*第六章 容器的集合—— 数组*

第一节：一维数组

第二节：一维数组与函数

第三节：字符数组

第四节：字符数组与函数

第五节：二维数组

第六节：二维数组与函数

第七节：多维数组

在任何一门程序设计语言中，变量都起着关键的作用，变量提供了储存各种数据的能力，没有变量的程序通常没有被执行的意义。然而，在面对大量的数据的时候，变量就显得很无力了。假设我们要储存一个班的所有同学的学号(int型数据)，如果我们用变量来实现这个需求的话，就不得不声明几十个变量，仅仅写声明变量的代码就足够我们写到手软。

既然一个班的同学的学号是一些相关联的数据，那么我们可不可以把它们融合成一个组呢？答案是可以的，而把相同类型的数据编成组就需要使用到数组（**array**）。

6.1一维数组

一维数组是最简单的数组。数组是变量的集合，因此数组本身也算是一种变量，在使用一个变量之前需要先声明它，同样地，使用一个数组前也许要先声明。

声明数组的语法格式是：数据类型 数组名[表达式];

可以看出，数组的声明方式和变量是十分类似的，中括号内的表达式就是数组的大小。比如int a[10];就声明了一个int型数组，这个数组的大小是10，能够储存最多10个int型数据，对于数组来说，单个的被称为数组的元素。

**小提示**

通常情况下，中括号内的表达式都应该是常量，但是C99之后的C语言标准规定表达式中可以含有变量，也就是说数组的大小不必一开始就固定，而可以根据实际情况来改变大小。在C99以前，C语言想要实现这种大小不固定的数组（称为变长数组）必须借助于指针（第7章）。事实上，实现变长数组的最佳方式确实是指针，所以允许中括号内出现变量的意义并不大。目前主流的IDE对变长数组的支持普遍不太好，Visual Studio系列的软件至今仍然不支持。

声明了变量之后，我们随时可以在它的作用域内对它进行各种操作。而一个数组是由多个元素组成的，我们该怎么访问某个元素呢？我们说数组是变量的组合，因此数组中的每个元素都有一个编号，这个编号叫做下标（subscript），访问数组元素的方式为：数组名[下标]。

1. #include <stdio.h>
2. int main()
3. {
4. int num[5];
5. for (int i = 0; i < 5; i++)
6. {
7. num[i] = i + 1;
8. printf("%d",num[i]);
9. }
10. getchar();
11. return 0;
12. }

程序6-1-1

程序6-1-1中，第4行声明了一个int型数组num，大小为5，这意味着它能容纳最多5个int型数据。在接下来的循环语句中，我们用i作为下标一一访问了num数组中的各个元素。可以看出，循环一共执行了5次，i的取值是0~4。一个大小为n的数组，它的有效下标是0~n-1，C语言中的数组下标从0开始，而不是从1开始。

前面第2章有一个表格，从表格上我们可以直观地感受到每种类型的变量所占内存空间的大小。而数组所占空间等于数组中每个元素所占空间之和。我们通过一个程序来验证这一点。

1. #include <stdio.h>
2. int main()
3. {
4. int num[5];
5. printf("int型数据的大小是：%d，数组num的大小是：%d",sizeof(int),sizeof(num));
6. getchar();
7. return 0;
8. }

程序6-1-2

程序运行结果是：

int型数据的大小是：4，数组num的大小是：20

sizeof是一个运算符（虽然它看起来更像一个函数），它的作用是求出括号内的数据类型（或者变量与常量）所占内存空间的大小，单位是字节。sizeof(int)求出的就是int型数据的大小（我们知道是4），sizeof(num)则能求出num数组的大小。从运行结果我们不难看出，数组num的大小是5\*4，也就是5个int型数据的大小。

在声明变量的时候，我们可以初始化它，同样地，声明数组的时候，我们也可以初始化它。初始化一个数组的格式如下：

数据类型 数组名[表达式x]={表达式1, 表达式2,....,表达式n};

n可以小于等于表达式x的值，但是不能大于表达式x。

1. #include <stdio.h>
2. int main()
3. {
4. int num1[5] = {1,2,3,4,5};
5. int num2[5] = {1,2,3};
6. for (int i = 0; i < 5; i++)
7. {
8. printf("%d %d\n",num1[i],num2[i]);
9. }
10. getchar();
11. return 0;
12. }

1

程序6-1-3

在程序6-1-3中，我们声明了两个数组num1和num2，然后初始化了num1中的所有元素以及num2中的部分元素。对于num2来说，只有前3个元素被初始化，至于剩下的元素会被初始化成什么值取决于编译器，在Visual Studio 2015中，num2[3]和num2[4]会被初始化成0。如果我们试图把第4行改成int num1[5] = {1,2,3,4,5,6};，程序将会不能通过编译。

此外，还有一个很有意思的东西，如果我们在声明一个数组的时候初始化了它，那么我们可以省略中括号里的表达式，程序会自动推断数组的大小。比如下面这个程序：

1. #include <stdio.h>
2. int main()
3. {
4. int num[] = {0,1,2,3,4};
5. int size = sizeof(num)/sizeof(int);
6. printf("数组大小为：%d",size);
7. printf("数组元素有：");
8. for (int i = 0; i < size; i++)
9. {
10. printf("%d ",num[i]);
11. }
12. getchar();
13. return 0;
14. }

程序6-1-4

程序6-1-4中，我们声明并初始化了一个数组num，接着用size来保存数组num的容量（num所占空间的大小除以int型数据的大小即为num所含元素的个数）。程序反馈的结果是num的大小为5，这正好与我们初始化的元素的个数相等。

