# 第六章　数据对象的顺序组合：数组

程序的工作都与数据有关，如输入数据、输出数据、存储数据、对数据进行各种计算和修改等。客观世界中需要处理的数据千姿百态，为能处理所关注事物的有关特征和性质，就需要在程序里表示它们。为此，程序语言需要提供一套手段，使人能在写程序时方便地处理与数据表示有关的各种问题，这就是语言的数据机制。

需要由程序处理的数据可能很简单，也可能很复杂，数据之间可能有丰富多采的各种关系。为了反映这些情况，语言的数据机制必须足够丰富。这里也有简洁性与方便性的问题，有关数据机制不能无限制地庞大，使语言过分臃肿，难以使用；也不应过于低级，以至要描述一点东西都非常烦琐。后一情况正是机器语言和汇编语言的一个弱点。

经过几十年的研究和实践，高级语言领域在这方面已经形成了一套公认的有效方式，数据机制基本框架通常包括下面几个互相联系的方面：

1. 把语言要处理的数据对象划分为一些类型，每个类型是一个数据值的集合。例如 C 语言里的 `int` 类型就是该类型能表示的所有整数值的集合。
2. 提供一组基本数据类型，确定其书写方式（文字量），提供一组基本操作（运算），支持基本数据对象的表示和使用。例如 C 语言的 `int`、`double` 等。
3. 提供一组由简单数据类型或数据对象构造更复杂数据类型或数据对象的手段。反复使用这些手段可以构造出任意复杂的数据结构，以满足复杂数据处理的需要。这里提到的*数据结构*是计算机科学中的一个重要概念，也是一门后续计算机课程的名称。

C 语言采用了通行的数据机制。它提供了一组基本数据类型，其数据构造机制是下面几章的基本内容，包括本章将要介绍的数组，后面章节要介绍的指针、结构、联合等。利用它们可以把多个（基本或非基本的）数据对象组合起来，作为整体在程序里使用。这样组合而成的数据对象称作*复合数据对象*，由所有“同类”复合对象形成的类型称为*复合数据类型*。一个复合数据对象的组成部分称为它的*成分*、*成员*或*元素*。

程序中可以建立存放复合类型数据的变量，这使复合数据对象也能命名。这种变量可以作为整体使用，通过变量名访问；还提供了访问复合数据对象里成分的操作，使用这些成分的值或者给它们赋值。C 语言提供了所有这些机制。

最常见的数据组合机制是*数组*和*结构*（structure，或称*记录*，record），另一种用于数据组织的重要机制是*指针*。本章将介绍 C 语言的数组机制，其意义、描述方式和使用。后面章节里介绍其他数据机制的情况及其在程序设计中的应用。

## 6.1 数组的概念、定义和使用

数组（array）是 C 语言中用于组合同类型数据对象的机制。一个数组里汇集一批对象（数组元素）。程序中既能从数组出发处理其中的个别元素，也能以统一方式处理数组的一批元素或所有元素。后一处理方式特别重要，是由一批成员构成的数组和一批独立命名的变量间的主要区别。

数组机制要解决三个问题：1）描述数组的性质，定义数组变量；2）使用数组，包括通过数组变量使用元素；3）实现数组，即在内存里为数组安排一种存储方式，使程序里可以方便地操作它们。当然，最后一个问题主要与语言的实现有关系，在实现 C 语言系统时，必须确定如何实现数组变量。了解这方面情况也有利于在编程时正确使用数组。

### 6.1.1 数组变量定义

根据数组的性质，在定义数组变量（下面简单说成“定义数组”）需要说明两个问题：1）该数组（变量）的元素是什么类型的；2）这个数组里包含多少个元素。C 语言规定，每个数组变量的大小是固定的，需要在定义时说明。

数组定义的形式与简单变量类似，但需要增加有关元素个数的信息。在被定义变量名之后写一对方括号就是一个数组定义，指定元素个数的方式是在括号里写一个整型表达式。人们常把数组元素类型看作数组的类型，把元素类型为整型的数组说成是*整型数组*，类似地说*双精度数组*等。本书下面也采用这种说法。

例如，下面的描述定义了两个数组：

```
int a[10];
double a1[100];
```

定义了一个包含有 10 个元素的整型数组 a 和一个 100 个元素的双精度数组 a1。数组元素个数也称为数组的*大小*或数组的*长度*。

数组定义可以与其他变量的定义写在一起，例如可以写：

```
int a2[16], n, a3[25], m;
```

数组变量也是变量，数组定义可以出现在任何能定义简单变量的地方。

数组变量也是变量，在作用域和存在期方面与简单变量没有差别。根据定义位置不同，数组也分为外部数组和函数内的局部数组，包括函数内的静态数组（用关键字 static）和普通的自动数组，定义方式（及位置）决定了它们的作用域与存在期。

可以写出数组的外部说明。C 语言规定，在写数组变量的外部说明时不必写数组大小，只要在数组变量名后写一对方括号。例如下面是两个数组的外部说明：

```
extern int a[];
extern double a1[];
```

这两个说明通知本源文件的其他部分，有两个数组（a 和 a1）在其他地方定义，它们的元素类型分别是整型和双精度类型。

数组元素个数必须能在编译时静态确定，因此这个表达式必须能静态求值，最简单的情况就是写一个整型字面量（整数）。根据这个规定，下面数组定义不合法[1]：

```
void f(int m, int n) {
    int b[n];
    ....
}
```

此时局部数组 b 的大小依赖于函数的参数值，这个值在编译时无法确定。

### 6.1.2 数组的使用

使用数组的最基本操作是元素访问，对数组的使用最终都通过对元素的使用而实现。数组元素在数组里顺序排列编号，首元素的编号规定为 0，其他元素顺序编号。这样，n 个元素的数组的元素编号范围是 0 到 n-1。如果程序里定义了数组：

```
int b[4];
```

b 的元素将依次编号为 0、1、2、3。数组元素的编号也称为元素的*下标*或*指标*。

数组元素访问通过数组名和表示下标的表达式进行，用下标运算符[]描述。下标运算符[]是 C 语言里优先级最高的运算符之一，它的两个运算对象的书写形式比较特殊：一个运算对象写在方括号前面，应表示一个数组（简单情况是数组名）；另一个应该是整型表达式，写在括号里面表示元素下标。元素访问是一种基本表达式，写在表达式里的 b[3]就是一个下标表达式，表示访问数组 b 中编号为 3 的元素，即上面定义的数组 b 的最后元素。

例如，有了上面定义之后，程序里就可以写下面这些语句：

```
b[0] = 1; b[1] = 1;
```

[1] 新的 C99 标准在这里的规定不同，请参看本书最后的有关介绍或其他相关材料。

```
b[2] = b[0] + b[1];
b[3] = b[1] + b[2];
```

显然，对于这些简单情况，完全可以用几个简单变量代替 b 这样的数组变量，例如写：

```
int b0, b1, b2, b3;
b0 = 1; b1 = 1;
b2 = b0 + b1;
b3 = b1 + b2;
```

这里还看不出数组的实际价值。

数组的真正意义在于它使我们可能以统一方式描述对一组数据的处理。由于下标表达式可以是任何具有整数值的表达式，也允许包含变量。例如可以写：

```
b[i] = b[i-1] + b[i-2];
```

这个语句执行时访问哪三个数组元素，要看变量 i 当时的值。改变下标表达式里变量的值，同一个访问数组元素的语句在每次执行时访问的可能是数组的不同元素。把这种形式的语句写在循环里，执行中实际访问的将是数组 b 的一组元素甚至全部元素。

例：写程序建立一个包含 Fibonacci 序列前 30 个数的数组，然后从大到小打印这个数组中所有的数。

根据上面的讨论，解决这个问题的程序可以写为：

```
#include <stdio.h>

int main () {
    long fib[30];
    int n;

    fib[0] = 1;
    fib[1] = 1;
    for (n = 2; n < 30; ++n)
        fib[n] = fib[n-1] + fib[n-2];

    for (n = 29; n >= 0; --n) {
        printf("%d", fib[n]);
        putchar(n % 6 == 5 ? '\n' : ' ');
    }

    return 0;
}
```

这里定义了一个 30 个整型元素的数组，用一个 for 循环语句实现大部分计算。如果用一组简单变量，那就需要写 28 个形式类似的语句。显然，采用数组带来许多方便，程序也更清晰。后面也利用循环完成输出，这个循环执行 30 次，相当于 30 个基本语句。

由这个简单例子可以看到，利用循环变量可以以统一形式访问一批数组元素，这样做可能带来许多方便。程序里采用 long 类型数组，是考虑到元素可能变得很大。最后的输出语句里用了一点小技巧，通过一个条件表达式，使程序能在输出 6 个元素后换一行。

如果需要操作数组中连续许的多个元素或者全部元素，人们通常用 for 语句和一个循环变量。最常见的结构就是令循环变量遍历数组的全部下标，其书写形式是：

```
for (n = 0; n < 数组长度; ++n) ... ...
```

今后读者将常常看到这种形式的循环。

这里还有一个问题。假设上面程序的第一个循环被写成如下形式：

```
for(n = 2; n <= 30; ++n)
    fib[n] = fib[n-1] + fib[n-2];
```

循环执行中将出现对 fib[30]的访问，按定义并没有这个元素，因为 fib 的下标范围是 0 到 29。访问不存在的东西显然是错误的，用超出数组下标合法范围的下标表达式进行访问的现象称为*越界访问*，这是使用数组的程序里最常见的一种语义错误。这种错误仅通过查看程序常常无法判断，因为越界情况产生的原因是下标表达式的值不合适，而表达式的值要在

程序运行过程中确定。

前面提到过 C 语言对程序错误的检查不够严格，一个重要方面就是它不检查数组元素访问的合法性。用 C 语言开发的程序在运行中不检查数组越界问题，在实际出现越界访问错误时也不报告出错信息。

当然，错误的存在与否是客观的，不报告错误不等于没有错误。数组的合法下标范围是确定的，超范围的访问一定是错误，由此引起的后果无法预料。C 语言不检查数组越界是为了保证程序执行的效率。这实际上要求写程序的人自己关心数组下标表达式的合法性问题，处理好这件事。我们在写与数组有关的程序时，在写访问数组元素的下标表达式时，一定要特别关注这个问题，保证不进行非法访问。

### 6.1.3 数组的初始化

我们可以通过语句和控制结构给数组元素设置初值，就像前面例子中所做的那样。为了方便使用，C 语言也允许在定义数组时直接初始化，提供了给数组元素指定初值的描述形式。无论是外部数组、函数内的静态数组或自动数组，都可以用这种形式初始化*。外部数组和局部静态数组同样在程序开始执行前建立并初始化；局部自动数组也是在程序执行进入相应函数时建立和初始化。

给数组指定初值也是在定义变量时通过附加描述给出元素初值：各元素值的表达式顺序在一对花括号里，表达式间用逗号分隔。例如：

```
int b[4] = {1, 1, 2, 3};
double ax[6] = {1.3, 2.24, 5.11, 8.37, 6.5};
```

这不但定义了整型数组变量 b 和双精度数组 ax，还给 b 的四个元素顺序地指定初始值 1、1、2 和 3，也顺序地给 ax 的各个元素指定了初值。

元素初值表达式必须是常量表达式，自动数组的初始化也只允许用常量表达式。这个规定与前面对简单类型自动变量初始化的规定不同。应特别指出，这种为数组元素指定值的写法只能用在初始化时，语句里不能采用这种写法。

