虽然代码清单 6.2 中的程序可以正常地运行[①]，但是处理的过程（算法）实在是够啰嗦的。如果需要排序的数据有 1000 个，那么就需要定义 1000 个变量。用于比较其中数值大小的 if 语句，更是需要约数十万个程序块。应该没有人想写这么麻烦的程序吧。也就是说，为了实现想要实现的算法，有时不能只依靠离散的变量。

6.2　要点 2：了解作为数据结构基础的数组

在实际应用的程序中经常需要处理大量的数据，比如那种用于统计 1000 名职员的工资之类的程序。在这类程序中存储数据时使用的是“数组”，而不是定义出 1000 个变量以供使用。通过使用数组，既可以同时定义出多个变量，又可以提高编写程序的效率。在上一节的例子中，分别定义了 a、b、c 三个变量，其实可以换一种定义变量的方法，那就是只定义一个含有 3 个元素（包含 3 个数据）的数组。在用 C 语言编写的程序中，是通过指定数组名和数组所包含的元素个数来定义数组、以供使用的（如代码清单 6.3 所示）。

代码清单 6.3　使用含有 3 个元素的数组

```
char x[3];  /* 定义数组 */
x[0] = 123; /* 把数据存入数组的第 0 个元素中 */
x[1] = 124; /* 把数据存入数组的第 1 个元素中 */
x[2] = 125; /* 把数据存入数组的第 2 个元素中 */
```

数组实际上是为了存储多个数据而在内存上集中分配出的一块内存空间，并且为这块空间整体赋予了一个名字。在代码清单 6.3 中，通过定义数组，操作系统就分配出了一块用于存储 3 个数据所需的内存空间，并将这块空间整体命名为 x。可以通过在“[”和“]”之间指定序号（索引）的方式分别访问数组内的各块内存空间。

① 需注意代码清单 6.2 给出的只是代码片段，无法直接运行。——译者注

本例中通过“char x[3];”这条语句就分配出了数组整体所需的内存空间，其中每个元素的内存空间可以通过 x[0]、x[1]、x[2] 的方式进行访问。虽然本质上还是定义出了 x[0]、x[1]、x[2] 三个变量，但是比起单独使用 a、b、c，使用数组可以更加高效地编写出能够实现排序等算法的程序。具体的例子将在稍后展示。

数组是数据结构的基础，之所以这么说是因为数组反映了内存的物理结构本身。在内存中存储数据的空间是连续分布的。而在程序中，往往要从内存整体中分配出一块连续的空间以供使用。如果用程序中的语句表示这种分配使用方式的话，就要用到数组（如图 6.2 所示）。

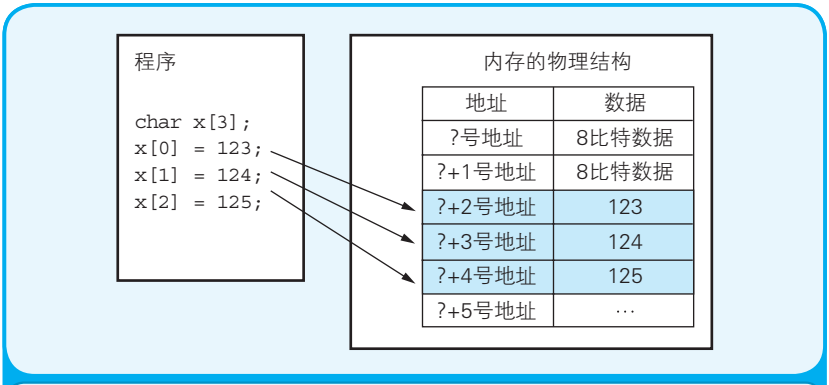

图 6.2 数组反映了内存物理结构本身

6.3 要点 3：了解数组的应用——作为典型算法的数据结构

数组是数据结构的基础，只要使用数组就能通过程序实现各种各样的算法以处理大量的数据。代码清单 6.4 中列出的程序使用了第 5 章中所介绍的名为“线性搜索”的典型算法，用于从数组 x 所存储的 1000 个数字中查找（Search）777 这个数字。在这段程序中没有使用“哨兵”。

