成字典中的任何一个单词), 则输出一行字符串 “NOT A VALID WORD”。以上两种情形都在列表后, 输出一行包含 6 个星号字符的字符串, 表示列表结束。

#### 样例输入:

```

tarp
given
score
refund
only
trap
work
earn
course
pepper
part
XXXXXX
aptr
sett
oresuc
XXXXXX

```

#### 样例输出:

```

part
tarp
trap
*****
NOT A VALID WORD
*****
course
*****

```

### 9.7 英文姓名排序

#### 题目描述:

在汉语里，对汉语姓名可以按拼音排序，也可以按笔画顺序排序。在英语里，对英语姓名主要按字母顺序排序。本题要求对给定的一组英文姓名按长短顺序排序。

#### 输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为一个正整数  $N(0 < N < 100)$ ，表示该测试数据中英文姓名的数目；接下来有  $N$  行，每行为一个英文姓名，姓名中允许出现的字符有大小写英文字母、空格、点号(.)，每个英文姓名长度至少为 2、但不超过 50。 $N = 0$  表示输入结束。

#### 输出描述：

对输入文件中的每个测试数据，输出排序后的姓名。排序方法为：先按姓名长短按从长到短的顺序排序，对长度相同的姓名，则按字母顺序排序。每两个测试数据的输出之间输出一个空行。

#### 样例输入：

```
8
Herbert Schildt
David A. Forsyth
Jean Ponce
Gerald Recktenwald
Tom M. Mitchell
Robin R. Murphy
John David Funge
Thomas H. Cormen
0
```

#### 样例输出：

```
Gerald Recktenwald
David A. Forsyth
John David Funge
Thomas H. Cormen
Herbert Schildt
Robin R. Murphy
Tom M. Mitchell
Jean Ponce
```

**提示：**声明一个结构体，包含姓名和姓名长度 2 个成员。本题要求对结构体进行二级排序。

## 9.3 竞赛题目解析

本节分别针对数值型数据、字符型数据、及混合数据的排序，分别讲解一道竞赛题目。

### 9.3.1 数值型数据的排序

#### 例 9.5 花生(The Peanuts)

##### 题目来源：

South Central USA 1995

##### 题目描述：

鲁宾逊先生和他的宠物猴，多多，非常喜欢花生。有一天，他们两个正沿着乡间小路散步，多多突然发现路边的告示牌上贴着一张小小的纸条：“欢迎免费品尝我种的花生！”。你可以想象当时鲁宾逊先生和多多是多么的高兴！

在告示牌背后，路边真的有一块花生田，花生植株整齐地排列成矩形网格（如图 9.8(a)所示）。在每个交叉点，有零颗或多颗花生。例如在图 9.8(b)中，只有 4 个交叉点上有多颗花生，分别为 15，13，9 和 7 颗，其他交叉行都只有零颗花生。多多只能从一个交叉点跳到它的四个相邻交叉点上，所花费的时间为 1 个单位时间。以下过程所花费的时间也是 1 个单位时间：从路边走到花生田，从花生田走到路边，采摘一个交叉点上的花生。

根据鲁宾逊先生的要求，多多首先走到花生数最多的植株。采摘这颗植株的花生后，然后走到下一个花生数最多的植株处，如此等等。但鲁宾逊先生并不耐烦，来等多多采摘所有的花生。而是要求多多在给定的时间内返回到路边。例如，在图 9.8(b)中，多多在 21 个单位时间内可以采摘到 37 颗花生，多多走到路线如图所示。

你的任务是，给定花生分布情况，以及给定时间限制，求多多最多能摘到的花生数。你可以假定每个交叉点的花生数不一样，当然除了花生数为 0 外。花生数为 0 的交叉点数目可以有多个。

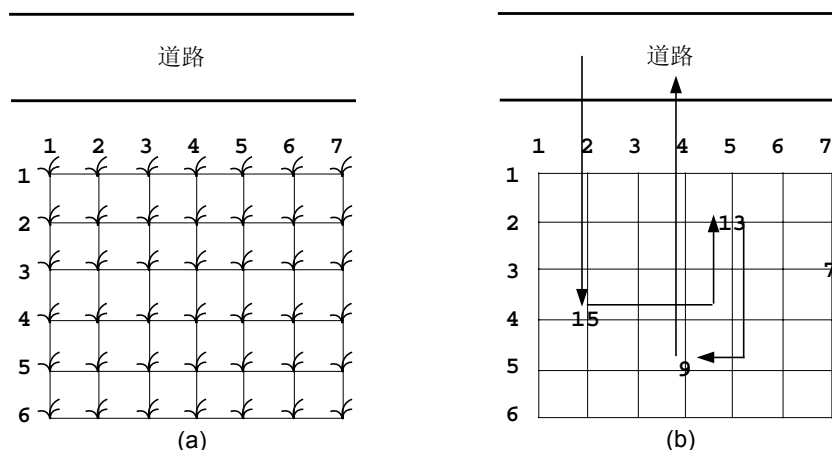


图9.8 采摘花生

### 输入描述:

输入文件的第 1 行为一个整数  $T$ ，代表测试数据的数目， $1 \leq T \leq 20$ 。对每个测试数据，第 1 行包含 3 个整数： $M$ 、 $N$  和  $K$ ， $1 \leq M, N \leq 50$ ， $0 \leq K \leq 20000$ 。接下来有  $M$  行，每行有  $N$  个整数。每个整数都不超过 3000。花生田的大小为  $M \times N$ ，第  $i$  行的第  $j$  个整数  $X$  表示在  $(i, j)$  位置上有  $X$  颗花生。 $K$  的含义是多多必须在  $K$  个单位时间内返回到路边。

### 输出描述:

对每个测试数据，输出多多在给定时间内能择到花生的最大数。

### 样例输入:

```
2
6 7 21
0 0 0 0 0 0 0
0 0 0 0 13 0 0
0 0 0 0 0 0 7
0 15 0 0 0 0 0
0 0 0 9 0 0 0
0 0 0 0 0 0 0
6 7 20
0 0 0 0 0 0 0
0 0 0 0 13 0 0
0 0 0 0 0 0 7
0 15 0 0 0 0 0
0 0 0 9 0 0 0
0 0 0 0 0 0 0
```

### 样例输出:

```
37
28
```

### 分析:

注意理解题目的意思。在图 9.9 所示的网格中，摘一株有 19 颗花生和一株有 18 颗花生的

植株所花费的时间为 7，摘一株有 20 颗花生的植株所花费的时间也为 7，如果给定时间限制为 7，那么答案到底是 37 还是 20 呢？题目中提到“根据鲁宾逊先生的要求，多多首先走到花生数最多的植株。采摘这颗植株的花生后，然后走到下一个花生数最多的植株处，如此等等。”因此正确答案是 20。

所以，本题只需把花生数按从大到小的顺序进行排序，在满足时间限制的前提下依次采摘花生即可。在以下程序中，定义了一个结构体，表示网格中的节点。它包含 3 个成员：节点的 x、y 坐标，及花生数。在排序时，对节点数组按花生数进行一级排序：按花生数从大到小排序。

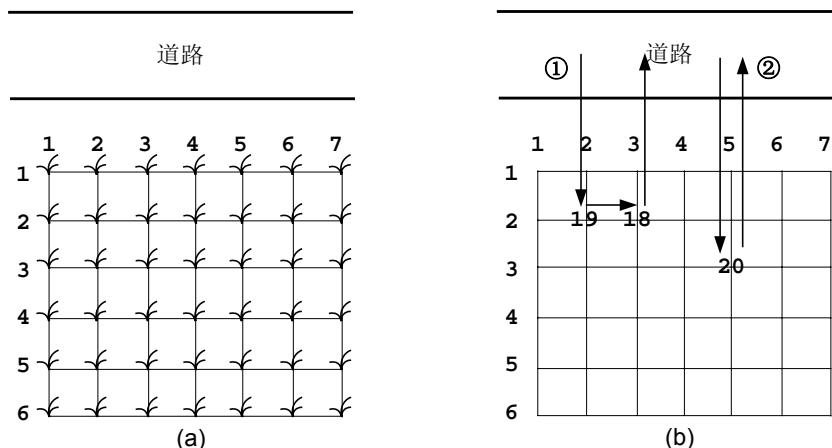


图9.9 采摘花生顺序的选择

代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
struct Node      //节点
{
    char x, y;    //位置
    short peanuts; //花生数目
};
int compare( const void * a, const void * b ) //按花生数从大到小排序的比较函数
{
    Node * m = ( Node * ) a;
    Node * n = ( Node * ) b;
    return n -> peanuts - m -> peanuts;
}
int main( )
{
    int T;    //测试数据的个数
    int m, n, time;    //每个测试数据中的数据(网格大小及规定的时间)
    int i, j, t; //循环变量
    scanf( "%d", &T );
    for( t = 0; t < T; t++ )
    {
        Node table[25000] = {0};
        scanf( "%d%d%d", &m, &n, &time );
        int count = 0; //有花生的植株数
        int p;    //读入的每棵植株下的花生数
        for( i = 1; i <= m; i++ )//读入网格，并记录有花生的节点信息
```

```

    {
        for( j = 1; j <= n; j ++ )
        {
            scanf( "%d", &p );
            if ( p )
            {
                table[count].x = i;  table[count].y = j;
                table[count].peanuts = p;
                count ++;
            }
        }
    }
    qsort ( table, count, sizeof (Node), compare );//按花生数从大到小排序
    int currX = 0, currY = table[0].y; //多多当前的位置
    int sum = 0;
    for ( i = 0; i < count; i ++ )
    {
        //从当前位置走到 table[i]节点并采摘花生所花费的时间
        int temp = abs ( currX - table[i].x ) + abs ( currY - table[i].y ) + 1;
        if ( temp + table[i].x <= time )//table[i].x 表示从 table[i]节点回到路边的时间
        {
            time -= temp; //剩余时间
            sum += table[i].peanuts;
            currX = table[i].x;
            currY = table[i].y;
        }
        else break;
    }
    printf( "%d\n", sum );
}
return 0;
}

```

### 9.3.2 字符型数据的排序

**例 9.6** Unix 操作系统的 ls 命令(Unix ls)

**题目来源:**

South Central USA 1995

**题目描述:**

你所工作的一家计算机公司准备引入一种新型的计算机,并打算开发一种类似 Unix 的操作系统。你的任务是为 ls 命令编写格式化显示程序。

你的程序最终是从管道读取输入数据,当然在本题中是从输入文件读入数据。输入文件包含  $N$  个文件名,你必须把这  $N$  个文件名按字符的 ASCII 编码值的升序排序,然后根据长度最长的文件名的长度  $L$ ,将这  $N$  个文件名输出到  $C$  列。文件名长度范围是  $1 \sim 60$ ,输出时是左对齐的。最右边一列的宽度是  $L$ ,即长度最长的文件名的长度,其他列的宽度是  $L+2$ 。你可以采用尽可能多的列,但各列宽度之和不得超过 60 个字符宽度。你的程序必须用最少行,记为  $R$  行,来输出  $N$  个文件名。

**输入描述:**

输入文件包含有限个文件名列表。每个列表的第 1 行为一个整数  $N$ ,  $1 \leq N \leq 100$ 。接下来有  $N$  行, 每一行为一个左对齐的文件名, 文件名的长度为  $1 \sim 60$ 。文件名中允许出现的字符包括数字字符和字母字符 (即 'a'~'z'、'A'~'Z' 以及 '0'~'9'), 以及 3 个字符 “.”、“\_” 和 “-”, 任何一个文件名都不包含除以上字符外的字符, 并且没有空行。

每个列表最后一个文件名之后就是下一个列表的数据, 或者如果该列表是最后一个列表, 则之后就没有其他数据了。

你必须读入输入文件中所有文件名列表并按要求的格式输出。

**输出描述:**

对每个文件名列表, 首先输出由 60 个短划线字符 “-” 组成的一行字符, 然后按格式排列的若干列文件名。按顺序, 第 1 个~第  $R$  个文件名显示在第 1 列, 第  $R+1$  个~第  $2R$  个文件名显示在第 2 列, 等等。

**样例输入:**

```
10
tiny
2short4me
very_long_file_name
shorter
size-1
size2
size3
much_longer_name
12345678.123
mid_size_name
12
Weaser
Alfalfa
Stimey
Buckwheat
Porky
Joe
Darla
Cotton
Butch
Froggy
Mrs_Crabapple
P.D.
```

**样例输出:**

```
-----
12345678.123      size-1
2short4me         size2
mid_size_name     size3
much_longer_name  tiny
shorter           very_long_file_name
-----
```

```
Alfalfa      Cotton      Joe      Porky
```



|           |        |               |        |
|-----------|--------|---------------|--------|
| Buckwheat | Darla  | Mrs_Crabapple | Stimey |
| Butch     | Froggy | P. D.         | Weaser |

分析:

本题首先要对读入的  $N$  个文件按字符的 ASCII 编码值的升序排序, 因为  $1 \leq N \leq 100$ , 数据量比较小, 所以排序可以直接用 9.1 节介绍的冒泡法或简单选择法。但是要注意, 比较两个文件名的大小只能采用 `strcmp` 函数。

本题的关键在于按照题目的格式要求输出  $N$  个文件名。有几点需特别注意:

1. 要准确地计算出输出这  $N$  个文件名所需的列数和行数。所需列数 `ncols` 就是  $62/(L+2)$ ,  $L$  为长度最长的文件名的长度。 $L+2$  是因为每列(除最后一列外)后有两个空格; 分母是 62 而不是 60, 因为最后一列后面不需要多输出两个空格, 这里为了统一考虑, 所以加上 2。如果显示出来时, 每列的文件名数一样, 即  $N$  能被 `ncols` 整数, 那么行数 `nrows` 就是  $N/\text{ncols}$ ; 如果  $N$  不能被 `ncols` 整除, 即  $(N\%ncols) > 0$ , 则行数 `nrows` 还要加 1。

2. 第 1 列~第 `ncols-1` 列, 在每个文件名后要输出多余的空格, 一直到总长为  $L+2$  为止; 而对第 `ncols` 列的文件名, 在每个文件名后不能输出多余的空格。

代码如下:

```
#include <stdio.h>
#include <string.h>
char filenames[100][61]; //存放 N 个文件名的字符数组
int N; //文件名的个数
void readfiles( void ) //读入文件列表中的 N 个文件名
{
    for( int i=0; i<N; i++ ) scanf( "%s", &filenames[i] );
}
void sortfiles( void ) //对 N 个文件名按字符的 ASCII 编码值的升序排序
{
    int i, j, k; //循环变量
    char holder[61]; //交换两个文件时用到的临时变量
    for( i=0; i<N-1; i++ ) //简单选择法排序
    {
        k = i;
        for( j=i+1; j<N; j++ )
        {
            //filenames[k]比 filenames[j]大
            if( strcmp(filenames[k], filenames[j])>0 )
                k=j;
        }
        if( k!=i )
        {
            strcpy(holder, filenames[k]);
            strcpy(filenames[k], filenames[i]);
            strcpy(filenames[i], holder);
        }
    }
}
void format_columns( void ) //按照题目要求的格式输出 N 个文件名
{
    int ncols, nrows; //显示文件名的列数和行数
```

```

int i, j;    //循环变量
int k;       //输出空格时用到的循环变量
unsigned widest = 0; //文件名的最大长度
for( i=0; i<N; i++ )
{
    if( strlen(filenamees[i]) > widest )
        widest = strlen(filenamees[i]);
}
//求输出 N 个文件名所需的列数和行数
ncols = 62/(widest+2);
nrows = N/ncols;
if( (N%ncols) > 0 )  nrows++;
printf( "-----\n" );
for( i=0; i<nrows; i++ )
{
    for( j=0; (j<ncols) && (j*nrows + i) < N; j++ )
    {
        printf( "%s", filenamees[j*nrows + i] );
        if( (j+1)*nrows+i<N ) //如果右边还有文件名，则要输出多余的空格
        {
            for( k=widest+2-strlen(filenamees[j*nrows + i]); k>0; k-- )
                printf( " " );
        }
    }
    printf( "\n" );
}
}
int main(void)
{
    while( scanf( "%d", &N )!=EOF )
    {
        readfiles( );
        sortfiles( );
        format_columns( );
    }
    return 0;
}

```

### 9.3.3 混合数据的排序

#### 例 9.7 混乱排序(Scramble Sort)

题目来源:

Greater New York 2000

题目描述:

在本题中，给定若干个包含单词和数值的列表，要求对这些列表按照如下的方式进行排序：所有的单词按照字母升序排列，所有数值按照大小升序排列；而且列表中的每个位置上的元素排序前是一个单词，则排序后还是一个单词，如果排序前是一个数值，排序后还是一个数值。并且对单词排序时对其中的字母是不区分大小写的。

**输入描述:**

输入文件中包含多个列表，每个列表占一行。列表中的每个元素用逗号“,”和空格隔开，列表以点号“.”结束。整个输入文件的最后一行为一个点号“.”，该行不需排序。

**输出描述:**

对输入文件中的每个列表，输出混乱排序后的列表，列表中的每个元素用逗号“,”和空格隔开，列表以点号“.”结束。

**样例输入:**

```
0.
banana, strawberry, OrAnGe.
Banana, StRaWbErRy, orange.
10, 8, 6, 4, 2, 0.
x, 30, -20, z, 1000, 1, Y.
50, 7, kitten, puppy, 2, orangutan, 52, -100, bird, worm, 7, beetle.
.
```

**样例输出:**

```
0.
banana, OrAnGe, strawberry.
Banana, orange, StRaWbErRy.
0, 2, 4, 6, 8, 10.
x, -20, 1, Y, 30, 1000, z.
-100, 2, beetle, bird, 7, kitten, 7, 50, orangutan, puppy, 52, worm.
```

**分析:**

本题的思路是用 **flag** 数组记录每个位置上是单词还是数值，如果第 *i* 个数据为单词，则 **flag[i]** 为 0，否则为 1。然后将读入的数值和单词分别存放到整型数组和字符串数组里，并分别对整型数组与字符串数组进行排序。

输出时，如果原先第 *i* 个位置上为数值(**flag[i]** 为 1)，则到整型数组中按顺序去找整数，并输出；如果原先第 *i* 个位置上为单词(**flag[i]** 为 0)，则到字符串数组中按顺序去找字符串，并输出。

本题有几个细节值得注意：

1. 输入时要正确地去掉单词(或数值)之间的逗号“,”，及最后一个数据之后的点号“.”。输出时要正确地加上单词(或数值)之间的逗号“,”，及最后一个数据之后的点号“.”。

2. 由于数值是混在字符型数据之间的，所以数值也只能采用字符形式读入，然后将其转换成数值，系统函数 **atoi** 可实现这个功能。下面的程序中用 **change** 函数实现该功能，思路是：  
`"6987" = (( ( 6*10 + 9 ) * 10 + 8 ) * 10 + 7 )`，并要判别是否有正号“+”或负号“-”。

**代码如下:**

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

char str[20]; //存放读入的每个单词或数值
int number[100]; //存放读入的数值
char word[100][20]; //存放读入的单词
int flag[100]; //第 i 个数据为单词，则 flag[i] 为 0，否则为 1
int ncount=0, wcount=0, count=0; //当前测试数据中数值、单词的个数及总的个数
```

```

int cmp1( const void* a, const void* b ) //数值比较大小
{
    return *(int*)a - *(int*)b;
}
//由于对单词排序时对字母不区分大小写，所以要先转换成小写字母再比较大小
int cmp2( const void* a, const void* b ) //单词比较大小
{
    char a1[20] = "", b1[20] = ""; //暂存 a 和 b 的临时变量
    int i; //循环变量
    strcpy( a1, (char*)a ); strcpy( b1, (char*)b );
    for( i = 0; a1[i]; i++ ) a1[i] = (a1[i]>=65&&a1[i]<=90)?a1[i]+32:a1[i];
    for( i = 0; b1[i]; i++ ) b1[i] = (b1[i]>=65&&b1[i]<=90)?b1[i]+32:b1[i];
    return strcmp( a1, b1 );
}
void change( char s[] )//将字符数组 s 中的数值转换成整数形式
{
    int len = strlen(s), value = 0; //字符数组的长度及转换后的数值
    int sign = 1, i = 0; //符号及循环变量
    if( s[0]=='-' ) { sign = -1; i = 1; }
    else if( s[0]=='+' ) i = 1;
    for( i<len-1; i++ )
    {
        value *= 10; value += s[i]-'0';
    }
    number[ncount] = sign*value; ncount++;
    flag[count] = 1; count++;
}
void solve( )
{
    qsort( number, ncount, sizeof(number[0]), cmp1 );//排序
    qsort( word, wcount, sizeof(word[0]), cmp2 );//排序
    int inumber = 0, iword = 0; //访问 number 数组和 word 数组的循环变量
    //输出: flag[i]为 0, 则到 word 数组中找单词输出, 否则到 number 数组中找数值输出
    for( int i=0; i<count; i++ )
    {
        if( flag[i] ) //数值
        { printf( "%d", number[inumber] ); inumber++; }
        else //单词
        { printf( "%s", word[iword] ); iword++; }
        if( i<count-1 ) printf( ", " );
        else printf( "." );
    }
    printf( "\n" );
}
int main( )
{
    while( scanf("%s",str) != EOF )
    {
        if( str[0] == '.' ) break; //整个输入结束

```

```

int len = strlen(str);
//读入的是数值，则将读入的数值转换成整数形式
if( (str[0]>='0' && str[0]<='9') || str[0]=='-' || str[0]=='+' )
    change(str);
else //读入的是单词
{
    strcpy( word[wcount], str );
    word[wcount][len-1] = 0; //去掉读入的单词后面的','或'.'
    wcount++; flag[count] = 0; count++;
}
if( str[ len-1 ] == '.' ) //当前测试数据输入结束
{
    solve( ); //排序并输出
    ncount = wcount = count = 0; continue;
}
}
return 0;
}

```

## 练习

### 9.8 古老的密码(Ancient Cipher)

#### 题目描述:

古老的罗马帝国有强大的政府体系，包含很多不同的部门，其中就有情报机构。从各省发给中央的重要文档都要经过加密，以防止泄密。那时两种最常用的加密方法是：替换加密法和置换加密法。

替换加密法是将原文中的每个字符替换成对应的其他字符。用来替换的字符必须是不同的。对某些字符来说，替换字符可能跟原始字符一致，即本身替换本身。例如一种替换加密法是将原文中所有字符('A'到'Y')替换成字母表中下一个字符，并把'Z'替换成'A'。如果原文为“VICTORIOUS”，采用此替换加密法得到的密文为“WJDUPSJPVT”。

置换加密法又称换位密码，并没有改变原文字母，只改变了这些字母的出现顺序。即这种加密方法是对原文施加一种置换。例如如果采取的置换为(2, 1, 5, 4, 3, 7, 6, 10, 9, 8)，则原文“VICTORIOUS”加密后得到“IVOTCIRSUO”。

很容易注意到，单独应用替换加密法或置换加密法，得到的加密效果都很弱。如果将这两种加密方法组合到一起，有时加密效果很好。因此，可以把原文先用替换加密法进行加密，然后将得到的文字再用置换加密法进行加密。例如，依次采用上述替换加密法和置换加密法，原文“VICTORIOUS”被加密成“JWPUDJSTVP”。

考古学家最近在一块石头上发现一些文字，初看这些文字好像是没有什么意义的，考古学家猜测这些文字是经过替换加密法和置换加密法加密过的。他们猜想加密前的原文是怎样的，他们希望验证他们的猜想。你的任务是编写程序，验证他们的猜想是否正确。

#### 输入描述:

输入文件中有多个测试数据。每个测试数据占2行。第1行为刻在石头上的文字。在加密前，所有的空格和标点符号都去除了，因此这些文字只包含大写英文字母。第2行考古学家所猜测的加密前的原文，同样也只包括大写英文字母。这两行文字长度都不超过100。

#### 输出描述:

如果测试数据中第1行文字可能是第2行文字经过替换加密方法和置换加密方法加密后的

密文，则输出“YES”，否则输出“NO”。

**样例输入：**

JWPUDJSTVP  
VICTORIOUS  
MAMA  
ROME  
NEERCISTHEBEST  
SECRETMESSAGES

**样例输出：**

YES  
NO  
NO

## 9.9 DNA 排序(DNA Sorting)

**题目描述：**

一个序列的逆序数定义为序列中无序元素对的数目。例如，在字符序列“DAABEC”中，逆序数为 5，因为字符“D”比它右边的 4 个字符大，而字符“E”比它右边的 1 个字符大。字符序列“AACEDGG”只有 1 个逆序，即“E”和“D”，它几乎是已经排好序的，而字符序列“ZWQM”有 6 个逆序，它是最大程度上的无序——其实就是有序序列的逆序。

在本题中，你的任务是对 DNA 字符串(只包含字符“A”、“C”、“G”和“T”)进行排序。注意不是按照字母顺序进行排序，而是按照逆序数从低到高进行排序，所有字符串的长度都一样。

**输入描述：**

输入文件中包含多组测试数据。输入文件的第 1 行为一个整数 N，然后是一个空行，接下来是 N 组测试数据。每两组测试数据之间有一个空行。每组测试数据的格式为：第 1 行为两个整数，一个正整数 n， $0 < n \leq 50$ ，表示字符串的长度，以及一个正整数 m， $1 \leq m \leq 100$ ，表示字符串的数目；然后是 m 行，每一行为一个字符串，长度为 n。

**输出描述：**

对应到输入文件中的 N 组测试数据，输出也有 N 组，每两组输出之间有一个空行。对每组输入数据，按逆序数从低到高输出各字符串，如果两个字符串的逆序数一行，则按输入时的先后顺序输出。

**样例输入：**

1  
  
10 6  
AACATGAAGG  
TTTTGGCCAA  
TTTGGCCAAA  
GATCAGATTT  
CCCGGGGGGA  
ATCGATGCAT

**样例输出：**

CCCGGGGGGA  
AACATGAAGG  
GATCAGATTT  
ATCGATGCAT  
TTTTGGCCAA  
TTTGGCCAAA

## 9.10 体重排序(Does This Make Me Look Fat?)

**题目描述：**

作为一台很受欢迎的脱口秀节目的主持人，你正在做一期关于节食的节目。你的嘉宾是 Kevorkian 博士。他最近推出了一项减肥计划“Do You Want To Diet?”，这项计划向它的用户保证每天减肥 1 磅。

节目录播那天，你准备让一些使用 Kevorkian 博士减肥计划的节食者上台秀一下。你准备

按他们体重的递减顺序来安排他们出场的先后次序。问题是他们报名时只提供了以下信息：姓名，节食的天数，节食前的体重。你要根据他们节食的天数来计算他们现在的体重。所有的节食者每天减肥 1 磅。

#### 输入描述：

输入文件包含至多 100 个测试数据。测试数据之间没有空行。每个测试数据包含 3 部分：

第 1 行为"START"；

接下来为节食者列表：包含 1~10 行，每行描述了一名节食者，包括姓名、节食的天数、节食前的体重。其中姓名为 1~20 个数字、字母字符组成的字符串；节食的天数不超过 1000 天；节食前的体重不超过 10,000。

最后 1 行为"END"。

#### 输出描述：

对每个测试数据，根据各节食者现在体重的递减顺序列出节食者的名字，每个节食者的名字占一行。每两个测试数据的输出之间有一个空行。

#### 样例输入：

```
START
Joe 10 110
END
START
James 100 150
Laura 100 140
Hershey 100 130
END
```

#### 样例输出：

```
Joe
James
Laura
Hershey
```

## 9.4 二分法思想及二分检索

本节介绍二分检索的算法思想、实现方法及其应用。

### 9.4.1 二分法的思想

二分法是分治算法的一种特例。分治算法的基本思想是将一个规模为  $N$  (比较大) 的问题分解为  $K$  个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

举个通俗的例子。32 支球队参加足球世界杯，要决出一个冠军。如果让这 32 支球队一起举行联赛，那么一年的时间恐怕也比赛不完。所以分成 8 个小组，每个小组 4 支球队。先在每个小组里决出一个冠军。然后 8 个冠军又分成 2 个小组，每个小组也是 4 支球队。这两个小组又分别决出一个冠军。这样就剩下两支球队，只需要再比赛一场就决出总冠军了。(这里说的规则跟实际世界杯的规则不完全一样)

在分治算法中，若将原问题分解成两个较小的子问题，我们称之为二分法。由于二分法划分简单，所以使用非常广泛。其中最经典的应用就是二分法检索。

### 9.4.2 二分法检索

所谓检索，即查找。假设有一个整型数组 `array`，其元素个数为 100，这些元素已经按照从小到大的顺序排好序了。要在该数组中查找某个数 `num`，可以采用的方法是：依次将数组元素与该数进行比较，如果相等，则找到。但这种方法查找一个数平均需要比较  $100/2 = 50$  次。如