如果定义数组时没做初始化，外部数组和局部静态数组的元素将自动初始化为 0，自动数组的元素将不初始化，这些元素的值处在没有明确初始化的状态。

语言允许只给数组前一段元素指定初值，这时未指定值的元素将自动初始化为 0（无论是外部数组还是自动数组）。另一方面，语言还要求初始化表示中元素的个数不超过数组元

**越界访问的可能后果**

在有些操作系统里，每个程序的合法数据访问范围受到严格监控。如果程序里的数据访问越出了程序数据区的范围，就会被这些操作系统认定为非法访问，从而导致一个动态错误，操作系统将强制性地终止造成这种错误的程序。

另一些系统（如 DOS）里根本不检查程序访问地范围，但越界访问操作可能破坏这个程序本身的代码，甚至破坏系统里正在运行的其他软件，例如操作系统的子程序或数据结构。这通常会造成系统死机或出现其他莫名其妙的现象。

即使数组越界访问没有超出本程序的合法数据区域，这种访问也是无意义的，甚至是很危险的。越界取得的数据显然不会是有意义的东西，在程序里使用这种数据没有任何价值。越界赋值则更加危险，即使这种操作没有被检查和禁止，其后果也是可怕的。这种赋值操作会破坏被赋值位置的原有数据，其后果难以预料，因为根本就无法知道被这个操作实际破坏的到底是什么。

所以，保证对数组元素访问在合法范围内进行是非常重要的。

* 一些 C 书籍和教科书说函数内的自动数组不能在定义时进行初始化。这是 ANSI C 标准前的过时规定。

素的个数。例如有下面定义：

```
int b1[4] = {1, 2};
int b2[4] = {1, 2, 0, 0};
```

因为 b1 的初值不够，b1[2]、b1[3]自动初始化为 0，b1 和 b2 中各元素的值完全一样。

为了编程方便，C 语言规定，在给出了所有元素初值的情况下，定义数组时可以不写大小。这时系统根据初始化表达式的个数自动确定数组大小。例如可以写下面定义：

```
int fib1[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
int fib2[10] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
```

数组 fib1 和 fib2 都是 10 个元素的整数数组，两种定义方式等价。前一写法能带来一点方便，因为这样能减少维护两部分描述之间一致性的麻烦，有利于程序修改。

### 6.1.4 数组的存储实现

实现数组时需要解决数组元素的存储问题，由于元素类型相同，其实现问题比较简单。C 语言系统将为每个数组分配一个连续存储区域，其中足以存放数组的所有元素。各元素顺序排列，下标为 0 的元素排在最前面，每个元素占的空间相同。例如，有了定义：

```
int a[8];
```

为数组 a 分配的存储区恰好能存放 8 个整型数据，也就是说，它占的存储空间相当于 8 个整型变量所用的空间。具体安排情况如图 6.1 所示。

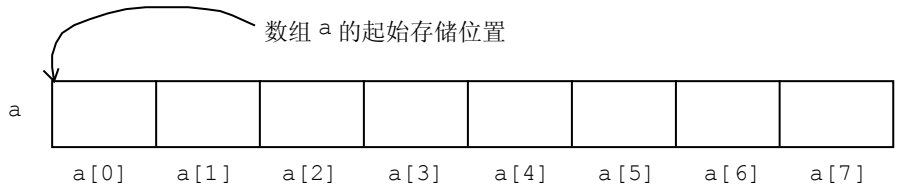



图 6.1 数组的存储实现

这样，数组 a 至少相当于 8 个整型变量，可以把 a[0]、…、a[7]看作是这些变量的“名字”。定义为数组能保证它们排在一起，可以统一通过名字 a 和下标表达式访问这些元素。

## 6.2 数组程序实例

本节将给出几个使用数组的程序设计实例。这些程序实例都比较简单，读者应该从这里了解的不仅仅是有关的程序本身，更应当注意在这些程序里使用数组的方式和与此相关的各种问题。这些程序里也显示了一些写 C 程序时常用的程序设计方法和技巧，其中的一些方法具有较广泛的意义。

### 6.2.1 从字符到下标

问题：写一个程序，统计由标准输入得到的文件中各数字字符出现的次数。

显然的办法是定义 10 个计数变量，用嵌套的 if 语句或用 switch 语句区分各种情况，在遇到某个数字字符时，将对应的计数变量加 1。前面已经写过许多读字符循环，按这一方式写出的程序请读者作为练习，其中不需要数组。

由于 C 语言将字符看作一种整数，由此本问题有一种更方便的解法，C 程序经常采用这种方法。首先，常见编码字符集里的数字字符都是顺序排列的。例如在 ASCII 字符集里，数字字符 0 的编码按十进制是 48，其他数字字符顺序向下排，9 的编码是 57。采用下面两种表达式都可以判断变量 c 存储的是不是数字字符（的编码）：

```
'0' <= c && c <= '9'
isdigit(c)
```

当然，用标准库函数判断更合适些。利用上述事实，我们可以在程序里用一个 10 个元素的计数器数组，以更简洁的形式完成对数字出现次数的统计工作。首先定义：

```
int cs[10];
```

设想用数组成员 cs[0]记录数字 0 的出现次数，用 cs[1]记录数字 1 的出现次数，其余成员顺序使用。如果变量 c 的值是数字字符，表达式 c - '0' 就是 c 中数字对应的计数器下标，将这个计数器加 1 的工作可用下面语句实现（无须判断是哪个字符）：

```
cs[c - '0']++;
```

这样，解决上面问题的程序就可以写成：

```
#include <stdio.h>

int main () {
    int c, i, cs[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++cs[c - '0'];

    for (i = 0; i < 10; ++i)
        printf("Number of %d: %d ", i, cs[i]);
    putchar('\n');

    return 0;
}
```

### 6.2.2 筛法求素数

例：求素数的一种著名方法叫“筛法”，其基本方法是取一个从 2 开始的整数序列，通过不断划掉序列中非素数的整数（合数），逐步确定顺序的一个个素数。具体做法是：

1. 令 n 等于 2，它是素数；
2. 划掉序列中 n 的所有倍数（2*n，3*n 等等）；
3. 找到 n 之后下一个未划掉的元素，它是素数，令 n 等于它，回到步骤 2。

现在要求写一个程序，输出从 2 到某个数之间的所有素数。

可以用一个整数数组表示整数序列，下面以数组元素下标表示该元素对应的整数（也可以在数组里存整数，那是另一种做法。有兴趣的读者可以自己做一做，并将它与这里的做法比较一下）。假设所用数组为 an，这里用 an[0]代表整数 0，an[1]代表整数 1，一般说，用 an[i]代表整数 i。对每个整数只需要表示两种情况：它尚未被划掉（还在序列里），或是已被划掉了。下面用 1 表示这个数还在序列里，用 0 表示它已经被划掉了。开始时数组元素都设置为 1，而后不断将一些元素置 0，直到确定了所需范围里的所有素数。

思路已经清楚了，对应的程序工作过程可以描述为（假设 NUM 是给定的范围）：

```
/* 建立初始数组an，元素都初始化为1，但应将an[0]和an[1]置0（它们不是素数）*/
for (int i = 2; i值不大于某个数; ++i)
    if (an[i] == 1) /* i是素数 */
        for (int j = i*2; j < NUM; j += i)
            an[j] = 0; /* 这些数都是i的倍数，因此不是素数 */
```

这里还有一个问题：外层循环在什么时候应该结束。可以用 NUM 作为界限，但仔细分析不难发现，只要 i 超过 NUM 的平方根，就可以划掉到既定范围内的所有合数了。

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

enum { NUM = 200 };
int an[NUM+1];

int main () {
    int i, j, upb = sqrt(NUM+1);
```

```
    an[0] = an[1] = 0; /* 建立起初始向量 */
    for (i = 2; i <= NUM; ++i) an[i] = 1;

    for (i = 2; i <= upb; ++i)
        if (an[i] == 1)
            for (j = i*2; j <= NUM; j += i)
                an[j] = 0;

    for (i = 2, j = 0; i <= NUM; ++i)
        if (an[i] != 0)
            printf ("%d%c", i, ((++j)%10 == 0 ? '\n' : ' '));
    putchar('\n');
    return 0;
}
```

这里定义的数组包含 NUM+1 个元素，因为考察范围直至 NUM。求平方根时加 1 是为了防止浮点误差带来麻烦。例如，无法保证 sqrt(9) 是 3，它也可能等于 2.9999999……，把这个数转换为整数就会变成 2。加 1 后的数再求平方根就没问题了。程序在需要将数组元素置 0 时没考察它原来是不是 0，因为即使元素已经是 0，这个程序也是正确的。检查值 0 不会带来实质性的效率提高，还会使程序变复杂。

这个程序用一个循环确定所有素数，用另一个循环输出。我们也可以将这两个循环合而为一，实现这种修改的工作留作练习。

### 6.2.3 成绩分类

现在要写一个程序，它通过输入得到一批双精度表示的学生成绩（不超过 200 个）。先输出不及格的成绩，而后再输出及格的成绩。最后输出不及格和及格的人数统计。

这个工作无法在输入过程中完成（因为输入的成绩序列未必是按所需顺序排列的，除非输入两遍。如果是从文件输入，也可能采用这种方式。但现在不讨论它），因此需要用一个数组，在输入过程中将成绩记录到数组里，而后输出和统计。

输入后的工作有许多可能解法。最简单的解法是两次扫描数组内容，第一次将遇到的不及格成绩输出，第二次将及格的成绩输出。统计人数的工作可以在某次扫描中完成（也完全可以在输入过程中做），因为两部分人数之和就是总人数，统计其中一部分就够了。下面是按这种方式写出的程序：

```
#include <stdio.h>

enum { NUM = 200 };
const double PASS = 60.0;
double scores[NUM];

int main () {
    int i, n = 0, fail;
    while (n < NUM && scanf("%lf", &scores[n]) == 1)
        ++n;

    for (fail = 0, i = 0; i < n; ++i)
        if (scores[i] < PASS) {
            printf("%f\n", scores[i]);
            ++fail;
        }
    for (i = 0; i < n; ++i)
        if (scores[i] >= PASS)
            printf("%f\n", scores[i]);

    printf("Fail: %d\n", fail);
    printf("Pass: %d\n", n - fail);
    return 0;
}
```

程序里大部分代码都很清楚，但也有几处需要说明。首先是输入循环，这里用两个条件控制循环的结束。第一个条件 n < NUM 是为了防止输入数据过多而导致数组越界，任何向数组里装填数据的循环都需要考虑数组越界问题。许多用 C 写的系统有这方面的缺陷，例如一些著名网络系统被黑客攻破，最后查出的原因就是黑客送入大量数据导致数组越界（称为缓冲区溢出），从而打入系统。还请注意这两个条件的顺序，调换它们位置就不对了。这里利用了二元逻辑运算符计算方式的规定，保证不会出现数组越界访问的问题。

这里定义了两个符号常量，今后这种方式是我们编写程序的常规。有了这些符号常量，我们就很容易将这一程序修改为能处理 400 个数据，或者以 55.0 分划分及格与不及格。

### 6.2.5 多项式求值

问题：设数组 p 中存着多项式 $a_0 + a_1x + a_2x^2 + \cdots + a_kx^k$ 的系数，其中元素 p[i]存着系数 $a_i$。现在要求写一个程序，求出由数组 p 表示的多项式在某个指定点的值。

假定指定的点由输入得到，存放于变量 x 中，TERMS 是表示多项式项数的常量（或者变量）。先考虑如何实现基本的多项式求值。这一计算过程有多种不同实现方法。例如：

**方法 1**：分别求出多项式各项的值并将其累加，最终得到整个多项式的值。设 sum 是存放累加和的变量，t 是用于计算项值的临时变量。另外用两个辅助循环变量，计算多项式值的工作可以由下面两重循环实现：