数组在内存中到底是怎样储存的呢？我们知道&运算符可以用来求一个变量的地址，而%p格式控制符则可以将一个数据按照内存地址的格式来输出。我们通过一个程序来分析数组在内存中的储存情况。

1. #include <stdio.h>
2. int main()
3. {
4. int num[5];
5. printf("数组的首地址是：%p\n",&num);
6. for (int i = 0; i < 5; i++)
7. {
8. printf("num[%d]的地址是：%p\n",i,&num[i]);
9. }
10. getchar();
11. return 0;
12. }

程序6-1-5

程序的某次运行结果如下：

数组的首地址是：0089FB3C

num[0]的地址是：0089FB3C

num[1]的地址是：0089FB40

num[2]的地址是：0089FB44

num[3]的地址是：0089FB48

num[4]的地址是：0089FB4C

如果读者尝试运行了这个程序，可能会发现运行结果有所出入，这是正常的，由于每次运行程序时，num被分配到的内存空间都是不同的，所以运行结果不一样就是很正常的事情了。

不过，虽然每次运行结果都不一样，但是有些东西却是固定不可能改变的。首先，我们可以看出，数组的首地址和num[0]的地址是相等的，&num[0]等价与&num（甚至还等价于num，C语言规定数组名可以作为该数组的首地址，&num和num都表示数组num的首地址）。同时，我们可以看出，num[0]的地址和num[1]的地址之间相差是4，至于为什么是4，这就和16进制的运算有关了，0~9可以用来表示10进制的数，而16进制的数则用0~9以及a b c d e f来表示，16进制的a等价于10进制的10，b等价于11，以此类推，f等价于15。从c到0，正好是4个数，因此0089FB40-0089FB3C等于4。而num[1]和num[2]、num[2]和num[3]的地址之间的差值也都为4。我们知道int型数据的大小为4，这说明了数组中的各个元素在内存中是连续储存的。

我们说过，有效的下标是0~n-1（n是数组的大小），那么对于一个大小为n的数组，可不可以访问下标为n或者下标为-1的元素呢？答案是可以的。

1. #include <stdio.h>
2. int main()
3. {
4. int num[5] = {1,2,3,4,5};
5. printf("num[0]的地址是：%p，num[0]的值为：%d\n", &num[0],num[0]);
6. printf("num[-1]的地址是：%p，num[-1]的值为：%d\n", &num[-1],num[-1]);
7. printf("num[4]的地址是：%p，num[4]的值为：%d\n", &num[4], num[4]);
8. printf("num[5]的地址是：%p，num[5]的值为：%d\n", &num[5], num[5]);
9. getchar();
10. return 0;
11. }

程序6-1-6

程序的某次运行结果如下：

num[0]的地址是：0023FEA8，num[0]的值为：1

num[-1]的地址是：0023FEA4，num[-1]的值为：-858993460

num[4]的地址是：0023FEB8，num[4]的值为：5

num[5]的地址是：0023FEBC，num[5]的值为：-858993460

可以看到，num[-1]和num[0]的地址之间相差为4，也就是说，当我们访问num[-1]的时候，程序认为num[-1]是num[0]的前一个元素。而num[4]和num[5]的地址也正好相差为4，程序同样认为num[5]是num[4]的后一个元素。从这个例子中我们可以看出，当程序访问数组中的某个元素时，首先求出数组的首地址，然后根据下标来寻找对应元素的地址。就程序6-1-6来讲，当下标为-1的时候，程序就在首地址的基础上往前移动一个int型数据的大小，当下标为5的时候，就后移5个int型数据的大小。

C语言不会检测下标是否合法。在程序6-1-6中，-1和5都不是num数组的有效下标，像这样访问非有效下标元素的做法被称为越界访问。越界访问会带来一些潜在的威胁，比如说num[-1]所在的内存空间不属于num数组，它可能属于其它的变量或数组，我们修改num[-1]的时候可能会修改到这些变量或数组，从而使整个程序的运行受到影响。

我们说Visual Studio会对栈空间里的变量做初始化，而这个初始化的值就是num[-1]和num[5]的值。

C语言通过指针可以作出很多很灵活多变的操作，而我们现在虽然还没有接触到指针，但是就见识了C语言很接近底层、语法限制少等特点。通常情况下，数组越界都是没有正面影响的，我们需要尽量避免程序运行过程中发生越界访问。

6.2一维数组与函数

函数的形参除了可以是变量，也可以是数组。事实上，用数组作为函数参数的情况并不少，本节将会有大量的实例用来演示数组与函数的关系。

第5章的程序5-3-2中，我们用一个变量作为参数来调用了一个函数，并且在函数中修改了该参数，结果发现不能影响到主函数中的变量。我们还证明了，主函数中的变量和被调用的函数只是数值上相等，它们的地址不同，也就是说它们不是同一个变量。如果我们用数组作为参数呢，主调函数与被调函数中的数组是否是同一个呢？

1. #include <stdio.h>
2. void fun(int []);
3. int main()
4. {
5. int num[5] = { 0,1,2,3,4 };
6. for (int i = 0; i < 5; i++)
7. {
8. printf("%d ",num[i]);
9. }
10. fun(num);
11. printf("\n");
12. for (int i = 0; i < 5; i++)
13. {
14. printf("%d ", num[i]);
15. }
16. getchar();
17. return 0;
18. }
19. void fun(int num[])
20. {
21. for (int i = 0; i < 5; i++)
22. {
23. num[i]++;
24. }
25. }