代码清单 6.4　使用线性搜索算法查找数据

```
for (i = 0; i < 1000; i++){
    if (x[i] == 777){
        printf("找到 777 了！ ");
    }
}
```

在 C 语言中，for 语句具备反复执行某种处理的功能。因此为了从头到尾连续地处理数组中的元素，往往需要使用 for 语句。这段程序中除了数组 x 还定义了一个变量 i，在 for 这个关键词后面的小括号中，要写上使变量 i 从 0 到 999 每循环一次就增加 1 的代码。于是就得到了这么一个代码片段。

```
for (i = 0; i < 1000; i++) {
```

在 C 语言中是通过用“{”和“}”将若干条语句括起来，表示程序中的程序块（具有一定意义的语句集合）的。通过这种方式写在 for 语句程序块当中的 if 语句就会随着变量 i 的值的增加而被反复执行 1000 次，在这里 if 语句的作用是判断是否已经找到了 777。

通常把像变量 i 这样的用于记录循环次数的变量称为循环计数器（Loop Counter）。数组之所以方便，就是因为可以把循环计数器的值与数组的索引对应起来使用（如图 6.3 所示）。

循环计数器的值	被处理的数组中的元素
0	x[0]（数组的开头）
1	x[1]
2	x[2]
…	…
999	x[999]（数组的结尾）

图 6.3　把循环计数器的值和数组的索引对应起来

接下来就试着用“冒泡排序”这种典型算法，将存储在数组中的1000个数字按降序排列吧。程序如代码清单6.5所示。在冒泡排序算法中，需要从头到尾地比较数组中每对儿相邻的元素的数值，然后反复交换较大的数值和较小的数值的位置。

代码清单6.5 通过冒泡排序算法排列数据

```
for (i = 999; i > 0; i--){
    for (j = 0; j < i; j++){
        if (x[i] > x[j]){
            tmp = x[i];
            x[i] = x[j];
            x[j] = tmp;
        }
    }
}
```

在这里没有必要去深究这个程序的流程，这之后展示出的代码也是如此，诸位只要粗略地浏览一下抓住其大意就OK了。这里只希望诸位能关注一点，即通过使用数组和for语句，就能编写出实现了线性搜索和冒泡排序算法的程序。

6.4 要点4：了解并掌握典型数据结构的类型和概念

数组是一种直接利用内存物理结构（计算机的特性）的最基本的数据结构。只需使用for语句，就可以连续地处理数组中所存储的数据，实现各种各样的算法。但是在现实世界中也有一些数据结构，仅凭借数组是无法实现的，比如有的数据结构可以把数据堆积得像小山一样，有的数据结构可以把数据排成一队，有的数据结构可以任意地改变数据的排列顺序，还有的数据结构可以把数据分为两路排列，等等。为了用程序实现这些数据结构，就必须要设法改造数组，但是与之相应的内存的物理结构又是改变不了的。这可怎么办才好呢？

就像在算法中有典型算法一样，在数据结构中也有典型数据结构（如表 6.1 所示），它们都是由老一辈程序员发明创造的。这些数据结构其实都是通过程序从逻辑上改变了内存的物理结构，即数据在内存上呈现出的连续分布状态。接下来笔者会依次介绍每种典型的数据结构，所以请诸位抓住它们各自的特点。

表 6.1　主要的典型数据结构

名称	数据结构的特征
栈	把数据像小山一样堆积起来
队列	把数据排成一队
链表	可以任意地改变数据的排列顺序
二叉树	把数据分为两路排列

“栈”（Stack）的本意是干草堆（如图 6.4 所示）。在牧场中，把喂家畜吃的干草堆积在地上就会形成一座小山。为了把干草堆成山就要从下往上不断地堆积。在程序中干草就相当于数据。而在给家畜喂食的时候，则要按照从上往下的顺序把堆积起来的干草（数据）取下来。也就是说，数据的使用顺序与堆积顺序是相反的。通常把这种存取方式称为 LIFO（Last In First Out，后进先出），即最后被存入的数据是最先被处理的。在那些作为程序处理对象的实际业务中，可以用栈来模拟诸如堆积在桌子上的文件等场景。既然无法马上处理，就暂且先都堆放在栈里吧。