```
for (sum = 0.0, n = 0; n < TERMS; ++n) {
    for (t = p[n], i = 1; i <= n; ++i)
        t *= x;
    sum += t;
}
```

大循环体内部的程序片段也可以改造为：

```
for (t = 1.0, i = 1; i <= n; ++i)
    t *= x;
sum += t * p[n];
```

这两段程序效果相同。分析这两段程序，不难发现其中有许多重复计算，后一种写法将问题表现得更清楚，其中每个项的计算总是从头开始做。后一循环形式中 t 值的递推公式非常简单，$t_{i+1} = t_i * x$，根据这个分析，整个大循环可以简化为：

```
for (sum = 0.0, t = 1.0, n = 0; n < TERMS; ++n) {
    sum += t * p[n];
    t *= x;
}
```

**方法 2**：根据数学知识，任何多项式都可以变形为下面规范形式（Horner 形式）：

$$((\cdots((a_kx + a_{k-1})\cdot x + a_{k-2})\cdots + a_2)\cdot x + a_1)\cdot x + a_0$$

按照这个公式，求值的循环可以写为：

```
for (sum = 0.0, n = TERMS - 1; n >= 0; --n)
    sum = sum * x + p[n];
```

请读者分析一下，对一个 $n$ 次多项式，分别采用上述几种算法，在整个求值过程中各需要做多少次加法，多少次乘法。据此判断各种方法的优劣。

下面程序采用第二种方法，采用其他方式的程序请读者自行写出，作为练习。

```
#include <stdio.h>

enum { NUM = 5 };
double p[] = {2.38, 3.142, 5.674, 8.257, 6.44};

int main()
{
    double sum, x;
    int n;
```

```
    while (1) {
        printf("Please enter next value for x: ");
        if (scanf("%lf", &x) != 1) break;
        for (sum = 0.0, n = NUM(p)-1; n >= 0; --n)
            sum = sum * x + p[n];
        printf("Value: %f\n", sum);
    }

    return 0;
}
```

### 6.2.6 定义数组的问题

上面一些实例中使用数组，是因为需要保存一批类型相同的数据，以便在处理中不断修改它们。许多实际应用问题的情况就是这样，这时采用数组就非常有必要。另一种情况是需要多次检查一批数据。即使数据来自文件，多次重复读入文件也不方便，效率很低（外存访问速度比内存慢 $10^6$ 倍或更多）。在可能的情况下（只要内存足够），将数据存入数组或其他类型结构里，在内存中直接处理，能大大提高处理效率。

可以看到，前面程序里的数组都被定义为外部数组（定义在函数之外）。究竟什么时候应该定义为函数的局部数组，什么时候应定义为外部的呢？在许多情况下，两者都是可能的选择。下面是一些常见的考虑：

1. 如果某个（某些）数组需要在多个函数里使用，它们表示的是全局性的数据集合，那么可以考虑将它们定义为外部数组。
2. 如果某个（某些）数组很大，应将它们定义为外部数组，以免占用运行栈上的大量空间。在许多 C 系统里不允许在函数内部定义特别大的数组。
3. 如果数组里保存的是递归定义的函数的局部数据，那么就必须定义为函数内部的自动数组。因为在递归调用时，可能需要这一数组的多份拷贝。
4. 其他情况下两种方式通常都能行。

另请读者参考下一节有关函数数组参数的讨论。

## 6.3 数组作为函数参数

函数是 C 语言中处理复杂问题的重要机制。有了数组，自然就需要考虑如何定义处理数组的函数。本节介绍这方面情况，并用一些例子说明用函数处理数组的方法。

如果某数组的元素是基本类型的，这些元素可以当作基本类型的变量使用，当然可以作为实参送给函数处理。这样做与处理简单变量没什么差别。另外，根据作用域规则，外部数组可以在任何函数里直接访问，这也是定义处理数组的函数的一种方式。前面的数组程序实例大都采用了这种方式。

上面两种方式各有缺陷。第一种只能处理数组的个别元素，实际程序中常要处理整个数组。第二种方式只能处理某个（某几个）固定的外部数组，因为数组名写在函数的语句里。假设某个程序里有 10 个数组，运行中需要多次对不同数组求平均值，采用第二种方法显然也不合适（请读者按这个假设写一个程序，看看有什么问题）。

问题的解决就是需要定义以数组为参数的函数。数组作为函数参数后，通过函数处理整个数组就不成为问题了。在不同调用中提供不同实参，可以处理不同的数组。假设能定义一个以数组为参数求元素平均值的函数，就可能解决程序里所有的求平均值问题了。

### 6.3.1 一个例子

还是从求平均值的简单例子开始。定义函数时遇到第一个的问题是不知道数组长度。假

设有符号常量 LEN，其值正是被处理数组的长度。此时函数可以定义为：

```
double avrg0(double a[]) {
    double x = 0.0;
    int i;

    for (i = 0; i < LEN; ++i)
        x += a[i];

    return x / LEN;
}
```

请读者注意数组形参的描述方式，参数表写成(double a[]) 表示函数有一个名字为 a 的形参，调用时的实参应是一个 double 数组。按规定，描述数组形参时只写一对方括号，不需要给出数组大小。即使在这里给了数组大小，编译程序也不会使用这个信息。有了上面定义后，如果数组 a1 长度是 LEN，下面语句将求它的元素的平均值：

```
x = avrg0(a1);
```

本函数有一定的通用性，如果数组 a2 的长度也是 LEN，也能正确求出它的元素的平均值：

```
y = avrg0(a2);
```

上面定义里借助符号常量得到数组大小，这样能解决许多具体问题。但这种做法也把函数能处理的数组的长度固定了，限制了函数的通用性。虽然上述函数能用于任何双精度数组（只要把数组作为实参），但只能正确处理长度为 LEN 的数组，大些或小些的数组都不能正确处理，甚至可能出现严重错误（在什么情况下执行中会出错？请读者考虑）。我们知道，引进参数的作用就是使函数能够处理一类问题。在定义处理数组的函数时，为使它更通用，应当引进描述长度的参数。下面是引进长度参数后实现同样功能的函数：

```
double avrg(int n, double a[]) {
    double x = 0.0;
    int i;

    for (i = 0; i < n; ++i)
        x += a[i];

    return x / n;
}
```

有了这个函数定义之后，可以写下面的主函数定义：

```
int main () {
    double b1[3] = {1.2, 2.43, 1.074},
           b2[5] = {6.54, 9.23, 8.463, 4.25, 0.386};

    printf("%f, %f\n", avrg(3, b1), avrg(5, b2));
    return 0;
}
```

引进长度参数提高了数组处理函数的通用性，也对调用提出了新要求，给使用带来了一点麻烦。在使用函数的地方都必须关注数组长度，正确提供长度参数，防止越界。例如，调用 avrg(5, b1)就不对，因为 b1 只有 3 个元素，执行中将出现数组越界访问。编译程序不能辨认这种错误，程序执行时也不检查，执行后果无法预料。另一方面，长度参数却可以小于实际数组的大小，avrg(3, b2) 将求出数组 b2 里前 3 个元素的平均值。这也说明了长度参数的另一作用，使函数可以把数组的前一段当作数组使用和处理。

### 6.3.2 修改实参数组的元素

前面说过，C 语言的函数参数都是值参数。也就是说，函数调用时求出实参表达式的值，用它设置函数的形式参数，而后执行函数体。在函数执行中，形参与函数里定义的自动变量

完全一样，它们与函数调用时的实际参数再没有任何联系了。这样，函数体里对形参的操作将不会对实际参数有任何影响。

数组参数的情况有所不同。假设有下面函数定义与调用：

```
int ff(int a[], ...) {
    ... ...
    a[3] = a[3] + a[1];
    ... ...
}
... ...
    ... ff(b) ...;
... ...
```

在 ff(b)执行时，数组形参 a 将与实参 b 建立联系，使函数体内对形参 a 的元素操作都直接表现为对实参 b 的元素操作。例如，在上面调用执行时，对 a[3] 的取值直接取自 b 里下标为 3 的元素，对 a[3]的赋值操作将给数组 b 里下标为 3 的元素赋值。

我们可以定义以数组为参数的函数，用它们改变实参数组元素的值，这与简单类型参数的情况不同。有关道理牵涉到指针的概念，现在请读者暂且接受这个事实：在有数组参数的函数执行时，对形参数组的元素操作就是直接对函数调用时的实参数组的元素操作。

下面是一个能反应数组参数特点的例子：函数 rev 将参数数组中的元素颠倒位置，把最后元素与最前面元素交换，其余类推。这个函数的定义如下：

```
void rev(int n, int a[]) {
    int x, i, j;

    for (i = 0, j = n-1; i < j; ++i, --j) {
        x = a[i];
        a[i] = a[j];
        a[j] = x;
    }
}
```

这个函数不需要返回值，因为它的作用就是修改实参数组。函数的循环中还是用两个下标，从两端向中间夹击。交换两个元素的值需要一个辅助变量，并要注意三个赋值的次序。可以用下面主程序测试函数 rev，确认它确实修改了被操作数组的内容：

```
int main () {
    int i, b[] = {1, 2, 3, 4, 5, 6, 7};

    for (i = 0; i < 7; i++)
        printf("b[%d] = %d\n", i, b[i]);

    rev(7, b);

    printf("After reversion:\n");
    for (i = 0; i < 7; ++i)
        printf("b[%d] = %d\n", i, b[i]);

    return 0;
}
```

## 6.4 字符数组与字符串

字符数组就是以字符为元素的数组，可用于保存文本等。由于人们常用 C 语言写与处理字符序列或文本的程序，C 为处理字符数组提供了特特别支持，本节介绍这方面情况。

### 6.4.1 字符数组

首先，字符数组也是数组，其定义方式与其他数组相同。例如，下面定义了一个包含

1000 个字符元素的数组 line：

```
char line[1000];
```

在这样定义之后，line 就可以像其他数组一样使用了。例如写：

```
for (i = 0; i < 1000 && (line[i] = getchar()) != '\n'; ++i)
    ;
```

这个循环将一行字符读入到数组 line，读入完毕后，i 记录着读入字符的个数。注意，这里我们也用条件保证对数组 line 的访问不越界。

定义字符数组时也可以像其他数组一样进行初始化，例如：

```
char city[15] = {'B', 'e', 'i', 'j', 'i', 'n', 'g'};
```

这定义了一个 15 个元素的数组，并给它的前面 7 个元素指定了初始值。与前面关于数组初始化的规定一样，city 中未指定初值的元素都自动置 0。对字符类型而言，就是把那些元素都设为编码为 0 的特殊字符值。编码为 0 的字符一般称为“0 字符”或空字符。请注意，空字符既不是表示数字 0 的字符（数字字符 0 的 ASCII 编码为 48），也不是表示空格的字符（空格的 ASCII 编码为 32），空字符字面量的表示形式是'\0'。空字符在 C 语言里有特殊作用，后面经常用到，我们马上会看到这方面的情况。

### 6.4.2 字符串

本书的第一个简单程序里已经用到字符串文字量，这是双引号括起的任意字符序列，对它们的唯一限制是不能跨越两行。关于字符串文字量的书写还有一个规定：如果顺序写出的两个或多个字符串，其间仅由空白字符分隔，编译程序就会将它们连成一个长字符串。这实际上为写程序的人提供一种写长字符串的方法。例如，下面输出语句是合法的，它将输出一个很长的字符串：