程序6-2-1

程序运行结果如下：

0 1 2 3 4

1 2 3 4 5

程序6-2-1中，我们在第2行声明了一个以数组为参数的函数，我们说过在声明函数的时候，只需给出形参的类型，而不必给出形参的名字。int []表示一个整形数组。

接下来，我们在主函数中初始化了一个数组，然后依次输出了每个元素的值。然后把num数组作为参数传递给了fun函数，注意到第19行，这里的中括号里面没有任何数字，因为当形参是数组的时候，没有必要给出数组的大小。fun函数中，把形参数组中的元素依次+1，然后神奇的事情发生了，主函数中的num数组受到了影响。

那么为什么会出现这种情况呢？6.1节我们已经介绍过，数组名代表着该数组的首地址，只要知道了首地址就能访问数组中的元素。一个数组可能含有数十个元素，如果把这些元素一一地传递过去，不免有些费时费力。在这种情况下，传递数组首地址就成了更好的选择，如果参数传递的时候，只传递了数组的地址，那么主函数与fun函数中的num数组就应该是同一个了，因为它们的地址相同，这一点可以通过程序来验证。

1. #include <stdio.h>
2. void fun(int []);
3. int main()
4. {
5. int num[5];
6. printf("主函数中num[0]的地址%p\n",&num[0]);
7. fun(num);
8. getchar();
9. return 0;
10. }
11. void fun(int num[])
12. {
13. printf("fun函数中num[0]的地址%p\n", &num[0]);
14. }

程序6-2-2

程序的某次运行结果如下：

主函数中num[0]的地址00E7FA68

fun函数中num[0]的地址00E7FA68

可以看出，主函数中的num数组和fun函数中的num数组共用一片内存空间，所以fun函数中对num数组的操作也就会影响到主函数中的num数组了。

有时候，我们可能会试图在一个数组中查找某个值，如果数组中包含与我们要查找的值相等的元素，则给出该元素的下标，否则提示没有找到。我们很容易想到查找的思路是将数组中的元素逐一地与待查找的值比较，直到遇见一个与待查找的值相等的元素或者找完了整个数组都没有找到为止。这种思路是可行的，它也是最简单的查找方法，这种方法被称为线性查找（**linear search**）。

1. #include <stdio.h>
2. void search(int[], int,int);
3. int main()
4. {
5. int num[5] = { 0,1,2,3,4 };
6. search(num,5,3);
7. search(num,5,5);
8. getchar();
9. return 0;

1. for (int i = 0; i < sizeOfArray; i++)
2. {
3. if (num[i] == target)
4. {
5. printf("数组中存在值为%d的元素，下标为：%d\n",target,i);
6. return;
7. }
8. }
9. printf("数组中不存在值为%d的元素\n", target);
10. }

程序6-2-3

程序6-2-3中，第2行声明了一个函数，这个函数有3个参数，第一个参数当然就是待查找的数组。而第2个参数是数组的长度，我们说过，当数组作为参数传递的时候，不传递数组的大小，这样被调函数中就不知道数组到底含有多少个元素，所以我们需要这样一个参数。至于最后一个参数，当然就是要查找的目标了。

容易看出，第13行的循环最多会执行sizeOfArray次，也就是数组的有多大，循环就会最多执行多少次。在循环中，依次地将数组元素与查找的目标相比较，一旦相等，立马输出提示信息并结束函数调用。而如果循环终止后，还没有查找到，那就说明数组中不含查找的目标。

主函数中，调用了两次search函数，分别查找3和5，两次调用的结果当然是一次找到，一次没找到。

**试一试**

修改程序6-2-2，使数组的每个值不是由初始化给出，而是由用户输入。

线性查找的好处在于代码简单，思路清晰，但是它有个明显的缺点，那就是在查找的时候，程序会尝试将数组中的所有元素与查找目标相比较（虽然最终不一定所有的元素都被比较过了，但是最坏的情况下是有可能的）。如果数据比较多，线性查找的效率就显得很低了，由此，人们发明了另外一种更快速的方法，这种方法叫做二分查找（**binary chop**），也叫折半查找。二分查找能够大幅度地减少需要比较的次数，但是它的缺点是待查找的数组必须保持有序（元素递增或者递减）。

我们通过一个程序来讲解二分查找

1. #include <stdio.h>
2. void search(int[], int,int);
3. int main()
4. {
5. int num[5] = { 0,1,2,3,4 };
6. search(num,5,3);
7. search(num,5,5);
8. getchar();
9. return 0;
10. }
11. void search(int num[], int sizeOfArray,int target)
12. {
13. int low = 0;
14. int high = sizeOfArray - 1;
15. int middle;
16. while (low<=high)
17. {
18. middle = (low + high) / 2;
19. if (num[middle] == target)
20. {
21. printf("数组中存在值为%d的元素，下标为：%d\n", target, middle);
22. return;
23. }
24. else if (num[middle] > target)
25. {
26. high = middle -1;
27. }
28. else
29. {
30. low = middle + 1;
31. }
32. }
33. printf("数组中不存在值为%d的元素\n", target);
34. }

程序6-2-4

程序6-2-4中，我们已知待查找的数组是升序排列的，这一点非常重要，二分查找是建立在数组有序的基础上的，并且代码会和排序方式有一定的关联。

第13行和第14行声明了两个变量low和high，low和high的作用是标记target存在的区间。一开始我们认为target存在的区间是整个数组，于是用low和high分别表示有效下标的下限0和上限sizeOfArray-1。

接下来是一个循环语句，由于数组是有序的，所以我们用target来和它可能存在的那个区间最中间的那个元素做比较（num[middle]或者num[(low+high)/2]）。如果target正好等于num[middle]，那说明已经查找到了，直接输出提示信息并返回即可。如果target和num[middle]不相等，那么就需要判断一下了。