“队列”（Queue）就是等待做某事而排成的队。笔者经常要在东京的西日暮里站从营团地铁换乘日本铁路。下了地铁就要去买日本铁路的车票，在购票窗口前买票的乘客会排成一队。这就是现实世界中的队列（如图 6.5 所示）。队列与栈正相反，排在队头的乘客可以最先买到车票。通常把这种形式称为 FIFO（First In First Out，先进先出），即最先被存入的数据也是最先被处理的。当无法一下子处理完数据的时

候，就可以暂且先把这些数据排成队。之后会介绍队列的数据结构，其实现方式一般是把数组的首尾相连，形成一个圆环。

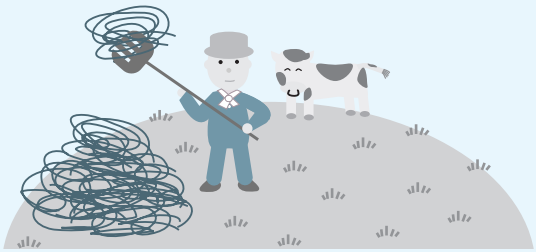

图 6.4 栈的示意图

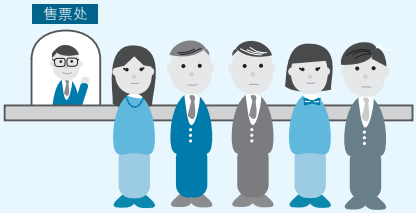

图 6.5 队列的示意图

“链表”的概念就相当于几个人手拉着手排成一排（如图 6.6 所示）。某个人只要松开拉住的那只手，再去拉住另一只手，这一排人（相当于数据）的排列顺序就改变了。而只要先松开拉住的手，再让一个新人加入进来并拉住他的手，就相当于完成了数据的插入操作。

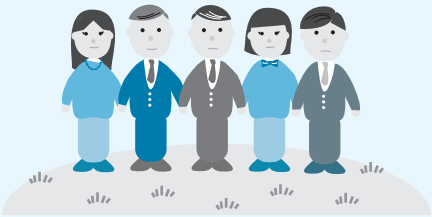

图 6.6　链表的示意图

“二叉树”的概念正如其名，就相当于一棵树。不过这棵树与自然界中的树稍有些不同，二叉树从树干开始分权，树枝上又有分权，但每次都只会分为两权，在每个分权点上有一片叶子（相当于数据）（如图 6.7 所示）。稍后诸位就会了解到二叉树其实是链表的特殊形态。

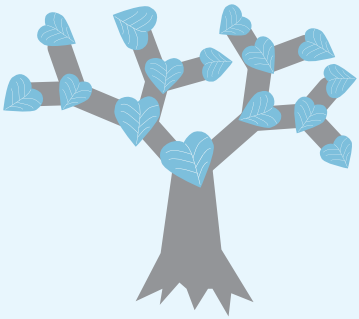

图 6.7　二叉树的示意图

6.5　要点 5：了解栈和队列的实现方法

栈和队列的相似点在于，它们都可以把不能立刻处理的数据暂时存储起来；不同点在于，栈对所存储数据的存取方式是 LIFO 的，而队

列对所存储数据的存取方式是 FIFO 的。既然诸位已经了解了栈和队列的概念，接下来笔者就开始讲解如何用程序表示这两种数据结构吧。同样是数组，处理手段不同，得到的数据结构也会不同，数组有时可以转化为栈，有时可以转化为队列。

在实现栈这种数据结构时，首先要定义一个数组和一个变量。数组中所包含的元素个数就是栈的大小（栈中最多能存放多少个数据）。变量中则存储着一个索引，指向存储在栈中最顶端的数据，该变量被称为“栈顶指针”。栈的大小可以根据程序的需求任意指定。假设最多也就有 100 个数据，那么定义一个能把它们都存储下来的栈就可以了，这样的话就可以定义一个元素数为 100 的数组。这个数组就是栈的基础。接下来编写两个函数，一个函数用于把数据存入到栈中，也叫作压入到栈中；另一个函数用于从栈中把数据取出来，也叫作从栈中弹出来。在这两个函数中，都需要更新栈中所存储的数据的总数，以及更新栈顶指针的位置。也就是说通过使用由数组、栈顶指针以及入栈函数和出栈函数所构成的集合，就能实现栈这种数据结构了（如代码清单 6.6 和图 6.8 所示）。