```
printf("In 1983," " the American National Standards "
  "Institute (ANSI) established a committee whose "
  "goal was to produce \"an unambiguous and machine-"
  "independent definition of the language C,\" while "
  "still retaining its spirit." "The result is the "
  "ANSI standard for C.");
```

在字符串文字量里显然不能直接写双引号，因为这将被认为是字符串的结束。读者可以看到，在上例中的双引号用换意序列“\"”表示。

字符串是与字符数组关系密切的概念。对程序中的字符串字面量，系统将用字符数组的形式存储它们：分配连续的若干存储单元，顺序存入字符串中的各个字符，每个字符存在一个字节里。这里有一个特殊规定：在存完字符串常量的所有字符之后，还要另存一个空字符'\0'作为字符串的结束标志。例如，如果在程序里写了字符串：

```
"Beijing"
```

虽然它只有 7 个字符，其内部表示却需要占 8 个字节存储，存储情况如图 6.2 所示，其中的\0 表示空字符。

| B | e | i | j | i | n | g | \0 |
|---|---|---|---|---|---|---|---|

图 6.2 字符串"Beijing"的内部表示

用这种方式表示字符串是为了处理方便。与基本类型的数据不同，不同的字符串可能有不同长度。这种情况下，程序怎么才能从内部表示确定字符串结束的位置呢？有了字符串末尾的空字符，处理字符串的程序就可以顺序检查，遇到空字符就知道遇到了字符串结束。虽然空字符不是字符串内容的一部分，但却是字符串表示中不可缺少的部分。标准库的字符串处理函数都是基于这种表示定义的，我们自己写字符串处理程序时也应该遵守这种规则。

下面的问题是：能在自己定义的字符数组里存放字符串吗？回答是肯定的，根据字符串存储形式的规定，只要在数组里顺序存入所需字符，随后存一个空字符，这个字符数组里的数据就有了字符串的表现形式，这个数组也就可以当作字符串使用了。在这种情况下，人们也说这个数组里存了一个字符串。例如有下面定义：

```
char a[5] = {'i', 's', 'n', 'o', 't'},
      b[5] = {'g', 'o', 'o', 'd'},
      c[5] = {'f', 'i', 'n', 'e', '\0'},
      d[5] = {'o', 'k', '\0'};
      e[5] = {'o', 'k', '\0', '?', '?'};
```

这时说数组 a 里存的不是字符串，因为缺少表示串结束的空字符。数组 b 和 c 里存的都是字符串。给数组 b 提供的初始化表达式个数不够，按规定，剩余位置自动置为空字符（字符值的 0 就是空字符），正好当作字符串结束标志。c 初始化时已在有效字符后加了一个空字符，所以它也存了一个字符串。数组 d 最后两个字符未给，将自动设为空字符，对 d 作为字符串没有影响。数组 e 的情况有些特殊，5 个元素都给了值，但在空字符后面又有另外两个字符，这该怎么看呢？作为字符数组，e 的 5 个元素分别有了值，意义很清楚。如果将 e 中数据当作字符串看待和处理，遇到空字符就认为串已结束，后面的东西对串处理已经没意义了。所以，如果将 e 看作字符串，那就是一个只包含两个字符的串。

为了方便使用，C 语言为字符数组提供了特殊的初始化形式：允许以字符串形式为字符数组的一系列元素指定初值。例如可以写：

```
char a1[20] = "Peking University";
```

这个定义给 a1 的前 18 个字符指定了值，不但有明确写出的 17 个字符，还有一个作为字符串结束的空字符。随后部分自动用空字符填充。这种初始化形式应看成一般形式的简写。使用时同样要注意数组长度。例如下面定义是不合法的，因为初始值的字符数过多：

```
char a2[15] = "Peking University";
```

下面定义也不合法，因为没有为存放作为字符串结束的空字符预留位置：

```
char a2[17] = "Peking University";
```

用字符串做字符数组初始化时也允许不直接给出数组元素个数。这时的数组大小规定为初始化字符串的字符数加 1，因为需要在数组最后存一个空字符。例如下面定义：

```
char a3[] = "Peking University";
```

这定义了一个 18 个元素的数组，其中依次存放各字符，最后元素存入一个空字符。

字符数组可以用在各种需要字符串的地方。例如，如果程序里许多输出语句都用同样输出格式，一种可能方法就是定义一个公用的格式描述数组。例如下面例子：

```
char outform1[] = "Two reals: %f, %f\n";
... /* 程序中可以使用这个格式描述 */
    printf(outform1, x, y);
    ... ...
    printf(outform1, s, t);
...
```

这样可以减少重复定义。对输入格式也可以采用同样技术。

### 6.4.3 程序实例

例 1（字符串复制），写一个函数，将一个字符串复制到一个字符数组里（同样做成字符串）的工作。这里有个隐含条件：假定复制用的字符串数组足够大，足以存放被复制字符和空字符。函数定义很简单，t 用 const 标记是想了把函数中不修改它的事实表示清楚：

```
void str_copy (char s[], const char t[]) {
    int i = 0;
    while (t[i] != '\0') {
        s[i] = t[i];
        ++i;
    }
    s[i] = '\0';
}
```

利用 C 语言的特点可以简化程序。由于循环结束时 t[i] 值正好是空字符，最后也需要给 s[i] 赋一个空字符。此外，赋值运算本身也有值。所以上述程序常被写成：

```
void str_copy (char s[], const char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != '\0')
        ++i;
}
```

或进一步简化为：

```
void str_copy (char s[], const char t[]) {
    int i = 0;
    while (s[i] = t[i]) ++i;
}
```

因为赋值表达式的值就是被赋的值。最后一次循环时复制空字符时，空字符的值就是 0，赋值之后检查这个值，发现是 0，正好可以控制循环结束。

例 2（二进制转换），写一个函数，给它一个表示二进制数的 0/1 字符串，它计算出这个串所表示的整数值。

将这一函数命名为 bin2int，这里取 2 的英文为 two（作为 to 的谐音，这种命名方式很常见）。它应该有一个字符数组参数，返回 int 值。由于假定字符数组里保存的是字符串，可以利用空字符判断结束，因此不必知道数组长度。

函数中的计算很简单，二进制数 $b_nb_{n-1}...b_2b_1b_0$ 的值可以用下面方式计算（与前面计算多项式值的方式同出一辙）：$(((...((b_n\times 2) + b_{n-1})\times 2 + ...)\times 2 + b_2)\times 2 + b_1)\times 2 + b_0$，据此可以直接写出一个循环，循环条件判断二进制数是否结束。由于二进制数的各个位只能是 0 和 1，只要在遇到 1 时加一就可以了：

```
int bin2int(const char s[]) {
    int i, n = 0;
    for (i = 0; s[i] != '\0' && (s[i] == '0' || s[i] == '1'); ++i) {
        n = n * 2;
        if (s[i] == 1) ++n;
    }
    return n;
}
```

由于各种字符集里的数字都是连续排列编码的，利用这种性质可以简化程序。人们常常将上述函数写成下面的样子，其中就不再需要条件语句了：

```
int bin2int(const char s[]) {
    int i, n = 0;
    for (i = 0; s[i] != '\0' && (s[i] == '0' || s[i] == '1'); ++i)
        n = n * 2 + (s[i] - '0');
    return n;
}
```

这种方式的另一优点是容易推广到其他进制（例如八进制和十进制）的数值转换，本章后面有这方面的练习。下面是一个简单的测试用主函数，写出完整程序还需要加上适当的头文件：

```
int main() {
    int n = bin2int("111111111");
    printf("%d\n", n);
    return 0;
}
```

### 6.4.4 标准库字符串处理函数

字符串是 C 程序的重要处理对象，标准库提供了许多与此有关的函数，这些函数的原

型在头文件 string.h 里说明。要使用标准字符串处理函数，程序前部应写：

```
#include <string.h>
```

下面介绍几个最常用的函数，其他函数在第十章介绍。

1. 字符串长度函数 strlen(const char s[])。本函数求出字符串的长度，也就是字符串里的字符个数。在计算字符个数时不计表示字符串结束的空字符。参数说明前面加了 const 修饰符说明函数执行中不会修改参数。
2. 字符串复制函数 strcpy(char s[], const char t[])。本函数与前面定义 str_copy 类似。第二个实参的应是字符串。strcpy 把字符串 t 复制到 s，s 应是一个足够大的字符数组，以保证字符串复制不越界。下面是使用 strcpy 的例子：

```
char a1[20], a2[20];
...
strcpy(a1, "programming");
...
strcpy(a2, a1);
...
```

   标准库还提供了一个字符串限界复制函数 strncpy，使用形式与 strcpy 类似，增加了第三个 int 类型的限界参数，用于限制复制的最大长度。字符串复制完毕或者达到限界长度时复制工作结束。例如：

```
strncpy(a1, s, 20);
```

   把字符串 s 里最多 20 个字符复制到 a1，如果 s 较短，a1 的多余部分补满\0 字符。
3. 字符串比较函数 int strcmp(const char s1[], const char s2[])。在两个字符串 s1 和 s2 相同时返回 0；字符串 s1 大于字符串 s2 时返回一个正值（并没有规定采用什么值），否则就返回负值。判断字符串大小的标准是字典序。简单地说，字典序就是普通英语词典里排列单词词条时所用的顺序，其严格定义见下面文字框里的解释。标准库还提供了一个限界比较函数，它只在给定范围内判断字符串的大小：

```
int strncmp(const char s1[], const char s2[], int n);
```

   返回值的规定与 strcmp 相同。
4. 字符串连接函数 strcat(char s[], const char t[])。第二个实参应当是一个字符串，对应第一个参数 s 的实参应是一个存放着字符串的字符数组。strcat 把作为第二个实参的字符串复制到这个实参字符数组中已有字符的后面，形成相当于两个串连在一起的字符串。这里也要求第一个实参数组足够大，使复制工作能合法完成。下面是函数使用的例子：

```
char b1[40] = "Programming", b2[10];
strcat(b1, " language");
strcpy(b2, " C");
strcat(b1, b2);
```

标准库里还提供了一个限界连接函数 strncat。

### 6.4.5 输出文本里的最长行

现在想写一个程序，它（通过标准输入）读入一个正文文件，最后输出其中最长的一行。如果同样长的最长行不止一个，则输出其中的某一行。

**字符串的字典序**

假设有两个字符串：

$$s_1 = a_1 a_2 \cdots a_n, \qquad s_2 = b_1 b_2 \cdots b_m$$

字符串字典序的定义是：逐个比较两个串中对应的字符，字符大小按照字符编码（作为一个整数）大小确定。从左向右比较，如果遇到不同字符，所遇第一对不同字母的大小关系就确定了两个字符串的大小关系；如果未遇到不同字符而某个字符串首先结束，那么这个字符串是较小的；否则，两个字符串相等。

这个问题比前面的问题复杂一些。根据我们做程序设计的经验，首先需要分析问题，将实现这个计算过程的程序结构弄清楚。不难看出，主函数基本部分的框架可以写成：