果数组中有 1000000 个整数，需要在数组中反复查找，则这种方法很费时。另外，这种方法并没有利用数组元素有序这个重要的信息。

假设 array 数组中的数已经按照从小到大的顺序排好了，现在要在 array 数组查找 num。

二分查找的思想是：先将 num 与 array 数组正中的元素进行比较，如果相等，则已经找到；如果 num 比正中的元素还要小，则如果 num 存在，则肯定位于前半段，不可能位于后半段，所以不需要考虑后半段；否则，num 肯定位于后半段。在前半段(或后半段)查找时，又是将 num 与正中的元素进行比较，……。一直到找到 num，或者判断 num 不存在为止。

二分查找的执行过程如图 9.10 所示。假设数组中有 10 个元素：15、17、18、22、35、51、60、88、93、99，这些数已经按照从小到大的顺序排好了。在二分查找里，有 3 个量很关键：low、mid 和 high，分别表示数组中某一段元素的最前面、中间及最后的元素的下标。

图 9.10(a)演示了在数组 array 中查找 num=18 的执行过程。

第 1 次比较时，low = 0, high = 9, mid = (low+high)/2 = 4, num 的值小于 array[mid]，所有如果 num 存在，则必然位于前半段，将 high 的值更新为 mid-1=3, low 的值不变。

第 2 次比较时，low = 0, high = 3, mid = (low+high)/2 = 1, num 的值大于 array[mid]，所有如果 num 存在，则必然位于后半段，将 low 的值更新为 mid+1=2, low 的值不变。

第 3 次比较时，low = 2, high = 3, mid = (low+high)/2 = 2, num 的值等于 array[mid]。至此，查找到 num。

以上过程要用循环来实现。现在的问题是，什么时候退出循环？

图 9.10(b)以在上述数组中查找 num=90 的情形解释了这个问题。当第 3 次比较完以后，因为 num 的值小于 array[mid]，所以如果 num 存在，则必然位于前半段，需要将 high 的值更新为 mid-1=7，而 low 的值不变，这样 high<low。这意味着 num 不存在，应该退出循环了。

因此，退出循环的条件是：high<low。

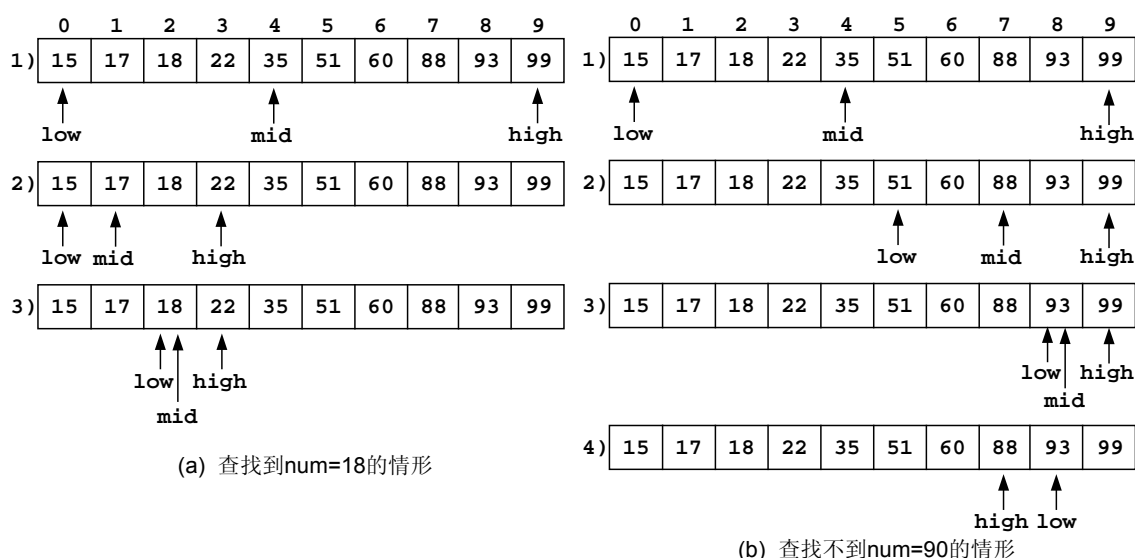


图9.10 二分检索

二分查找的实现代码详见例 9.8。

**例 9.8** 二分检索：随机产生 100000 个正整数，存储到数组 array 里，然后对这些正整数按从小到大的顺序排序。实现：反复从键盘上输入一个整数 num（直到输入的整数为负值为止），以二分法查找该整数在数组中**第一次出现**的位置；如果查找到，则输出其位置，否则输出“该数不存在”的信息。

本题要用到随机函数 rand()，关于 rand()函数的详细介绍请参考 9.3 节。

注意：二分检索方法找到的位置并不一定是 num 在排序后的 array 数组中第一次出现的位置。



置，但可以找到第一次出现的位置。方法是：因为 **array** 数组中的元素已经有序，相等的元素位置肯定相邻，所以只需要从当前位置向前扫描，直到数组元素不等于 **num** 为止。

代码如下：

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
int cmp( const void *elem1 , const void *elem2 )//qsort 函数用的比较函数
{
    return *(int *)elem1 - *(int *)elem2;
}
int BinSearch( int a[ ], int n )    //采用二分查找思想在数组 a 中查找 n
{
    int low=0, high=100000, mid;
    while( low <= high )
    {
        mid = ( low + high ) / 2;
        if( n<a[mid] ) high = mid-1;    //如果 n 比中间的数还小，则在前半段
        else if( n>a[mid] ) low = mid+1; //如果 n 比中间的数还大，则在后半段
        else return mid;
    }
    return -1;
}
void main( )
{
    int i; //循环变量
    int array[100000], num;    //存储 100000 个数的数，及要查找的数
    srand( (unsigned)time( 0 ) );
    for( i = 0; i < 100000; i++ ) //产生 100000 个随机数
        array[i] = rand( );
    qsort( array, 100000, sizeof(array[0]), cmp ); //按从小到大的顺序排序
    printf( "Please input a number: " );
    while( scanf( "%d", &num ) && num>0 )
    {
        int pos = BinSearch( array, num );    //num 在 array 中的位置
        if( pos>0 )
        {
            do
            {
                pos--;
            }while( array[pos]!=num ); //找 num 在 array 中第一次出现的位置
            printf( "%d is the %dth number in array.\n", num, pos+1 );
        }
        else printf( "There is no such a number in array.\n" );
        printf( "Please input a number: " );
    }
}
```

该程序的运行示例如下：

Please input a number: 99✓

99 is the 286th number in array.  
 Please input a number: 100✓  
 100 is the 291th number in array.  
 Please input a number: 888✓  
 888 is the 2740th number in array.  
 Please input a number: 999999✓  
 There is no such a number in array.  
 Please input a number: -2✓  
 从以上运行结果可以看出，数组中第 286~290 个数的值都是 99。

### 9.4.3 竞赛题目分析

下面以例 9.9 为例介绍二分法在具体题目中的应用。

#### 例 9.9 赌徒(Gamblers)

题目来源:

Waterloo, June 2, 2001

题目描述:

n 个赌徒一起决定玩一个游戏:

游戏刚开始的时候，每个赌徒把赌注放在桌子上并遮住，侍者要查看每个人的赌注并确保每个人的赌注都不一样。如果一个赌徒没有钱了，则他要借一些筹码，因此他的赌注为负数。假定赌注都是整数。

最后赌徒们揭开盖子，出示他们的赌注。如果谁下的赌注是其他赌徒中某 3 个人下的赌注之和，则他是胜利者。如果有多于一个的胜利者，则下的赌注最大的赌徒才是最终的胜利者。

例如，假定赌徒为：Tom, Bill, John, Roger 和 Bush，他们下的赌注分别为：\$2, \$3, \$5, \$7 和 \$12。因此最终的胜利者是 Bush，因为他下的赌注为 \$12，而其他的人下的赌注之和也等于 12：\$2 + \$3 + \$7 = \$12，因此 Bush 是胜利者(并且没有其他人是胜利者)。

输入描述:

输入文件中包含了多组赌徒下的赌注。每组赌注的数据第 1 行是一个整数 n,  $1 \leq n \leq 1000$ ，代表赌徒的个数，然后是他们下的赌注，每个人的赌注占一行，这些赌注各不相同，并且范围是[-536870912, +536870911]。输入文件的最后一行为 0，代表输入结束。

输出描述:

对每组赌注，输出胜利者下的赌注，如果没有解，则输出 “no solution”。

样例输入:

5  
2  
3  
5  
7  
12  
5  
2  
16  
64  
256  
1024

样例输出:

12  
no solution

0

分析:

本题要求的是一个最大的赌注, 满足: 它是其他 3 个赌注的和。假设赌注存放在 `data` 数组中, 先将 `n` 个人的赌注按从小到大的顺序排序。可以采用的思路是枚举, 即从最大的赌注开始, 看是否存在一个赌注是其他 3 个不同赌注之和。这个过程需要用四重循环实现。代码如下:

```
void work( )
{
    qsort( data, n, sizeof(data[0]), cmp );    //按从小到大的顺序排序
    int i, j, k, m;    //循环变量
    for( i = n - 1; i >= 0; i-- )
    {
        for( j = 0; j < n; j++ )
        {
            if( i == j ) continue;
            for( k = j + 1; k < n; k++ )
            {
                if( i == k ) continue;
                for( m = k + 1; m < n; m++ )
                {
                    if( i == m ) continue;
                    //判断 i 的赌注是否是 j,k,m 赌注之和
                    if( data[i] == data[j]+data[k]+data[m] )
                    {
                        printf( "%d\n", data[i] );    return;
                    }
                }
            }
        }
    }
    printf( "no solution\n" );
}
```

上述代码使用了四重循环, 由于 `n` 的值最大可以取到 1000, 如果 `n` 的值取 1000, 则上述循环中 `if` 语句里的比较语句在最坏情况下要执行  $1000 \times 1000 \times 1000 \times 1000$  次, 很显然这不是一种好方法。读者可以到 OJ 上去提交试试, 看是否会超时, 如果没有超时, 记录运行时间。

本题采用二分查找减少一重循环。方法是: 对上述代码, 取消第四重循环, 即最里面的 `for` 循环, 也就说不是枚举 `m` 的所有取值, 而是在 `data` 数组里查找 `data[i] - data[j] - data[k]`, 如果能找到, 即存在某个人的赌注  $M = data[i] - data[j] - data[k]$ , 也就是  $data[i] = data[j] + data[k] + M$ , 因此 `data[i]` 满足题目的要求。但同时要保证  $(data[i] - data[j] - data[k])$  既不等于 `data[i]`、`data[j]`, 也不等于 `data[k]`, 为什么呢? 以 `data[i]` 为例, 如果  $data[i] - data[j] - data[k] = data[i]$ , 则只要 `data[j] = -data[k]`, 所有的 `data[i]` 都满足条件。

代码如下:

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

const int MAXN = 1001;
int data[MAXN], n;    //赌徒们下的赌注及赌徒的人数
```

```

int cmp( const void *elem1 , const void *elem2 )//qsort 函数用的比较函数
{
    return *(int *)elem1 - *(int *)elem2;
}
int search( int x ) //在 data 数组中查找 x, 如果查找到, 返回 1, 否则返回 0
{
    int low = 0, high = n - 1, mid;
    while( low <= high )
    {
        mid = (low + high) / 2;
        if( data[mid] == x ) return 1;
        if( data[mid] > x ) high = mid - 1;
        if( data[mid] < x ) low = mid + 1;
    }
    return 0;
}
void work( )
{
    qsort( data, n, sizeof(data[0]), cmp );    //按从小到大的顺序排序
    for( int i = n - 1; i >= 0; i-- )
    {
        for( int j = 0; j < n; j++ )
        {
            if( i == j ) continue;
            for( int k = j + 1; k < n; k++ )
            {
                if( i == k ) continue;
                //存在第 4 个人下的赌注为 data[i] - data[j] - data[k]
                //并且这个人不是 i,j, 也不是 k, 则 i 就是胜利者
                if( search(data[i] - data[j] - data[k]) &&
                    data[i] - data[j] - data[k] != data[i] &&
                    data[i] - data[j] - data[k] != data[j] &&
                    data[i] - data[j] - data[k] != data[k])
                {
                    printf( "%d\n", data[i] ); return;
                }
            }
        }
    }
    printf( "no solution\n" );
}
int main( )
{
    while( scanf("%d", &n) )
    {
        if( n==0 ) break;
        for( int i = 0; i<n; i++ ) scanf( "%d", &data[i] );    //输入下的赌注
        work( ); //求解
    }
    return 0;
}

```

}

## 练习

## 9.11 半素数(Semi-Prime)

## 题目描述:

素数的定义: 对于一个大于 1 的正整数, 如果除了 1 和它本身没有其他的正约数了, 那么这个数就称为素数。例如, 2, 11, 67, 89 是素数, 8, 20, 27 不是素数。

半素数的定义: 对于一个大于 1 的正整数, 如果它可以被分解成 2 个素数的乘积, 则称该数为半素数, 例如 6 是一个半素数, 而 12 不是。

你的任务是判断一个数是否是半素数。

## 输入描述:

输入文件中有多个测试数据, 每个测试数据包含一个整数  $N$ ,  $2 \leq N \leq 1,000,000$ 。

## 输出描述:

对每个测试数据, 如果  $N$  是半素数, 则输出 Yes, 否则输出 No。

## 样例输入:

3  
4  
6  
12

## 样例输出:

No  
Yes  
Yes  
No

## 9.12 棍子的膨胀(Expanding Rods)

## 题目描述:

当一根长度为  $L$  的细长金属棍子加热  $n$  度后, 它会膨胀到一个新的长度  $L' = (1+n*C)*L$ , 其中  $C$  为该金属的热膨胀系数。

当一根细长的金属棍子固定在两堵墙之间, 然后加热, 则棍子会变成圆弓形, 棍子的原始位置为该圆弓形的弦。如图 9.11 所示。

你的任务是计算棍子中心的偏离距离。

## 输入描述:

输入文件包含多个测试数据, 每个测试数据占一行。每个测试数据包含 3 个非负整数: 棍子的初始长度, 单位为毫米; 加热前后的温差, 单位为度; 该金属的热膨胀系数。输入数据保证膨胀的长度不超过棍子本身长度的一半。输入文件的最后一行为 3 个负数, 代表输入结束, 该测试数据不需处理。

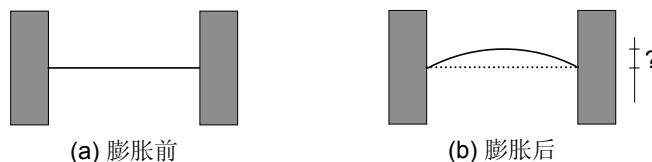


图9.11 膨胀的金属棍子

## 输出描述:

对每个测试数据, 输出金属棍子中心加热后偏离的距离, 单位为毫米, 保留小数点后 3 位有效数字。

## 样例输入:

## 样例输出:

|                  |         |
|------------------|---------|
| 1000 100 0.0001  | 61.329  |
| 15000 10 0.00006 | 225.020 |
| 10 0 0.001       | 0.000   |
| -1 -1 -1         |         |

## 第四篇 课程设计

本篇的内容包括第 10 章，课程设计。第 10 章以“字符界面的扫雷游戏”为例，通过任务分解，循序渐进地讲解整个程序的开发过程。在第 10 章的练习题中，还给出了 3 个课程设计选题的例子。

### 程序实践提示

读者在学完本教材介绍的程序设计方法和算法后，可以通过课程设计来强化这些方法和算法的综合应用。很多 **ACM/ICPC** 题目是取材于一些经典游戏，通过对这些游戏的规则进行简化来构造题目。读者在做这些题目的时候可以思考能否把题目的游戏规则进行完善，把程序扩充，这样就能找到一个很好的课程设计题目。

---



# 第 10 章 课程设计：字符界面扫雷游戏的开发

本章循序渐进地讲解了字符界面扫雷游戏的开发过程。首先在 10.1 节给出了该软件的需求说明，接下来将整个软件的开发过程分成三部分来讲解，每部分内容都有测试程序。读者在理解每部分内容后可以运行、分析测试程序来进一步理解程序的思路。

## 10.1 软件需求说明

编程实现一个字符界面的扫雷游戏。功能需求如下：

- 1. 允许用户输入地图的大小，为简化起见，地图的大小是  $N \times N$  的， $N$  的值从键盘输入， $5 \leq N \leq 20$ 。默认为 5。
- 2. 地图中地雷的个数允许用户输入。建议用户输入的地雷个数小于  $N * N / 4$ 。地雷的分布是随机的。
- 3. 初始时显示地图，各个位置都没有点开，也没有标记，格式如图 10.1 所示(该地图大小为  $10 \times 10$ )。

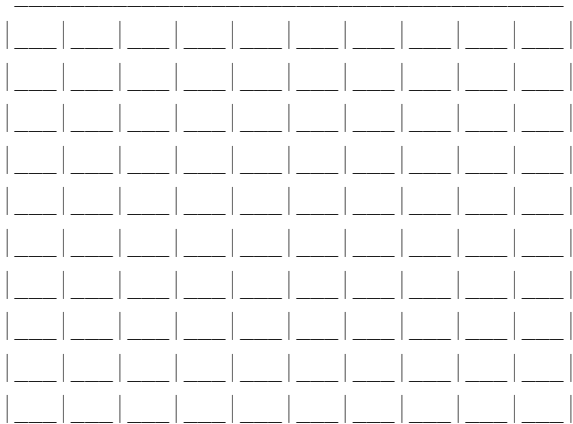


图 10.1 字符界面扫雷游戏的地图

- 4. 玩游戏时，用户每次操作是从键盘输入 3 个数据： $x\ y\ op$  来表示的， $(x,y)$ 表示操作的位置，字符  $op$  表示要执行的操作。可以执行的操作有：点开(用字符 “.” 表示)、标记地雷(用字符 “!” 表示)、取消标记地雷(用字符 “c” 表示)、退出游戏(用字符 “q” 表示)。
  - a) 如 windows 扫雷游戏一样，当点开的位置 $(x,y)$ ，其 8 个相邻位置上地雷总数为 0，则点开连成一片的空白区域。某个位置的 8 个相邻位置从左上角开始按顺时针顺序依次为：左上角、上、右上角、右、右下角、下、左下角、左。
  - b) 如果在游戏过程中踩到地雷，则游戏失败。
  - c) 如果用户标记出来的每个地雷位置跟实际的位置一致，并且标记出来的地雷个数与实际地雷个数一致，则游戏成功完成。并且，这时把还没有点开的位置(这些位置上都没有地雷)都点开。
- 5. 每执行一次操作后，显示操作后的地图。

某次游戏过程，如图 10.2 所示。

请输入地图的大小  $N$ (地图是  $N \times N$  大小的， $5 \leq N \leq 20$ ): 5↵(↵表示输入，下同)

请输入地图中地雷的个数(地雷是随机分布的, 建议个数小于  $N*N/4$ ): 3✓

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Start Game!

请输入执行操作的位置及要执行的操作, 格式为: x y op

1 1 .✓

|  |   |   |   |   |
|--|---|---|---|---|
|  | 0 | 1 |   |   |
|  | 0 | 1 |   |   |
|  | 0 | 1 | 3 |   |
|  | 0 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 |

还有 3 颗地雷没有标记!

请输入执行操作的位置及要执行的操作, 格式为: x y op

2 3 !✓

|  |   |   |   |   |
|--|---|---|---|---|
|  | 0 | 1 |   |   |
|  | 0 | 1 | ! |   |
|  | 0 | 1 | 3 |   |
|  | 0 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 |

还有 2 颗地雷没有标记!

请输入执行操作的位置及要执行的操作, 格式为: x y op

2 4 !✓

|  |   |   |   |   |
|--|---|---|---|---|
|  | 0 | 1 |   |   |
|  | 0 | 1 | ! | ! |
|  | 0 | 1 | 3 |   |
|  | 0 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 |

还有 1 颗地雷没有标记!

请输入执行操作的位置及要执行的操作, 格式为: x y op

3 4 !✓

|  |   |   |   |   |
|--|---|---|---|---|
|  | 0 | 1 | 2 | 2 |
|  | 0 | 1 | ! | ! |
|  | 0 | 1 | 3 | ! |
|  | 0 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 |

游戏成功!

图 10.2 某次游戏过程

接下来的 10.2~10.4 节将整个软件的开发过程分成三部分来讲解, 每部分内容都有测试程

序。

## 10.2 地图的表示与输出

### 10.2.1 Windows 操作系统扫雷游戏简介

Windows 操作系统提供的扫雷游戏功能及玩法大致如下：

- 1) 启动游戏后的界面如图 10.3(a)所示。刚开始的时候可以通过鼠标左键单击“地图”，试探性地点开地图。在游戏过程中，如果不小心点中了地雷所在的位置，则游戏失败并将所有地雷的位置显示出来。
- 2) 如果点开的位置 8 个相邻位置上都没有地雷（这种位置在本文中称为零位置），则扫雷游戏会自动点开一片连续的没有地雷的区域。注意：没有地雷的位置不一定是零位置。例如图 10.3(b)中，(7,4)这个位置没有地雷，但它的 8 个相邻位置上有 1 颗地雷。单击(7,4)这个位置不会点开一片连续的没有地雷的区域。如果单击的是零位置，如(6,5)这个位置，则(4,5)、(9,6)等这些零位置会点开，因为这些零位置通过若干个零位置与(6,5)相邻。另外，(6,4)、(3,7)等这些没有地雷的位置也被点开，因为它们在这片零位置中某个零位置的相邻位置。
- 3) 鼠标右键可以标记地雷的位置，如图 10.3(c)所示。
- 4) 对于不是地雷的位置，可以用鼠标左键点开。当把所有地雷正确地标记出来，并且所有其他位置都点开，游戏成功结束，如图 10.3(d)所示。

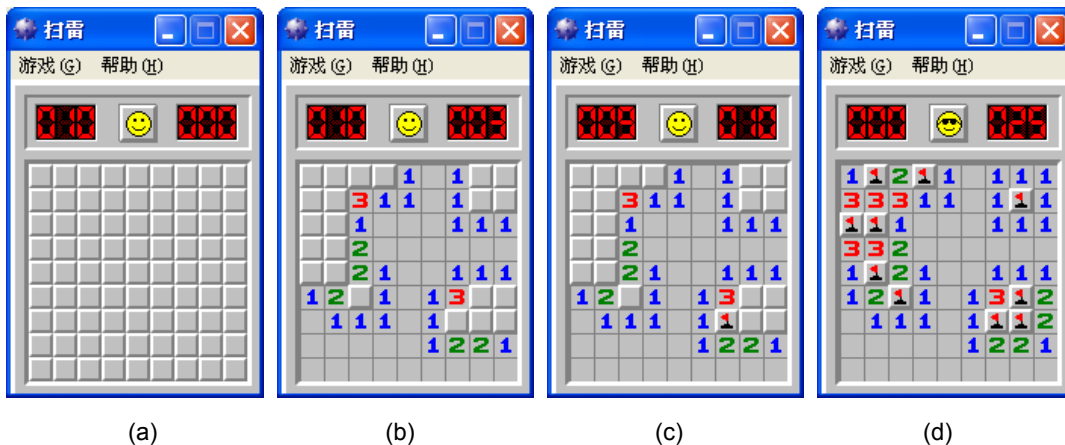


图 10.3 Windows 操作系统的扫雷游戏

本文所设计的程序采用字符界面来模拟该游戏。

### 10.2.2 如何表示地图

在程序中用  $N \times N$  大小的“棋盘”表示扫雷游戏的地图。在程序中要用二维数组 `map` 来表示这个地图。为了与日常生活中的习惯一致，二维数组 `map` 中的第 0 行和第 0 列元素不用，如图 10.4 所示。

第 0 行和第 0 列的元素不用还有另外一个好处，我们在计算每个位置周围 8 个相邻位置上的地雷个数时，不需要判断该位置是否为左边界和上边界位置(即第 0 行和第 0 列)。具体分析如下。

如果从第 0 行、第 0 列的元素开始算起，则左边界和上边界上的元素，即第 0 行和第 0 列的元素，比如 `[0][0]` 这个元素，它的 8 个相邻位置中有 5 个是在二维数组范围外的，不能统计这 5 个位置上地雷的个数。因此，对边界元素需要单独考虑。

而从第 1 行、第 1 列开始算起的话，地图中每个位置(包括左边界和上边界上的位置，即第 1 行和第 1 列上的位置)的 8 个相邻位置都是在二维数组范围内，可以统一处理。例如[1][1]这个位置的 8 个相邻位置(从左上角位置开始按顺时针顺序)分别为：[0][0]、[0][1]、[0][2]、[1][2]、[2][2]、[2][1]、[2][0]、[1][0]，我们要统计这 8 个相邻位置中地雷的个数。这 8 个位置中位于第 0 列或第 0 行的有 5 个，都是有效的数组元素，而且它们的值为 0，在统计时可以统一处理。因此对左边界和上边界上的位置不必专门处理。

同样的道理，对  $N \times N$  大小的地图，为了对右边界和下边界(即第  $N$  列和第  $N$  行)的位置统一处理，需要把第  $N+1$  列和第  $N+1$  行包括在内。因此如果最大的地图为  $20 \times 20$ ，则二维数组 map 需定义成  $22 \times 22$ 。

map 数组

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| [0][0] | [0][1] | [0][2] | [0][3] | [0][4] | [0][5] |
| [1][0] | [1][1] | [1][2] | [1][3] | [1][4] | [1][5] |
| [2][0] | [2][1] | [2][2] | [2][3] | [2][4] | [2][5] |
| [3][0] | [3][1] | [3][2] | [3][3] | [3][4] | [3][5] |
| [4][0] | [4][1] | [4][2] | [4][3] | [4][4] | [4][5] |
| [5][0] | [5][1] | [5][2] | [5][3] | [5][4] | [5][5] |

说明：  
Map 数组第 0 行及第 0 列不用，即左图中下标为红色的元素不用。

图10.4 用二维数组来表示扫雷游戏的地图

map 数组中各元素  $\text{map}[x][y]$  的取值及其含义分别为：

- 1: (x,y)位置上为地雷；
- 0~8: (x,y)位置上没有地雷，该数字为 8 个相邻位置上地雷的个数。

图 10.5 给出了一个扫雷游戏地图及对应的 map 数组示例。

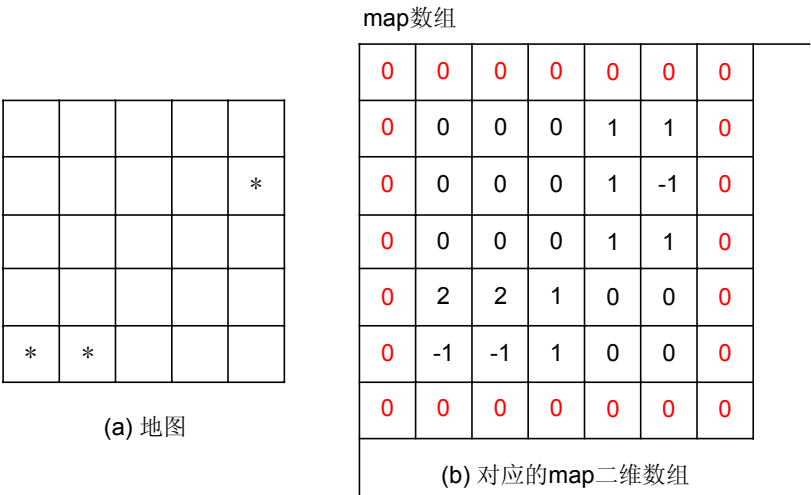


图10.5 扫雷游戏地图及对应的二维数组示例

10.2.3 如何表示一个位置的 8 个相邻位置

在统计某个位置(x,y)的 8 个相邻位置上地雷个数时，需要用到 8 个相邻位置。本程序很巧妙地处理了这个问题。

如图 10.6 所示。(x,y)位置的左上角位置，其行坐标为  $x-1$ ，列坐标为  $y-1$ ，行、列坐标的“增量”均为-1；右边位置相对于(x,y)位置，行、列坐标的“增量”分别为 0 和 1。因此在程序中可以定义一个  $8 \times 2$  的二维数组 dir，表示 8 个相邻位置的行、列坐标相对于(x,y)位置行、

列坐标的增量。

```
int dir[8][2] = { {-1,-1}, {-1,0}, {-1,1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1} };
```

dir 二维数组所表示的 8 个位置的顺序是从左上角位置开始，按顺时针顺序。

有了这个二维数组，统计(x,y)位置的 8 个相邻位置上地雷的个数，可以用下面的代码段：

```
int k, mines = 0;
for( k=0; k<8; k++ )
{
    if( map[ x+dir[k][0] ][ y+dir[k][1] ] == -1 ) mines++;
}
```

|         |        |        |
|---------|--------|--------|
| (-1,-1) | (-1,0) | (-1,1) |
| (0,-1)  | (0,0)  | (0,1)  |
| (1,-1)  | (1,0)  | (1,1)  |

说明：

(x,y)位置为中心位置，其8个相邻位置的行、列坐标相对于(x,y)位置行、列坐标的“增量”在图中列出来了。

图10.6 位置(x,y)的8个相邻位置

### 10.2.4 如何输出地图

对于图 10.5 所示的 5×5(有 3 颗地雷，\*表示地雷)的地图，存储到 map 数组里后，要按如下的格式输出：

```
_0_	_0_	_0_	_1_	_1_
_0_	_0_	_0_	_1_	_*_
_0_	_0_	_0_	_1_	_1_
_2_	_2_	_1_	_0_	_0_
_*_	_*_	_1_	_0_	_0_
```