代码清单 6.6　使用数组、栈顶指针、入栈函数和出栈函数实现栈

```
char Stack[100];         /* 作为栈本质的数组 */
char StackPointer = 0;   /* 栈顶指针 */

/* 入栈函数 */
void Push(char Data){
    /* 把数据存储到栈顶指针所指的位置上 */
    Stack[StackPointer] = Data;
    /* 更新栈顶指针的值 */
    StackPointer++;
}

/* 出栈函数 */
char Pop(){
    /* 更新栈顶指针的值 */
```

```
    StackPointer--;
    /* 把数据从栈顶指针所指的位置中取出来 */
    return Stack[StackPointer];
}
```

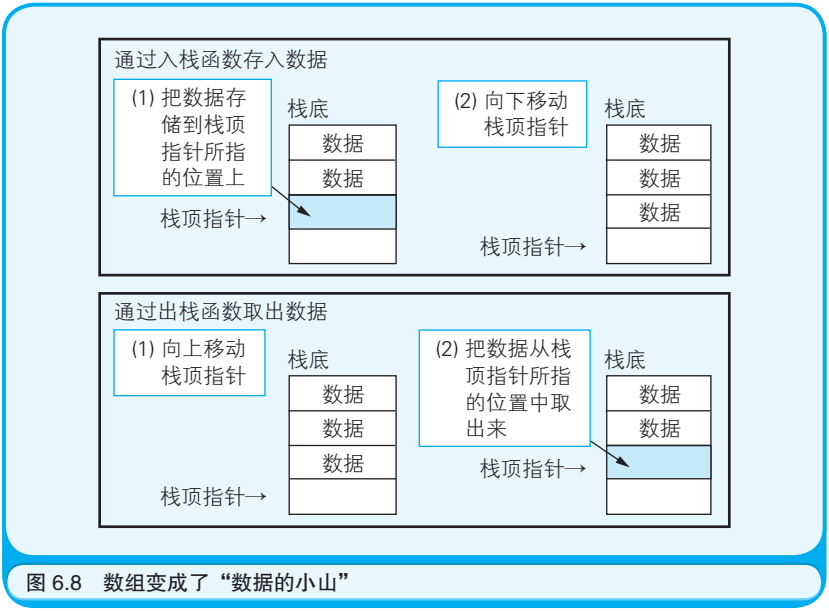

图 6.8　数组变成了“数据的小山”

为了实现队列这种数据结构，以下元素是必不可少的：1. 一个任意大小的数组；2. 一个用于存放排在队头的数据对应的索引的变量；3. 一个用于存放排在队尾的数据对应的索引的变量；4. 一对儿函数，分别用于把数据存入到队列中和从队列中把数据取出来。如果数据一直存放到了数组的末尾，那么下一个存储位置就会折回到数组的开头。这样就相当于数组的末尾就和它的开头连接上了，于是虽然数组的物理结构是“直线”，但是其逻辑结构已经变成“圆环”了（如代码清单 6.7 和图 6.9 所示）。

代码清单 6.7 使用一个数组、两个变量和两个函数实现队列

```
char Queue[100];          /* 作为队列本质的数组 */
char SetIndex = 0;        /* 标识数据存储位置的索引 */
char GetIndex = 0;        /* 标识数据读取位置的索引 */

/* 存储数据的函数 */
void Set(char Data){
    /* 存入数据 */
    Queue[SetIndex] = Data;
    /* 更新标识数据存储位置的索引 */
    SetIndex++;
    /* 如果已到达数组的末尾则折回到开头 */
    if (SetIndex >= 100){
        SetIndex = 0;
    }
}

/* 读取数据的函数 */
char Get(){
    char Data;
    /* 读出数据 */
    Data = Queue[GetIndex];
    /* 更新标识数据读取位置的索引 */
    GetIndex++;
    /* 如果已到达数组的末尾则折回到开头 */
    if (GetIndex >= 100){
        GetIndex = 0;
    }
    /* 返回读出的数据 */
    return Data;
}
```