```
while (还有新输入行)
    if (新行比以前记录的最长行更长)
        记录新行及其长度;
输出所记录的最长行;
```

这里的几个操作都可以考虑用函数实现。

先考虑在处理中需要记录哪些数据。显然我们需要记录已经遇到的最长一行，因为最后需要输出这个行。记录这个行应当用一个字符数组，假定命名为 maxline。为了处理方便，这个数组的内容可以采用字符串的存储形式，在所有有效字符之后放入一个空字符。程序读入的新行也需要记录，因为如果这一行更长，就应该将它转存入 maxline。记录新行需要用另一个数组，命名 line。

下一个问题是数组长度，这必须在定义数组时给定。因为无法确知实际文件里最长行的长度，因此一般说，定义多大的数组都无法保证它足够大。这里需要做一个假定，下面假定文件中各行长度都不超过 1023 个字符，定义一个符号常量规定数组的大小。

读入行的操作用函数 getline 实现，令它把读入的东西存入参数数组，返回读入行的长度。为防止越界，用另一个参数给出数组长度。本函数的原型可以是：

```
int getline(int limit, char line[]);
```

getline 把读入的行按字符串形式存储。一般说，只要还有输入，行里一定有字符，空行也包含一个换行字符。也就是说，在有输入的情况下，getline 的返回值必定大于 0。令它输入结束时返回 0 值，用这个值控制大循环。假设 MAXLEN 是表示行数组长度的符号常量，现在大循环的条件就可以写为：

```
while (getline(MAXLEN, line) != 0) ....
```

处理中每次遇到更长的行，就需要把 line 的内容转存到 maxline，需要做一次行的复制。新行并不更长时就直接丢掉。由于数组里保存的是字符串，行复制工作可以借助于前面定义的字符串复制函数完成，原型是：

```
void str_copy(char s[], const char t[]);
```

也可以用标准库函数 strcpy。

还有一些需要解决的细节，例如，每次读完一行后需要记录其长度；完成到最长行的复制后需要更新最长行的长度记录等。有了这些分析和考虑，最后的程序已经不难完成了。下面是这个程序中主函数的定义，前面给出了所用辅助函数的原型：

```
#include <stdio.h>

enum { MAXLEN = 1024 };

int getline(int limit, char line[]);
void str_copy (char s[], const char t[]);

int main () {
    int n, max = 0; /* 记录当前行和最长行的长度 */
    char line[MAXLEN], maxline[MAXLEN];

    while ((n = getline(MAXLEN, line)) > 0)
        if (n > max) {
            str_copy(maxline, line);
            max = n;
        }

    if (max > 0)
        printf("%s\n", maxline);
    return 0;
}
```

上面程序的 main 函数里用了一个前面没用过的转换描述串：%s 表示输出时转换的数据对象是字符串。有读者可能提出，能直接用 maxline 作为函数 printf 的转换描述串吗？实际上，那样做有时会出问题，请读者自己思考，什么时候可能出问题？

函数 str_copy 非常简单，可以从前面字节抄过来，也可以用标准库的 strcpy 函数。函数 getline 也不复杂：

```
void str_copy (char s[], const char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != '\0') ++i;
}

int getline(char line[], int limit) {
    int c, i = 0;

    while (i < limit - 2 && (c = getchar()) != EOF && c != '\n') {
        line[i] = c;
        ++i;
    }

    if (c == '\n') {
        line[i] = '\n';
        ++i;
    }
    line[i] = '\0';

    return i;
}
```

请注意 getline 的定义，这里有一些细节，主要是需要避免数组越界访问。还要考虑遇到换行时的处理，正确安放结束的空字符等等。读字符循环能正确处理还依赖于 && 的求值方式，请读者自己分析。

在上面实现方式中，函数间的信息传递都通过参数完成，这种方式的优点是各函数相互独立。例如这里的 str_copy 和 getline 各自完成了一项独立功能，它们工作中所需的信息都通过参数获得，因此完全可以自由地复制到其他程序中使用。这种方式的缺点也在这里：如果这样定义的一个函数需要从外部获得许多信息，那么就需要为它提供许多参数，函数的定义和使用都比较麻烦。

下面考虑同一程序的另一种实现方式。我们将程序里的两个数组定义为外部变量，使程序里的各函数都能直接访问它们，因此也就不必作为参数传递了。下面是按照这种想法写出的完成同样工作的另一程序。

```
#include <stdio.h>

enum { MAXLEN = 1024 };

int getline(void);
void str_copy (void);

char line[MAXLEN], maxline[MAXLEN];

int main () {
    int n, max = 0; /* 记录当前行和最长行的长度 */

    while ((n = getline()) > 0)
        if (n > max) {
            max = n;
            str_copy();
        }

    if (max > 0)
```

```
        printf("%s\n", maxline);

    return 0;
}

void str_copy () {
    int i = 0;
    while ((maxline[i] = line[i]) != '\0') ++i;
}

int getline() {
    int c, i = 0;

    while (i < MAXLEN - 2 && (c = getchar()) != EOF && c != '\n') {
        line[i] = c;
        ++i;
    }

    if (c == '\n') {
        line[i] = '\n';
        ++i;
    }
    line[i] = '\0';

    return i;
}
```

请读者对这两个程序做一个比较。

从后一个程序中可以看到，外部变量同样能作为函数间传递信息的通道。因为外部变量由所有函数共享，一个函数存入外部变量的信息能传给随后使用同一外部变量的函数。由于函数可以直接访问外部变量，这种做法可以省去一些参数。此外，这种方式定义的函数的意义比较简单，因为被它们操作的变量是固定的。通过外部变量传递信息的缺陷是：这样做使所定义函数在一定意义上变成了专用函数，因为它们总对一些确定的外部变量工作。这种方式的另一缺点是造成程序不同部分间密切的互相依赖关系。

一般说，在比较复杂的程序里，人们倾向于把比较大、具有唯一出现，而且是在程序中被许多函数公用的数据对象（例如很大的数组）定义为外部变量。对于一般数据对象则采用参数方式传递。对于一个具体问题应当怎样处理，要考虑软件系统实现的方便、清晰、数据安全等许多因素。

现在回到上面的程序，请读者考虑下面问题，并对程序做一些修改：首先可以考虑增加一个语句，输出最长行的长度，这种信息也有用处。进一步说，如果实际读入的文件里确实存在超过 1023 个字符的行，上述程序将怎样工作？显然这时程序无法输出最长的行（因为 maxline 的容量不够），得到的行长信息也不对了。请读者设法修改程序，保证程序能正确输出最长行的长度，而且实际输出的也一定是最长行或者最长行的前 1023 个字符。

## 6.5 两维和多维数组

前面讨论的数组也称为一维数组，因为它们只有一个下标，所有元素呈线性一维排列，这种结构可以直接用于表示数学中的向量、数据的有限序列、成组的被处理数据对象等等。实际计算中有时需要更复杂的结构，例如在计算机的数值计算应用方面经常要表示和处理矩阵，其他方面的例子也很多。矩阵不是线性结构的东西，它们有两个维，其元素需要通过两个下标指定，程序里怎样表示这种数据呢？

C 语言里也能定义两维的和更多维的数组。在这里把两维数组看作一维数组的数组。也就是说，两维数组里的每个元素都是（成员类型相同，成员个数也相同的）一维数组。下面

定义了两个两维数组：

```
double a[3][2];
int b[4][4];
```

数组 a 包含三个元素，每个元素各是一个双精度数组，其中包含了两个 double 元素。人们常说 a 是 3×2 的双精度数组。有时也说 a 是 3 行 2 列的数组，这是从矩阵概念那里借用的说法。类似地，也说 b 是 4×4 的整型数组。

更多维的数组的定义也与此类似。下面定义了一个三维数组：

```
int a1[3][2][4];
```

如果需要，我们也可以用这种形式定义更高维的数组。

### 6.5.1 多维数组的初始化

多维数组也可以在定义时直接初始化。下面是一个例子：

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

内嵌各括号里的数据将用于初始化各个成员数组。这里要求各括号里表达式的个数都不超过成员数组长度，数据的组数不超过成员数组的个数。如果给出的表达式不够，相应成员数组的其他成分也将被自动设置为 0。

多维数组的初始化表示中也可以不写内嵌括号，采用平铺的写法。例如，上面定义也可以用下面形式给出，其效果完全相同：

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

在采用这种形式时，所列出的初始值按顺序，依次给各个成员数组的各成分置初值。如果初始值的个数不够，剩余成分也置 0。第二种形式写起来简单，但不如前一形式清晰。

如果定义时提供了对数组初始化的部分，并实际给出了全部元素的初始值，那么就可以不写被定义数组第一个下标的元素数，因为这个数可以根据初始化表示和该数组其余维的长度计算出来。这样，下面两个定义的效果与上面两个定义都一样：

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

和

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

### 6.5.2 多维数组的表示和使用

假设有了下面数组定义：

```
double a[3][2];
int b[4][4];
```

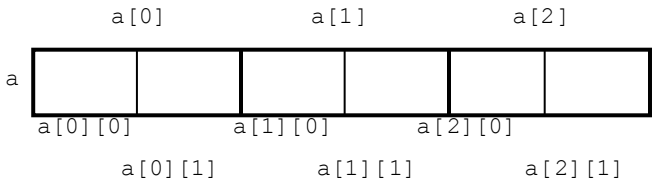



图 6.3 二维数组 a 的内部表示

在程序里，a 代表所定义的整个数组，a[0]、a[1]和 a[2]是数组 a 的 3 个元素，即 a 的 3 个成员数组，它们可以像普通一维数组一样用，特别是可以访问它们的成员。这样，a[0][1]就表示 a 的下标为 0 的成员数组中下标为 1 的元素，它可以像其他变量一样取值和赋值。下面是两维数组使用的两个例子：

```
a[2][1] = a[0][1] + a[1][1];

for (i = 0; i < 4; ++i)
    for (j = 0; j < 4; ++j)
        b[i][j] = i + j;
```

一维数组采用连续顺序存储元素的方式实现。多维数组的内部表示也完全一样，同样是依次连续存放数组元素，即其中的各成员数组），而这些成员数组也按同样方式存放它们的元素。这样，两维数组 a 的存储形式将如图 6.3 所示，在它所占据的内存中依次存放着三个成员数组。还有一个情况值得注意：数组 a 的存储开始位置也是其首成员 a[0]的存储开始位置，还是 a[0]的首成员 a[0][0]的存储开始位置。按这种存放方式，一行（一个成员

数组）的元素连续存放，这种形式又被称为*按行存放*，或者*行优先存放*。

**程序示例**

对于两个以两维数组表示的 4×4 矩阵，下面程序求出其乘积存入另一两维数组里，而后输出它。如果读者要在计算机上试验这个程序，请自己为被乘数组填入适当的数据。

我们知道，乘积矩阵的元素 $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ ，其中的 $a_{ik}$ 和 $b_{kj}$ 分别是两个被乘矩阵的元素。求一个元素的乘积需要一个循环，完成整个矩阵乘积需要用一个三重循环。

```
#include <stdio.h>

enum { N = 4 };

double A[N][N] = { ... ...}, /* 数组A和B的实际数据需要填充 */
       B[N][N] = { ... ...},
       C[N][N];

int main () {
    int i, j, k;
    double x;

    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            for (C[i][j] = 0.0, k = 0; k < N; ++k)
                C[i][j] += A[i][k] * B[k][j];