即对于 N×N 的地图，共输出 N+1 行：

第 1 行：第 0 个字符是空格字符，其 ASCII 编码值为 32，或者用'\40'表示也可以。然后是 4\*N-1 个下划线字符“\_”。最后是字符串结束标志'\0'。

后面 N 行，每行：第 0 个字符是字符“|”，然后是 N 组字符。每组字符，首先是下划线字符“\_”，然后是 map[i][j]这个位置上对应的字符，如果是地雷，显示为“\*”，否则显示该位置周围 8 个相邻位置上地雷的个数，最后两个字符是“\_”和“|”。最后也是字符串结束标志'\0'。

因此地图的输出可以用下面的函数实现。其中字符数组 line 为全局变量，用于输出地图时，表示地图每一行的字符串。

```
void Output_Map( ) //输出地图
{
    int i,j; //循环变量
    line[0] = '\40'; //'\40'表示空格
    for( i=1; i<=4*N-1; i++ )
        line[i] = '_';
    line[i] = '\0'; puts(line); //输出第 0 行
    for( i=1; i<=N; i++ ) //输出地图中的后面 N 行
    {
        line[0] = '|';
        for( j=1; j<=N; j++ ) //每行在 line[0]后面有 N 组字符
        {
```

```

        line[4*j-3] = '_';
        if( map[i][j] == -1 ) line[4*j-2] = '*';    //地雷，显示***
        else line[4*j-2] = 48+map[i][j];    //不是地雷，显示 8 个相邻位置上地雷个数
        line[4*j-1] = '_';
        line[4*j] = '|';
    }
    line[4*N+1] = '\0'; //串结束符标志
    puts(line);    //输出
}
}

```

程序里有一点要说明，如果 `map[i][j]` 的值是 0~8 之间的数，要输出对应的数字字符，需要在数字字符“0”的 ASCII 编码值 48 的基础上加上 `map[i][j]` 的值。例如，如果 `map[i][j]` 的值为 2，则 `48+map[i][j]` 的值为 50，就是数字字符“2”的 ASCII 编码值，显示出来的就是数字字符“2”。

### 10.2.5 测试程序

以下测试程序可以表示和输出上一节中的地图。其中 `Manual_Map1( )` 函数用于手动设置一个地图，其中(2,5)、(5,1)、(5,2)这 3 个位置上为地雷。

程序代码如下：

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

```

int N = 5;           //实际的扫雷地图是 N×N 大小的，N 的最大值为 20
//存储地图(从第 1 行、第 1 列开始表示地图，第 0 行、第 0 列不用)
//map[i][j]==-1 表示地雷；map[i][j]的值在[0,8]之间，表示(i,j)位置周围 8 个位置上地雷个数
int map[22][22] = {0};
char line[100];      //在输出地图时，表示地图每一行的字符串
int MineNum = 10;    //地雷的个数
//每个位置的 8 个相邻位置(从左上角位置开始按顺时针顺序列出)
int dir[8][2] = { {-1,-1}, {-1,0}, {-1,1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1} };

```

```
void Manual_Map1( ) //手动设置地图
```

```

{
    N = 5;    MineNum = 3;    //5×5 的地图，有 3 颗地雷
    map[2][5] = -1;    map[5][1] = -1;    map[5][2] = -1;    //手动设置 3 颗雷
    int i, j, k; //循环变量
    int mines;    //每个位置上周围 8 个位置上地雷的个数
    for( i=1; i<=N; i++ )//对其余位置，统计其四周 8 个位置上地雷的个数
    {
        for( j=1; j<=N; j++ )
        {
            if( map[i][j] == -1 ) continue;    //[i][j]位置上为地雷，不用统计
            mines = 0;
            for( k=0; k<8; k++ )
            {
                if( map[ i+dir[k][0] ][ j+dir[k][1] ] == -1 ) mines++;
            }
            map[i][j] = mines;
        }
    }
}

```

```

    }
}

void Output_Map( ) //输出地图
{
    int i, j; //循环变量
    line[0] = '\40'; //'\40'表示空格
    for( i=1; i<=4*N-1; i++ )
        line[i] = '_';
    line[i] = '\0'; puts(line); //输出第 0 行
    for( i=1; i<=N; i++ ) //输出地图中的后面 N 行
    {
        line[0] = '|';
        for( j=1; j<=N; j++ ) //每行在 line[0]后面有 N 组字符
        {
            line[4*j-3] = '_';
            if( map[i][j] == -1 ) line[4*j-2] = '*'; //地雷，显示 "*"
            else line[4*j-2] = 48+map[i][j]; //不是地雷，显示 8 个相邻位置上地雷个数
            line[4*j-1] = '_';
            line[4*j] = '|';
        }
        line[4*N+1] = '\0'; //串结束符标志
        puts( line ); //输出
    }
}

int main( )
{
    Manual_Map1( ); //手动设置地图
    Output_Map( ); //输出地图
    return 0;
}

```

该程序的输出结果为：

```

_0_	_0_	_0_	_1_	_1_
_0_	_0_	_0_	_1_	_*_
_0_	_0_	_0_	_1_	_1_
_2_	_2_	_1_	_0_	_0_
_*_	_*_	_1_	_0_	_0_

```

## 10.3 随机生成地图

在 10.2 节实现了手动设置地雷的个数及位置，然后输出地图。10.3 节这部分内容讲述随机地产生不重复的 MineNum(注：MineNum 是程序中表示地图中地雷个数的变量)个地雷的位置，从而生成了一个随机的地图。这里需要用到产生随机数的函数 rand( )。

### 10.3.1 随机函数 rand( )

这个函数包含在<stdlib.h>头文件中，函数的原型为：

```
int rand( void );
```

该函数用于产生 0 到 RAND\_MAX 之间的伪随机数。RAND\_MAX 是计算机里定义的符号常量，其值为 32767。rand( )函数采用线性同余法来产生随机数，它产生的随机数不是真正的随机数，是“伪”随机数。

为了保证每次运行程序时产生的随机数不同，在使用 rand( )函数前，需要调用 srand( )函数设置随机数种子。srand( )函数的原型为：

```
void srand( unsigned int seed );
```

通常该函数的调用形式为：

```
srand( (unsigned)time( 0 ) );
```

即用当前系统时间作为随机种子。其中 time 函数是<time.h>头文件中定义的函数，用于获取当前系统时间。

例如下面的代码用于产生 10 个随机数。

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
void main( void )
{
    int i;
    srand( (unsigned)time( 0 ) );
    for( i = 0; i < 10; i++ )
        printf( " %6d\n", rand( ) );
}
```

可能的输出为：

```
6929
8026
21987
30734
20587
6699
22034
25051
7988
10104
```

如果要产生 1~10 的随机数，可以采用表达式 “1+(int)( 10.0\*rand( )/(RAND\_MAX+1) )”。

### 10.3.2 随机生成地图

要随机地生成地图，需要随机地生成 MineNum 个地雷的位置(num1, num2)，这个位置的行坐标 num1、列坐标 num2 都是 1~N 的随机数。而且 MineNum 个位置不能重复。

rand( )函数生成的随机数范围是 0~RAND\_MAX。为了产生一对 1~N 之间的随机数，需要用下面的式子转换：

```
num1 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) );
```

```
num2 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) );
```

生成一对随机数(num1,num2)后，还得判断 map[num1][num2]是否为-1，如果为-1 表示之前已经在该位置上设置了地雷，要重新生成一对随机数。

生成一个随机的地图可以用下面的函数实现。

```
void Random_Map( ) //随机生成地图
```



```

{
    //随机地产生 MineNum 个位置，这些位置上就是地雷
    int i=1, j, k; //循环变量
    srand( (unsigned)time( 0 ) );
    int num1, num2;
    while( i <= MineNum ) //随机地产生 MineNum 个地雷的位置
    {
        num1 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) ); //产生 1~N 之间的随机数
        num2 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) ); //产生 1~N 之间的随机数
        //排除第 0 行,第 0 列，及重复的地雷位置
        if( num1<1 || num1>N || num2<1 || num2>N || map[num1][num2]==-1 )
            continue;
        map[num1][num2] = -1;
        i++;
    }

    //对其余位置，统计其四周 8 个位置上地雷的个数
    int mines; //每个位置上周围雷的个数
    for( i=1; i<=N; i++ )
    {
        for( j=1; j<=N; j++ )
        {
            if( map[i][j] == -1 ) continue;
            mines = 0;
            for( k=0; k<8; k++ )
            {
                if( map[ i+dir[k][0] ][ j+dir[k][1] ] == -1 ) mines++;
            }
            map[i][j] = mines;
        }
    }
}

```

### 10.3.3 测试程序

以下测试程序可以实现：输入地图大小  $N$  和地雷个数  $MineNum$ ，然后在地图中随机地生成  $MineNum$  个地雷的位置，并输出地图。

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int N = 5; //实际的扫雷地图是  $N \times N$  大小的， $N$  的最大值为 20
//存储地图(从第 1 行、第 1 列开始表示地图，第 0 行、第 0 列不用)
//map[i][j]==-1 表示地雷；map[i][j]的值在[0,8]之间，表示(i,j)位置周围 8 个位置上地雷个数
int map[22][22] = {0};
char line[100]; //在输出地图时，表示地图每一行的字符串
int MineNum = 10; //地雷的个数
//每个位置的 8 个相邻位置(从左上角位置开始按顺时针顺序列出)
int dir[8][2] = { {-1,-1}, {-1,0}, {-1,1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1} };

```

```

void Output_Map( ) //输出地图
{
    int i, j; //循环变量
    line[0] = '\40'; //'\40'表示空格
    for( i=1; i<=4*N-1; i++ )
        line[i] = '_';
    line[i] = '\0'; puts(line); //输出第 0 行
    for( i=1; i<=N; i++ ) //输出地图中的后面 N 行
    {
        line[0] = '|';
        for( j=1; j<=N; j++ ) //每行在 line[0]后面有 N 组字符
        {
            line[4*j-3] = '_';
            if( map[i][j] == -1 ) line[4*j-2] = '*'; //地雷，显示"*"
            else line[4*j-2] = 48+map[i][j]; //不是地雷，显示 8 个相邻位置上地雷个数
            line[4*j-1] = '_';
            line[4*j] = '|';
        }
        line[4*N+1] = '\0'; //串结束符标志
        puts(line); //输出
    }
}

void Random_Map( ) //随机生成地图
{
    //随机地产生 MineNum 个位置，这些位置上就是地雷
    int i=1, j, k; //循环变量
    srand( (unsigned)time( 0 ) );
    int num1, num2;
    while( i <= MineNum ) //随机地产生 MineNum 个地雷的位置
    {
        num1 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) ); //产生 1~N 之间的随机数
        num2 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) ); //产生 1~N 之间的随机数
        //排除第 0 行,第 0 列, 及重复的地雷位置
        if( num1<1 || num1>N || num2<1 || num2>N || map[num1][num2]==-1 )
            continue;
        map[num1][num2] = -1;
        i++;
    }

    //对其余位置，统计其四周 8 个位置上地雷的个数
    int mines; //每个位置上周围雷的个数
    for( i=1; i<=N; i++ )
    {
        for( j=1; j<=N; j++ )
        {
            if( map[i][j] == -1 ) continue;
            mines = 0;
            for( k=0; k<8; k++ )
            {
                if( map[ i+dir[k][0] ][ j+dir[k][1] ] == -1 ) mines++;
            }
        }
    }
}

```

```

    }
    map[i][j] = mines;
}
}
}
int main( )
{
    printf( "请输入地图的大小 N(地图是 N×N 大小的, 5=<N<=20): ");
    scanf( "%d", &N );
    printf( "请输入地图中地雷的个数(地雷是随机分布的, 建议个数小于 N*N/4): ");
    scanf( "%d", &MineNum );
    Random_Map( ); //随机生成地图
    Output_Map( ); //输出地图
    return 0;
}

```

该程序的运行示例如下：

请输入地图的大小 N(地图是 N×N 大小的, 5=<N<=20): 10✓

请输入地图中地雷的个数(地雷是随机分布的, 建议个数小于 N\*N/4): 10✓

```

_0_	_0_	_0_	_0_	_1_	_*_	_2_	_1_	_1_	_0_
_0_	_0_	_0_	_0_	_2_	_2_	_3_	_*_	_2_	_1_
_0_	_0_	_0_	_0_	_1_	_*_	_3_	_3_	_*_	_1_
_0_	_0_	_0_	_1_	_2_	_2_	_2_	_*_	_3_	_2_
_0_	_0_	_0_	_1_	_*_	_1_	_1_	_1_	_2_	_*_
_0_	_0_	_0_	_1_	_1_	_1_	_0_	_0_	_1_	_1_
_0_	_0_	_0_	_0_	_0_	_0_	_0_	_0_	_0_	_0_
_0_	_0_	_0_	_1_	_1_	_1_	_0_	_0_	_0_	_0_
_0_	_1_	_1_	_2_	_*_	_1_	_0_	_0_	_1_	_1_
_0_	_1_	_*_	_2_	_1_	_1_	_0_	_0_	_1_	_*_

```

## 10.4 如何实现玩游戏

### 10.4.1 显示给用户看的地图

要实现扫雷游戏，除了要用 `map` 数组来存储地图外，还要一个二维数组 `usermap` 用来记录用户的操作(点开某个位置，或标记某个位置为地雷)，并且显示给用户看的地图也是根据 `usermap` 数组来显示的。

`usermap` 数组中各元素 `usermap[x][y]` 的取值及其含义分别为：

0：表示用户没有点开(x,y)位置；

1：表示用户已经点开(x,y)位置；

2：表示用户已经将(x,y)位置标记为地雷。

以图 10.7 解释 `map` 数组与 `usermap` 数组的关系、`usermap` 数组各元素值的含义、显示给用户看的地图的格式。

该游戏的地图如图 10.7(a)所示，地图为 5×5 大小的，其中有 3 颗地雷。`map` 数组各元素的值如图 10.7(b)所示，`map` 数组各元素的值在游戏过程中是不变的。假设用户首先点开(1,1)

这个位置，则成片的空白区域都点开了。然后用户依次将(5,1)、(5,2)、(2,5)这 3 个位置上的地雷标记出来。这样整个地图中只有(1,5)这个位置既没有点开，也没有标记(当然程序会判断标记出来的 3 个地雷位置跟实际的位置都吻合，游戏成功完成)。这时，usermap 数组各元素的值如图 10.7(d)所示。显示给用户看到的地图为图 10.7(c)。其具体格式及含义为：

- 1) 对没有点开的位置，显示为下划线 “\_”，如[1][5]这个位置没有点开。
- 2) 对点开的位置，显示周围 8 个相邻位置上地雷的个数，如[4][1]这个位置上的数字为 2，表示其 8 个相邻上地雷的个数为 2 个。
- 3) 对于用户已做标记的位置，显示 “!”，如[5][1]这个位置已经标记为地雷。

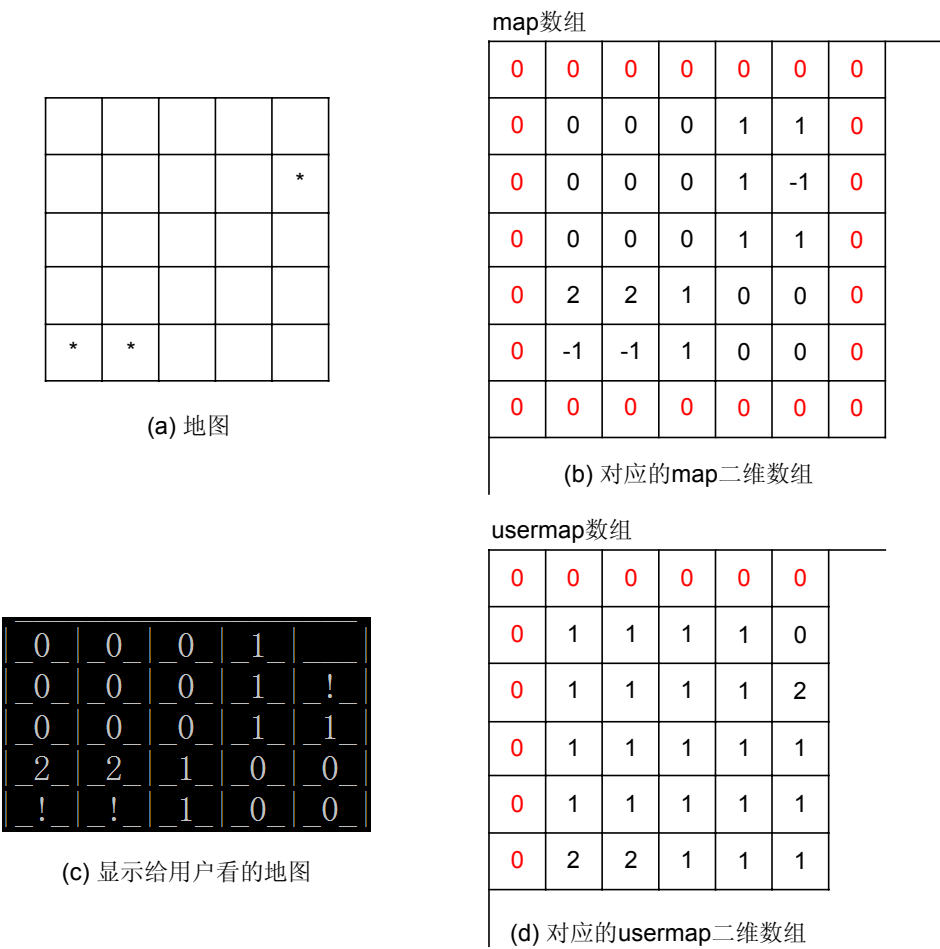


图10.7 map数组与usermap数组

10.4.2 输出用户地图

以下的 Output\_UserMap 函数用于输出用户地图，输出方法给跟 10.1 节中的 Output\_Map 函数类似，这里不再赘述。需要注意的是：

- 1) Output\_UserMap 函数除了可以输出用户地图外，还可以判断游戏是否成功结束，方法是判断用户标记的每个地雷位置是否跟实际地雷的位置吻合，并且正确标记出的地雷个数跟地图中地雷的总数是否相等，如果相等，则游戏已经成功完成。游戏成功时，把没点开的位置(这些位置没有地雷)全部点开。
- 2) 用户每次操作后，都要重新显示地图，所以要调用 Output\_UserMap 函数。

```
void Output_UserMap( ) //输出用户地图
{
    RightMarkNum = 0;
    int i, j; //循环变量
```

```

for( i=1; i<=N; i++ )    //统计用户正确标记地雷的个数
{
    for( j=1; j<=N; j++ )
        if( usermap[i][j]==2 &&map[i][j]==-1 )    RightMarkNum++;
}
if( RightMarkNum==MineNum )//成功，把没点开的位置(这些位置没有地雷)全点开
{
    bsuccess = true;
    for( i=1; i<=N; i++ )
    {
        for( j=1; j<=N; j++ )
            if( usermap[i][j]==0 )    usermap[i][j]=1;
    }
}
//接下来显示用户地图
line[0] = '\40';//'\40'表示空格
for( i=1; i<=4*N-1; i++ )
    line[i] = '_';
line[i] = '\0';    puts(line);    //输出第 0 行
for( i=1; i<=N; i++ )    //输出地图中的后面 N 行
{
    line[0] = '|';
    for( j=1; j<=N; j++ )    //每行在 line[0]后面有 N 组字符
    {
        line[4*j-3] = '_';
        if( usermap[i][j]==0 )    line[4*j-2]='_';//没有点开
        else if( usermap[i][j] == 1 )    line[4*j-2] = 48+map[i][j]; //已经点开
        else if( usermap[i][j] == 2 )    line[4*j-2] = '!';    //标记
        line[4*j-1] = '_';
        line[4*j] = '|';
    }
    line[4*N+1] = '\0'; //串结束符标志
    puts(line);    //输出
}
}

```

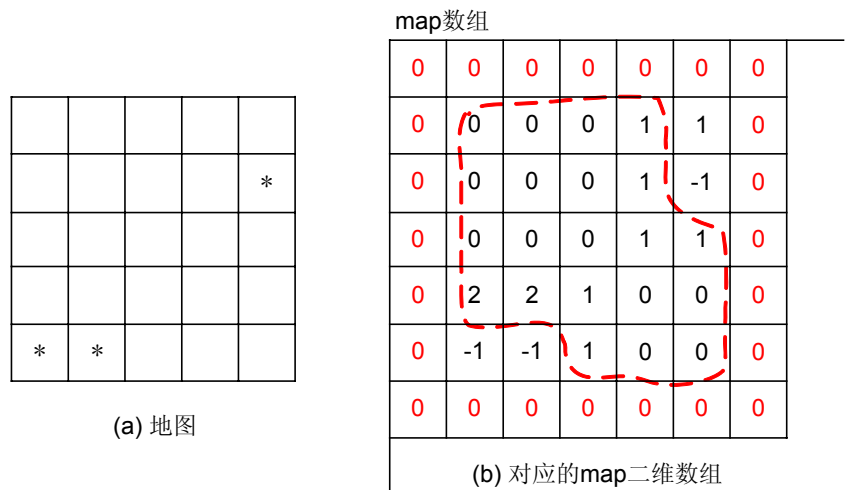
### 10.4.3 点开一片连续的没有地雷的区域

这个功能的实现需要用到 8.3 节介绍的搜索思想，search0 函数实现了该搜索功能。search0 函数的原型为：

```
void search0( int posx, int posy );
```

其中参数 posx 和 posy 表示点开的位置。

如果用户点开的位置是一个零位置，则会调用 search0 函数，search0 函数自动点开周围连续的零位置及它们的相邻位置，详见 10.2.1 中的描述。对图 10.8(a)所示的地图，当点开(1,1)这个零位置，search0 函数处理的结果如图 10.8(b)所示。



说明：在(a)所示的地图中，如果点开(1,1)这个位置，则图(b)中红色虚线内所有的位置都会自动点开，显示对应的数字。实际的扫雷游戏也是这样处理的。

图10.8 点开一片连续的没有地雷的区域

10.4.4 游戏的玩法

用户在通过这个程序玩扫雷游戏时，首先可以设置地图的大小 **N** 和地图中地雷的个数 **MineNum**。在玩游戏时，是通过输入 3 个数据表示一次操作的，格式为：**x y op**。

- x、y**：整型数据，表示操作施加到地图的(x,y)位置上。
- op**：用户可以执行的操作，有 4 种操作：
- 1) '!'：表示将(x,y)位置标记为地雷；
  - 2) ' ': 表示点开(x,y)位置；
  - 3) 'c'：取消标记，如果该位置之前已标记，该操作有效，否则该操作无效；
  - 4) 'q'：表示退出游戏。

具体玩游戏的过程是在 **playgame()** 函数中实现的。所有的操作都是在永真循环内执行的：

```
while(1)
{
    ...
}
```

当用户退出游戏、或踩到地雷、或游戏成功完成时，退出 **while** 循环。该函数的代码详见下面的测试程序。

10.4.5 测试程序

以下测试程序实现了整个扫雷游戏，读者可以对该程序建立工程，然后运行该程序。整个程序的运行过程如图 10.2 所示。

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

int N = 5;          //实际的扫雷地图是 N×N 大小的，N 的最大值为 20
//存储地图(从第 1 行,第 1 列开始表示地图;第 0 行,第 0 列不用;第 N+1 行,第 N+1 列也不用)
//map[i][j]==-1 表示地雷；map[i][j]的值在[0,8]之间，表示(i,j)位置周围 8 个位置上地雷个数
int map[22][22] = {0};
//显示给用户看的地图(记录用户的操作)，0 表示没有点开，1 表示点开，2 表示标记地雷
```

```

int usermap[22][22] = {0};
char line[100];    //在输出地图时，表示地图每一行的字符串
int MineNum = 10;    //地雷的个数
int RightMarkNum = 0; //用户正确地标记出地雷的个数
int MarkNum = 0; //用户标记出地雷的个数
//每个位置的 8 个相邻位置(从左上角位置开始按顺时针顺序列出)
int dir[8][2] = { {-1,-1}, {-1,0}, {-1,1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1} };
bool bsuccess = false; //是否成功的标志,在 Output_UserMap( )函数中判断是否游戏成功

void Output_UserMap( )    //输出用户地图
{
    RightMarkNum = 0;
    int i, j;    //循环变量
    for( i=1; i<=N; i++ )    //统计用户正确标记地雷的个数
    {
        for( j=1; j<=N; j++ )
            if( usermap[i][j]==2 && map[i][j]==-1 ) RightMarkNum++;
    }
    if( RightMarkNum==MineNum ) //游戏成功,把没点开的位置(这些位置没有地雷)全点开
    {
        bsuccess = true;
        for( i=1; i<=N; i++ )
        {
            for( j=1; j<=N; j++ )
                if( usermap[i][j]==0 ) usermap[i][j]=1;
        }
    }
    //接下来显示用户地图
    line[0] = '\40'; //'\40'表示空格
    for( i=1; i<=4*N-1; i++ )
        line[i] = '_';
    line[i] = '\0'; puts(line);    //输出第 0 行
    for( i=1; i<=N; i++ )    //输出地图中的后面 N 行
    {
        line[0] = '|';
        for( j=1; j<=N; j++ )    //每行在 line[0]后面有 N 组字符
        {
            line[4*j-3] = '_';
            if( usermap[i][j]==0 ) line[4*j-2]='_'; //没有点开
            else if( usermap[i][j] == 1 ) line[4*j-2] = 48+map[i][j]; //已经点开
            else if( usermap[i][j] == 2 ) line[4*j-2] = '!'; //标记
            line[4*j-1] = '_';
            line[4*j] = '|';
        }
        line[4*N+1] = '\0'; //串结束符标志
        puts( line );    //输出
    }
}

void Output_Map( )    //输出地图

```

```

{
    int i, j;    //循环变量
    line[0] = '\40'; //'\40'表示空格
    for( i=1; i<=4*N-1; i++ )
        line[i] = ' ';
    line[i] = '\0'; puts(line);    //输出第 0 行
    for( i=1; i<=N; i++ )          //输出地图中的后面 N 行
    {
        line[0] = '|';
        for( j=1; j<=N; j++ )    //每行在 line[0]后面有 N 组字符
        {
            line[4*j-3] = ' ';
            if( map[i][j] == -1 ) line[4*j-2] = '*'; //地雷，显示"*"
            else line[4*j-2] = 48+map[i][j]; //不是地雷，显示 8 个相邻位置上地雷的个数
            line[4*j-1] = ' ';
            line[4*j] = '|';
        }
        line[4*N+1] = '\0'; //串结束符标志
        puts( line );    //输出
    }
}

```

void Random\_Map( ) //随机生成地图

```

{
    //随机地产生 MineNum 个位置，这些位置上就是地雷
    int i=1, j, k;    //循环变量
    srand( (unsigned)time( 0 ) );
    int num1, num2;
    while( i <= MineNum ) //随机地产生 MineNum 个地雷的位置
    {
        num1 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) ); //产生 1~N 之间的随机数
        num2 = 1 + int( 1.0*N*rand( )/(RAND_MAX+1) ); //产生 1~N 之间的随机数
        //排除第 0 行,第 0 列，及重复的地雷位置
        if( num1<1 || num1>N || num2<1 || num2>N || map[num1][num2]==-1 )
            continue;
        map[num1][num2] = -1;
        i++;
    }

    //对其余位置，统计其四周 8 个位置上地雷的个数
    int mines;    //每个位置上周围雷的个数
    for( i=1; i<=N; i++ )
    {
        for( j=1; j<=N; j++ )
        {
            if( map[i][j] == -1 ) continue;
            mines = 0;
            for( k=0; k<8; k++ )
            {
                if( map[ i+dir[k][0] ][ j+dir[k][1] ] == -1 ) mines++;
            }
        }
    }
}

```



```

    }
    map[i][j] = mines;
}
}
}

```

//如果用户点开的位置上雷的个数为 0，则程序自动点开一片连续的没有地雷的区域

```

void search0( int posx, int posy )
{
    usermap[posx][posy] = 1; //点开(x,y)位置
    int dx, dy;
    for( int i=0; i<8; i++ ) //点开(x,y)周围 8 个位置
    {
        dx = posx+dir[i][0];
        dy = posy+dir[i][1];
        if( dx<1 || dx>N || dy<1 || dy>N ) continue;
        if( map[dx][dy]!=0 ) usermap[ dx ][ dy ] = 1;
    }
    for( i=0; i<8; i++ )
    {
        dx = posx+dir[i][0];
        dy = posy+dir[i][1];
        if( dx<1 || dx>N || dy<1 || dy>N ) continue;
        //如果(x,y)周围相邻位置没有地雷，且也没有点开过，则沿着这些位置递归地点开
        if( map[dx][dy]==0 && usermap[dx][dy]==0 )
            search0( dx, dy );
    }
}

void playgame( )
{
    Output_UserMap( );
    int x, y; //输入的位置
    char op; //要执行的操作
    /*
    ! 表示标记为雷
    . 表示点开该位置
    c 取消标记，如果该位置之前已标记，该操作有效，否则该操作无效
    q 表示退出游戏
    */
    printf( "Start Game!\n" );
    while( 1 )
    {
        printf( "请输入执行操作的位置及要执行的操作，格式为：x y op\n" );
        scanf( "%d%d %c", &x, &y, &op );
        if( x==0 && y==0 && op=='q' ) { printf("用户退出程序!\n"); break; } //用户结束游戏
        else if( x<1 || y<1 || x>N || y>N ) { printf("错误的操作\n"); continue; }
        else if( op=='.' ) //点开(x,y)位置
        {
            if( map[x][y]==-1 ) { printf("踩到了地雷！游戏结束！\n"); break; } //踩到地雷

```