第一种情况，num[middle]比target还大，由于数组是升序排列的，我们可以马上得出一个结论，那就是target可能存在的区域是low~(middle-1)，而middle~high这个区间的值肯定都比target更大，我们就不用再考虑它了。经过这样的比较，我们把target所在的区间范围缩小到low~(middle-1)，所以需要重新设定high的值为middle-1。

至于第二种情况，则是num[middle]还小，同样地，我们可以立马确定target所在的区域是(middle+1)~high，因为low~middle这个区间肯定都比target更小。于是我们重新设定low的值为middle+1。

最后再来看循环继续的条件，如果一直没能查找到目标，那么若干次循环之后，必然会出现low+1=high或者low等于high，如果是low+1=high，那么middle的值等于(low+high)/2等于low（这一点是前面的章节中多次提到的），这时候如果num[middle]和target依然不相等，那么又出现两种情况，要么high被赋值为middle-1也就是low-1，要么是low被赋值为middle+1也就是high，前一种情况，则high<low，这时候有效区间就不存在了，我们当然没有必要继续比较。至于low等于high的时候，middle也就等于low等于high，这时候要是target依然不等于num[middle]，不管是low会加1还是high会减1，都会造成high<low，同样，有效区间不再存在。

可以看出，当我们已知数组有序的情况下，使用二分查找的效率远远优于使用线性查找，因为有相当一部分的数据是不必比较的。但是二分查找的缺点是需要数组有序，对一个无序的数组使用二分查找是没有意义的。

**试一试**

修改程序6-2-4，将数组由升序改成降序，并且适当地修改search函数，让函数依然能得到正确的查找结果。

一个数组并不总是保持有序的，我们有没有办法来重新排列数组中的元素，让它们变得有序呢？答案是可以的。

排序的方法多种多样，并且各有所长，接下来我们将介绍3种不同的排序方式。

**小提示**

用C语言解决某个问题的时候，需要有准确而清晰的思路，这种思路为我们提供了解决方案。所谓的解决方案，通常被称为算法（**algorithm**）。解决排序问题就要用到排序算法，排序算法几乎是各个语言都通用的，即使最终写出来的代码不一样。

首先我们要介绍的第一种排序算法被称为“冒泡排序（**bubble sort**）”。所谓冒泡排序，就是根据排序方式（升序或降序），在不断的循环中让最大或最小的元素渐渐地往数组尾部（或头部）移动，经过多次迭代之后，数组就会变得有序了。

再多的文字都不足以说清楚这个道理，我们借助代码来分析，一切就会变得更直观。

1. #include <stdio.h>
2. void sort(int[], int);
3. void showArray(int[], int);
4. int main()
5. {
6. int num[10] = {7,9,5,2,1,3,0,8,6,4};
7. printf("排序前的数组：\n");
8. showArray(num, 10);
9. sort(num, 10);
10. printf("排序后的数组：\n");
11. showArray(num, 10);
12. getchar();
13. return 0;
14. }
15. void sort(int num[], int sizeOfArray)
16. {
17. int i, j, temp;
18. for (i = 0; i < sizeOfArray-1; i++)
19. {
20. for (j = 0; j < sizeOfArray-1-i; j++)
21. {

1. if (num[j] > num[j + 1])
2. {
3. temp = num[j];
4. num[j] = num[j + 1];
5. num[j + 1] = temp;
6. }
7. }//第二层for循环
8. }//第一层for循环
9. }
10. void showArray(int num[], int sizeOfArray)
11. {
12. for (int i = 0; i < sizeOfArray; i++)
13. {
14. printf("%d ",num[i]);
15. }
16. printf("\n");
17. }

程序6-2-5

程序6-2-5的第2、3行声明了两个函数，sort函数用来给数组排序，showArray用来输出数组中的所有元素。假设我们按照升序的方式来排序。

主函数中，我们初始化了一个大小为10的数组，可以看出这个时候数组是无序的。然后我们输出了排序前的数组。接着调用了排序的函数，在排序函数中，我们声明了3个变量，前两个是循环控制变量，后一个则是临时变量，关于临时变量的作用，我们稍后会详细说明。

我们先来看第二层循环，首先，循环开始前j被赋值为0，循环继续的条件是j<sizeOfArray-1-i，读者可能对这个条件有些疑惑，不过没关系，我们暂时不必理会它。每次循环的内容是，把数组中的某个元素和它的下一个元素相比较，如果该元素小于它的下一个元素，那么就交换它和它的下一个元素。在交换的时候，临时变量就派上用场了，首先我们把num[j]的赋值给temp，再把num[j+1]的值赋给num[j]，最后再把temp的值赋给num[j+1]。这就好比要交换A、B个杯子里的水，我们必须要借助第3个杯子C，首先把A中的水倒进C，再把B中的水倒进A，接着把C中的水倒进B，交换变量的思想和交换杯子中的水很类似。

我们再来看第二层循环的循环继续条件，首先，假设i等于0，这说明外层循环才刚刚开始，数组还没有被动过，我们就应该设法把数组中最大的那个元素移动到num[9]的为止，由于在交换的时候，每个元素都要和它的下一个元素比较，而num[9]没有下一个元素，所以j必须小于9，也就是小于10-1-0。而当i等于1的时候，说明外层循环已经进行过一次了，num[9]已经是最大的那个元素，这时候我们要考虑的就是在数组的前9个元素中找到最大的那个，然后把它放进num[8]，同样的道理num[8]不再应该和它的下一个元素比较，因此j必须小于8，也就是小于10-1-1。以此类推，我们不难得出循环继续的条件是j<sizeOfArray-1-i。