    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            printf("%f%c", C[i][j], j == N-1 ? '\n' :' ');

    return 0;
}
```

程序最后的循环将矩阵的一行输出在一行里。

### 6.5.3 多维数组作为函数的参数

前面介绍了数组参数的基本特性，以及给函数附加长度参数的必要性。现在也需要考虑定义以两维及多维数组为参数的函数的问题。举例说，实际中可能需要完成矩阵乘法或求行列式值的函数等。下面的讨论以两维数组为例，更高维数组的情况与此类似。

以多维数组作为参数时，不必给出参数说明中最左一维的长度，系统也不检查它（这一规定实际上与一维数组一致），但必须给出参数的其余各维长度。下面是一个例子，所定义函数可求出任何 n×5 的数组中所有数据的平均值（n 是任意的，通过参数指定）。

```
double aaverage(int n, double a[][5]) {
    int i, j;
    double sum = 0.0;

    for (i = 0; i < n; ++i)
        for (j = 0; i < 5; ++j)
            sum += a[i][j];
    return sum / (5 * n);
}
```

定义这种函数时要求除一维以外各维的长度，是为了确定数组参数的元素位置。但这一要求也使人没办法定义操作多维数组的通用函数。例如上例的 5 就是固定的，它能以 3×5 或 8×5 的矩阵为实参，但不能以 4×7 矩阵作为实参。这种情况当然不令人满意，因为实际中确实常常需要这方面的通用函数，例如能对任意 n×n 矩阵求和、求乘积、求行列式值的

函数。要实现这种函数就需要借助于指针，下章将介绍一种方法。

## 6.6 编程实例

本节讨论一些基于数组的编程实例，也讨论一些常见的问题和技术。原则上说，对一个问题总可以写出许多不同的程序，产生这种情况的原因在于编程工作的性质。从问题到程序要经过一个较长的工作过程，其中有许多大大小小的步骤，在许多步骤中编程序的人都需要做出选择。有些选择牵涉到对问题的不同考虑或认识，这可能引起程序之间显著的差异。有时是要在不同方式间做出简单选择，例如需要循环，用 for 或 while 结构都可以实现，这种选择产生的差异不太重要。下面提出的问题将使读者更多体会到这方面的情况。

我们不说这里给出的程序是所提问题的标准答案，只是说这是一个解，一个合理解或可能解，总希望读者自己去考虑其他解法。在学习（或者阅读有关书籍）时，应该特别注意隐藏在给出的解后面的那些选择：作者考虑了哪些问题，是怎样考虑的？做了什么选择，这些选择的合理性或不合理性？还可能怎样考虑、怎样处理等等。如果读者在学习程序设计、阅读书籍的过程中能反复提出这类问题，必然能从中受益无穷。

### 6.6.1 成绩直方图

假设文件里保存着一批学生成绩，现在要写程序读入这些成绩，产生其平均值 $M$ 和标准差 $S$，其中 $M=\frac{1}{N}\sum_{i=1}^{N}x_i$，$S^2=\frac{1}{N-1}\sum_{i=1}^{N}(x_i-M)^2$，并输出一个分段成绩的直方图。

**问题分解**

由于程序中需要反复使用这些数据，应考虑将它们存入一个数组。考虑到这个程序的工作比较复杂，我们首先将主要工作划分为若干函数。用一个双精度数组保存所有成绩，作为这个程序的基本数据。在最高层，可以将这一程序的工作分为三个步骤：输入，计算并输出统计量，计算并输出直方图。这样就可以做出程序的第一级分解：

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

/* 最大数据项数和直方图的最大高度 */
enum { NUM = 200,
       HISTOHIGH = 60 };

double scores[NUM];

int readscores(int lim, double tb[]); /*输入不超过lim个数据到tb，返回项数*/
void statistics(int num, double tb[]); /*统计tb里的num个数据项*/
void histogram(int num, double tb[], int high); /*做最高为high的直方图*/

int main()
{
    int n;
    n = readscores(NUM, scores);
    statistics(n, scores);
    histogram(n, scores, HISTOHIGH);
    return 0;
}
```

这里数组的大小 200 是一个比较随意的选择，定义为符号常量以方便修改。直方图的最大高度也定义为符号常量。输入函数的原型很自然，前面有关输出最长行的实例研究过类似问题。当然，这里的情况有所不同，需要输入的是一组数据。但那里的经验和设计值得参考。其他

函数的原型说明它们都是典型的数组处理函数。

**实现**

简单的输入很容易实现，下面是输入函数的一种实现：

```
int readscores(int limit, double tb[]) {
    int i = 0;
    while (i < limit && scanf("%lf", &tb[i]) == 1) ++i;
    return i;
}
```

求出各种统计量并输出也没有特殊困难。因为这里需要用数据项数做分母，在项数不大于 2 时会出现问题（参看计算公式）。因此程序里做了特别处理。

```
void statistics(int n, double tb[]) {
    int i;
    double s, sum, avr;

    if (n < 1) {
        printf("Data too few."
               " Can't produce mean and standard-deviation.\n");
        return;
    }

    for (sum = 0.0, i = 0; i < n; ++i) sum += tb[i];
    avr = sum/n;
    for (sum = 0.0, i = 0; i < n; ++i)
        sum += (tb[i] - avr)*(tb[i] - avr);
    s = sqrt(sum/(n-1));

    printf("Total students: %d\n", n);
    printf("Average score: %f\n", avr);
    printf("Standard deviation: %f\n\n", s);
}
```

下面考虑直方图生成。为了简单起见，考虑用字符输出横向的直方图。每个成绩段输出一段字符，这里选用 H 作为基本字符。为描述的方便，先定义一个简单的字符输出函数：

```
void prtHH(int n) {
    int i;
    for (i = 0; i < n; ++i) putchar('H');
}
```

分段长度可以用符号常量表示，根据它又可以算出分段数。为此最好是定义两个常量，而且在定义中明确刻画并利用它们之间的关系：

```
enum { SEGLEN = 5,
       HISTONUM = (100/SEGLEN)+1 };
```

现在考虑各分段的成绩数统计的实现问题。显然程序里应该用一个计数器数组保存各分段的成绩人数，将这个数组命名为 segs，其中应有 HISTONUM 个计数器。由于现在处理的是等长分段，因此可以找到一种从成绩直接得到对应计数器下标的方便方法。对 scores 里的每个元素，下面语句能够正确更新对应的计数器：

```
++segs[((int)scores[i])/SEGLEN];
```

将成绩值强制转换到 int，除以分段长度（这里是整除）后就得到计数器下标。

为了使分段统计值中最大项的直方图正好输出 HISTOHIGH 个字符，还需要求出这个最大值，以便用它去规范化其他计数值。有了所有这些数据之后，剩下的就是为输出设计一种形式的问题了。下面函数定义产生的每行输出都具有如下形式：

```
< 80:  23|HHHHHHHHHHHHHH
```

开头表示这里是那段的成绩（小于 80 分的成绩），随后是这段的人数。一个竖线符号后是表示直方图的字符序列。下面是直方图生成函数 histogram 的定义：

```
void histogram(int n, double tb[], int high) {
    int i, mx;
    int segs[HISTONUM];

    if (n == 0) return;

    for (i = 0; i < HISTONUM; ++i) segs[i] = 0; /* 初始化 */
    for (i = 0; i < n; ++i) /* 统计各分段人数 */
        segs[(int)tb[i]/SEGLEN]++;

    for (mx = 1, i = 0; i < HISTONUM; ++i) /* 为规范化找出最大个数 */
        if (segs[i] > mx) mx = segs[i];

    for (i = 0; i < HISTONUM; ++i) { /* 输出 */
        printf("<%3d: %4d|", (i+1)*SEGLEN, segs[i]);
        prtHH(segs[i]*high/mx);
        putchar('\n');
    }
    putchar('\n');
}
```

还有一些细节需要解释。这里将语句“segs[(int)tb[i]/SEGLEN]++;”中的增量运算符写成后缀形式，是为了使人更容易看清加一的对象是数组的指定元素。在使用 printf 输出时，转换描述中还可以包含许多信息（详情见第九章）。上面用的“%3d”除表示要输出一个整数外，还要求这一输出的宽度为 3 个字符。“%4d”的情况类似。

### 分析和改进

如果文件中的数据是一组 0 到 100 的数值，将它们送给这个程序（可以通过输入定向），就能得到所需的统计显示。可是如果文件中出现了不合要求的数据呢？例如，成绩中可能不小心混入了一个 178，此时上述程序会怎么样？一个程序除了应该能正确处理所有合法的输入外，实际软件系统还应该能在输入有错的情况下做出合理处置。请想一想，如果我们所用的编译程序遇到不合法程序就垮台，而且还可能破坏操作系统，还有人愿意用它吗？程序（软件）能抵御不合法数据的破坏的能力称为程序的强健性。上面的程序强健吗？

不难看到，输入函数 readscores 不会受到错误数据的破坏。由于循环条件中考虑了对输入数据量的控制，这个函数不会出现“输入缓冲区溢出”错误，在数据过多时，它完成的只是前面一部分数据的统计。在这里程序也有缺陷：在数据没有用完的情况下，它不声不响地产生了输出，却没有通知我们“工作结果可能不正确”。

统计函数很简单，它检查了输入项数以防止除零的问题。直方图函数虽然检查了项数，但却隐含着一个危险情况：由于采用了优化的实现方法，如果成绩的值不在 0 与 100 之间，“segs[(int)tb[i]/SEGLEN]++;”就可能出现数组越界，可能导致严重后果。问题还是在数据入口，那里没有对输入数据做任何检查，导致非法数据混入了后续步骤。

上面提出的问题都与数据输入阶段有关。由于已将有关工作实现为函数，现在只要修改这个函数就可以了。首先，需要检查每个输入数据项，只将合法数据存入数组。有关数据是否处理完的情况只能在循环结束后检查。修改后的 readscores 如下：

```
int readscores(int limit, double tb[]) {
    int i = 0, line = 1;
    double x;

    for ( ; i < limit && scanf("%lf", &x) == 1; ++line) {
        if (0.0 <= x && x <= 100.0) {
```

```
                tb[i] = x;
                ++i;
            }
            else printf("Data error, line %d\n", line);
        }

        if (i == limit && scanf("%lf", &x) == 1) { /* 还有数据 */
            printf("Too many data. Output is not correct.\n");
            return 0;
        }

        return i;
    }
```

修改后的程序比原来更强健了，它能合理地处置更多客观情况，遇到错误还能提供有用信息。当然，这个程序也还有许多值得改进地地方。请读者考虑如下问题：如果输入数据中不甚混入了非数字字符，例如某个成绩被输入为 8O（数字 8 和大写字母 O），修改后的程序将会如何反应？当然，需要首先想清楚，对于这一情况的合理处理方式是什么。而后再考虑应该怎样修改程序？你还能发现程序中值得改进的地方吗？请认真地想一想，如果发现了什么问题，请设法修改程序，针对你所提出的问题改进它。

### 6.6.2 一个通用带检查的整数输入函数

前一章最后的猜数实例里写了两个不同的整数输入函数。其中强调对错误输入的处理，包括格式错误和超数值范围的问题。那里提出，可以为这类函数创建一个统一模式，写出通用的带错误检查和范围检查的输入函数。现在研究如何定义一个这样的函数，以便在今后的程序里使用。这里以输入整数的函数为例，对其他类型也可以模仿这里的定义。

首先考虑函数需要哪些参数。由于需要检查输入数值范围，可以给函数提供最大值和最小值，这需要两个整型参数。在提示输入时，不同地方需要不同的提示串，为此需要增加一个提示字符串参数。还有允许用户出错的次数，允许永远错下去时一种策略，另一种策略是在若干次错误后就强行停止。这可以通过一个 `int` 参数描述，例如用小于等于 0 表示不限制出错次数，正数表示次数限制。有了这些分析，就可以给出函数的原型了：

```
int getnumber(char prompt[], int imin, int imax, int repeat);
```