```

        else if( map[x][y]==0 && usermap[x][y]==0 )
            search0( x, y );    //从(x,y)开始点开成片空白区域
        else usermap[x][y] = 1;
    }
    else if( op=='!' )    //标记(x,y)位置
    {
        if( usermap[x][y]==1 ) { printf("该位置已经点开过，不能再标记!\n"); continue; }
        usermap[x][y] = 2;
        MarkNum++;
    }
    else if( op=='c' && usermap[x][y]==2 )    //取消(x,y)位置的标记
    { usermap[x][y]=0; MarkNum--; }
    else { printf("错误的操作\n"); continue; }

    Output_UserMap( );

    if( bsuccess ) { printf("游戏成功！\n"); break; }    //游戏成功
    if( MarkNum<MineNum )    //提示还有多少颗地雷没标记
        printf( "还有%d 颗地雷没有标记!\n", MineNum-MarkNum );
    else if( MarkNum==MineNum && !bsuccess )//标记出 MineNum 颗雷,但有标记错误
        printf( "%d 颗地雷已经全部标记，但部分地雷标记错误，请重新考虑!\n",
MineNum );
    }
}

int main( )
{
    printf( "请输入地图的大小 N(地图是 N×N 大小的，5=<N<=20): " );
    scanf( "%d", &N );
    printf( "请输入地图中地雷的个数(地雷是随机分布的，建议个数小于 N*N/4): " );
    scanf( "%d", &MineNum );
    Random_Map( ); //随机生成地图
    playgame( );
    return 0;
}

```

#### 10.4.6 完善程序

10.4.5 节列出了字符界面扫雷游戏的代码，与 Windows 系统下的扫雷游戏相比，该代码可以在以下方面进行完善(编程难度在本教材及上一节代码的难度以内)：

- 1) 用户如果在点开某个位置时踩到地雷，该程序只是简单地提示“踩到了地雷！游戏结束！”，程序就结束了，没有显示踩到了地雷后的地图。而 Windows 系统下的扫雷游戏对这种情形的处理是：把所有的地雷都显示出来，踩到的地雷作了专门标注。字符界面的程序可以这样处理：踩到地雷时，被踩到的地雷显示为'x'，其他地雷显示为'\*'。
- 2) 对于已经点开的位置，如果再去点开，没有提示。

读者如果有兴趣的话，可以在理解上述代码的基础上，往这些方向对该程序做改进，以进一步理解该程序的思路。

## 课程设计选题例子

### 10.1 机房座位随机编排系统

实验课上，经常需要随机的给学生编排座位，如：

- 1) 《程序设计基础》课程期末考试有上机考试，班上有  $N$  个学生，老师提供了  $N$  份试卷，需要随机地给每个学生安排座位、抽取试卷。机房座位分布如图 10.9 所示。安排座位时，要隔位就座，比如第 1 排，只能在 1、3、4、6、8、10 这 6 个位置上安排学生。

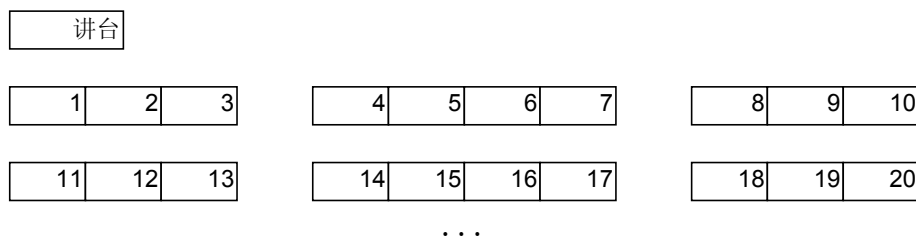


图10.9 机房座位分布图

给  $N$  个学生随机地安排座位、随机地抽取试卷的结果如图 10.10 所示。

| 学号  | 试卷号 | 座位号 | 学号 | 试卷号 | 座位号 |
|-----|-----|-----|----|-----|-----|
| 01  | 20  | 23  | 27 | 42  | 53  |
| 02  | 16  | 48  | 28 | 52  | 66  |
| 03  | 48  | 46  | 29 | 43  | 56  |
| 04  | 33  | 58  | 30 | 03  | 36  |
| ... |     |     |    |     |     |

图 10.10 机房座位随机编排系统运行界面

- 2) 《程序设计基础》课程实验课，老师希望 3 人一个团队，团队的所有成员挨着坐，以便讨论。团队和团队之间不能交错，比如图 10.9 中，第 1 排可以安排 3 个团队。而且团队的成员是随机指定的。现需要将  $N$  个学生组成若干个团队，并随机地安排团队的座位。

请编写程序，实现上述功能，并展开你的想象，设想有更多的座位编排需求，使得你的程序提供尽可能多的座位编排功能。

### 10.2 石头、剪刀、布走方格游戏

编写一程序，实现人跟电脑在  $N \times N$  大小的棋盘上玩石头、剪刀、布游戏，赢者走一格，直到某一方到达棋盘最后一个位置为止。要求：

- 1) 游戏双方一方为玩家，另一方为电脑。
- 2) 允许用户输入棋盘的大小，为简化起见，棋盘的大小是  $N \times N$  的， $N$  的值由用户从键盘输入。
- 3) 双方从棋盘上 1#方格处开始进行石头、剪刀、布游戏。石头可以打败剪刀，剪刀可以打败布，布可以打败石头；玩家从键盘输入出石头、剪刀或布，而电脑随机决定出什么。（为简化起见，分别用 1 代表石头、2 代表剪刀、3 代表布，输入 0 代表退出游戏，这点将会在输入时给予提示）。
- 4) 每进行一次石头、剪刀、布游戏就输出一次地图，显示双方所在的位置，P 代表玩家，C 代表电脑。
- 5) 赢的一方可以前进一格，而输的一方则原地不动；直到其中一方先走到终点，该方为胜利者，这一局游戏结束，并输出胜利者。
- 6) 该程序可进行多次游戏，是否进行下一局游戏由玩家决定，只需输入 Y 或 N。

该程序的某次运行示例如下：

请输入地图的大小 N(地图是 N×N 大小的, 3<=N<=15): 5

|      |  |  |  |  |
|------|--|--|--|--|
| _PC_ |  |  |  |  |
|      |  |  |  |  |
|      |  |  |  |  |
|      |  |  |  |  |
|      |  |  |  |  |

Start Game!

P 表玩家, C 代表电脑, 请输入您要出的招(1-3 分别石头、剪刀、布, 退出请按 0): 2✓

电脑出: 3

|     |     |  |  |  |
|-----|-----|--|--|--|
| _C_ | _P_ |  |  |  |
|     |     |  |  |  |
|     |     |  |  |  |
|     |     |  |  |  |
|     |     |  |  |  |

P 表玩家, C 代表电脑, 请输入您要出的招(1-3 分别石头、剪刀、布, 退出请按 0): 0✓

10.3 “万年历的查询及输出” 软件

设计一个“万年历的查询及输出”软件，具备的功能如下：

- 1) 查询某年是否是闰年；
  - 2) 查询某年某月某日是星期几；
  - 3) 输出某年的全年日历，每行输出一个月份的日历和每行并排输出 4 个月份的日历。
- 系统流程如图 10.11 所示。

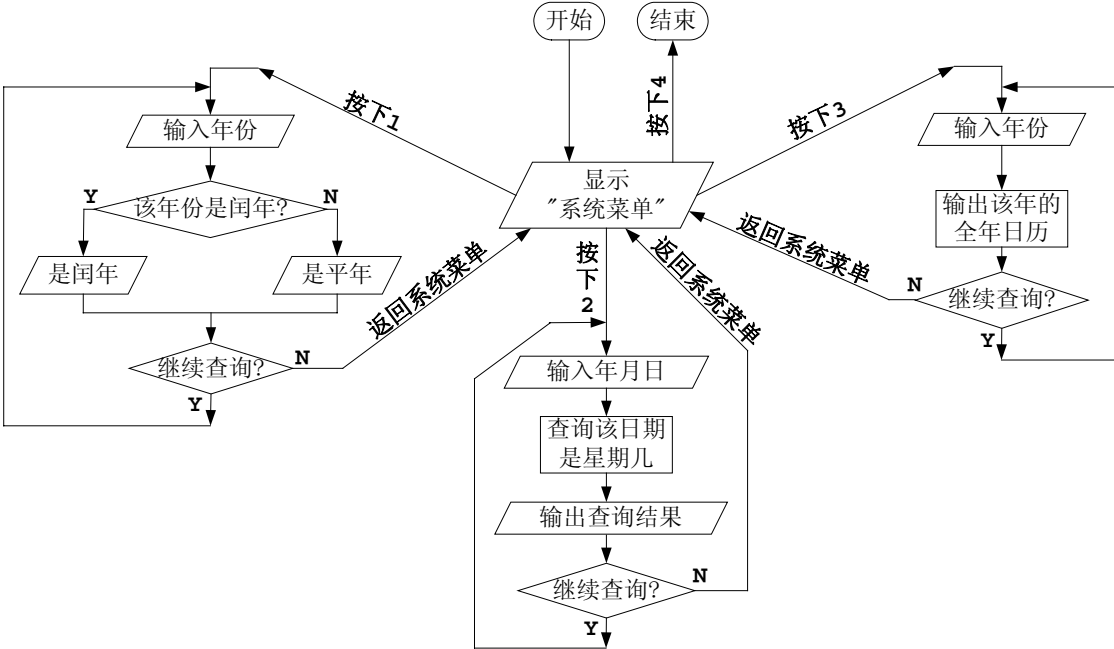


图10.11 “万年历的查询及输出” 软件流程图

## 第五篇 附录

由于篇幅的限制，很多 C/C++ 语言的知识没有在本教材的正文介绍，本附录正是为弥补这一缺憾而设置的。本附录的内容主要有两部分。

第一部分：包括附录 A～附录 C，为读者介绍在进行 C/C++ 程序设计实践时需要掌握的一些方法。特别地，附录 C 还专门为程序设计竞赛初学者介绍了在做 ACM/ICPC 题目时需要掌握的一些基本方法和技巧。

第二部分：包括附录 D～附录 G，补充介绍一些有用的、在平时学习中经常需要查阅的 C/C++ 语言知识。

另外，为了方便读者借助国内外一些 OJ 来学习 C/C++ 语言并实践本教材介绍的程序设计思想和方法，在附录 H 中还列出了本教材例题和练习题在 ZOJ、POJ 及 UVA 上的题号。

---

## 附录 A C/C++程序的编写与运行

本附录介绍 C/C++程序从编写到运行的完整过程，及 Visual C++ 6.0 开发环境的使用。

### A.1 C/C++程序编写及运行

一个 C/C++程序从编写到最后得到正确的运行结果要经历以下几个步骤。

#### 1. 在编辑器中用 C/C++语言编写程序

用高级语言编写的程序称为“源程序”(source program)。C 语言的源程序是以.c 作为后缀的，而 C++的源程序是以.cpp(英文 C++的名称“C plus plus”的简写)作为后缀的。因为 C++语言完全兼容 C 语言，所以在.cpp 文件中也可以使用 C 语言。

#### 2. 对源程序进行编译

用一种称为“编译器(compiler)”的软件(也称编译程序或编译系统)，把源程序翻译成二进制形式的“目标程序(object program)”。目标程序一般以.obj 或.o 作为后缀(object 的缩写)。

编译是以源程序文件为单位分别编译的。编译的作用是对源程序进行词法检查和语法检查。编译时对文件中的全部内容进行检查，编译结束后会显示出所有的编译出错信息。一般编译系统给出的出错信息分为两种：一种是错误(error)；一种是警告(warning)。

#### 3. 将目标文件连接

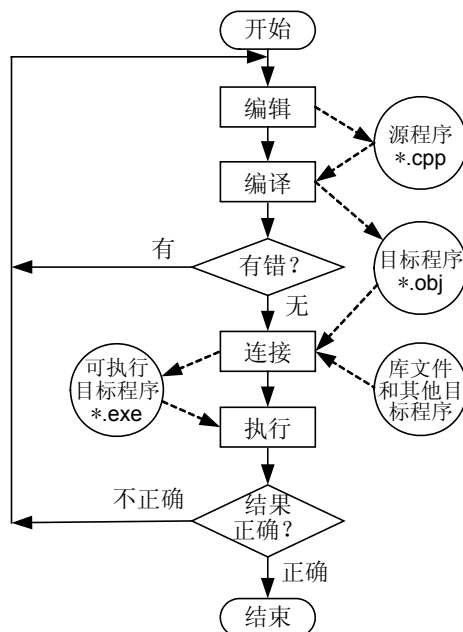
在改正所有的编译错误并全部通过编译后，得到一个或多个目标文件。此时要用系统提供的“连接程序(linker)”将一个程序的所有目标程序和系统的库文件以及系统提供的其他信息连接起来，最终形成一个可执行的二进制文件，它的后缀是.exe，是可以直接执行的。

#### 4. 运行程序

运行最终形成的可执行文件(.exe 文件)，得到运行结果。

#### 5. 分析运行结果

一个程序编写完毕，能够运行了，不一定就大功告成了。通常还要对程序的运行结果进行分析，从而得出程序是否正确的结论。如果程序运行结果不正确，则还要对程序进行分析改正等。以上过程可以用图 A.1 来表示。



图A.1 C/C++程序从编写到运行的完整过程

### A.2 Visual C++ 6.0 开发环境简介

Visual C++ 6.0(以下简称 VC)的集成开发环境(IDE, Integrated Development Environment)集成了编写、运行一个 C/C++程序所需的所有工具：编辑器、编译器、连接程序、调试工具等。本节以例 1.2 为例介绍用 VC 编写、运行一个 C/C++程序的完整过程。

### 1. 新建工程及源文件

VC 是通过项目(project, 或者称为工程)来管理一个 C/C++程序中的所有文件的, 而项目又是存放在工作空间(workspace)的。一个工作空间可以存放多个项目, 初学 VC 时一般在一个工作空间只存放一个项目。在新建源文件之前, 要先新建工程。

启动 VC 以后, 执行“File | New...”菜单命令 (注: 表示“File”菜单下的“New...”菜单命令), 在弹出的“New”对话框中选择工程的类型为“Win32 Console Application”, 以及工程名称为“1\_2”, 并选择工程的位置(假设将工程新建在“D:\C++”目录下)。如图 A.2 所示。

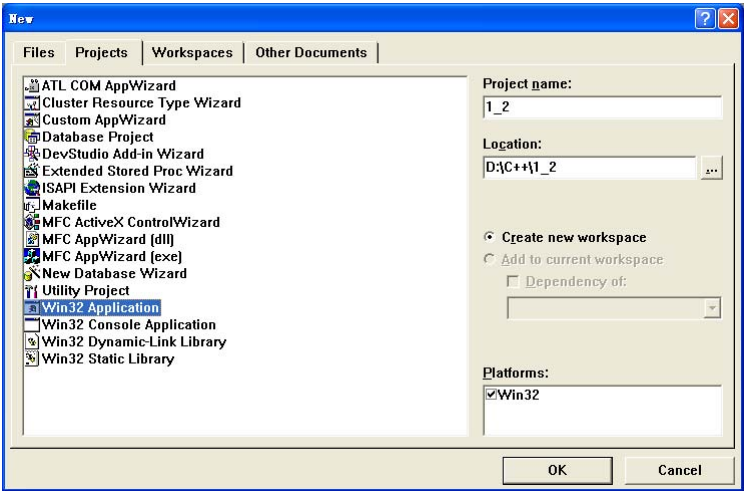


图 A.2 新建“Win32 Console Application”工程

点击“OK”以后, VC 向导会帮助我们生成工程文件, 这时有些选项需要选择, 如图 A.3 所示, 我们选择默认的“An empty project.”选项。

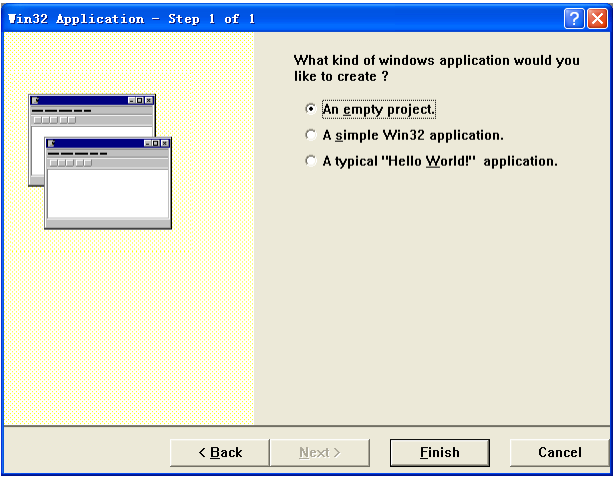


图 A.3 选择“An empty project.”选项

观察所创建的文件夹“D:\C++\1\_2\”, 有两个特殊的文件, \*.dsp 和\*.dsw 文件, 分别是项目(project, 或者称为工程)文件和工作空间(workspace)文件。如图 A.4 所示。



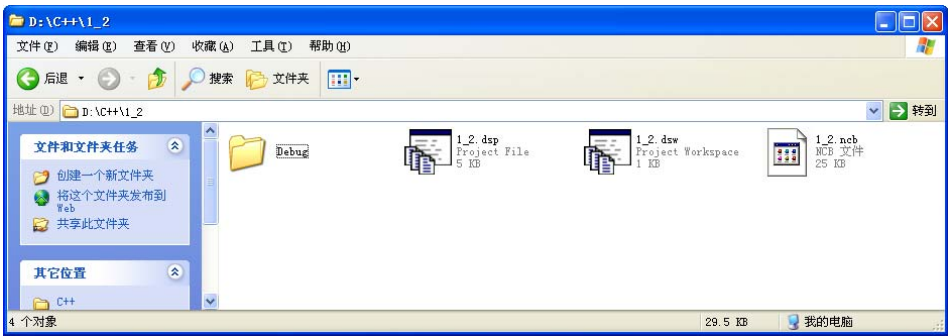


图 A.4 生成的工程文件和工作空间文件

在文件夹“D:\C++\1\_2\”中，并没有找到 C++源文件(.cpp)，所以还必须新建 C++源文件。新建方法是执行“File | New...”命令。在弹出的“New”对话框中选择文件类型为“C++ Source File”，文件名为“1\_2.cpp”，确保文件的位置为“D:\C++\1\_2”，然后确定。如图 A.5 所示。

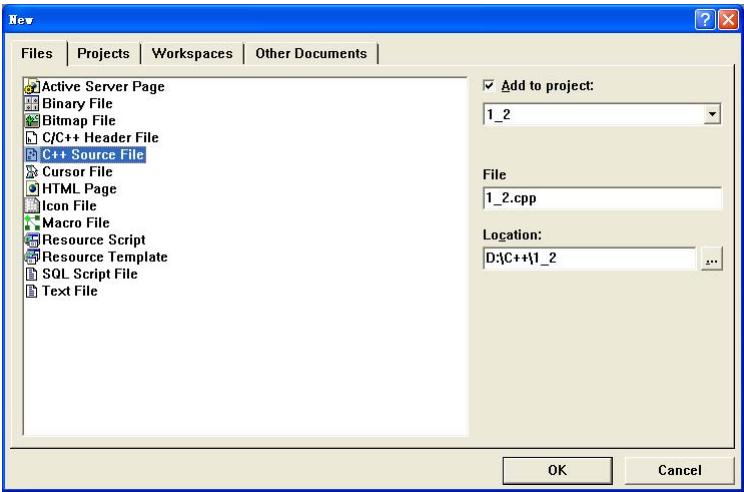


图 A.5 新建 C++源文件

这时，在 VC 中会出现一个文档窗口(这个文档窗口可以最大化，而且通常都是已经最大化了)，标题是文件名“1\_2.cpp”，如图 A.6 所示。我们是在这个文档窗口中编写源文件“1\_2.cpp”的。

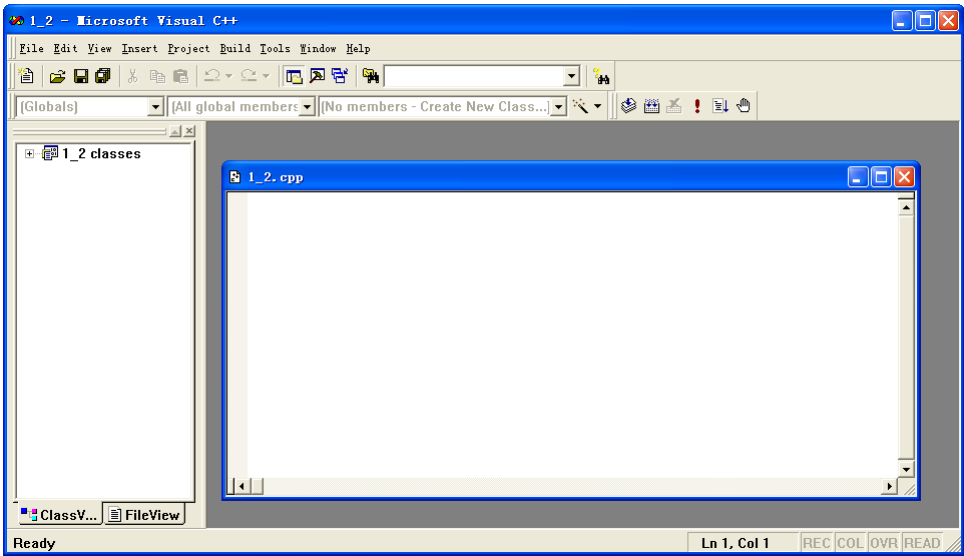


图 A.6 VC 中的源程序文档窗口

另外，我们在“D:\C++\1\_2”文件夹下还可以观察到新生成的源文件，如图 A.7 所示。

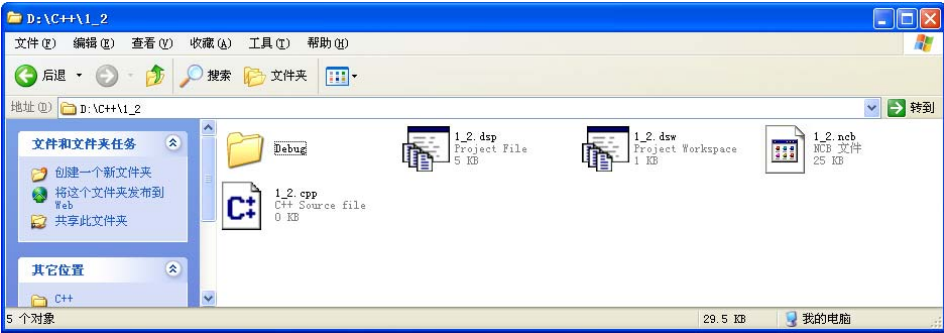


图 A.7 新生成的源文件

## 2. 编写源程序

在“1\_2.cpp”文件的文档窗口中编写源程序(注释部分可不输入), 如图 A.8 所示, 并保存。

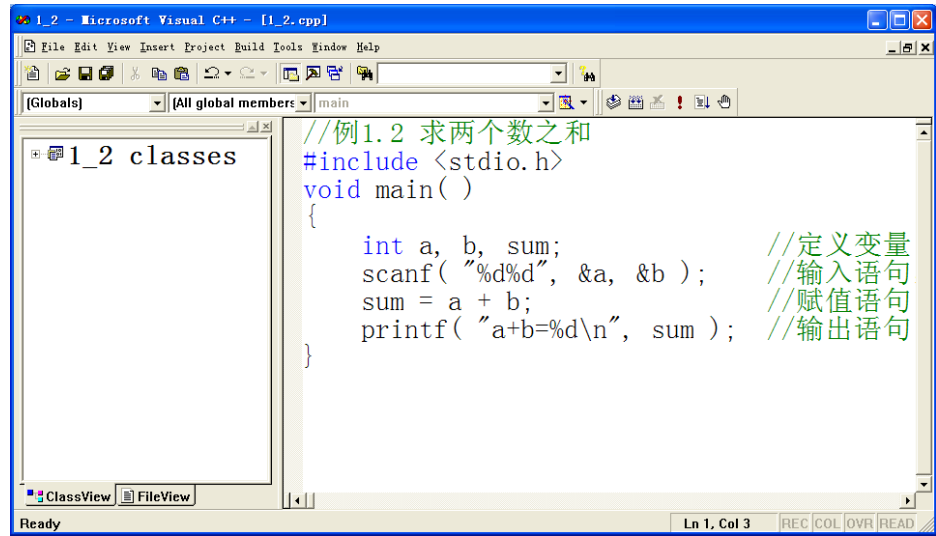



图 A.8 编写源程序

## 3. 编译

源程序编写好以后, 可以通过执行“Build | Compile 1\_2.cpp”菜单命令, 或点击“Build MiniBar(编译微型条)”工具栏上的“”按钮对源文件进行编译。编译的结果(编译错误和编译警告信息)显示在输出窗口中, 如图 A.9 所示。如果有编译错误或编译警告, 还需改正。

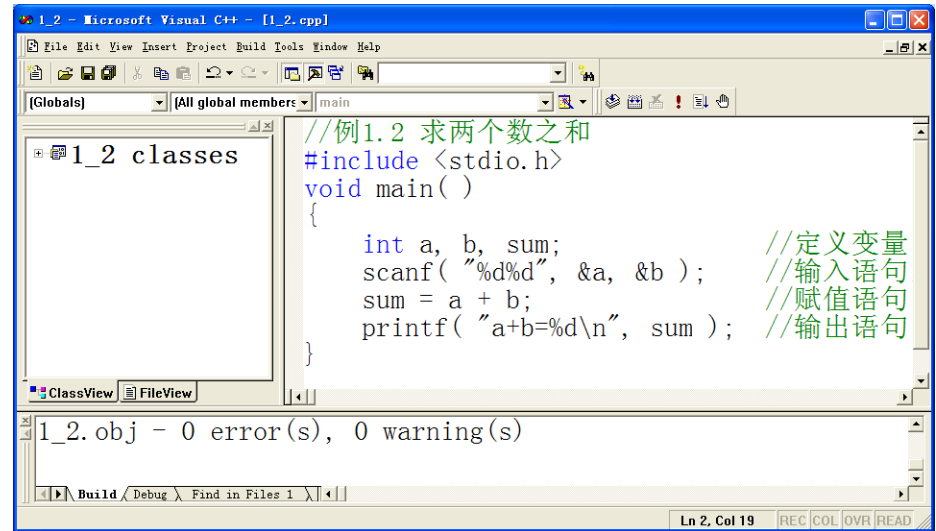



图 A.9 编译源程序


编译以后，可以在“D:\C++\1\_2\Debug”目录下观察到生成的目标文件“1\_2.obj”。

#### 4. 连接

源程序编译好以后，可以通过执行“Build | Build 1\_2.exe”菜单命令，或点击“编译微型条”工具栏上的“”按钮进行连接。连接的结果(连接错误和连接警告信息)显示在输出窗口中。如果有连接错误或连接警告，还需改正。

连接成功以后，可以在“D:\C++\1\_2\Debug”目录下观察到生成的可执行文件“1\_2.exe”。

#### 5. 运行

源程序连接好以后，可以通过执行“Build | Execute 1\_2.exe”菜单命令，或点击“编译微型条”工具栏上的“”按钮运行程序。该程序运行的过程及结果如图 A.10 所示。需要说明的是，“Press any key to continue”这一行是系统自动添加上去的，并不是程序输出的。

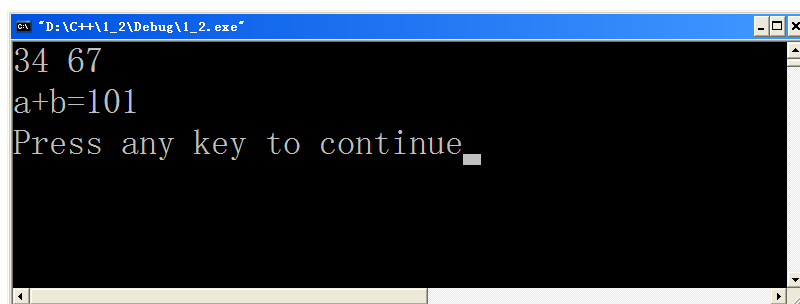


图 A.10 运行程序

另外，也可以直接执行“D:\C++\1\_2\Debug”目录下的可执行文件“1\_2.exe”来运行程序，但与在 VC 环境下运行该程序不同的是，它不会显示“Press any key to continue”这一行，而是运行完程序后马上关闭该窗口，所以用户无法观察到程序的输出结果。

## 附录 B 程序测试与调试

初学者在编写 C/C++ 程序时可能会犯很多错误，本附录介绍常见的错误原因及检测和改正这些错误的方法。

### B.1 常见的错误原因

一个 C/C++ 程序出错的原因主要有：录入错误、语言运用错误、算法逻辑错误。前两种错误编译器都能给出编译错误(详见 B.2 节)；一个程序如果编译出错，则无法执行后续的连接和运行步骤。后一种错误需要通过其他工具和方法来查找(详见 B.4 节)。

#### B.1.1 录入错误

录入错误是初学者经常犯的错误，包括：

- 1) 程序中的标识符或关键字字母遗漏、大小写误用、拼写问题；
- 2) 字符常量定界符、字符串常量定界符、各种括号左右不匹配；
- 3) 字符“o”和数字“0”混淆，字母 L 的小写字符“l”和数字字符“1”混淆。

初学者经常会犯这种错误，主要是因为指法不熟练、对 C/C++ 语言的关键字不熟悉等。

应对方法是：熟记常用的 C/C++ 语言关键字，变量命名时尽量规范。

#### B.1.2 语法错误

这类错误是因为程序中的语句违反了 C/C++ 的语法规则，比如：