然后我们再来分析第一层循环，循环继续的条件是i<sizeOfArray-1，也就是说循环执行的次数是sizeOfArray-1次，之所以会这样，是因为在sizeOfArray次循环之后，num[1]到num[sizeofArray-1]都已经保持有序，并且此时num[0]是最小的，那么相当于num[0]到num[sizeofArray-1]的元素都已经保持有序，我们也就没有必要继续循环。

现在我们能理解为什么这个排序算法被称为冒泡法了。就第一层循环的第一次循环来讲，我们的目的是把最大的元素移动到num[9]，最大的元素是num[1]，也就是9。第二层循环每执行一次，9就和它的下一个元素交换一次，知道它最终来到num[9]的位置，9就像一个气泡一样，在循环的过程中不断地往上浮动。

冒泡法是最简单、最容易理解的算法之一，虽然它在某些情况下的性能表现不好，但是作为初学者，我们还是应该熟练掌握它。

除了冒泡法，还有另外一种比较简单的排序算法，理解它甚至比理解冒泡法还简单，这种排序方法叫做选择排序（**selection sort**）。

选择排序的思路是，在一次循环中，找到最大（或最小）的元素的下标，然后把这个位置的元素移动到数组的头部或尾部。和冒泡法类似的，经过多次迭代之后，数组就能变得有序。我们来看代码实例。

1. #include <stdio.h>
2. void sort(int[], int);
3. void showArray(int[], int);
4. int main()
5. {
6. int num[10] = {7,9,5,2,1,3,0,8,6,4};
7. printf("排序前的数组：\n");
8. showArray(num, 10);
9. sort(num, 10);
10. printf("排序后的数组：\n");
11. showArray(num, 10);
12. getchar();
13. return 0;
14. }
15. void sort(int num[], int sizeOfArray)
16. {
17. int i, j, min, temp;
18. for (i = 0; i<sizeOfArray; i++)
19. {
20. min = i;
21. for (j = i+1; j < sizeOfArray; j++)
22. {
23. if (num[j] < num[min])
24. {
25. min = j;
26. }
27. }
28. if (min != i)
29. {
30. temp = num[i];
31. num[i] = num[min];
32. num[min] = temp;
33. }
34. }
35. }
36. void showArray(int num[], int sizeOfArray)
37. {
38. for (int i = 0; i < sizeOfArray; i++)
39. {
40. printf("%d ",num[i]);
41. }
42. printf("\n");
43. }

程序6-2-6

程序6-2-6和程序6-2-5唯一的区别在于sort函数的内容。程序的第17行我们声明了4个变量，i和j是循环控制变量，temp是交换变量时用到的临时变量，而min的作用则是用来标记最小的元素的下标，我们每一次都把最小的元素移动到数组的前端，这样就能保证在若干次循环之后，数组能升序排列。

我们声明了4个int型变量，其中i和j分别是两层for循环中的循环控制变量，temp是临时变量，在交换数组的2个元素时需要用到，而min则用来记录数组还没有排序好的部分的最小值的下标。第一层循环每进行一次，就会有一个最小的元素移动到数组前面部分，这样没排序好的部分会越来越小。

首先看到第一层循环，在这一层循环的开头，我们假定下标为i的元素是最小的，然后进入第二层循环。

在第二层循环中，我们不断地将num[i]之后的元素与num[min]相比较，一旦出现比num[min]更小的元素，马上更改min的值，这样就能保证在第二层循环结束之后，min是num[i]~num[sizeOfArray-1]之间最小的元素的下标。我们以i=0的时候为例，起初min的值是0，然后用num[1]来和num[0]比较，因为num[1]大于num[0]，所以min的值不会变，接下来用num[2]来和num[0]比较，由于num[2]小于num[0]，所以min的值变成了2，如此反复，最终min的值变成了6。当i等于1的时候，min被设置为1，然后用num[1]和num[2]比较，num[2]小于num[1]，于是min的值改变……。当i等于2或i等于3的时候，也是类似的分析方式。

接下来程序将判断min是否等于初始值，如果是，那就没有比较进行交换了，如果不是，则交换。

这样，第一层循环每执行一次，就有一个最小的元素被移动到num数组的前面。比如当i等于0时，循环一次后num[0]就是最小的，i等于1时，循环一次后num[1]就是除了num[0]之外最小的，以此类推，第一层循环结束以后，数组就升序排列了。

选择排序比冒泡排序更容易理解，但是这两种排序方式的性能表现是一样的，选择排序并不比冒泡排序更优秀。选择排序也是初学者应该熟练掌握的一种排序方式。

**小提示**

不管是冒泡排序还是选择排序，想要直接通过代码来看懂排序的原理都是很难的。在学习某个算法的时候，通常都需要看相关的解释文字，我们在介绍这两种算法的时候，都用了很详细的描述来讲解。事实上，如果只有原理的解释，初学者要看懂还是很难的，所以我们就用程序本身作为实例来进行分析，虽然程序中的情况是一种特例，但是用它作为例子来帮我们理解算法的原理却起到了很大的作用。在学习C语言的过程中，这种找特例的方式是很有用的，它可以让我们避免直接接触抽象的原理，降低学习的难度。