还有一个问题值得考虑：如果函数运行中出错，调用函数的地方如何得到出错的信息？前一章例子里采用让函数返回特殊值的方式（那里选用的分别是 0 和-1。现在要定义的是通用函数，这个问题就不容易处理了。因为我们不知道实际调用时需要的数值范围究竟是什么。完全解决这一问题需要靠下一章的技术。现在假定存在比允许最小值 `imin` 更小的 `int` 值。这样就可以给出下面定义，其中用 `imin-1` 表示输入出错：

```
int getnumber(char prompt[], int imin, int imax, int repeat) {
    int i, n;

    for (i = 0; repeat <= 0 || i < repeat; ++i) {
        printf("%s", prompt);
        if (scanf("%d", &n) != 1 || n < imin || n > imax) {
            printf("Wrong input. Correct range [%d, %d].\n",
                   imin, imax);
            while (getchar() != '\n')
                ;
        }
        else return n;
    }
    return imin - 1;
}
```

在参数 `repeat` 小于等于 0 时循环的条件成立，因此就不会有重复次数的限制。有了这个

函数，上一章所定义两个函数的调用可以分别用下面函数调用取代：

```
m = getnumber("Choose a range [0, n]. Input n: ", 2, 32767, 5);
guess = getnumber("Your guess: ", 0, m-1, 5);
```

当然，在这两个调用之后都应该检查函数返回值，看实际输入中是否出了问题（这里是用户连续 5 次输入错误数据）。很容易修改原程序，用这个函数替代原来的函数。

我们还可以将错误信息串参数化，为此只需再给函数增加一个字符串参数。从这里可以看到一种普遍现象，这就是，在函数的通用性（使用范围）和函数使用的方便性之间有一种此消彼长的关系。一个函数越通用，它的参数就可能越多。这样，我们在使用该函数时就可以更好地根据需要指定函数的行为，但在每次使用时需要提供的信息也更多了。在考虑函数定义时需要权衡两方面的情况，确定适当的取舍。

此外，上面这类函数可能很有用，可以用在许多需要这类功能的程序里。如果我们积累起一批常用的函数，编写有关应用程序就会更方便了。这是最基本的软件重用形式，标准库的想法也源于此。随着对程序和软件研究和实践的发展，人们提出了更多重用的技术。

### 6.2.4 计算数组变量的大小

这里以一个简单问题讨论一种有时有用的技术。

假定现在需要求一组数据的平均值，设给定了数据 2.38、3.142、5.674、8.257、6.44。现在考虑用一个数组解决这个问题。这个例子要求完成的计算很简单，使用数组的定义时初始化方式等，立即可以写出如下程序：

```
#include <stdio.h>

double a[5] = {2.38, 3.142, 5.674, 8.257, 6.44};

int main () {
    int n;
    double sum = 0.0;

    for (n = 0; n < 5; ++n)
        sum += a[n];

    printf("Average: %f\n", sum / 5);
    return 0;
}
```

这个程序是对的，但它有一个小缺点：假如想用这个程序计算另一组数据，如果数据的个数不是 5，那就需要在程序中找出整数 5 的所有出现，并把它们都修改好。如果希望程序更易修改，数组定义写 a[]更合适，这可以去掉了一个需要维护的依赖关系。另一个小修改是定义一个符号常量，把程序里的两个 5 都换为这个符号常量：

```
#include <stdio.h>

enum { NUM = 5 };
double a[] = {2.38, 3.142, 5.674, 8.257, 6.44};

int main () {
    int n;
    double sum = 0.0;

    for (n = 0; n < NUM; ++n)
        sum += a[n];
    printf("Average: %f\n", sum / NUM);
    return 0;
}
```

经过这些改动后，如果需要换一组数据，除填入数据外，所有修改都只需在程序最前面进行。

唯一需要注意的是数清改动后数组的元素个数，根据它定义 NUM 的值。

忘记修改符号常量定义还是会带来问题。如果新数据较多，求和循环将不能处理完所有数据，程序输出的是数组中一部分数据的平均值，这是一个计算错误。如果新数据少，产生的问题更严重。因为循环仍会按 NUM 的值进行，从而造成数组越界访问，最后给出的结果也不会任何意义。这种情况还可能造成程序无法完成计算，甚至造成系统的破坏。这里问题的实质是：在数组大小和常量值定义之间存在一种内在的互相依赖关系，而上面给出了两个互相独立的定义，其内在关系并没有得到反映。

利用 sizeof 运算符可以处理这里的问题。sizeof 是 C 语言里的一个运算符，它能求出一个类型或者一个变量的大小。类型的大小是个整数，表示该类型的变量所占内存单元个数（C 语言规定这个数值以 char 类型大小的倍数计算。实际上可以不关心到底以什么大小为单位，因为系统保证这一计算总以同样单位进行。例如，在这里的程序中关心只是相对值，与具体的值无关。）；变量大小就是它所占的内存单元数。对于目前问题，可用 sizeof(a) 求出数组 a 的大小，用 sizeof(a[0])求出元素大小。这样，sizeof(a)/sizeof(a[0]) 的值正好是数组 a 的元素个数。利用宏定义机制，可以把程序改成下面的样子：

```
#include <stdio.h>

double a[] = {2.38, 3.142, 5.674, 8.257, 6.44};
#define NUM(ax) (sizeof(ax) / sizeof(ax[0]))

int main () {
    int n;
    double sum = 0.0;

    for (n = 0; n < NUM(a); ++n)
        sum += a[n];
    printf("Average: %f\n", sum / NUM(a));
    return 0;
}
```

对这个程序，如果需要换数据，就只需把新数据写进数组，重新编译后就可以运行了。无论所提供的数据个数是否改变，这个程序都能算出正确结果，也不会出现任何问题。

运算符 sizeof 看作普通一元运算符，优先级和结合顺序与其他一元运算符相同。C 语言规定：变量名或表达式可直接作为 sizeof 的运算对象（计算变量或表达式结果类型的大小）；类型名需要写在括号里。因此，上面计算数组元素个数的表达式也可以写为：

```
(sizeof a / sizeof a[0])
```

在 sizeof 和变量名之间必须有空格。对类型的规定是考虑到类型描述可能由多个标识符构成，例如：sizeof(unsigned long)，没有括号就不行了。

### 6.6.3 统计 C 程序里的关键字

考虑下面编程实例，完成它可能用到许多学过的东西。这里不打算给出完整程序，而是把重点放在问题分析上，将其余工作留给读者。最后还要提出许多问题供读者考虑。

**问题：**写一个程序统计在一个 C 语言源程序文件中各 ANSI C 关键字出现的次数。

首先要确定程序如何得到输入。根据到目前为止的知识，一个合理安排是让程序从标准输入读文件，实际文件可以通过输入定向的方式送给程序处理。C 程序也能处理命名文件，但现在还没有讨论有关情况（有关内容在第八章介绍），现在暂不考虑。

C 语言源程序文件就是一个字符序列，其中的换行符只是一种分隔符，不需要特殊处理。由于要处理字符序列，可以考虑用 getchar 完成基本输入。函数 scanf 是另一种选择。读者不妨考虑一下如何以 scanf 作为基本输入函数实现这个程序，有什么困难等等。

**函数参数与 sizeof 运算符**

读者可能想起前面介绍的 sizeof 运算符，可能提出能否利用 sizeof 运算符，由形式参数出发计算出实际参数数组的大小呢？这个问题的回答是否定的。

函数里对数组形参使用 sizeof，求出的是这个形参（这是一个局部变量）的大小，而不会是某次调用的实际参数数组的大小。sizeof 是编译时处理的运算符，编译后“sizeof 变量名”被实际求出的一个整数值取代。对数组形参的 sizeof 求出的是一个指针的大小，与实参数组无关。指针是下一章将要讨论的问题。

定义包含数组参数的“通用”函数，最合理方式就是加一个数组长度参数。这方面的技术细节牵涉到指针概念和数组参数的实现方式，在下一章里详细讨论。C99 标准在这里有些改变，请参考有关材料。

现在需要统计关键字，而关键字具有标识符的形式。C 语言标识符有严格形式定义，识别它们并不困难。这提供了一种解决问题的线索：顺序识别源程序里的一个个标识符；得到一个标识符后判断它是否关键字；在遇到关键字时更新统计数据。这套想法形成了一个解决方案。注意，这是一个大选择，采纳它决定了后面工作中的许多东西。另一可能方案是在读入字符的过程中直接统计关键字，例如，读到 f 后考查能否读到 or，成功时将 for 的计数值加 1。读者不妨沿这条思路继续考虑能否写出程序，有什么困难，如何克服。

识别标识符是具有逻辑独立性的工作，应当考虑定义为函数。例如 getident。为了后续处理的需要，getident 不但要识别标识符的开始和结束，还应把标识符存起来。这里可以用函数参数传递信息，也可以采用外部数组变量。下面考虑用数组参数。

getident 需要通告是否真读到标识符，通过返回值是一种可行方式。我们让函数返回读到的标识符长度，返回 0 表示没有标识符了，文件已经处理完毕（有没有其他方式向调用函数的地方传递信息？请考虑可以用什么方式？有什么优缺点？）。再为 getident 引进一个读入长度限制，就可以得到它的原型了：

```
int getident(int limit, char id[]);
```

至此，主函数中核心部分的框架已经可以写出来了：

```
while (getident(MAXLEN, s) > 0)
    if (s是某个关键字)
        相应计数器加1;
输出结果;
```

其中的 MAXLEN 是一个定义好的符号常量，表示保存标识符的数组的长度。

getident 的实现可以参考前面单词计数程序实例，可以先画出一个状态转换图，弄清什么情况下遇到了标识符开始，什么时候标识符结束。读者应记得 C 语言的规定：标识符是由字母开头的连续字母数字序列，下划线字符也作为字母看待。显然，前面介绍的字符分类标准库函数可以在这里发挥作用。

getident 还需要把读到的字符存入参数数组，并记录存入的字符个数。为了使用方便，这里可以采用字符串形式（下面假定这样做）。完成一个标识符的识别后，应在数组中已有字符后面放空字符。当然，也可以不用字符串形式（此时应该怎样处理？）。还要考虑一个麻烦的问题：C 标识符没有长度限制。你的程序能正确处理极长的标识符吗？

这个函数还有许多细节：源程序里可能有注释、字符常量、字符串常量，还有数和标点符号等。这些都应在查寻下一标识符的过程中跳过去。将标识符识别的工作定义为函数之后，问题也都孤立在 getident 里了，与函数外的程序无关，可以推后再仔细考虑。

要判断标识符是否关键字，必须记录所有关键字的信息。前面确定 getident 把取得的标识符做成字符串形式，如果也用字符串表示关键字，比较起来就很容易。ANSI C 共有 32 个关键字，可考虑用数组表示。可以用一个两维数组，其每个元素是一个字符数组，存放一个关键字的字符串。ANSI C 最长的关键字有 8 个字符，我们可以定义一个 32×9 的两

维字符数组。下面定义还包含了初始化：

```
char keywords[32][9] = {
    "auto",     "break",    "case",     "char",
    "const",    "continue","defualt", "do",
    "double",   "else",     "enum",     "extern",
    "float",    "for",      "goto",     "if",
    "int",      "long",     "register","return",
    "short",    "signed",   "sizeof",   "static",
    "struct",   "switch",   "typedef", "union",
    "unsigned","void",     "volatile","while"
};
```