```
char str[10] = 'Hello'; //错误：字符串常量的定界符为双引号
```

```
int a = 4.5%2; //错误：模运算符左右两侧的操作数都必须是整数
```

#### B.1.3 算法逻辑错误

一个没有语法错误的程序并不一定正确，即并不一定能达到正确的结果，程序中可能隐含着算法逻辑错误。

例如，判断闰年的程序，如果判断闰年的条件表示成“(year % 4 == 0 && year % 100 != 0) && year % 400 == 0”，那么很明显得出的结论是错误的，尽管该条件并不存在语法错误。

又如，初学者经常在需要用判断相等的关系运算符“==”时误用成赋值运算符“=”，尽管可能没有语法错误，但很可能会导致程序运行结果不正确。

对这一类错误通常需要借助 VC 提供的调试工具来检查。

### B.2 编译检测

VC 的编译系统能自动检测出大部分语法错误(error)，对于合法但可能会引起后续错误的潜在问题提出警告(warning)，并尽可能正确地定位错误和警告的出错位置。

#### B.2.1 编译错误

编译系统能报告程序中违反 C/C++ 语法规则的各种错误，显示错误所在的程序行号、错误代码、错误提示信息等。例如，图 B.1 所示的程序编译以后有两个错误。

VC 提供了十分方便的错误行定位手段：在输出窗口中双击一条错误，将在源程序窗口左侧的文本选定区出现一个箭头，指示错误所在行；或者连续按快捷键 F4，可以陆续找到各错误的所在行。

图 B.1 所示程序有两个错误，其出错位置都在第 5 行，错误代码均为 C2065，错误信息提示的都是“未定义标识符(undeclared identifier)”。这是因为单独一条语句“b;”和“sum;”都是合法的语句，但是由于在定义变量时只定义了变量 a，而变量 b 和 sum 都没有定义，所以提示 b 和 sum 都是未定义的标识符。很明显程序中的语句“int a; b; sum;”存在一个录入错误，正确的语句应该是“int a, b, sum;”。

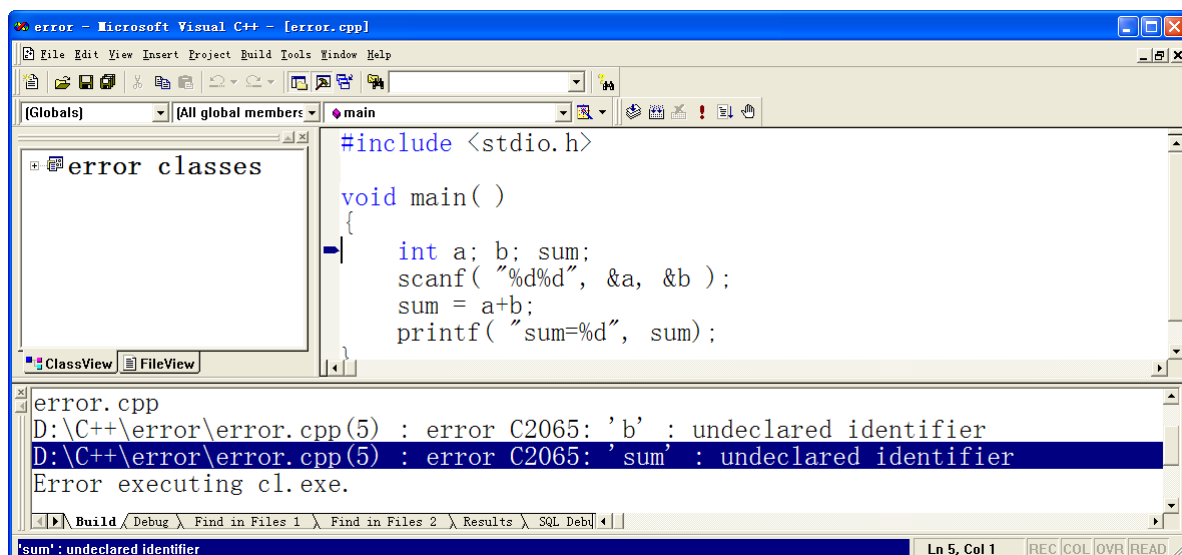


图 B.1 编译错误

## B.2.2 编译警告

编译系统除了能报告错误信息外，还能报告出“警告(warning)”信息。对于这些警告信息，一般不要轻易放过。因为这些警告信息常常能指出一些隐藏的错误。

例如，图 B.2 中的程序，编译以后报告出有 3 个警告信息。

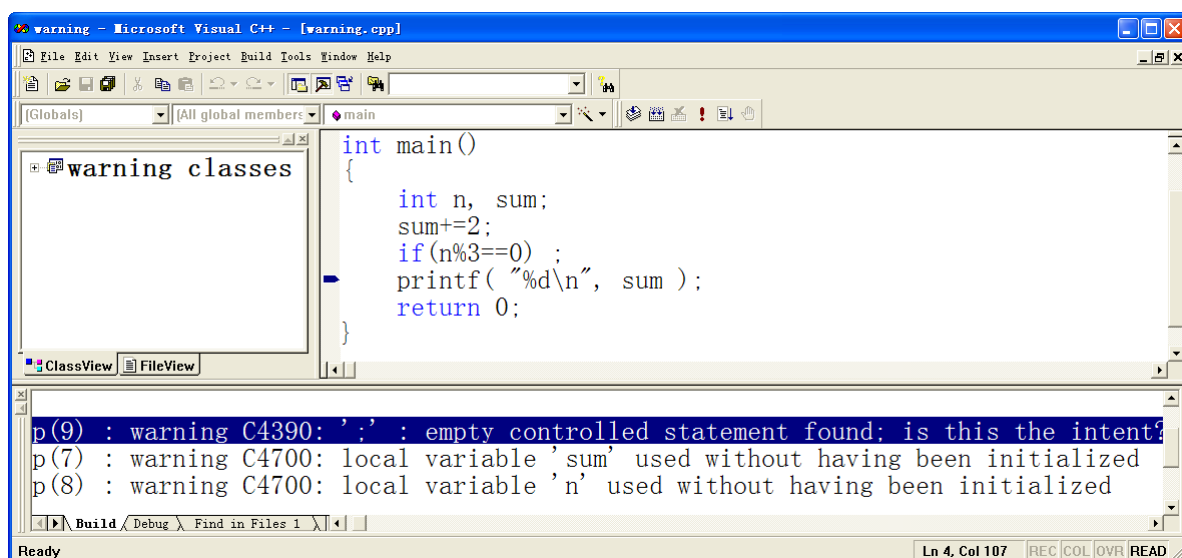


图 B.2 警告信息

第 1 个警告信息报告的位置是第 9 行，警告代码是“C4390”，提示信息表明“发现了一个

空的控制语句”，即 if 结构的执行语句是一个空语句，提示信息善意地询问用户“是否是故意的(is this the intent?)”。这个警告信息未能正确地报告出出错的位置，因为 if 语句是在第 8 行（编译系统将这条警告信息的出错位置定位在第 9 行是有道理的，因为如果第 8 行末尾的分号不是故意输入的、而是用户的录入错误，则第 9 行的语句是 if 结构的执行语句）。

第 2、3 个警告信息都是提示相关变量在使用前没有初始化。一个局部变量在定义时如果没有赋初值，则它的值是未知的。使用这些变量得到的结果也是未知的。例如程序中将 n 对 3 取余，n 的值既不是通过赋值语句赋予的，也不是从键盘上输入的，它的值是未知的，因此将 n 对 3 取余的结果也是未知的。

## B.3 程序测试

如果一个程序编写完毕，编译通过，连接成功，是不是就大功告成了呢？远没有!!! 我们还要测试该程序是否正确！

### B.3.1 程序测试的目标

对于程序设计者而言，常常希望验证自己设计的程序是正确的，因此，在测试过程中，会选择那些导致程序出错概率小的测试数据，有意或无意地回避易于暴露程序错误的测试数据。然而，如果不把着眼点放在尽可能查找错误的基础上，程序中隐藏的错误和缺陷就不容易找出来。

程序是软件的一部分，程序的测试是软件测试的一部分工作。在软件工程的教程中，通常引用 Grenford J. Myers 关于软件测试的规则：

- 1) 测试是为了发现程序中的错误而执行程序的过程；
- 2) 好的测试用例在于发现至今未发现的错误；
- 3) 成功的测试是发现了至今为止尚未发现的错误的测试。

这些规则，对于程序测试一样适用：程序测试不是要证明程序的正确，而是要检测程序的错误。

### B.3.2 测试方法

对程序的测试，一般可以通过两个方面进行：

- 1) 根据程序设计的说明，检测每个实现了的功能是否符合要求，称为**黑盒测试**。
- 2) 根据程序内部的逻辑结构，检测每个设计细节是否都符合要求，称为**白盒测试**。

接下来以例 1 为例讲解这两种测试方法，该程序中有一个逻辑错误，详见程序中的注释。

#### 例 1 黑盒测试与白盒测试

```
#include <stdio.h>
void main( )
{
    int year;
    bool leap;
    printf( "please enter year: " ); //输入提示
    scanf( "%d", &year );           //输入年份
    if( year%4==0 )                  //年份能被 4 整除
    {
        if( year%100==0 )           //又能被 100 整除
        {
            if( year%400==0 ) leap = true; //又能被 400 整除，是闰年
        }
    }
}
```

```
        else leap = false;    //非闰年
    }
    else leap = false;        //这里有错(被 4 整除但不能被 100 整除肯定是闰年)
}
else leap = false;          //年份不能被 4 整除肯定不是闰年
if( leap ) printf( "%d is ", year );    //若 leap 为真，就输出年份和“是”
else printf( "%d is not ", year );    //leap 为假，输出年份和“不是”
printf( "a leap year.\n" );    //闰年
}
```

1. 黑盒测试

黑盒测试方法把程序看成是一个封装的系统，测试只在程序接口处进行。测试过程不考虑程序内部的逻辑结构，只是根据程序设计说明书，检查程序的功能是否符合程序的设计说明。因此黑盒测试又称为**功能测试**。

例 1 程序的设计说明(功能说明)为：

输入一个闰年，如 2000，输出 2000 is a leap year.

输入一个平年，如 2001，输出 2001 is not a leap year.

分别输入 1996、2000、2001，输出结果如图 B.3 所示。从输出结果可知，该程序对某些输入数据的判断是错误的。

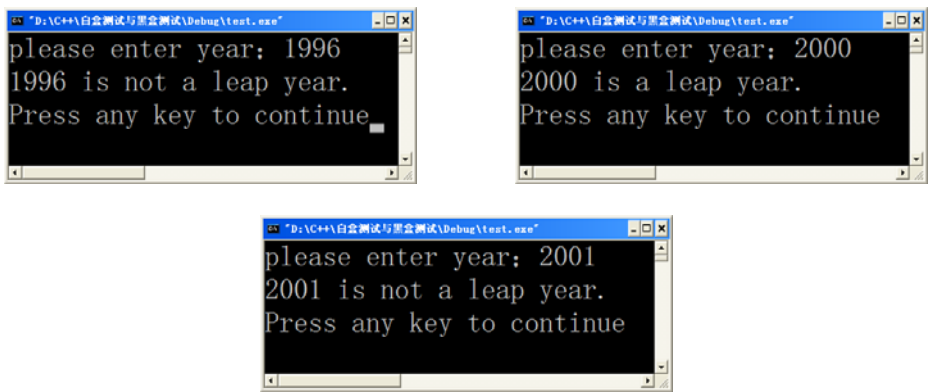
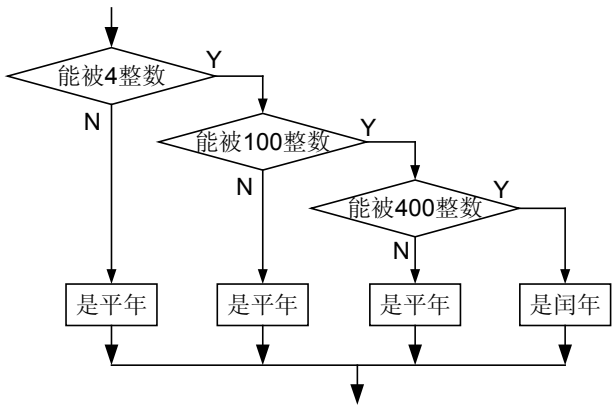


图 B.3 黑盒测试

2. 白盒测试

白盒测试方法把程序看成是一个打开的盒子，让测试人员检查程序的内部逻辑结构。测试人员要设计有针对性的测试数据，对程序的所有逻辑路径进行测试，检查程序的运行流程是否按设计要求进行工作。白盒测试又成为**结构测试**。



图B.4 例1的流程图

对例 1，通过分析该程序，可知该程序有 4 个分支，如图 B.4 所示。

- 1) 不能被 4 整除 — 平年，如 2001 年。
- 2) 能被 4 整除，但不能被 100 整除 — 平年，如 1996 年。
- 3) 能被 4 整除，但不能被 400 整除 — 平年，如 1900 年。
- 4) 能被 400 整除 — 闰年，如 2000 年。

因此，针对每个分支，输入一个年份，判断输出结果是否正确，马上可以判断第 2) 个分支的逻辑判断是错误的。

## B.4 程序调试

### B.4.1 调试的目的

考虑以下两个问题：

- 1) 如果要观察程序(特别是其中的分支结构和循环结构)的执行过程，或查看程序执行过程中某个变量的值，该怎么办？
- 2) 如果一个语法正确的程序，经测试得知程序的运行结果是错误的，想一步一步运行程序，以便找出程序的逻辑错误，该怎么办？

以上两个问题都需要借助 VC 提供的调试工具。这也是调试的目的所在。

利用 VC 提供的调试(Debug)工具，可以设置断点、单步执行程序、观察变量和表达式的值，也可以跟踪程序的执行流程，观察不同时刻变量值的变化。

### B.4.2 调试方法

使用 VC 的调试工具进行调试可按如下步骤执行：


- 1) 设置断点。断点设置方法是：将光标移动到需要设置断点的语句处，然后点击“编译微型条”工具栏上的手形按钮“”。断点的含义是程序在每次执行到该条语句处就暂停。因此应该在哪儿条语句处设置断点，应视具体题目、具体要求而定。需要注意的是：
  - a) 断点只能设置在有效行(否则 VC 会弹出如图 B.5 所示的对话框，并自动将断点移动到下一个有效行)，有效行的概念类似于执行语句的概念。
  - b) 如果断点前有输入语句，程序会在断点前的输入语句处就暂停，等待从用户键盘上输入数据。用户输入数据后才在断点处暂停。
- 2) 断点设置好以后，执行菜单命令“Build | Start Debug | Go”，或者按快捷键 F5，进入调试状态。
- 3) 进入调试状态后，可单步执行程序，也可在 Watch 窗口观察程序执行过程中变量值的变化，具体方法请参照例 2 和例 3。



图 B.5 提示对话框（提示断点只能设置在有效行）

接下来通过两个例子详细介绍 VC 调试工具的使用方法，这两个例子分别对应到上述提到的两个调试目的。

**例 2** 调试例 1.28 的程序，观察程序执行过程中变量 x0 和 x1 值的变化过程。



以下程序是正确的程序，需要借助 VC 的调试工具观察程序执行过程中相关变量的值及其变化过程，这是调试的第 1 个目的。

```
#include <stdio.h>
#include <math.h>
void main( )
{
    double a;    //输入的数据(第 5 行)
    double x0, x1;    //前后两次迭代的结果
    printf( "Please enter a positive number: a=" );
    scanf( "%lf", &a ); //输入 a，要求的是根号 a
    x0 = a/2;
    x1 = ( x0 + a/x0 ) / 2;
    //当前后两次迭代结果之差的绝对值大于 10 的-5 次方,就继续迭代
    while( fabs(x1-x0) >= 1e-5 )
    {
        x0 = x1;
        x1 = ( x0 + a/x0 ) / 2; //从前一项 x0 递推到后一项 x1
    }
    printf("The square root of %10.6f is %10.6fn", a, x1);
}
```

说明：

- 1) 如果在第 5 行或第 6 行设置断点，进入调试状态时会弹出图 B.5 所示的对话框，然后 VC 自动将断点移动到第 7 行。
- 2) 如果在第 9 行或者后面的语句处设置断点，则会因为前面第 8 行有输入语句，则会在输入语句处先暂停，等待用户从键盘上输入数据。输入数据后再执行到断点处暂停。

调试步骤：

- 1) 设置断点。将断点设置在第 9 行。此时在源程序窗口左侧的文本选定区第 9 行处出现一个红点，这是断点的标志，如图 B.6 所示。然后执行菜单命令“Build | Start Debug | Go”，或者按快捷键 F5，进行调试。

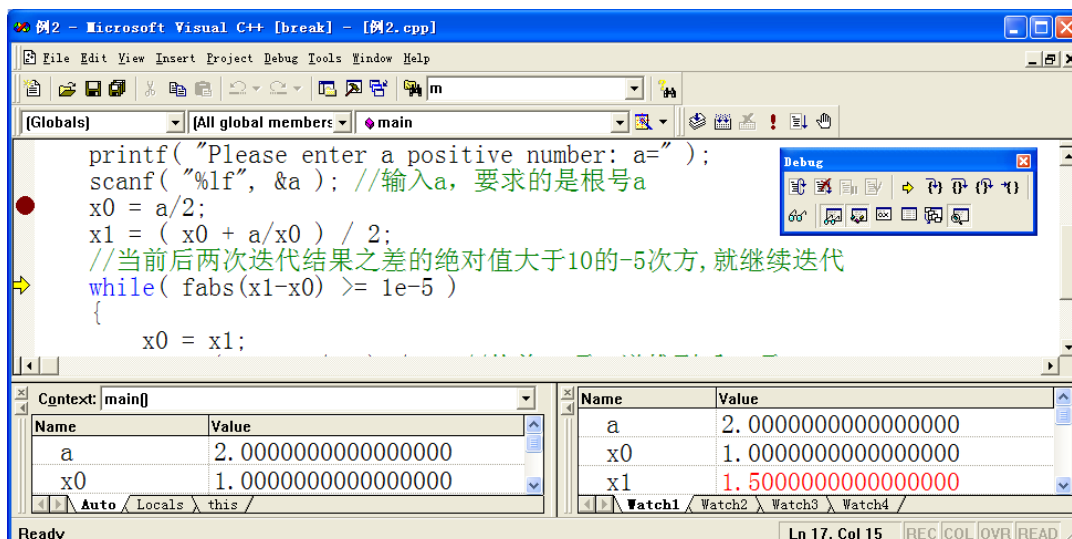


图 B.6 进入调试状态，观察相关变量的值

- 2) 输入 a 的值为 2，回车后 VC 就进入调试状态并在断点处暂停。在调试状态下，源程序窗口左侧的文本选定区出现一个亮黄色的箭头，指示当前程序将要执行的语句，如图 B.6 所示。另外，进入调试状态后，窗口菜单中新增一个菜单“Debug”，并显示“Debug”

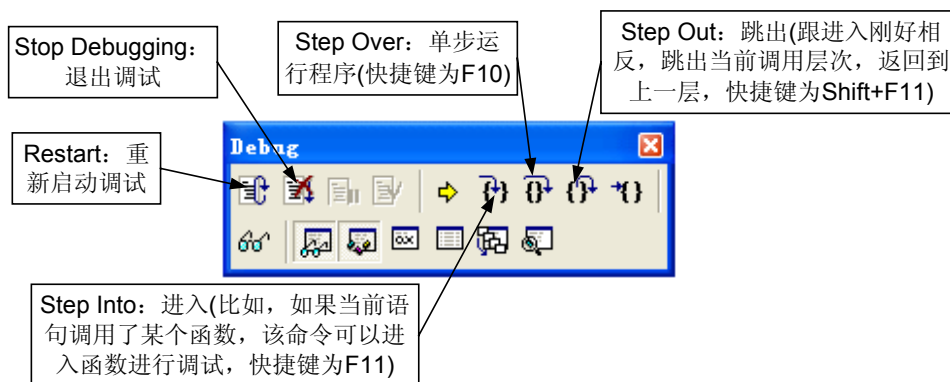
工具栏，所有有关调试的操作都可以通过执行相关菜单命令或点击“调试工具栏”上的按钮来执行。

3) 执行菜单命令“Debug | Step Over”，单步运行程序，图 B.6 是单步运行两步后的结果。在调试状态下，有两个窗口经常要用到：

- 1) Variable 窗口(图 B.6 左下角的窗口)：包含了 Auto、Locals 和 this 选项卡；常用的是 Auto 选项卡，用于显示当前行语句以及上一行语句中所使用的变量及其值。
- 2) Watch 窗口(图 B.6 右下角的窗口)：可以观察任何变量或表达式的值。

例如，图 B.6 中，在 Watch 窗口中分别输入 a、x0 和 x1，在单步运行程序的过程中，可以观察并记录这些变量的值。

在调试过程中，经常需要用到“调试(Debug)工具栏”上的按钮，如图 B.7 所示，其中简要地描述了“Debug”工具栏上常用按钮的功能。



图B.7 调试工具栏

**例 3** 以下程序是例 1.27 的程序，作了一些改动。程序没有语法错误，但输出结果是 6 7，很明显是错误的。在此，需要借助 VC 的调试工具来找出程序中隐含的逻辑错误，这是调试的第 2 个目的。

```
#include <stdio.h>
void main( )
{
    int m, i; //循环变量
    int S = 0; //累加因子的变量(第 5 行)
    for( m=6; m<=10000; m++ )
    {
        for( i=1; i<m; i++ )
        {
            if( m%i==0 ) //把 m 的因子累加起来
                S = S + i;
        }
        if( S==m ) printf( "%d ", m ); //输出完数 m
    }
}
```

在第 5 条语句处设置一个断点，然后进入调试状态，如图 B.8 所示。在 Watch 窗口中观察变量 S、m 和 i 的值。单步运行至循环变量 m 的值为 6 时，内循环执行完毕，各因子之和 S 也为 6，从而得出“m 是完全数的结论”，这是正确的。

继续单步运行，循环变量 m=7 时，在执行内循环之前，S 的值应该为 0。而在 Watch 窗口中观察到此时 S 的值为 6，如图 B.8 所示，为上一次内循环执行完毕后 S 的值。这显然是不对的。这就是程序中隐含的一个逻辑错误。

退出调试状态，在内循环前面加上一条语句：`S = 0;`。

编译修改后的程序，再运行，程序依次输出 6、28、496、8128，结果正确。

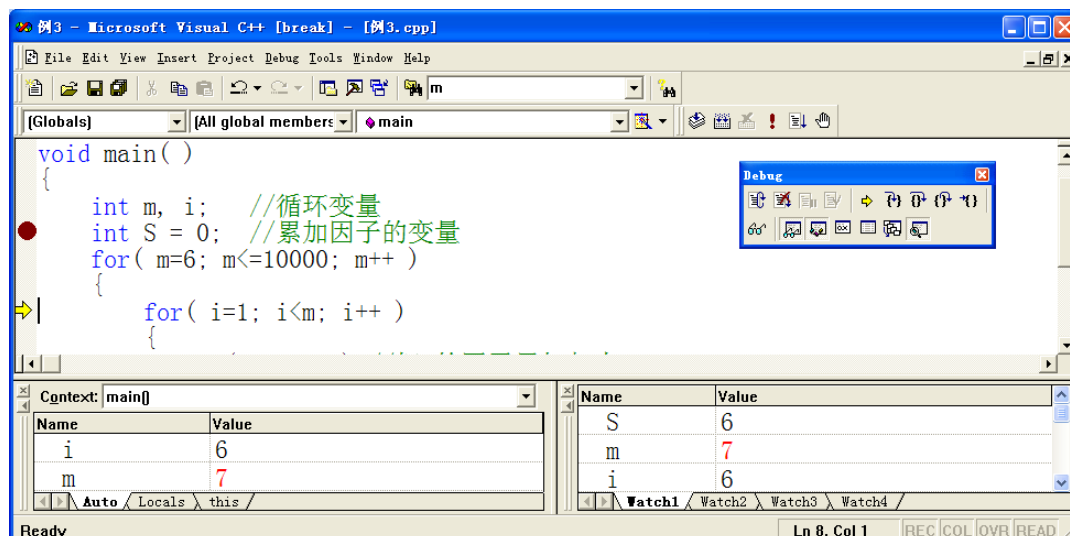


图 B.8 借助 VC 调试工具查找逻辑错误

### B.4.3 函数调用过程的调试

当一个函数(主调函数)调用另一个函数(被调函数)时，需要暂停主调函数的执行，转而去执行被调函数；执行被调函数前，先将实参的值传递给形参；执行完被调函数后，返回到主调函数中的函数调用语句处。这个过程可以通过调试方法观察到，详见下面的例 4。

**例 4** 调试例 1.34 的程序，观察函数调用的执行过程。