不知道读者们是否还记得上一章我们说过，递归调用的函数并不一定效率很低，我们还说将会在这一章介绍一个这样的例子。

冒泡排序和选择排序的性能都很一般，有一种速度相对快很多的排序方式，叫做快速排序（**quick sort**），而快速排序通常是使用递归的方式来实现。

**小提示**

在衡量一个排序算法好还是不好的时候，通常取决于两个方面，性能表现和排序的稳定性。所谓性能表现，是指当需要排序的数据的量增大的时候，排序所消耗的时间增长的幅度，冒泡排序和选择排序用到双重循环，所以在数据量增加的时候，消耗的时间将会以平方的级别来增加，而快速排序的时间增长幅度则远远没有这么高，有一门学科叫《数据结构与算法》，这门学科就专门研究这种时间增长幅度（在该学科中叫做时间复杂度）。数据结构算法是一门相对高深的学问，有兴趣的读者可以在学习完C语言基础知识后学习算法的相关知识。

我们再来看排序的稳定性，假设数组中的各个元素是这样的：7、9、9、2、3……，出现了两个9，如果一个算法能保证排序之后，第一个9一定在第二个9前面，那么就称这种算法是稳定的，反之，则称它是不稳定的。

快速排序的基本思想是，在第一轮排序的时候，将数组分为两部分，让其中一部分中的所有元素都比另外一部分中的元素更小，然后再对这两部分进行同样的操作，这样数组就会越来越有序，当进行到某个程度的时候，数组会被分成若干个只含1个或2个元素的部分，如果让这些最小的部分保持有序，那么整个数组都会变得有序。快速排序的这种思想被称为分治法（**divide and conquer**）。快速排序的基本思想介绍完了，我们来看具体实例。

1. #include <stdio.h>
2. void sort(int[], int,int);
3. void showArray(int[], int);
4. int main()
5. {
6. int num[10] = {7,9,5,2,1,3,0,8,6,4};
7. printf("排序前的数组：\n");
8. showArray(num, 10);
9. sort(num,0,9);
10. printf("排序后的数组：\n");
11. showArray(num, 10);
12. getchar();
13. return 0;
14. }
15. void sort(int num[],int low,int high )
16. {
17. if (low >=high)
18. {
19. return;
20. }
21. int first = low;
22. int last = high;
23. int key = num[first];
24. while (first < last)
25. {
26. while (first < last&&num[last] >= key)
27. {
28. last--;
29. }
30. num[first] = num[last];
31. while (first < last&&num[first] <= key)
32. {
33. first++;
34. }
35. num[last] = num[first];
36. }
37. num[first] = key;
38. sort(num, low, first - 1);
39. sort(num, last + 1, high);
40. }
41. void showArray(int num[], int sizeOfArray)
42. {
43. for (int i = 0; i < sizeOfArray; i++)
44. {
45. printf("%d ",num[i]);
46. }
47. printf("\n");
48. }

程序6-2-7

不管是冒泡排序还是选择排序，实现它们的时候都可以不依赖函数，而快速排序则不同，实现快速排序必须借助于函数（如果用递归的方式来实现快速排序的话）。

可以看出，调用快速排序的函数的时候需要提供3个参数，第一个依然是待排序数组，后面两个则用来限定数组中待排序的区域，low表示这个区域的最小下标，high则表示最大下标。我们说在整个排序过程中，函数会不断地递归调用，而在递归调用的过程中，都是在对同一个数组进行处理，只不过处理的区域各不相同，因此，如果我们像冒泡排序或选择排序那样只设定一个参数来表示数组的长度，自然无法满足对不同区域的处理的需求，因此，我们必须使用两个参数来分别表示目标区域的两个边界。

第17行对low与high的相对关系进行了判断，如果low等于high，那么我们要排序的区域实际上就只剩下一个元素了，实际上已经有序了，我们当然没有必要对这样的区域进行操作。而如果low大于high，那么它们表示的区间也就不存在了（这一点类似于二分查找中的low和high）。主函数中调用sort函数的时候，我们认为需要排序的区域是整个数组，于是把数组的有效下标的最小值0和最大值9作为参数传递给了sort函数。

接下来我们声明并初始化了3个变量，first和last分别被初始化为low和high，它们的作用将在排序中起到关键的作用，而key则作为一个比较的基准，通常我们把待排序区域中的第一个元素设定为这个基准。

由于我们复制了num[first]中的内容，所以num[first]可以暂时作为一个临时区域用来随意储存数组中的其它元素。

第24行开始了一个while循环，循环继续的条件是first小于last，为什么会是这个条件呢？看完接下来的分析，读者就明白了。

在第一层while循环中，我们又看见了一个循环，循环继续条件是first小于last并且num[last]的值大于等于我们之前选定的基准值，last的值不断减小，直到num[last]小于基准的时候循环结束。这样我们就找到了第一个小于基准值的元素，然后我们把这个值放到num[first]里面去，我们说过num[first]的值已经被复制过了，所以num[first]可以储存其它的元素。而到这个时候，num[last]的值也已经被复制过了，因此我们可以用num[last]来储存其它的数组。如果始终没有找到一个比基准值更小的元素，那么最终last会等于first，这时候再继续循环，就会访问到超出当前区域的元素，显然就不对了，因此这一层循环的继续条件是first小于last。而如果last等于first，那么num[first]=num[last]相当于把自身的值赋值给自身，自然也不会造成负面影响。

第一层while循环中还有另外一个循环，这个循环继续的条件是first小于last并且num[first]小于等于基准值。同样地，当循环结束的时候，我们找到了第一个大于基准值的元素，这时候我们就应该把它放到num[last]中（num[last]的值已经被复制），这时候num[first]的值又被复制了，num[first]又可以用于储存其它的值。如果始终没能找到一个比基准值大的元素会怎样呢？如果出现了这种情况，first会等于last，然后num[last]=num[first]又会变成赋值给自身的操作，同样不会造成负面影响。