这样，keywords[n]就是表示某个关键字的字符串，可以用标准库函数 strcmp 完成字符串比较。将一个标识符字符串顺序与 keywords 中的关键字字符串比较，就可以确定这个标识符是否关键字，是哪一个关键字。

要分别统计各关键字的出现次数，可以采用一个计数器数组，定义：

```
int counters[32];
```

最方便的方式是用 counters[n]记录关键字 keywords[n]的出现次数。程序开始时应把所有计数器置 0。有了这些准备之后，程序的计数部分就可以写为：

```
for (n = 0; n < 32; ++n)
    if (strcmp(s, keywords[n]) == 0) {
        counters[n]++;
        break;
    }
```

假设字符数组变量 s 里面存放着读入的标识符。

这个程序片段里做了一件许多程序都经常做的工作：设法确定某个东西是否存在于一组数据之中，这种工作称为*检索*或*查找*。这里检索是一组字符串，它们存在一个数组里；工作方式是顺序比较被查数据与数组里的数据。这种方式称为*顺序存储*和*顺序检索*。由于许多程序里都需要做检索，有时被处理的数据集合非常大，顺序检索中一个个地比较的速度很慢。人们对检索问题做了许多研究，提出了许多数据组织方法和检索方法。采用不同数据组织方法和检索方法，又会导致许多不同的程序。

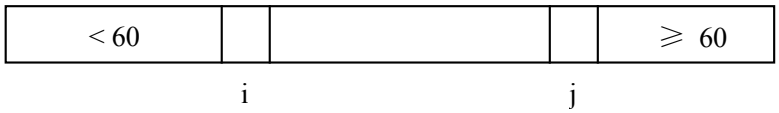
至此，我们已对问题的许多方面做了深入分析，提出了一个解决方案，写出程序已不太困难了。剩下的主要问题是 getident 的实现，这个工作留给读者完成。虽然读者可以在某些参考书上找到别人的程序，但我们相信，在任何地方能找到的程序都不是完美的。下面再提出一些实现程序时可以考虑的问题。如果读者看到书籍上的程序，那么也不妨看看想想，它们解决了下面的哪些问题？解决时是怎么考虑的？其解决方案有什么优点和不足？什么时候程序的统计会出问题等等。自己实现程序时也应该仔细考虑这些问题。如果在分析中还能提出更多的问题，那就更好了，说明你在学习中真正动了脑筋。

1. 应该用多大的数组存标识符？C 语言对标识符长度没有限制，而前面考虑 getident 定义时加了长度限制参数。请考虑该参数的作用，如果遇到超长标识符，你的程序（或其他书上的程序）能正确统计吗？什么时候会出问题？出什么问题？
2. 遇到标识符超长时应该采取什么处理原则？怎样保证统计结果正确？你的程序(或你读到的程序）正确处理了这一问题吗？
3. 考虑 C 语言关键字，你注意到它们的长度都不超过 8 个字符这一重要特性了吗？你的程序利用了这个特性吗？利用它能否使统计的工作更简单？能否及早发现某标识符不可能是关键字？（提示：把 getident 实现为“取得下一长度不超过 8 的标识符”）。
4. 最后，要使你的程序完善，还必须考虑 C 程序中的各种成分。如注释、字符常量、字符串常量等。仔细分析这些问题，修改你的程序，使它在任何情况下都能够立于不败之地。这绝不是很容易的事，但是在努力做到这一点的过程中，你会学到许多东西。

### 6.6.4 数组划分*

现在回到 6.2.3 节提出的问题，考虑将输入的学生成绩分两段输出，首先输出不及格的成绩，而后输出及格的成绩。那里用两次循环解决了这一问题。

下面考虑另一种解决问题的方法，并以此为例介绍一种复杂程序中有时可能用到的技术，元素划分（这一方法技术性比较强，读者可以将来再读）。在将数据存入数组后，我们可以考虑调整数组中元素的位置，将所有小于 60 的分数调到所有大于等于 60 的分数之前，而后就可以直接输出了。为正确完成元素的位置调整，请考虑下面的示意图：




我们考虑用两个整型变量 i 和 j 作为数组下标，调整中将它们从数组的两端逐步向中间移动，在移动过程中始终保证位于 i 左边的数组元素都小于 60，而位于 j 右边的数组元素都不小于 60。这样，当 i 移至 j 的右边时，数组元素就整理好了。这一图形也表示了下面（完成调整的）循环中所维持的不变关系。

循环开始时让 i 取值 0，j 取值 n-1，这时不变关系成立（参看上图）。下面考虑 i 的右移和 j 的左移。只要下标为 i（或者 j）的元素满足条件，就可以直接增加 i 值（减小 j 值）而不会破坏不变关系。这样直接移下标直至相应数组元素不符合要求为止。在下面程序段中两个内部 while 循环结束时，i 移过的数组元素都小于 60，而 j 移过的数组元素都大于等于 60。如果这时 i<j 仍成立，就说明 i 所指元素不小于 60，而 j 所指元素小于 60。交换这两个元素后将 i 和 j 各向内移一步，循环不变关系仍然成立。

```
for (i = 0, j = n - 1; i < j; ) {
    while (i <= j && scores[i] < PASS) ++i;
    while (i <= j && scores[j] >= PASS) --j;
    if (i < j) {
        x = scores[i];
        scores[i] = scores [j];
        scores [j] = x;
        ++i; --j;
    }
}
if (i == j && scores[i] < PASS) ++i;
```

这段程序结束时，scores 里下标 0 到 i-1 的一段都是小于 60 分的成绩，下标 i 到 n-1 的一段都是不小于 60 的成绩，i 就是不及格学生人数。注意，两段都可能为空。顺序输出数组元素就能满足题目要求，有关主函数可以写为：

```
int main () {
    int i, j, n = 0;
    double x;

    while (n < NUM && scanf("%lf", &scores[n]) == 1) ++n;
    for (i = 0, j = n - 1; i < j; ) {
        while (i <= j && scores[i] < PASS) ++i;
        while (i <= j && scores[j] >= PASS) --j;
        if (i < j) {
            x = scores[i];
            scores[i] = scores [j];
            scores [j] = x;
            ++i; --j;
        }
    }
    if (i == j && scores[i] < PASS) ++i;

    for (j = 0; j < n; ++j) printf("%f\n", scores[j]);
    printf("Fail: %d\n", i);
    printf("Pass: %d\n", n - i);
```

```
    return 0;
}
```

上面讨论了如何按某个值将数组元素划分为两部分，现在将那里的代码包装成一个划分函数。这也是一个修改实参数组的函数，给它一个数组和一个划分值，它将小于该值的元素值移到数组前一段，将大于等于该值的元素移到数组后一段。我们让函数返回划分后的分界位置。有了前面的分析，不难给出下面定义：

```
int partition(int num, double a[], double cut) {
    double x;
    int i = 0, j = num - 1;
    while (i < j) {
        while (i <= j && a[i] < cut) ++i;
        while (i <= j && a[j] >= cut) --j;
        if (i < j) {
            x = a[i];
            a[i] = a [j];
            a[j] = x;
            ++i;
            --j;
        }
    }
    if (i == j && a[i] < cut) ++i;

    return i;
}
```

有了这个函数，完成前面程序所要求的划分工作就非常简单了。只要写：

```
i = partition(n, scores, PASS);
```

### 问题解释

1) (6.3) 除非对每个数组写一个函数。
2) (6.4.3) 直接用字符串 maxline 作为函数 printf 的转换描述串可能引起问题，因为如果行里有字符 %，它会被看作是一个格式描述，这样就可能引起执行中的错误。

### 本章讨论的重要概念

数据构造机制，复合数据对象，复合数据类型，数组，数组元素，下标（指标），下标运算符，越界访问，sizeof 运算符，数组参数，字符数组，字符串，头文件<string.h>，字符串的字典序，字符串连接，多维数组，划分。

### 练习

1. 写一个程序，它能够计算并输出杨辉三角形（帕斯卡三角形）前面的 n 行。
2. 写程序统计输入文件中各字符出现的频率，并打印输出一个频率图，用形象的方式显示各个字符在文件中出现的数目。首先写出用横向的图显示的程序，然后考虑如何做出纵向的字符频率图。
3. 写一个程序，它从标准输入读入字符文件，输出其中所有的字符数不超过 10 个的行。
4. 修改正文中的“筛法”程序，使它在循环中确定了素数时就直接输出。将这样写出的程序与正文中的程序做一些比较（从简单，清晰等各种角度）。
5. 1）写一个把数字字符串转换成整数的函数，它只有一个字符数组参数。2）为这个函数增加一个表示“基数”的参数。3）类似地，写一个把合乎 C 语言实数文字量形式的字符串转换成一个双精度数的函数。

6. 写函数 squeeze(char s1[], char s2[])，它从字符串 s1 中删除串 s2 里包含的所有字符（而且保证剩下的字符仍然按照原来顺序连续排列，形成字符串）。
7. 写一个函数，它判断一个整数（或浮点数）是否在一个数组中出现。如果出现，给出第一次出现位置的下标；不出现时给出值-1。
8. 写一个函数，它统计出一个整数在一个数组（都通过参数提供）里出现的次数。
9. 设有 n 个人围成一个圆圈，从编号 m 的人开始由 1 开始报数，每个正好报到数 k 的人退出游戏，后面的一个人重新由 1 开始报数。求出最后剩下的那个人的编号。
10. 写程序读入正文文件，并统计其中各种长度的单词出现次数。设法用比较形象的方式显示统计结果。这是一项很简单的文字材料统计工作。
11. 写一个程序，它读入一个文件，输出其中最长的词（参考前面章节对“词”的定义，你也可以自己规定，例如规定词是由字母开头的字母数字序列）。考虑用函数的局部变量或者外边变量实现程序的两个版本。
12. 1）实现一个求由数组表示的多项式的值的函数；
    2）写一个求数值数组中最大值的函数；
    3）用上面的函数求出一个多项式在一系列等距点的值，利用其中的最大值将前一步得到的所有值进行规格化（这样可以防止产生的图形太长或太宽），并设法用星号在屏幕上显示多项式的近似图形。
13. 回文是从前向后和从后向前读起来都一样的句子。例如英文中的：

    amanaplanacanalpanama

    其原文是：A man, a plan, a canal, Panama。再比如中文的“落叶秋叶落”等。写一个函数，它能够判断一个字符串是否为一个回文。如果你 C 的系统能够处理中文的字符串，也为中文定义一个函数。
14. 设法写一个程序，对于给定整数 n，它能输出 1 到 n 之间的数的所有不同排列。
15. 1）请写一个程序，它输入一个学生成绩文件，输出按照每 10 分一个成绩段的学生人数。
    2）请写一个程序，它输入一个学生成绩文件，输出其中的不及格（小于 60 分）、及格（60≤X＜75）、良好（75≤X＜85）、优秀（85≤X）的学生人数。
16. 修改正文中有关有关求最长行程序实例，使之“总能”正确输出最长行的长度，输出的总是最长的那一行或者那一行中前面一段（如果该行超过数组容量）。
17. 按照文中有关定义处理多维数组的函数的规定，写出求 4×4 的矩阵和、数乘，求其行列式值的函数。
18. 根据需要扩展文中定义的通用的带检查的输入函数。
19. 按照本章最后的讨论实现一个 C 语言关键字的统计程序。回答讨论中所提出的问题。修改你的程序，使它能够正确处理任何 C 程序。