```
#include <stdio.h>
int max( int x, int y )      //定义有参函数 max
{
    int m;
    m = x>y ? x : y;
    return(m);
}
void main( )
{
    int a, b, c;
    printf( "please enter two integer numbers: " );
    scanf( "%d%d", &a, &b );
    c = max( a, b );        //调用 max 函数，给定实参为 a,b。函数值赋给 c(设置断点)
    printf( "max=%d\n", c );
}
```

调试步骤如下：

- 1) 在主函数的函数调用语句处设置断点，进行调试。输入两个整数 5 和 7，并在 watch 窗口观察变量 a, b, x, y 的值，如图 B.9 所示。在 Watch 窗口中，x 和 y 的值显示为"Error"，这是因为 x 和 y 是在 max 函数中定义的局部变量，而此时程序是在 main 函数中执行。
- 2) 点击调试工具栏上的"Step Into"按钮进入到 max 函数中执行。在 Watch 窗口中可以观察到 x 和 y 的值分别为 5 和 7，说明数据传递已经完成，如图 B.10 所示。同样的道理，此时 a 和 b 的值显示为"Error"。
- 3) 执行完 max 函数后，返回到主函数时，在 Variables 窗口的最后一行可以观察到 max

函数的返回值为 7，如图 B.11 所示。

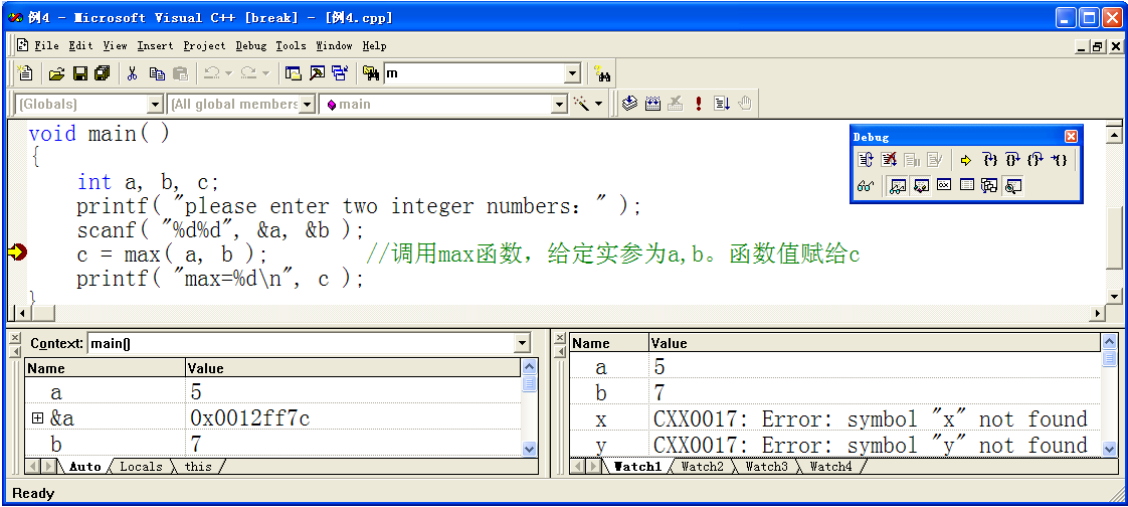


图 B.9 函数调用过程的调试：在主调函数中设置断点

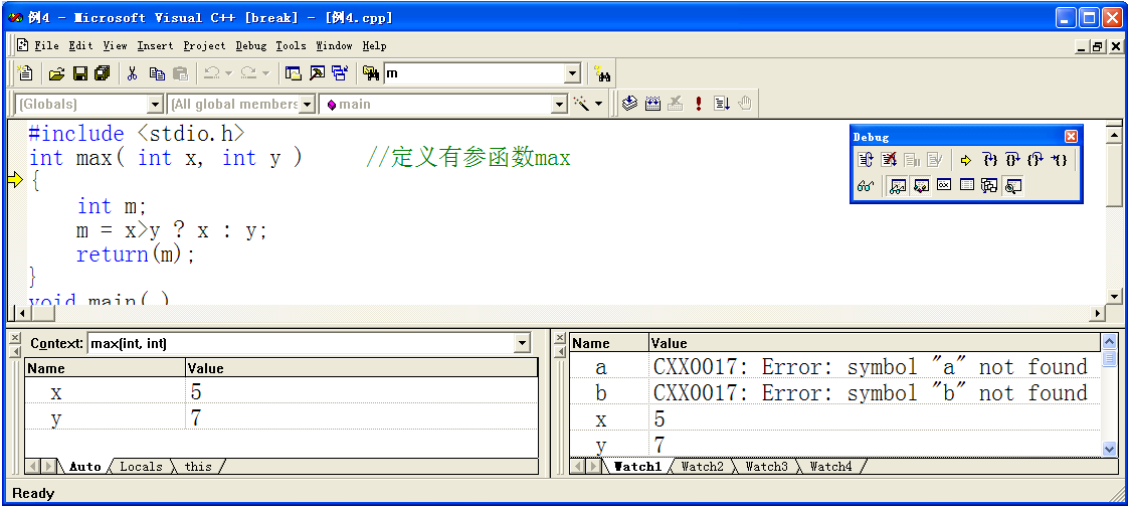


图 B.10 函数调用过程的调试：进入到被调函数中执行

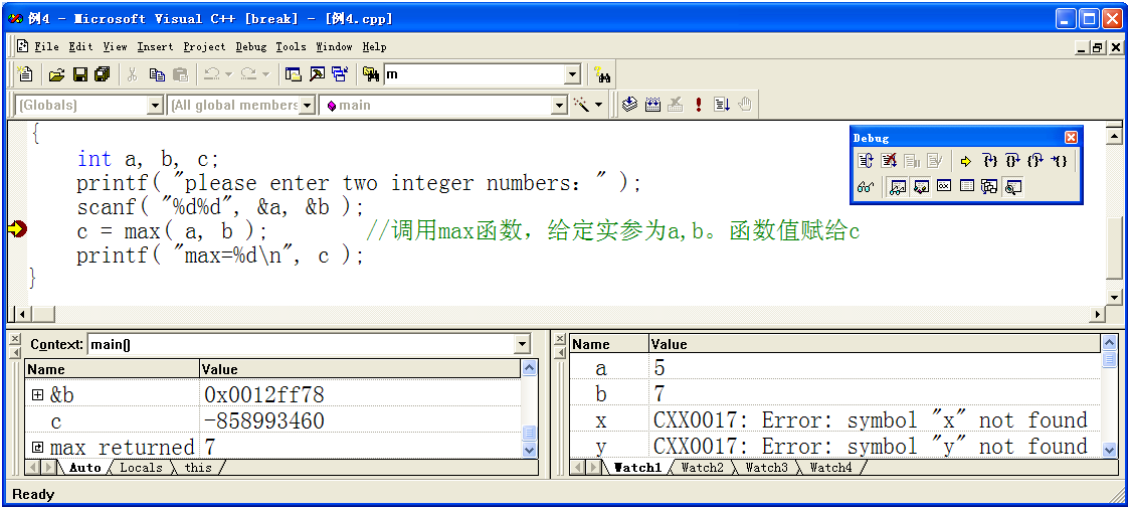


图 B.11 函数调用过程的调试：从被调函数返回到主调函数

#### B.4.4 指针程序的调试

学完 1.11 节后,读者知道,指针(即变量的地址)和指针变量在 C/C++语言中是很重要的概念。下面在例 5 中以例 1.47 的程序为例,介绍采用调试的方法来观察程序中各变量的地址及指针变量的值。

**例 5** 调试例 1.47 程序,观察程序中各变量的地址及指针变量的值,以及指针变量与所指向变量的关系。

```
#include <stdio.h>
int main( )
{
    int *p1, *p2;
    int t, a, b;
    scanf( "%d %d", &a, &b ); //输入两个整数
    p1 = &a; //使 p1 指向 a(设置断点)
    p2 = &b; //使 p2 指向 b
    t = *p1;
    *p1 = *p2;
    *p2 = t;
    printf( "a=%d, b=%d\n", a, b );
    return 0;
}
```

调试步骤如下:

- 1) 在 main 函数的“p1 = &a;”语句处设置断点,然后进行调试,输入两个整数: 57 和 43。如图 B.12 所示。在 Watch 窗口中输入 4 个表达式: a、&a、b、&b,并记录它们的值。由图 B.12 可知,在 32 位机器中,地址是 32 位的,用十六进制表示,有 8 位,例如变量 a 的地址为 0x0012ff70,前缀 0x 表示是十六进制形式。注意,在本教材的正文中,为了简化起见,一般假定变量的地址为 2000、3000 等这样的一些十进制值。

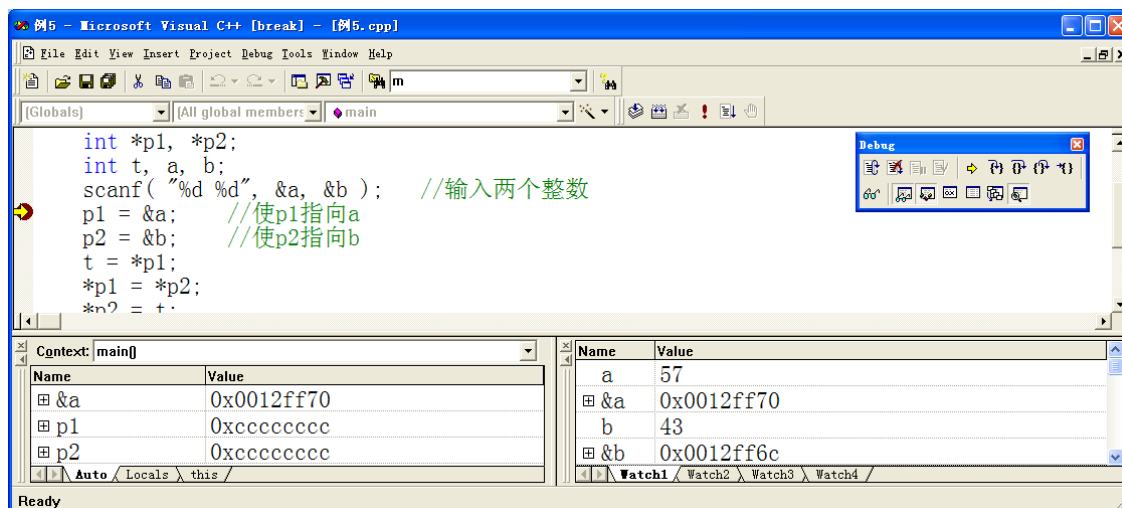


图 B.12 指针程序的调试: 变量的值与变量的地址

- 2) 单步运行两步后,在 Watch 窗口中输入 4 个表达式: p1、&p1、p2、&p2,并记录它们的值,如图 B.13 所示。其中指针变量 p1 的值为变量 a 的首地址,表达式&p1 表示指针变量 p1 本身所占存储空间的首地址。

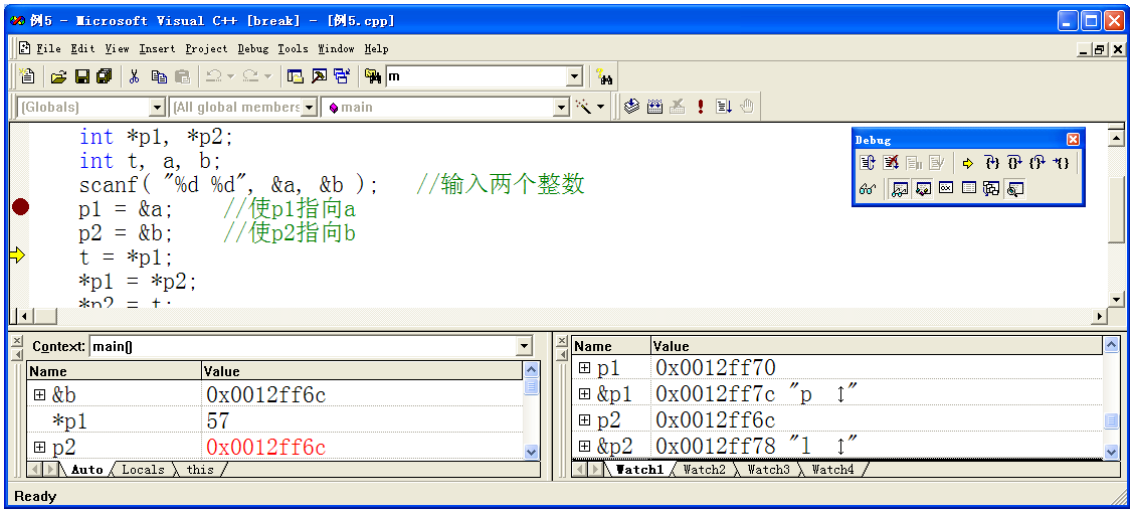
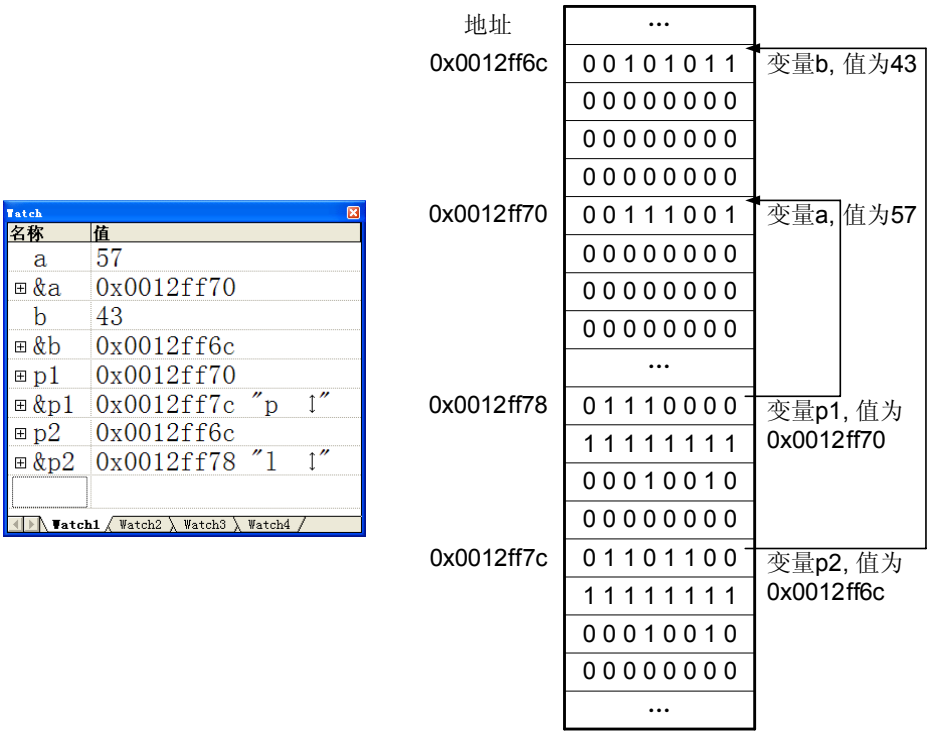


图 B.13 指针程序的调试：指针变量的值与指针变量的地址

在例 5 的程序中，对 a、b、p1、p2 这 4 个变量，根据图 B.12 和图 B.13 所显示的地址，我们可以画一幅描述这 4 个变量的值和它们的地址，以及指针变量与被指向变量之间的关系的示意图，即图 B.14。



图B.14 指针程序的调试：指针变量与被指向的变量

在图 B.14 的表格中，每一行代表一个字节，其内容为 8 位二进制。表格的左边为 4 个变量的地址，表格右边标明了 4 个变量的值。例如，变量 a 的值为 57(十进制)，对应的二进制为 111001，因为变量 a 占 4 个字节，共 32 位，所以其余 26 位均为 0。又如，变量 p1 的值为 0x0012ff70(十六进制)，其中最高两位，即“00”，为最高字节的内容，对应二进制为“00000000”；最低两位，即“70”，为最低字节的内容，对应二进制为“01110000”。从图 B.14 可以看出，这 4 个变量构成 2 对指向关系，分别为：指针变量 p1 指向变量 a，p2 指向变量 b。

### B.4.5 调试技巧

在调试过程中，经常需要掌握以下两个技巧：

- 1) 在调试时，如果希望改变某个变量的值继续调试，而不想重新调试程序的话，可以在 watch 窗口改变变量的值，继续调试。

例如，例 3 的程序在加上语句“S = 0;”前存在逻辑错误，错误地输出了 7，而其他正确的完数没有输出来。这是因为用来累加的变量 S 为上一次内循环执行完毕的值。为了验证这一点，可以在图 B.8 所示的调试状态下，在 Watch 窗口中人为地将 S 的值修改为 0，这时变量 S 的值将会以红色字体显示，如图 B.15 所示。继续调试，此后不会再输出 7。这进一步验证了逻辑错误的存在。

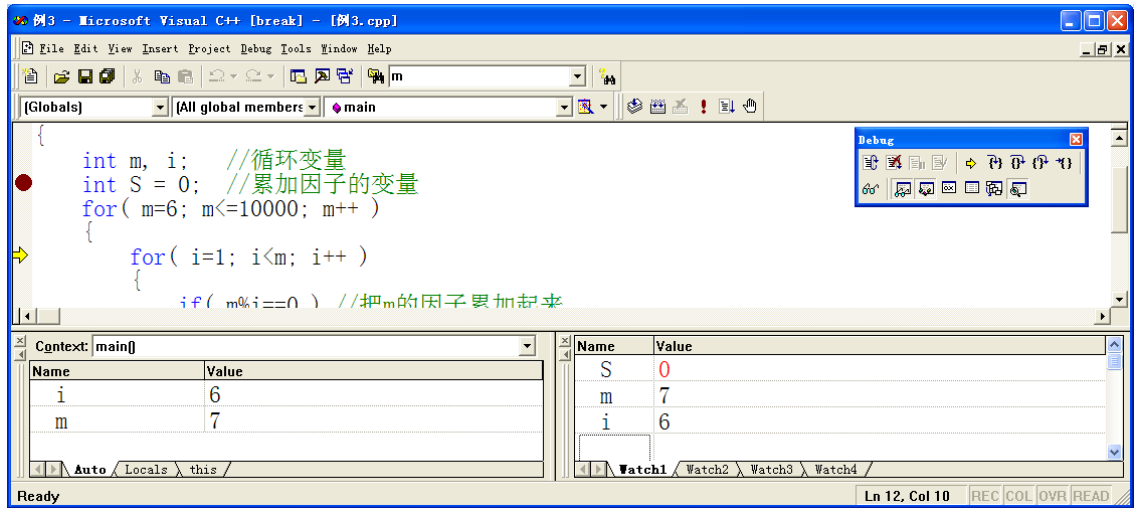


图 B.15 在 Watch 窗口中修改变量的值，继续调试

- 2) 在调试过程中，还可以继续插入断点，同样当程序执行到该断点处时，程序会自动停下来。



## 附录 C ACM/ICPC 入门指导

从编写一般的程序过渡到完成 ACM/ICPC 题目，程序设计竞赛初学者会面临很多问题。本附录介绍在做 ACM/ICPC 题目时应该掌握的一些基本方法和技巧。

### C.1 ACM/ICPC 题目的输入/输出

ACM/ICPC 题目对输入/输出的要求是极其严格的，所以程序设计竞赛初学者在做这些题目时面临的第一个问题是理解这类题目对输入/输出的要求，并正确地进行处理。

本书第 3 章详细地总结了这类题目的 4 种输入方式，以及常见的输出格式要求。程序设计竞赛初学者除了需要认真阅读第 3 章外，在平时做 ACM/ICPC 题目时还需注意积累一些技巧——实现一些特殊的输入/输出要求的技巧。

### C.2 测试程序错在哪里

初学者在做 ACM/ICPC 题目，TLE(Time Limit Exceeded, 超时)、WA(Wrong Answer, 答案错误)、PE(Presentation Error, 格式错误)无数次，历经千辛万苦最终才得到 AC(Accepted, 程序通过)，这是再常见不过了！所以不必害怕 TLE、WA、PE 等。初学者在历经无数次 TLE→WA→PE，最终收获 AC，这个过程不仅能锻炼耐心、持之以恒的毅力，更能全面掌握程序设计思想、方法、技巧！

一旦程序出错了，我们需要通过以下方法来测试程序在哪里出错了。

#### C.1.1 调试

如果程序编写完后，运行时出现以下情形，首先想到的就是调试程序：

- 1) 运行程序时，输入题目中的样例输入数据，程序运行出错、或者得不到结果；
- 2) 运行程序时，输入题目中的样例输入数据，某些测试数据的输出跟样例输出对不上。

对于这样的情形，我们需要用导致程序运行出错(或答案不对)的输入数据进行程序调试，以便找出程序中隐含的逻辑错误。所以说，调试是程序设计的基本功。关于调试方法，请参考本书附录 B，此处不赘述。

#### C.1.2 利用测试数据文件判断程序对哪些数据的处理是错误的

如果题目中的样例数据通过了，但提交程序后，得到的反馈结果是 WA，则需要更多的数据来测试程序对哪些数据的输出是错误的。这需要分两种情形来处理：

- 1) 如果有标准输入/输出文件，则可以利用这些文件来测试程序；
- 2) 如果没有标准输入/输出文件，则需要自己生成测试数据文件。

##### 1 如果有标准输入/输出文件

如果是正式的 ACM/ICPC 各预赛区的竞赛题目，赛后竞赛主办方一般会公布标准测试数据，只要找到预赛区官方网站，就能找到标准测试数据。找到标准测试数据后，则我们可以采取以下步骤来测试程序对哪些数据的处理结果是错误的。

##### 1) 文件输入/输出

要利用标准数据文件来测试程序，因为输入数据是在文件中，所以首先要将程序中的标准输



入/输出方式，改成文件输入/输出。

由标准输入/输出改成文件输入/输出，不管是 C 语言还是 C++ 语言，都有很多种方法，以下只介绍最简单的方法。

对 C 语言程序，由标准输入/输出改成文件输入/输出，只需在所有输入语句之前(一般在 main 函数的最前面)加上两行：

```
freopen( "in.txt", "r", stdin );
freopen( "out.txt", "w", stdout );
```

其中，"in.txt"和"out.txt"分别为输入文件名和输出文件名；"r"的含义是 read，表示从输入文件读数据；"w"的含义是 write，表示输出数据到输出文件。对不同的输入/输出文件，用户只需要修改文件名即可。

对 C++ 语言程序，由标准输入/输出改成文件输入/输出，只需要重新定义 cin 和 cout：

```
ifstream cin( "in.txt" );
ofstream cout( "out.txt" );
```

并包含头文件"fstream.h"，以支持文件输入/输出。

对于上述方法，如果想重新使用标准输入/输出，只需把以上两条语句注释即可。所以这种方法在文件输入/输出和标准输入/输出之间进行切换时是很方便的。

## 2) 利用标准输入文件生成输出文件

有了标准输入文件和标准输出文件后，我们可以利用标准输入文件、运行自己的程序，生成用户输出文件；接下来就是比对标准输出文件和用户输出文件，检查程序对哪些数据的输出是错误的。

例如在图 C.1 中，左图为标准输出文件(每个测试数据的输出占一行)，右图为用户输出文件。从图中可以看出，用户程序对第 9 个测试数据的输出是错误的，因此用户可以用第 9 个测试数据来调试程序，找出程序中的逻辑错误。

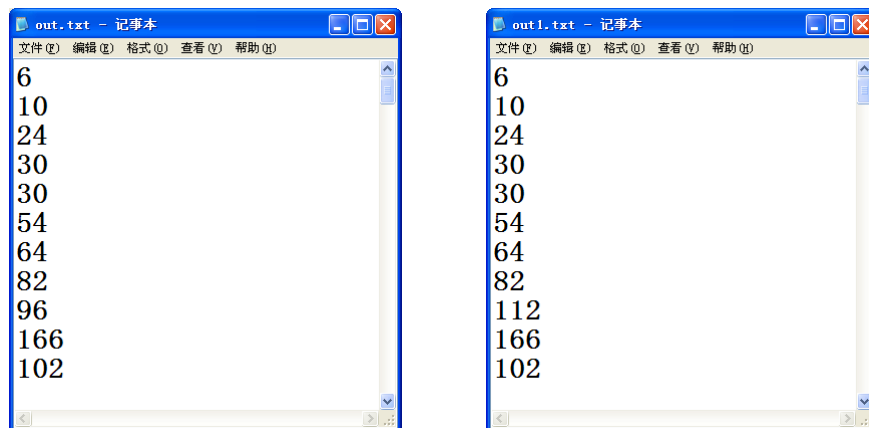


图 C.1 标准输出文件和用户输出文件

## 3) 利用 UltraEdit 快速比对用户输出文件和标准输出文件

输出文件中通常有成千上万组数据，如果需要人工去比对，很麻烦。很多文本编辑软件(如 UltraEdit 软件)有比对两个文件的功能，我们可以借助这些软件帮我们比对标准输出文件和用户输出文件。

例如，用 UltraEdit 软件比对图 C.1 中的两个输出文件，比对的结果为图 C.2。从图中可以看出，UltraEdit 软件将两个输出文件中不匹配的行清晰地标明出来了。

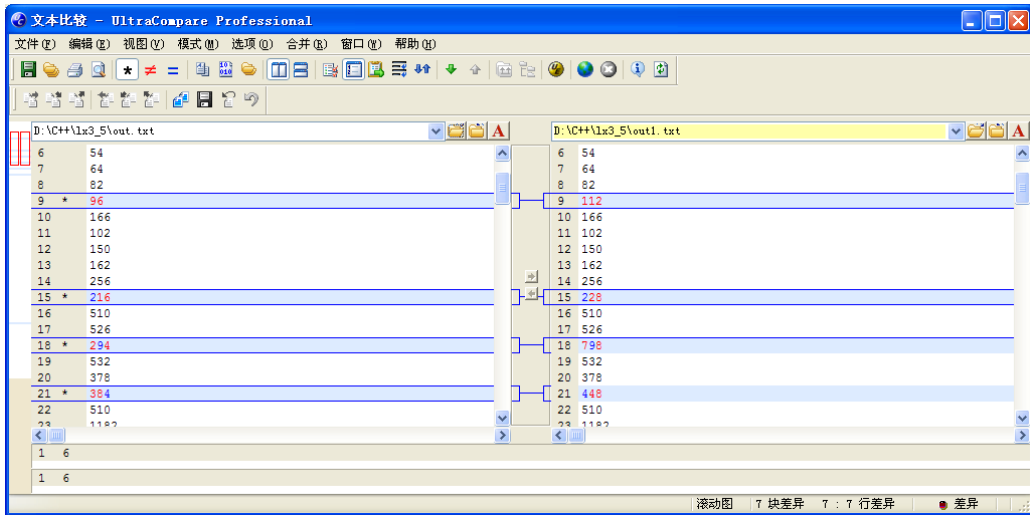


图 C.2 用 UltraEdit 软件比对标准输出文件和用户输出文件

## 2 如果没有标准输入/输出文件

如果没有标准输入/输出文件，则只能自己生成输入文件了（生成测试数据的方法见下一节）。但由于没有标准输出文件，我们只能分析程序对一些典型的输入数据产生的输出结果（必要时需要进行程序调试），如果结果不正确，则可以用这些输入数据来调试程序，以找出程序中的逻辑错误。

## C.3 测试程序运行时间及生成测试数据

### C.3.1 测试程序运行时间的必要性

ACM/ICPC 题目都是有时间限制的，一般为 1 秒钟。很多题目只有算法设计得很巧妙才能通过。当提交的程序后反馈结果为超时（如果超时，评判系统会在时间界限到了后强行终止用户程序的执行，因此用户无法得知程序的最终运行时间），则需要测试程序运行具体需要花多长时间；有时也需要比较采用不同算法编写的程序的运行时间，从而比较算法的优劣。那么如何测试程序的运行时间？

测试程序运行时间时，如果采用从键盘输入数据的方式来运行程序，所得到的运行时间大部分是输入数据所花的时间，这种时间是没有意义的。另外，评判系统在评判用户程序时也是从文件读入测试数据，以及将用户程序的输出结果写入到文件中的。所以测试程序运行时间时需要采用文件输入/输出方式。

ACM/ICPC 题目的样例数据中只有几组数据，这些数据只是用来初步测试程序正确与否。没有足够多的数据，就无法模拟实际的数据文件，测试出来的运行时间研究没有多大意义。那么如何生成足够多的测试数据？

接下来以下面的例题为例介绍生成测试数据的方法和测试采用不同算法编写的程序的运行时间。

#### 例 1 求两个正整数的最大公约数

##### 输入描述：

输入文件包含多个测试数据，每个测试数据占一行，为两个正整数  $m, n$ ， $1 \leq m, n \leq 1000$ 。输入数据一直到文件尾。

**输出描述:**

对每个测试数据中的两个正整数，输出其最大公约数。

**样例输入:**

33 18  
45 15

**样例输出:**

3  
15

**分析:**

求最大公约数有两种算法。

方法一：试除法。试除法是一种很朴素的算法：从定义出发，最大公约数 `gcd` 初始为 1，依次判断 2、3、… 能否整除 `m` 和 `n`（设  $m \geq n$ ），如果能同时整除，则更新 `gcd` 的值；循环一直到 `n` 为止。

方法二：辗转相除法。本书 8.1 节介绍了辗转相除法，此处不赘述。

**方法一的代码为:**

```
#include <stdio.h>
int main( )
{
    int m, n, t;
    while( scanf( "%d%d", &m, &n )!=EOF )
    {
        if( m<n )    //如果 m 比 n 大，则交换 m 和 n
        { t = m; m = n; n = t; }
        int i;
        int gcd = 1;
        for( i=2; i<=n; i++ ) //求最大公约数
        {
            if( m%i==0 && n%i==0 ) gcd = i;
        }
        printf( "%d\n", gcd );
    }
    return 0;
}
```

**方法二的代码为:**

```
#include <stdio.h>
int main( )
{
    int m, n, t;
    while( scanf( "%d%d", &m, &n )!=EOF )
    {
        if( m<n )    //如果 m 比 n 大，则交换 m 和 n
        { t = m; m = n; n = t; }
        int r;    //余数
        while( (r=m%n)!=0 ) //求最大公约数
        {
            m = n; n = r;
        }
        printf( "%d\n", n );
    }
}
```

```

    return 0;
}

```

### C.3.2 生成测试数据的方法

生成测试数据主要有以下三种方法：

- 1) 生成所有数据。如果题目告诉了数据的取值范围，我们可以根据这个范围生成所有可能的数据。例如，上面例 1 告诉了  $m$  和  $n$  的范围， $1 \leq m, n \leq 1000$ ，我们可以生成一个数据文件，该数据文件中包含了  $m$  和  $n$  的所有组合，即从(1 1)到(1000 1000)。代码如下：

```

#include <stdio.h>
void main( )
{
    freopen( "gcd.in", "w", stdout ); //输出数据到文件 gcd.in
    for( int m=1; m<=1000; m++ )
    {
        for( int n=1; n<=1000; n++ )
            printf( "%d %d\n", m, n );
    }
}

```

如果无法生成所有测试数据，可以考虑以下两种方法：

- 2) 如果生成数据文件仅仅是为了测试程序运行时间，则可以将现有数据复制若干份。在复制时，要掌握技巧。例如，在例 1 中，假设初始时只有以下 3 组数据，要生成一个包含 10000 组数据的数据文件，如果只是简单地复制粘贴以下 3 组数据，则要粘贴 3000 多次才能达到 10000 组数据。正确的方法是：粘贴几次后，再重新全部选择复制、粘贴若干次，再重新全部选择复制、粘贴若干次，...，这样数据量将会以几何方式增长，只需复制、粘贴很少的次数就可以达到 10000 组数据。