当first大于等于last的时候，第一层while循环中的两个循环都不会再被执行，这种情况下第一层循环也就没有必要继续了，因此这一层循环的条件是first<last。

当第一层循环结束后，num[first]的值是肯定已经被复制了的，此时数组中还有一个空位可以用来容纳元素，当然我们就要把之前取出来的基准值放回去，所以才有了第37行语句。此时，num[first]前面的元素全都比它小，它后面的元素则全都比它大，如果我们的目标是让数组升序排列，那么此时我们可以认定，num[first]已经处于正确的位置了。

由于第24行的循环的继续条件是first<last，并且循环中first和last增长或降低的幅度都只有1，因此我们可以认为循环结束后first等于last。num[first]的位置正确了还远远不够，我们还需要让它前面的元素和后面的元素都分别有序，因此我们再次调用sort函数，用来排序剩余的元素。那么应该怎样递归调用呢？我们可以设想，就当前这一层的调用来讲，它的目的是使low~high这个区间变得有序，而其它的区域我们不必理会（其它的调用会帮我们解决），而low~high这个区间里，又只有first（或者说last）这一个位置是正确的（或者说只有这一个位置能确保是正确的），所以我们应该设法让low~(first-1)以及(first+1)~high这两个区间变得有序，所以需要再调用两次sort函数，就像第38行和第39行一样。

快速排序的设计非常精妙，它把一个相对大的问题拆分成了若干个更小的问题，当所有的更小的问题都被解决之后，最大的那个问题也就被解决了。我们只需要掌控局部，不需要总揽全局，这也就是分治法的优势所在。

快速排序采用递归的方式来实现，而其效率反而很高，这说明了递归并不等价于低效率。前面第5章里面展示的递归的例子中，几乎都是低效率的，但这只不过这是因为它们做了大量的重复的工作。

比起冒泡排序和选择排序，快速排序在性能表现上有明显的优势，在实际项目的开发中被广泛使用，但它的缺点是相对难以理解得多，并且快速排序是一种不稳定的排序方式。

除了我们介绍的这3种排序方式，常见的排序方式还有直接插入排序和归并排序等。有兴趣的读者可以查阅数据结构与算法的相关文献来学习这些排序方式。

说完了排序，我们再来说一些比较有趣的数组的应用。

生活中，我们使用的数字通常是10进制的，而事实上，对于一个程序员来说，除了会大量地使用10进制以外，还会接触到很多其它的进制，比如我们前面见到的16进制。我们可以自己写一个程序来实现进制的转换。由于16进制的数需要用一些字母来表示，而这无疑会使程序变得更复杂，因此，为了简化程序，我们只实现将一个10进制的数转换成一个2到9之间进制的数。

进制转换的依据是什么呢？我们只分析10进制转换成更低进制的情况，假设我们需要将一个10进制的数转换成一个n进制的数（n大于等于2，小于等于9）。首先，对于一个n进制的数来讲，它的进制规则是逢n进1，也就是说，在n位数的某一位上绝对不会出现n（就像10进制的数只用0~9来表示一样）。既然n位数的某一位上最多只能出现n-1，那么我们很容易想到除法运算，假设我们要把10进制的x转换成n进制，那么x/n的余数肯定就不满n，这个余数就能作为转换之后的“个位”。假设x/10的商是y，余数是z，那么现在x暂时可以表示成yz，z是个位上的数。但是这样有个问题，那就是y可能大于等于n，这时候就需要再次用y除以n，然后取余数作为“十位”的数。如此反复，直到商为0。

看完上面的介绍，可能读者还是很迷惑，没关系，看完下面这个程序实例之后，一切就会豁然开朗。

1. #include <stdio.h>
2. int conversion(char [],int, int);
3. int getNum();
4. int main()
5. {
6. int num, base;
7. char result[33];
8. printf("请输入一个10进制整数：\n");
9. num = getNum();
10. printf("请输入要转换成几进制（2~9）：\n");
11. while (1)
12. {
13. base = getNum();
14. if ((base >= 2) && (base <= 9))
15. {
16. break;
17. }
18. printf("请输入2~9之间的数！\n");
19. }
20. int lenth=conversion(result,num, base);
21. printf("%d转换成%d进制的数之后是：",num,base);
22. if (lenth == 0)
23. {
24. putchar('0');
25. }
26. else
27. {
28. for (int i = 0; i < lenth; i++)
29. {
30. putchar(result[i]);
31. }
32. }
33. getchar();
34. return 0;
35. }
36. int conversion(char result[],int num, int base)
37. {
38. int tempArray[32], tempSub=0, resultSub = 0;
39. if (num == 0)
40. {
41. return 0;
42. }
43. else if (num < 0)
44. {
45. result[resultSub] = '-';
46. resultSub++;
47. num = -num;
48. }
49. while (1)
50. {
51. tempArray[tempSub] = num%base;
52. num = num/base;
53. if (num == 0)
54. {
55. break;
56. }
57. tempSub++;
58. }
59. for (int i = tempSub; i >= 0; i--)
60. {
61. result[resultSub] = tempArray[i]+'0';
62. resultSub++;
63. }
64. return resultSub;
65. }
66. int getNum()
67. {
68. int num,ok;
69. while (1)
70. {
71. rewind(stdin);
72. ok=scanf("%d",&num);
73. if (ok == 1)
74. {
75. break;
76. }
77. printf("输入有误，请重新输入！\n");
78. }
79. rewind(stdin);
80. return num;
81. }