```

32768 32767
33 18
1024 768

```

- 3) 如果生成数据文件的目的是为了测试程序运行是否正确，则需要生成随机数据。这需要用到生成随机数的函数 `rand( )`，详见 9.3.1 节。例如，以下代码可以生成 10000 组随机的测试数据(每个测试数据有两个整数  $m$  和  $n$ )。

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
void main( )
{
    freopen( "gcd.in", "w", stdout ); //输出数据到文件 gcd.in
    srand( (unsigned)time( 0 ) );
    int m, n;
    for( int i=1; i<=10000; i++ )
    {
        m = rand( ); n = rand( ); //产生随机数
        printf( "%d %d\n", m, n );
    }
}

```

生成的数据文件如图 C.3 所示。

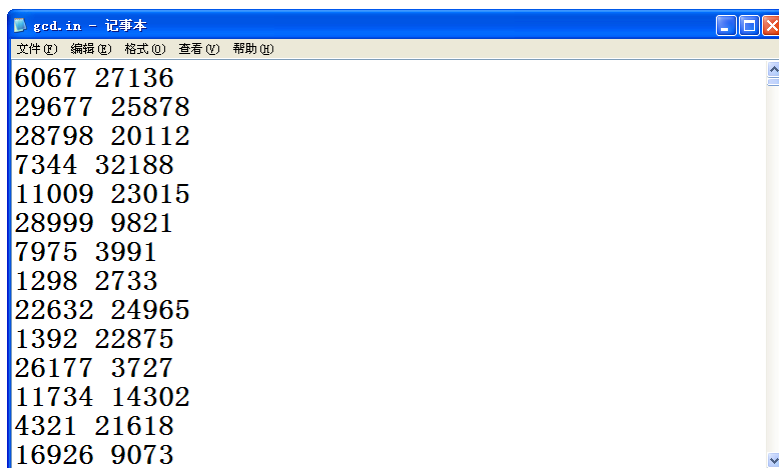


图 C.3 生成的测试数据文件

### C.3.3 测试程序运行时间的方法

测试程序运行时间有很多种方法。方法之一是使用 `clock()` 函数，该函数的功能是返回系统 CPU 的当前时间，单位是毫秒。该函数是在头文件 `time.h` 中声明的。

测试方法：在程序处理前和处理后分别测试当前时间，将这两个时间相减就是程序处理所花的时间。代码如下：

```
#include <stdio.h>
#include <time.h>
int main( )
{
    time_t time, start, end;
    start = clock( );    //处理前
    freopen( "in.txt", "r", stdin );
    freopen( "out.txt", "w", stdout );

    //...处理

    end = clock( );    //处理后
    time = end - start;
    printf( "%d\n", time );
    return 0;
}
```

上述代码执行后，程序处理所花的时间也被输出到文件 `"out.txt"` 中(在最后一行)。如果要将程序的处理结果输出到文件、将运行时间输出到屏幕上，则需要对这两种输出分别采取文件输入/输出和标准输入/输出实现，这需要使用文件指针（C 语言）或文件流（C++ 语言）实现，本书不作进一步介绍。

利用图 C.3 所示的测试数据，测试出上面例 1 中方法一的运行时间为 781 毫秒，方法二的运行时间为 15 毫秒。可见两种算法的效率差别是比较大的。

## 附录 D C/C++的输入/输出

C/C++的输入/输出方式按输入/输出对象可分为三种：标准输入/输出，文件输入/输出，以及对内存中指定的存储空间进行输入和输出(对这种输入方式，本附录不作介绍)。所谓标准输入/输出，是指以标准输入/输出设备为对象进行输入/输出；对输入设备来说，常见的就是键盘，对输出设备来说，常见的就是显示器。所谓文件输入/输出，就是以磁盘上的文件作为输出/输出对象。C 和 C++语言是采用不同的方式来实现这些输入/输出的。

### D.1 C 语言输入/输出

在 C 语言中，不管是标准输入/输出，还是文件输入/输出，都是通过函数实现的。这些函数都是在头文件中声明的，要使用这些函数必须包含相应的头文件。

#### D.1.1 标准输入/输出

在 C 语言中，标准输出是通过 `printf` 函数实现的，标准输入是通过 `scanf` 函数实现的。另外，`puts` 函数和 `gets` 函数分别可以实现字符串的输出与输入。

##### 1. printf 函数

功能：向终端(显示器)输出若干个指定类型的数据。

格式：**`printf( "格式控制", 输出列表 )`**

说明：

① 由“格式控制”部分控制后面的输出列表按指定的输出格式在显示器屏幕上输出具体的内容，因此必不可少；

② 输出列表可以由 0 到多个具体输出数据组成，其中的数据可以是常量、变量或表达式，也可以没有任何输出数据(这种形式就是下面的第一种情况)。

格式控制可分为三种情况：

**1) 不含有“%”的普通字符串。**此时输出列表中将没有输出数据，其结果是将字符串原样显示。如：

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

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

**2) 带有格式控制符的格式输出。**

格式控制符由“%”和跟随其后的一个字符构成，常用的有：`%d`、`%f`、`%c`、`%s` 等。格式控制符控制输出列表中数据的输出格式。注意：格式控制符的个数与输出数据个数应相等，且前后位置要一一对应。格式控制符中能够使用的字符及其含义如表 D.1 所示。

表 D.1 printf 函数格式控制符所包含的字符及其含义

| 格式字符 | 说 明                            |
|------|--------------------------------|
| d    | 以带符号的十进制形式输出整数(正数不输出符号)        |
| o    | 以八进制无符号形式输出整数(不输出前导符0)         |
| x    | 以十六进制无符号形式输出整数(不输出前导符0x)       |
| u    | 以无符号十进制形式输出整数                  |
| c    | 以字符形式输出，只输出一个字符                |
| s    | 输出字符串                          |
| f    | 以小数形式输出单、双精度数，隐含输出6位小数         |
| e    | 以标准指数形式输出单、双精度数，小数位数为6位        |
| g    | 选用%f或%e格式中输出宽度较短的一种格式，不输出无意义的0 |

在格式控制符中，%和后面跟随的单个字符之间又可以插入以下几种修饰符，如表 D.2 所示。

表 D.2 格式控制中的修饰符及其含义

| 字 符   | 说 明                                  |
|-------|--------------------------------------|
| 字母 l  | 用于长整型及 double 型数据，可加在 d、o、x、u、f、e 前面 |
| 正整数 m | 数据最小宽度                               |
| 正整数 n | 对实数，表示输出几位小数；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                       |

### 例 1 格式控制输出。

```
#include <stdio.h>

void main( )
{
    double d = 3.1415926;
    printf( "%6.2f", d );      //(1)
    printf( "%-6.2f\n", d );  //(2)
    long i = 1234567890;
    printf( "%d\n", i );       //(3)
    printf( "%0ld\n", i );     //(4)
    printf( "%20d\n", i );     //(5)
    printf( "%2d\n", i );      //(6)
    char ch[20] = "Hello";
    printf( "%s\n", ch );      //(7)
    printf( "%20s\n", ch );    //(8)
    printf( "%-20s\n", ch );   //(9)
}
```

[illegible]

图D.1 格式控制输出效果示意图

标准输出函数 `printf` 是在头文件 `stdio.h` 中声明的，因此如果程序中使用到该函数，就需要把头文件 `stdio.h` 包含进来。

例 1 的程序输出结果如图 D.1 所示。为了让读者更清晰地理解格式输出的效果，图 D.1 特意用表格将各个输出效果中的每个字符隔开。

程序中第(1)个 `printf` 函数输出浮点数 `d` 时, 总的宽度为 6 位, 其中小数为 2 位, 这样前面就有两个空格, 因为小数点也要占 1 个字符的宽度。

第(2)个 `printf` 函数输出浮点数 `d` 时，虽然也是 6 位，但加上了字符“`-`”，输出时是左对齐的，因此后面有 2 个空格。

第(3)个 printf 函数输出整数 i 时, 按照实际的宽度输出。

第(4)个 printf 函数输出整数 i 时，虽然加上字符“l”，但在目前的 32 位机器里，int 和 long int 类型数据都是 4 个字节，所以按 %d 和 %ld 输出来结果都是一样的。

第(5)个 printf 函数输出整数 i 时，控制输出的宽度为 20 位，因此前面有 10 个空格。

第(6)个 printf 函数输出整数 i 时,控制输出的宽度为 2 位,但实际宽度为 10 位,结果还是按照实际宽度输出。

第(7)个 printf 函数输出字符串 ch 时, 没有加上宽度控制, 按照实际的宽度输出。

第(8)个 printf 函数输出字符串 ch 时, 控制输出的宽度为 20 位, 因此后面有 15 个空格。

第(9)个 `printf` 函数输出字符串 `ch` 时，虽然也是 20 位，但加上了字符“`-`”，输出时是左对齐的，因此后面有 15 个空格。

### 3) 普通字符串与格式控制符混合使用。

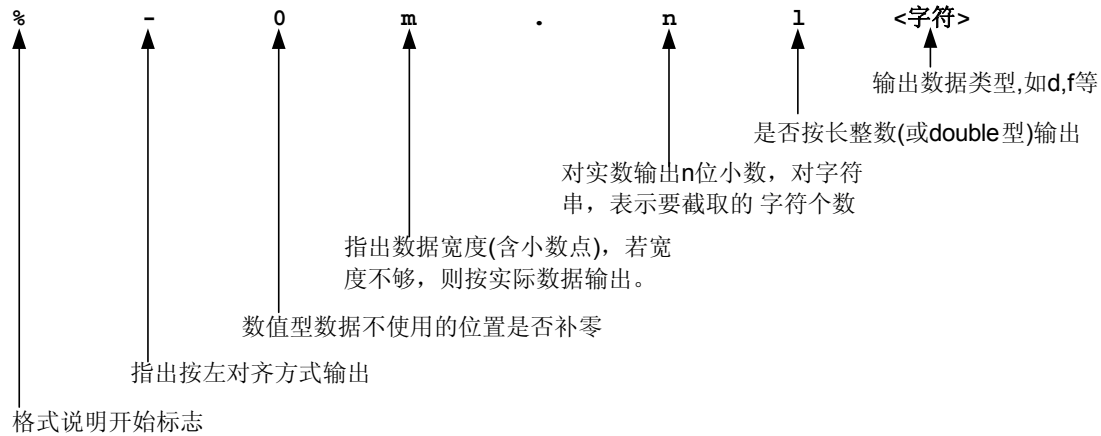
格式控制符用后面相应位置的常量、变量或表达式的值代替，其余普通字符一律原样显示。如下面的程序段：

```
void main( )
```

```
{
    printf( "2+3=%d, 2*3=%d\n", 2+3, 2*3 );
}
```

该输出的结果是：2+3=5, 2\*3=6。其中“2+3=”和“， 2\*3=”是原样输出出来的，而“5”和“6”是后面输出列表中求得的表达式的值。

printf 函数中的格式控制可归纳图 D.2(其中-、0、m、n、l 都是可以缺省的)：



图D.2 格式控制归纳

下面再举几个例子。

**例 2** printf 函数输出整数的例子。

```
printf ( "%d\n", 100 );
printf ( "%5d\n", 100 );
printf ( "%8d\n%8d", 100, 100*100 );
```

程序输出结果如图 D.3 所示。

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 |   |   |   |   |   |
|   | 1 | 0 | 0 |   |   |   |   |
|   |   |   |   | 1 | 0 | 0 |   |
|   |   |   | 1 | 0 | 0 | 0 | 0 |

图D.3 printf函数输出整数的例子

|   |   |   |   |   |  |  |  |  |  |  |  |  |  |   |   |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|---|---|--|
|   |   | A | , | A |  |  |  |  |  |  |  |  |  |   |   |  |
| N | a | m | e |   |  |  |  |  |  |  |  |  |  | N | a |  |

图D.4 printf函数输出字符型数据的例子

**例 3** printf 函数输出字符行数据的例子。

```
printf ( "%4c,%c\n", 'A', 'A' );
printf ( "%-8s,%8.2s", "Name", "Name" );
```

程序输出结果如图 D.4 所示。

## 2. scanf 函数

功能：等待用户从键盘上输入数据，然后按格式控制的要求对数据进行转换后存储到相应的变量中去。

格式：**scanf( "格式控制", 地址列表 )**

说明：

- ①、由“格式控制”部分控制输入的数据按指定的格式送到相应变量的地址单元中；
- ②、“地址列表”由若干个地址组成，可以是变量的地址或字符串的首地址，注意不能是变量名，如下面的例子。

```
scanf("%d%d%d",&a,&b,&c)    //(√)
```

```
scanf("%d%d%d", a, b, c)    //(×)
```

格式控制符：以%开始，以一个格式字符结束。常用的格式字符有：d、f、c、s等。各字符



及其含义如表 D.3 所示。

表 D.3 scanf 函数格式控制符所包含的字符及其含义

| 格式字符 | 说 明                                                                  |
|------|----------------------------------------------------------------------|
| d    | 用来输入带符号的十进制整数                                                        |
| o    | 用来输入无符号的八进制整数                                                        |
| x    | 用来输入无符号的十六进制整数                                                       |
| 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，在输入时以非空格字符开始，以第一个空格字符结束。字符串以串结束标志'\0'作为最后一个字符。 |
| f    | 用来输入实数，可以用小数形式或指数形式输入                                                |
| e    | 与f作用相同，e与f可以相互替换                                                     |

例 4 scanf 函数的使用。

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

同样，标准输出函数 `scanf` 也是在头文件 `stdio.h` 中声明的，因此如果程序中使用到该函数，就需要把头文件 `stdio.h` 包含进来。

不含其它字符的格式控制部分，在输入数据时，两个数据之间以一个或多个空格、用 `Tab` 键或回车键分隔。因此以上程序运行时，可以采用如下的输入方式：

```
35 67 14✓    或
35  67  14✓    或
35✓
67✓
14✓
```

对输入的以上数据，程序的输出为：

```
a=35,b=67,c=14
a+b+c=116
```

以上程序错误的输入方式有：

```
35, 67, 14✓
```

在%和格式字符之间可插入附加的格式说明符，含义如表 D.4 所示。请特别注意，**如果要输入数据到 `double` 型变量中，必须使用格式控制"`%lf`"。**

表 D.4 scanf 函数中附加的格式说明符及含义

| 字符     | 说 明                                           |
|--------|-----------------------------------------------|
| l      | 用于输入长整型数据（可用%ld、%lo、%lx）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）                      |
| m(正整数) | 域宽，指定输入数据所占宽度（列数）                             |
| *      | 表示本输入项在读入后不赋给相应的变量                            |

另外，对于 `scanf` 函数中除格式控制外的字符，在输入时，必须原样输入，否则程序无法正确读入数据。如下面的例子。

例 5 scanf 函数中含有其他字符。

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

对第 1 个 `scanf` 函数，正确的输入为：`i=30`。仅仅输入 `30`，是不能将 `30` 读入到变量 `i` 的。

对第 2 个 `scanf` 函数，正确的输入为：`35, 67, 14`。

要注意的是：在 `scanf` 函数中不能企图用格式控制来规定输入数据的精度。如下面的例子。

```
scanf( "%7.2f", &a )    //(×)
scanf( "%f", &a )       //(√)
```

**例 6** 读入多行字符时的注意事项。

在读入多行字符时，不管是 `scanf` 函数（使用 `%c` 格式控制符）还是 `getchar` 函数，都无法跳过上一行的“回车换行”当中的换行字符；对上一行的换行字符，必须单独读取。而在 `scanf` 函数中使用 `%s` 格式控制符读入一串字符到数组时，能自动跳过上一行的换行字符。例如，假设要读入下面的迷宫数据，其中第 1 行的两个整数表示迷宫的行和列大小，接下来有 5 行字符(每行有 6 个字符)描述了一个迷宫。

```
5 6
S...X.
..X.X.
..X..X
.....D
..XX.X
```

可以采用以下两种方法读取上面的迷宫数据：

方法一：

```
#include <stdio.h>
void main( )
{
    char map[10][10]; //存储地图
    int m, n, i, j;   //地图大小和循环变量
    scanf( "%d%d", &m, &n );
    for( i=0; i<m; i++ )
    {
        getchar( ); //读入上一行的换行字符
        for( j=0; j<n; j++ )
        {
            scanf( "%c", &map[i][j] );
        }
    }
}
```

方法二：

```
#include <stdio.h>
void main( )
{
    char map[10][10]; //存储地图
    int m, n, i;      //地图大小和循环变量
    scanf( "%d%d", &m, &n );
    for( i=0; i<m; i++ )
    {
        scanf( "%s", &map[i] );
    }
}
```

在方法一中，在读入每一行字符前，执行 `getchar( )` 函数读入上一行的换行字符，但并没有赋值给某个变量，从而跳过上一行的换行字符。

### 3. puts 函数

功能：将一个字符串输出到终端(显示器)。

格式：**`int puts( const char *string );`**

说明：

①、参数可以是字符数组的数组名，或者字符指针变量。

②、`puts` 函数可以用 `printf` 函数(使用 `%s` 格式控制)替换，但是要注意的是：`puts` 函数会在输出的字符串后面自动加上换行符，而 `printf` 函数不会。如下面的例子。

**例 7** 使用 `puts` 函数。

```
#include <stdio.h>
void main( )
{
    char *s1 = "I love China!\n";
    char s2[50] = "China Beijing 2008!\n";
    puts(s1);    //等效于 printf("%s\n",s1); puts 函数会加上换行符
    puts(s2);    //等效于 printf("%s\n",s2); puts 函数会加上换行符
}
```

程序输出结果如下：

I love China!

China Beijing 2008!

(有一空行)

#### 4. gets 函数

功能：从终端(键盘)输入一个字符串到字符指针变量所指向的存储空间。

格式：***char \*gets( char \*buffer );***

说明：参数可以是字符数组的数组名，或者字符指针变量。

**例 8** 使用 gets 函数。

```
#include <stdio.h>
void main( )
{
    char s1[50];
    char* s2 = s1;
    gets(s1);
    puts(s1);
    gets(s2);
    puts(s2);
}
```

程序输出结果如下：

I love China!✓

I love China!

China Beijing 2008!✓

China Beijing 2008!

注意：scanf( )函数在接收字符串时，以空格或回车符作为分隔符；而 gets( )函数仅以回车符作为结束符。所以输入带空格的字符串时候必须用 gets( )函数。

### D.1.2 文件输入/输出

#### 1. 文件输入/输出基本概念

在程序中除了可以以标准输入/输出设备(键盘和显示器)作为输入/输出对象外，还可以以外存中的文件作为输入/输出对象。

程序在内存运行的过程中与外存交互主要是通过以下两种方法：

- (1) 以文件为单位将内存中的数据写到外存中；
- (2) 从外存中根据文件名读取文件中的数据到内存中。

我们将“向外存写入数据”称为“输出”，“由外存读取数据”称为“输入”。

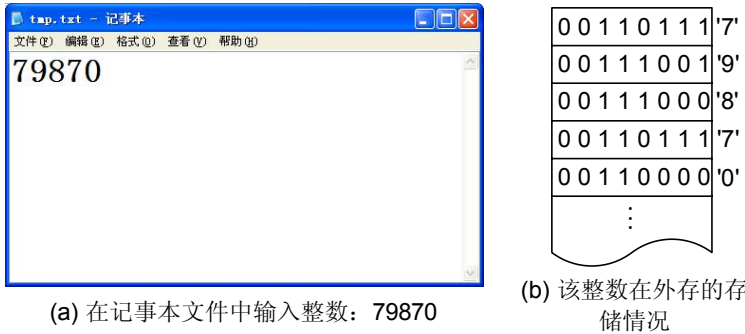
#### 2. ASCII 文件和二进制文件

根据文件中数据的组织形式，可分为 **ASCII 文件**和**二进制文件**。

### 1) ASCII 文件

如果文件中的每一个字节均以 **ASCII** 代码形式存放数据，即一个字节存放一个字符，这个文件就是 **ASCII 文件**(或称**字符文件**)。**ASCII** 文件中的数据与显示出来的内容中的字符是一一对应的，一个字节代表一个字符。

例如在记事本中输入一个整数：**79870**，这个整数包含了 **5** 个字符，每个字符以 **ASCII** 码形式存储在文件中，如图 **D.5** 所示。因此记事本文件的存储格式就属于 **ASCII 文件**。



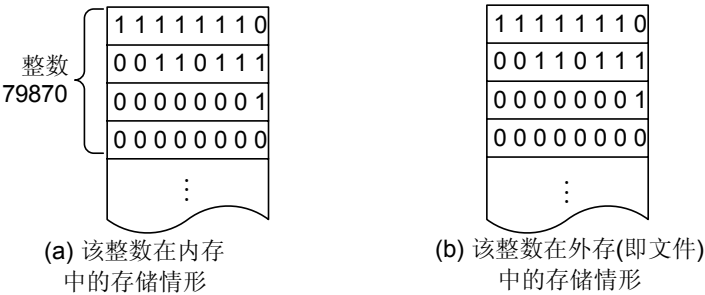
图D.5 ASCII文件

**ASCII** 文件格式的特点是：方便、直观、占用存储空间较多，存储时需要转换。

### 2) 二进制文件

二进制文件格式是直接将内存中的内容存储到外存的文件里。因为数据在内存中是以二进制形式存储的，因此这种文件格式称为**二进制文件**。另外，二进制文件中的信息不是字符信息，而是字节中的二进制形式的信息，因此又称为**字节文件**。

如果在内存中有一个整数：**79870**，则这个整数在内存中占 **4** 个字节，而且是以二进制形式存储的，即为：**00000000 00000001 00110111 11111110**。将这个整数以二进制形式写入到文件中，是直接将内存中的内容存放到文件中的。如图 **D.6** 所示。



图D.6 二进制文件

二进制文件格式的特点是：节省存储空间，存储时不需要转换，但不能直接显示文件的内容。

## 3. 文件的打开、关闭

使用文件输入/输出时，必须先打开文件，输入/输出完毕后，再关闭文件。

### 1) 打开文件

格式：**File\* fopen( "文件名", "使用方式" )**

说明：**fopen** 函数返回值为 **FILE** 类型的指针(称为文件指针)，如果打开文件失败，返回值为 **NULL**。在调用 **fopen** 函数前必须定义一 **FILE** 类型的指针，见下面的代码段。

```
FILE* p; //FILE 必须大写
p = fopen( "design.txt", "w" ) //以写方式打开 ASCII 文件 “design.txt”
if( p==NULL ) //判断是否成功打开文件
```

```
{
    printf( "cannot open file!\n" );
    return ;
}
```

在打开文件时，必须指定文件的使用方式。在 **fopen** 函数中可以采用的文件使用方式及其含义如表 D.5 所示。

表 D.5 文件使用方式及其含义

| 符号  | 文件使用方式     | 含 义                    |
|-----|------------|------------------------|
| r   | 文本文件读      | 为输入打开一个文本文件            |
| w   | 文本文件写      | 为输出打开一个文本文件            |
| a   | 文本文件写(追加)  | 向文本文件尾添加数据             |
| rb  | 二进制文件读     | 为输入打开一个二进制文件           |
| wb  | 二进制文件写     | 为输出打开一个二进制文件           |
| ab  | 二进制文件写(追加) | 向二进制文件尾增加数据            |
| r+  | 文本文件读      | 为读/写打开一个文本文件           |
| w+  | 文本文件写      | 为读/写建立一个新的文本文件         |
| a+  | 文本文件写(追加)  | 为读/写打开一个文本文件，在文件尾添加数据  |
| rb+ | 二进制文件读     | 为读/写打开一个二进制文件          |
| wb+ | 二进制文件写     | 为读/写建立一个新的二进制文件        |
| ab+ | 二进制文件写(追加) | 为读/写打开一个二进制文件，在文件尾添加数据 |

2) 关闭文件

格式: **fclose( 文件指针 )**

调用形式如: **fclose(p)**，其中 **p** 是之前已经定义好的一个 **FILE** 型的指针，且在前面已经通过 **fopen** 函数打开了某个文件。

4. ASCII 文件的输入/输出

ASCII 文件的输入/输出方式跟标准输入/输出方式完全类似，输出采用的是 **fprintf** 函数，输入采用的是 **fscanf** 函数，这两个函数跟 **printf** 函数和 **scanf** 函数的唯一差别只是多了一个参数，需要指定文件指针。

1) **fprintf** 函数—ASCII 文件输出函数

功能: 向文件输出若干个指定类型的数据。

格式: **fprintf( 文件指针, "格式控制", 输出列表 )**

说明:

- ①、**fprintf** 函数的第一个参数为文件指针，即在之前必须定义一个 **FILE** 类型的指针变量，并且通过 **fopen** 函数打开一个文件。具体使用方法见例 9。
- ②、**fprintf** 函数的其他两个参数含义同 **printf** 函数。

2) **fscanf** 函数—ASCII 文件读入函数

功能: 从文件中读入数据。

格式: **fscanf( 文件指针, "格式控制", 地址列表 )**

说明:

- ①、**fscanf** 函数的第一个参数为文件指针，即在之前必须定义一个 **FILE** 类型的指针变量，并且通过 **fopen** 函数打开一个文件。具体使用方法见例 9。
- ②、**fscanf** 函数的其他两个参数含义同 **scanf** 函数。

例 9 **fscanf** 函数和 **fprintf** 函数使用例子。

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

```

int main( )
{
    double d1 = 3.1415926, d2;
    FILE* p;
    p = fopen( "design.txt", "w" ); //以写方式打开 design.dat 文件
    if( p==NULL )
    {
        printf("cannot open file!\n"); return 1;
    }
    fprintf( p, "%10.5f", d1 ); //往文件写入数据
    fclose(p); //关闭文件
    p = fopen( "design.txt", "r" ); //以读方式打开 design.dat 文件
    if( p==NULL )
    {
        printf("cannot open file!\n"); return 1;
    }
    fscanf( p,"%lf", &d2 ); //从文件读入一个浮点数到 d2
    printf( "d2=%f\n", d2 ); //输出到屏幕上
    fclose(p); //关闭文件
    return 0;
}

```

该程序的输出为：d2=3.141590。

在本程序中，使用 `fprintf` 函数向文件 “design.txt” 中写入一个浮点数，并使用了格式控制，使得该浮点数占 10 位，其中小数点后 5 位，这样前面就有 3 个空格(运行该程序后，可以从当前目录下找到 “design.txt” 文件，打开该文件查看其内容，如图 D.7 所示)。从该文件读入浮点数到变量 `d2`，最终将 `d2` 的值以标准输出方式输出到显示器(小数点后有效数字位数为默认的 6 位)。

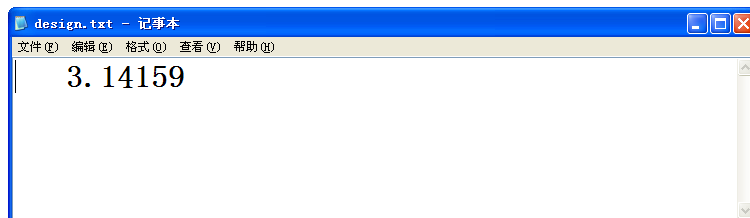


图 D.7 程序生成的 ASCII 文件

## 5. 二进制文件的输入/输出

前面已提及 ASCII 文件的输入/输出方式跟标准输入/输出方式很类似，但二进制文件的输入/输出方式差别就比较大了。二进制文件输出采用的是 `fwrite` 函数，二进制文件输入采用的是 `fread` 函数。

### 1) fwrite 函数—二进制文件输出函数

功能：向指定文件以二进制方式写入若干个字节的数据。

格式：**`fwrite( buffer, size, count, fp )`**

说明，各参数的含义为：

**buffer**：待输出数据的起始地址；

**size**：要写入到文件中的字节数；

**count**：要写多少个 **size** 字节的数据；

**fp**：文件型指针。

`fwrite` 函数具体使用方法见例 10。

## 2) fread 函数—二进制文件读入函数

功能：从指定的二进制文件中读入若干个字节的数据。

格式：**fread( buffer, size, count, fp )**

说明，fread 函数各参数的含义类似于 fwrite 函数，具体为：

**buffer**：读入数据的存放地址，指的是起始地址；

**size**：要从文件中读入的数据项占多少个字节；

**count**：要读入多少个 size 字节的数据；

**fp**：文件型指针。

fread 函数具体使用方法见例 10。

**例 10** fwrite 函数和 fread 函数使用例子。

```
#include <stdio.h>
int main( )
{
    int a1 = 79870, a2;
    FILE *fp;
    fp = fopen( "design.dat", "wb" ); //以写方式打开二进制 design.dat 文件
    if( fp==NULL )
    {
        printf( "cannot open file!\n" ); return 1;
    }
    fwrite( &a1, sizeof(a1), 1, fp ); //将整数 a1 以写入到二进制文件中
    fclose( fp ); //关闭文件
    fp = fopen( "design.dat", "rb" ); //以读方式打开 design.dat 文件
    if( fp==NULL )
    {
        printf( "cannot open file!\n" ); return 1;
    }
    fread( &a2, sizeof(a2), 1, fp ); //从二进制文件读入一个整数到 a2
    printf( "a2=%d\n", a2 ); //输出到屏幕上
    fclose(fp); //关闭文件
    return 0;
}
```

该程序的输出为：a2=79870。

本程序生成的二进制文件“design.dat”，如果用记事本打开，只能看到乱码，不能看到整数“79870”，这说明二进制文件中的内容不是字符信息，而是数据的二进制形式。如果要查看该文件每个字节的内容，可以采用 UltraEdit 等文本处理软件。图 D.8 以十六进制方式显示出该二进制文件的 4 个字节，从低到高内容分别为：FE 37 01 00。这正是整数“78970”的十六进制形式。

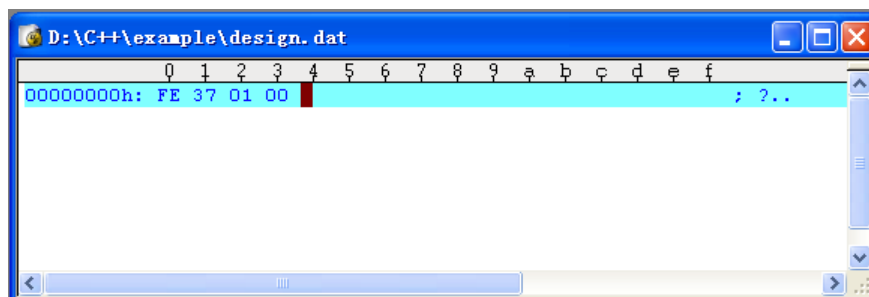


图 D.8 程序生成的二进制文件

## D.2 C++语言输入/输出

C++语言的输入/输出方式是采用流对象来实现的，由于本教材不涉及面向对象的内容，另外C++语言的文件输入/输出方式也比较复杂，超出了本教材的难度，所以本节只简单地介绍C++语言标准输入/输出方式的使用方法。

C++语言的标准输入是采用输入流对象 `cin` 来实现的，标准输出是采用输出流对象 `cout` 来实现的。一般可以把使用 `cin` 和 `cout` 对象进行输入/输出的语句简单地称为 `cin` 语句和 `cout` 语句。

`cout` 语句的一般格式为：

**`cout <<表达式 1 <<表达式 2 <<…… <<表达式 n;`**

`cin` 语句的一般格式为

**`cin >>变量 1 >>变量 2 >>…… >>变量 n;`**

要使用 `cin` 和 `cout` 对象进行输入/输出，必须包含头文件 `<iostream.h>`。C++语言的标准输入/输出方式使用方法见例 11。

**例 11** C++语言的标准输出/输出。

```
#include <iostream.h>
int main( )
{
    char c1,c2;
    int a;
    double b;
    cin >>c1 >>c2 >>a >>b;
    cout <<"c1=" <<c1 <<" ,c2=" <<c2 <<" ,a=" <<a <<" ,b=" <<b <<endl;
    return 0;
}
```

该程序的运行示例 1：

h i 24 45.12349 ✓

c1=h,c2=i,a=24,b=45.1235

说明：使用 C++的输入方式，在输入数据时，也是以空格、Tab 键或回车分隔数据的，对于如上的输入，系统会提取第一个数据即字符 ‘h’ 给字符变量 `c1`，提取第二个字符 ‘i’ 给字符变量 `c2`，再提取 24 给整数变量 `a`，最后提取 45.12345 给浮点型变量 `b`。输出时，对变量 `b`，(默认) 只输出了 6 位有效数字。

该程序的运行示例 2：

1234 56.78 ✓

c1=1,c2=2,a=34,b=56.78

说明：对于如上的输入，系统会提取第一个字符 ‘1’ 给字符变量 `c1`，提取第二个字符 ‘2’ 给字符变量 `c2`，再提取 34 给整数变量 `a`，最后提取 56.78 给浮点型变量 `b`。



## 附录 E ASCII 编码表

| ASCII 值 | 控制字符 | 含义     | ASCII 值 | 字符      | ASCII 值 | 字符 | ASCII 值 | 字符  |
|---------|------|--------|---------|---------|---------|----|---------|-----|
| 000     | NULL | 空字符    | 032     | (space) | 064     | @  | 096     | `   |
| 001     | SOH  | 标题开始   | 033     | !       | 065     | A  | 097     | a   |
| 002     | STX  | 正文开始   | 034     | "       | 066     | B  | 098     | b   |
| 003     | ETX  | 正文结束   | 035     | #       | 067     | C  | 099     | c   |
| 004     | EOT  | 传输结束   | 036     | \$      | 068     | D  | 100     | d   |
| 005     | ENQ  | 请求     | 037     | %       | 069     | E  | 101     | e   |
| 006     | ACK  | 响应     | 038     | &       | 070     | F  | 102     | f   |
| 007     | BEL  | 响铃     | 039     | '       | 071     | G  | 103     | g   |
| 008     | BS   | 退格     | 040     | (       | 072     | H  | 104     | h   |
| 009     | HT   | 水平制表   | 041     | )       | 073     | I  | 105     | i   |
| 010     | LF   | 换行     | 042     | *       | 074     | J  | 106     | j   |
| 011     | VT   | 垂直制表   | 043     | +       | 075     | K  | 107     | k   |
| 012     | FF   | 换页     | 044     | ,       | 076     | L  | 108     | l   |
| 013     | CR   | 回车     | 045     | -       | 077     | M  | 109     | m   |
| 014     | SO   | 移位输出   | 046     | .       | 078     | N  | 110     | n   |
| 015     | SI   | 移位输入   | 047     | /       | 079     | O  | 111     | o   |
| 016     | DLE  | 数据链路转义 | 048     | 0       | 080     | P  | 112     | p   |
| 017     | DC1  | 设备控制 1 | 049     | 1       | 081     | Q  | 113     | q   |
| 018     | DC2  | 设备控制 2 | 050     | 2       | 082     | R  | 114     | r   |
| 019     | DC3  | 设备控制 3 | 051     | 3       | 083     | S  | 115     | s   |
| 020     | DC4  | 设备控制 4 | 052     | 4       | 084     | T  | 116     | t   |
| 021     | NAK  | 拒绝接收   | 053     | 5       | 085     | U  | 117     | u   |
| 022     | SYN  | 同步空闲   | 054     | 6       | 086     | V  | 118     | v   |
| 023     | ETB  | 传输块结束  | 055     | 7       | 087     | W  | 119     | w   |
| 024     | CAN  | 取消     | 056     | 8       | 088     | X  | 120     | x   |
| 025     | EM   | 介质中断   | 057     | 9       | 089     | Y  | 121     | y   |
| 026     | SUB  | 置换     | 058     | :       | 090     | Z  | 122     | z   |
| 027     | ESC  | 溢出     | 059     | ;       | 091     | [  | 123     | {   |
| 028     | FS   | 文件分隔符  | 060     | <       | 092     | \  | 124     |     |
| 029     | GS   | 组分分隔符  | 061     | =       | 093     | ]  | 125     | }   |
| 030     | RS   | 记录分隔符  | 062     | >       | 094     | ^  | 126     | ~   |
| 031     | US   | 单元分隔符  | 063     | ?       | 095     | -  | 127     | DEL |

备注(ASCII 编码表的一些规律):

- ① 第 1 列共 32 个字符是控制字符，是不可以显示的，必须用转义字符来表示。例如，'\n'表示换行字符。
- ② ASCII 编码值为 32 的字符是空格字符。
- ③ 数字字符 0 对应编码是 0110000B (48D)，0~9 码值以 1 递增，数字字符“0”到“9”的 ASCII 码值减去 48D，可得到对应的数值。
- ④ 大写英文字母码值比小写英文字母码值小。
- ⑤ A 的 ASCII 码是 65，B 是 66，以此类推。
- ⑥ a 的 ASCII 码是 97，b 是 98，以此类推。
- ⑦ 同一字母的大小写 ASCII 码值相差 32。

## 附录 F C/C++关键字

所谓**关键字**是由 C/C++语言规定的具有特定意义的字符串，通常也称为**保留字**。用户定义的标识符不应与关键字相同。

C/C++语言的关键字分为以下几类：

- 1) 类型说明符：用于定义、说明变量、函数或其它数据结构的类型，如 **int**、**double** 等。
- 2) 语句定义符：用于表示一个语句的功能，如 **if**、**else** 等。
- 3) 预处理命令字：用于表示一个预处理命令，如 **include** 等。

另外，C/C++的关键字在 Visual C++编辑器显示为蓝色字体。

下表列出了 C/C++语言中所有的关键字。

|           |          |           |                  |          |         |
|-----------|----------|-----------|------------------|----------|---------|
| asm       | auto     | bool      | break            | case     | catch   |
| char      | class    | const     | const_cast       | continue | default |
| delete    | do       | double    | dynamic_cast     | else     | enum    |
| explicit  | export   | extern    | false            | float    | for     |
| friend    | goto     | if        | include          | inline   | int     |
| long      | mutable  | namespace | new              | operator | private |
| protected | public   | register  | reinterpret_cast | return   | short   |
| signed    | sizeof   | Static    | static_cast      | struct   | switch  |
| template  | this     | throw     | true             | try      | typedef |
| typeid    | typename | union     | unsigned         | using    | virtual |
| void      | volatile | wchar_t   | while            |          |         |

## 附录 G 运算符及其优先级与结合性

| 优先级 | 运算符                                                                       | 含义                                                                                                                                          | 结合方向 |
|-----|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|------|
| 1   | ::                                                                        | 域运算符                                                                                                                                        | 自左至右 |
| 2   | ()<br>[]<br>-><br>.<br>++<br>--                                           | 括号, 函数调用<br>数组下标运算符<br>指向成员运算符<br>成员运算符<br>自增运算符(后置)(单目运算符)<br>自减运算符(后置)(单目运算符)                                                             | 自左至右 |
| 3   | ++<br>--<br>~<br>!<br>-<br>+<br>*<br>&<br>(类型)<br>sizeof<br>new<br>delete | 自增运算符(前置)<br>自减运算符(前置)<br>按位取反运算符<br>逻辑非运算符<br>负号运算符<br>正号运算符<br>指针运算符<br>取地址运算符<br>类型转换运算符<br>长度运算符<br>动态分配空间运算符<br>释放空间运算符<br>(以上均为单目运算符) | 自右至左 |
| 4   | *<br>/<br>%                                                               | 乘法运算符<br>除法运算符<br>求余运算符                                                                                                                     | 自左至右 |
| 5   | +<br>-                                                                    | 加法运算符<br>减法运算符                                                                                                                              | 自左至右 |
| 6   | <<<br>>>                                                                  | 按位左移运算符<br>按位右移运算符                                                                                                                          | 自左至右 |
| 7   | <、<=、>、>=                                                                 | 关系运算符                                                                                                                                       | 自左至右 |
| 8   | ==<br>!=                                                                  | 等于运算符<br>不等于运算符                                                                                                                             | 自左至右 |
| 9   | &                                                                         | 按位与运算符                                                                                                                                      | 自左至右 |
| 10  | ^                                                                         | 按位异或运算符                                                                                                                                     | 自左至右 |
| 11  |                                                                           | 按位或运算符                                                                                                                                      | 自左至右 |
| 12  | &&                                                                        | 逻辑与运算符                                                                                                                                      | 自左至右 |
| 13  |                                                                           | 逻辑或运算符                                                                                                                                      | 自左至右 |
| 14  | ? :                                                                       | 条件运算符(三目运算符)                                                                                                                                | 自右至左 |
| 15  | =、+=、-=、*=、/=、%=<br>>>=、<<=、&=、^=、 =                                      | 赋值运算符                                                                                                                                       | 自右至左 |
| 16  | Throw                                                                     | 抛出异常运算符                                                                                                                                     | 自右至左 |
| 17  | ,                                                                         | 逗号运算符                                                                                                                                       | 自左至右 |

## 附录 H 本教材例题和练习题在 ZOJ、POJ 及 UVA 上的题号

| 章节    | 本教材题号  | 题目名称                                       | 题目来源                                                       | ZOJ 题号 | POJ 题号 | UVA 题号 |
|-------|--------|--------------------------------------------|------------------------------------------------------------|--------|--------|--------|
| 第 1 章 | 例 1.12 | 海狸(Beavergnaw)                             | Waterloo, June 1, 2002                                     | 1904   | 2405   |        |
| 第 3 章 | 例 3.1  | 数字阶梯(Number Steps)                         | Asia 2000, Tehran (Iran)                                   | 1414   | 1663   |        |
|       | 例 3.2  | 假票(Fake Tickets)                           | South America 2002, Practice                               | 1514   |        |        |
|       | 例 3.3  | 纸牌(Deck)                                   | South Central USA 1998                                     | 1216   | 1607   |        |
|       | 例 3.4  | 特殊的四位数(Specialized Four-Digit Numbers)     | Pacific Northwest 2004                                     | 2405   | 2196   |        |
|       | 练习 3.1 | 二进制数(Binary Numbers)                       | Central Europe 2001, Practice                              | 1383   |        |        |
|       | 练习 3.2 | 完数(Perfection)                             | Mid-Atlantic USA 1996                                      | 1284   | 1528   |        |
|       | 练习 3.3 | 求三角形外接圆周长(The Circumference of the Circle) | Ulm 1996                                                   | 1090   | 2242   |        |
|       | 练习 3.4 | 根据公式计算 e (u Calculate e)                   | Greater New York 2000                                      | 1113   | 1517   |        |
| 第 4 章 | 例 4.2  | 切换状态(Switch)                               | Zhejiang University 2003 Summer Camp Qualification Contest | 1622   |        |        |
|       | 例 4.4  | 歌德巴赫猜想(Goldbach's Conjecture)              | Asia 1998, Tokyo (Japan)                                   | 1657   |        |        |
|       | 例 4.5  | 绑定正多边形(Bounding Box)                       | University of Waterloo Local Contest 2001.09.22            | 1892   | 2504   |        |
|       | 例 4.6  | 假银币(Counterfeit Dollar)                    | East Central North America 1998                            | 1184   | 1013   |        |
|       | 例 4.7  | 自我数(Self Numbers)                          | Mid-Central USA 1998                                       | 1180   | 1316   |        |
|       | 例 4.8  | 各位数码全为 1 的数(Ones)                          | University of Waterloo Local Contest 2001.06.02            | 1889   | 2551   |        |
|       | 练习 4.1 | 几何问题变得简单了 (Geometry Made Simple)           | Southwestern Europe 1997, Practice                         | 1241   |        |        |
|       | 练习 4.2 | 歌德巴赫猜想(Goldbach's Conjecture)              | University of Ulm Local Contest 1998                       | 1951   | 2262   |        |
|       | 练习 4.3 | 围住多边形的边(Frame Polygonal Line)              | Zhejiang University Local Contest 2004                     | 2099   |        |        |
|       | 练习 4.4 | 假币(False Coin)                             | Northeastern Europe 1998                                   | 2034   | 1029   |        |
|       | 练习 4.5 | 积木(Blocks)                                 | University of Waterloo Local Contest 2002.09.21            | 1910   | 2363   |        |
|       | 练习 4.6 | 关灯游戏增强版(Extended Lights Out)               | Greater New York 2002                                      | 1354   | 1222   |        |
| 第 5 章 | 例 5.1  | 醉酒的狱卒(The Drunk Jailer)                    | Greater New York 2002                                      | 1350   | 1218   |        |
|       | 例 5.3  | 网络拥堵解决方案(Eeny Meeny Moo)                   | University of Ulm Local Contest 1996                       | 1088   | 2244   |        |
|       | 例 5.4  | 三子棋游戏(Tic Tac Toe)                         | University of Waterloo Local Contest 2002.09.21            | 1908   | 2361   |        |
|       | 例 5.5  | 扫雷游戏(Mine Sweeper)                         | University of Waterloo Local Contest 1999.10.02            | 1862   | 2612   |        |
|       | 例 5.6  | 弹球游戏(Linear Pachinko)                      | Mid-Central USA 2006                                       | 2813   | 3095   |        |
|       | 例 5.7  | 分糖果的游戏(Candy Sharing Game)                 | Greater New York 2003                                      | 1814   | 1666   |        |
|       | 例 5.8  | 爬动的蠕虫(Climbing Worm)                       | East Central North America 2002                            | 1494   |        |        |
|       | 例 5.9  | 遍历迷宫(Maze Traversal)                       | University of Waterloo Local Contest 1996.09.28            | 1824   |        |        |

| 章节    | 本教材题号   | 题目名称                                           | 题目来源                                                | ZOJ题号 | POJ题号 | UVA题号 |
|-------|---------|------------------------------------------------|-----------------------------------------------------|-------|-------|-------|
|       | 练习 5.1  | 约瑟夫环问题(Joseph)                                 | Central Europe 1995                                 |       | 1012  |       |
|       | 练习 5.2  | 另一个约瑟夫环问题(Yet Another Josephus Problem)        | Zhejiang University Local Contest 2006              | 2731  |       |       |
|       | 练习 5.3  | 石头、剪刀、布(Rock, Scissors, Paper)                 | University of Waterloo Local Contest 2003.01.25     | 1921  | 1339  |       |
|       | 练习 5.4  | 贪吃蛇游戏(The Worm Turns)                          | East Central North America 2001, Practice           | 1056  |       |       |
|       | 练习 5.5  | 纸牌游戏(Undercut)                                 | East Central North America 2001, Practice           | 1057  |       |       |
|       | 练习 5.6  | 货币兑换(Currency Exchange)                        | East Central North America 2001, Practice           | 1058  |       |       |
|       | 练习 5.7  | 古怪的钟(Weird Clock)                              | ZOJ Monthly, December 2002                          | 1476  |       |       |
|       | 练习 5.8  | 金币(Gold Coins)                                 | Rocky Mountain 2004                                 | 2345  | 2000  |       |
| 第 6 章 | 例 6.1   | 曾经最难的题目(the hardest problem ever)              | South Central USA 2002                              | 1392  | 1298  |       |
|       | 例 6.2   | 打字纠错(WERTYU)                                   | University of Waterloo Local Contest 2001.01.27     | 1884  | 2538  |       |
|       | 例 6.3   | Soundex 编码(Soundex)                            | University of Waterloo Local Contest 1999.09.25     | 1858  | 2608  |       |
|       | 例 6.4   | 圆括号编码(Parencodings)                            | Asia 2001, Tehran (Iran)                            | 1016  | 1068  |       |
|       | 例 6.5   | 回文的判断                                          | 自编                                                  |       |       |       |
|       | 例 6.6   | 构造回文                                           | 自编                                                  |       |       |       |
|       | 例 6.7   | 镜像回文(Palindromes)                              | South Central USA 1995                              | 1325  | 1590  |       |
|       | 例 6.8   | 字符串的幂(Power Strings)                           | University of Waterloo Local Contest 2002.07.01     | 1905  | 2406  |       |
|       | 例 6.9   | 字符串包含问题(All in All)                            | University of Ulm Local Contest 2002                | 1970  | 1936  |       |
|       | 例 6.10  | 数字字符                                           | 自编                                                  |       |       |       |
|       | 例 6.11  | 英语数字翻译<br>(English-Number Translator)          | Czech Technical University Open 2004                | 2311  | 2121  |       |
|       | 例 6.12  | 单词的 anagrammatic 距离<br>(Anagrammatic Distance) | Southeastern Europe 2005                            | 2585  | 2681  |       |
|       | 练习 6.1  | 字符减一(IBM Minus One)                            | Southwestern Europe 1997, Practice                  | 1240  |       |       |
|       | 练习 6.2  | 置换加密法(Substitution Cypher)                     | University of Waterloo Local Contest 1996.10.05     | 1831  |       |       |
|       | 练习 6.3  | Quicksum 校验和<br>(Quicksum)                     | Mid-Central USA 2006                                | 2812  | 3094  |       |
|       | 练习 6.4  | 字符宽度编码(Run Length Encoding)                    | University of Ulm Local Contest 2004                | 2240  | 1782  |       |
|       | 练习 6.5  | 摩尔斯编码(P,MTHBGWB)                               | Greater New York 2001                               | 1068  | 1051  |       |
|       | 练习 6.6  | 添加后缀构成回文<br>(Suffdromes)                       | University of Waterloo Local Contest 1999.10.02     | 1865  | 2615  |       |
|       | 练习 6.7  | 粗心的 Tony(Careless Tony)                        | Zhejiang University Local Contest 2003, Preliminary | 1582  |       |       |
|       | 练习 6.8  | 令人惊讶的字符串<br>(Surprising Strings)               | Mid-Central USA 2006                                | 2814  | 3096  |       |
|       | 练习 6.9  | LC 显示器(LC-Display)                             | Mid-Central European Regional Contest 1999          | 1146  | 1102  |       |
|       | 练习 6.10 | 单词逆序(Word Reversal)                            | East Central North America 1999,                    | 1151  |       |       |

| 章节    | 本教材题号              | 题目名称                              | 题目来源                                                | ZOJ题号 | POJ题号 | UVA题号 |
|-------|--------------------|-----------------------------------|-----------------------------------------------------|-------|-------|-------|
|       |                    |                                   | Practice                                            |       |       |       |
|       | 练习 6.11            | 多项式表示问题(Polynomial Showdown)      | Mid-Central USA 1996                                | 1720  | 1555  |       |
| 第 7 章 | 例 7.1              | 回文数(Palindrom Numbers)            | South Africa 2001                                   | 1078  |       |       |
|       | 例 7.2              | 初等算术(Primary Arithmetic)          | University of Waterloo Local Contest 2000.09.23     | 1874  | 2562  |       |
|       | 例 7.3              | Skew 二进制(Skew Binary)             | Mid-Central USA 1997                                | 1712  | 1565  |       |
|       | 例 7.4              | 整数探究(Integer Inquiry)             | Central Europe 2000                                 | 1205  |       |       |
|       | 例 7.5              | 高精度数的乘法                           | 自编                                                  |       |       |       |
|       | 例 7.6              | 八进制小数(Octal Fractions)            | South Africa 2001                                   | 1086  | 1131  |       |
|       | 例 7.7              | Fibonacci 数(Fibonacci Numbers)    | University of Waterloo Local Contest 1996.10.05     | 1828  |       |       |
|       | 例 7.8              | Niven 数(Niven Numbers)            | East Central North America 1999, Practice           | 1154  |       |       |
|       | 练习 7.1             | 设计计算器(Basically Speaking)         | Mid-Central USA 1995                                | 1334  | 1546  |       |
|       | 练习 7.2             | 进制转换(Number Base Conversion)      | Greater New York 2002                               | 1352  | 1220  |       |
|       | 练习 7.3             | Wacmian 数(Wacmian Numbers)        | South Pacific 2003                                  |       |       | 2371  |
|       | 练习 7.4             | 火星上的加法(Martian Addition)          | Zhejiang University Local Contest 2002, Preliminary | 1205  |       |       |
|       | 练习 7.5             | 总和(Total Amount)                  | Zhejiang Provincial Programming Contest 2005        | 2476  |       |       |
|       | 练习 7.6             | 循环数(Round and Round We Go)        | Greater New York 2001                               | 1073  | 1047  |       |
|       | 练习 7.7             | 余数(Basic Remains)                 | University of Waterloo Local Contest 2003.09.20     | 1929  | 2305  |       |
|       | 练习 7.8             | 有多少个 Fibonacci 数(How Many Fibs?)  | University of Ulm Local Contest 2000                | 1962  | 2413  |       |
|       | 练习 7.9             | 颠倒数的和(Adding Reversed Numbers)    | Central Europe 1998                                 | 2001  | 1504  |       |
|       | 练习 7.10            | 数基(Digital Roots)                 | Greater New York 2000                               | 1115  | 1519  |       |
| 第 8 章 | 例 8.6              | 另一个 Fibonacci 数列(Fibonacci Again) | ZOJ Monthly, December 2003                          | 2060  |       |       |
|       | 例 8.7 <sup>①</sup> | 分形(Fractal)                       | Asia 2004, Shanghai (Mainland China), Preliminary   | 2423  | 2083  |       |
|       | 例 8.8              | 骨头的诱惑(Tempter of the Bone)        | Zhejiang Provincial Programming Contest 2004        | 2110  |       |       |
|       | 例 8.9              | 图形周长(Image Perimeters)            | University of Waterloo Local Contest 2001.09.22     | 1047  | 1111  |       |
|       | 例 8.10             | 素数环问题(Prime Ring Problem)         | Asia 1996, Shanghai (Mainland China)                | 1457  |       |       |
|       | 例 8.11             | 保险箱解密高手(Safecracker)              | Mid-Central USA 2002                                | 1403  | 1248  |       |
|       | 例 8.12             | 方形硬币(Square Coins)                | Asia 1999, Kyoto (Japan)                            | 1666  |       |       |
|       | 例 8.13             | 求和(Sum It Up)                     | Mid-Central USA 1997                                | 1711  | 1564  |       |
|       | 例 8.14             | 正方形(Square)                       | University of Waterloo Local                        | 1909  | 2362  |       |

<sup>①</sup>这道题在ZOJ和POJ上有区别：在ZOJ上，每行最后的一个'X'字符后不能有多余的空格；而在POJ上，要求每行的宽度相同，这样某些行最后的一个'X'字符后会有多余的空格。

| 章节    | 本教材题号   | 题目名称                              | 题目来源                                                           | ZOJ题号 | POJ题号 | UVA题号 |
|-------|---------|-----------------------------------|----------------------------------------------------------------|-------|-------|-------|
|       |         |                                   | Contest 2002.09.21                                             |       |       |       |
|       | 练习 8.6  | 欧几里得游戏(Euclid's Game)             | University of Waterloo Local Contest 2002.09.28                | 1913  | 2348  |       |
|       | 练习 8.7  | 幸存者游戏(Recursive Survival)         | ZOJ Monthly, January 2004                                      | 2072  |       |       |
|       | 练习 8.8  | 抽签(Lot)                           |                                                                | 1539  |       |       |
|       | 练习 8.9  | 礼物(Gift?!)                        | OIBH Online Programming Contest #1                             | 1229  |       |       |
|       | 练习 8.10 | 火力配置网络(Fire Net)                  | Zhejiang University Local Contest 2001<br>Mid-Central USA 1998 | 1002  | 1315  |       |
|       | 练习 8.11 | 字母排列(Anagram)                     | Southwestern European Regional Contest 1995                    |       | 1256  |       |
|       | 练习 8.12 | 抽奖游戏(Lotto)                       | University of Ulm Local Contest 1996                           | 1089  | 2245  |       |
|       | 练习 8.13 | 分配大理石(Dividing)                   | Mid-Central European Regional Contest 1999                     | 1149  | 1014  |       |
| 第 9 章 | 例 9.4   | 快乐的蠕虫(The Happy Worm)             | Asia 2004, Tehran (Iran), Sharif Preliminary                   | 2499  | 1974  |       |
|       | 例 9.5   | 花生(The Peanuts)                   | South Central USA 1995                                         | 2235  | 1928  |       |
|       | 例 9.6   | Unix 操作系统的 ls 命令(Unix ls)         | South Central USA 1995                                         | 1324  | 1589  |       |
|       | 例 9.7   | 混乱排序(Scramble Sort)               | Greater New York 2000                                          | 1225  | 1520  |       |
|       | 例 9.9   | 赌徒(Gamblers)                      | Waterloo, June 2, 2001                                         | 1101  | 2548  |       |
|       | 练习 9.6  | 单词重组(Word Amalgamation)           | Mid-Central USA 1998                                           | 1181  | 1381  |       |
|       | 练习 9.7  | 英文姓名排序                            | 自编                                                             |       |       |       |
|       | 练习 9.8  | 古老的密码(Ancient Cipher)             | Northeastern Europe 2004                                       | 2658  | 2159  |       |
|       | 练习 9.9  | DNA 排序(DNA Sorting)               | East Central North America 1998                                | 1188  | 1007  |       |
|       | 练习 9.10 | 体重排序(Does This Make Me Look Fat?) | South Central USA 2001                                         | 1431  | 2218  |       |
|       | 练习 9.11 | 半素数(Semi-Prime)                   | Zhejiang University Local Contest 2006, Preliminary            | 2723  |       |       |
|       | 练习 9.12 | 棍子的膨胀(Expanding Rods)             | University of Waterloo Local Contest 2004.06.12                | 2370  | 1905  |       |

备注:

ZOJ, 浙江大学 ACM 网站, <http://acm.zju.edu.cn/onlinejudge/>POJ, 北京大学 ACM 网站, <http://acm.pku.edu.cn/JudgeOnline/>UVA, 西班牙 Valladolid 大学 ACM 网站, <http://acm.uva.es/problemset/>

## 参考文献

- [1] 李文新, 郭炜, 余华山编著. 程序设计导引及在线实践[M]. 清华大学出版社. 2007 年 11 月.
- [2] 谭浩强编著. C++程序设计[M]. 清华大学出版社. 2004 年 7 月.
- [3] 谭浩强著. C 程序设计 (第 2 版) [M]. 清华大学出版社. 1999 年 12 月.
- [4] 吴文虎编著. 程序设计基础 (第 2 版) [M]. 清华大学出版社. 2004 年 9 月第 2 版.
- [5] 王桂平, 冯睿. 以在线实践为导向的程序设计课程教学新思路[J]. 计算机教育, 2008, (22): 100-102.
- [6] 王桂平, 冯睿. 突出实践能力培养的程序设计课程教学方法[J]. 实验室科学, 2009, (1): 81-84.