程序6-2-8

很长的一个程序，原理并不复杂，但是却包含了很多技巧，接下来我们将详细分析。首先，我们声明了两个函数，conversion和getNum。conversion的作用是转换进制，而getNum的作用则是从用户那里接受一个int型数据。

在主函数中，我们声明了2个int型变量，num和base，num用来保存需要被转换进制的数，而base则保存需要转换的进制。然后我们声明了一个char型的数组用来保存结果，数组长度为33，至于为什么是33，将会在讲解conversion函数的时候解释。第8行提示用户输入待转换的10进制数，而第9行则通过getNum函数来接收用户的输入。

在getNum函数中，又声明了2个int型变量，num用来接受用户的输入，并作为函数的返回值，而ok则用来判断是否输入成功了。在接收输入前，我们首先清空输入缓冲区，避免之前的数据造成干扰。接下来的72行，我们用ok来接受scanf函数的返回值，如果ok等于1，则说明输入的是一个合法的数据（关于scanf的返回值请参考第5.7节的相关内容），否则说明没有输入成功。如果输入成功，则退出循环，并在清空缓冲区之后退出程序；如果输入没有成功，则提示用户重新输入。getNum就是我们自定义的安全输入函数。

程序的第11行开始的while循环的作用是接收转换后的进制。程序6-2-7的目的是将一个10进制数转化为2~9进制的数，如果用户输入的数据不在这个区间内，程序将会无法运行，所以我们要对用户的输入加以判断，循环接收输入，直到输入正确为止。

conversion函数一共有3个参数，第1个参数是一个char类型的数组，用来保存转换后的结果，第2个参数是待转换的数，而第3个参数则表示转换后的进制。

那么，为什么要用数组来表示转换后的数呢？一个int型的数据能表示的最大的数是232-1（我们不妨认为是232）。在10进制中，仅用一个数位最多能表示10个数（0~9），而如果用两个数位则可以表示102个数（0~99）。我们知道，4转换成2进制的数之后是100，这时候我们当然可以用int型数据来储存它，但是用int型数据储存的数实际上并不是2进制的数，而是10进制的。4转换成2进制的数之后，需要占用3个数位，在2进制中，3个数位能表示8个数，而对于int型的数据来讲，3个数位意味着能表示1000个数，所以用int型数据来储存2进制的数是很浪费的。而如果我们要把**232**转换成2进制的数，那么转换后的数将会占用32个数位，32个数位对于int型数据来说是1032个数，显然这远远大于int型数据的上限232，因此用int型来储存转换后的数就不合适了。既然用一个变量不足以表示转换后的结果，那我们就只好考虑用数组了。转换后的数最多是9进制的数，因此我们可以用char类型的数组来表示（事实上用int型也是可以的，只是int型数据会比char型的数据占用更多的内存）。考虑到最极端的情况，我们至少需要把数组的长度设定为33位，再考虑到符号，我们需要一个长度为33的数组，这也就是为什么主函数中，result数组的长度是33。而conversion函数中的tempArray的长度为32，是因为它没有必要储存符号。

conversion的返回值是转换后的数据的位数（符号位也算一位）。因为result的33个数位不一定都会用来储存转化后的数据，所以我们需要把有效的数据的数量作为返回值，这样主函数中才能输出正确的数据。

conversion函数中的tempSub和resultSub分别用来表示tempArray数组和result数组的下标，这里被初始化为0。

接下来对num的值进行判断，如果它为0，那么就不用转化了，因为任意进制中的0都是0。我们把0作为返回值，主函数中会对返回值加以判断，如果发现返回值是0，就说明num的值为0，这种情况下直接输出结果即可，不需要再访问result数组中的数据。

如果num小于0，那么我们就把result数组的第一个元素设定为’-’，由此来表示num是一个负数，然后我们增加了resultSub的值，这样下一次往result数组中写入数据的时候，接收数据的就是下一个元素。最后我们把num的相反数赋值给它，num和-num转换后的结果只是相差一个符号，而result中已经记录了num是负数，所以让num暂时变成一个正数会有利于我们的操作。

接下来的第50行开始了一个while循环，在这个循环中，我们用之前说到过的进制转换的方法来对num进行转换，并把结果储存到tempArray中。而num%base是一个int型的数据，要把它转化成对应的字符需要加上’0’，这也就是我们多次提到的，0和’0’不等价，前者为值为0，后者的值为49（ASCII码）。假设我们要把10进制的2转换成2进制，那么转换后的结果是10，但是tempArray中保存的数据实际上是’0’和’1’（参照第50行的循环语句来分析，第一次循环，2%2的值是0，所以tempArray[0]就被赋值成了’0’，第53行的语句会让2变成1。接下来的循环中，2%1的值是1，所以tempArray[1]就是’1’）。也就是说，tempArray中储存的数据和最终的结果实际上是相反的，所以我们在第60行开始的for循环中，从tempArray的最后一个有效数据开始，依次将前面的数据转移到result数组中，这样result数组中保存的就是正序的结果了。

如果num的值不为0，那么函数的返回值将会是result数组中有效数据的个数，主函数则会以此为依据来输出result数组中的数据。最终程序会输出转化后的结果。

**小提示**

对于程序6-2-7中的tempArray来讲，我们往里面存入数据的时候，顺序是0、1、2……这样的从小到大的顺序。而当我们从里面取出数据的时候，顺序却反过来了。对于tempArray数组来讲，数据存取的顺序是后进先出，最后被存入的数据会最先被取出。而栈空间的存取顺序也正好是这样，程序6-2-7相当于是在用数组来模拟栈空间。

6.3字符数组

字符数组是一维数组的特殊情况，声明方式和普通的一维数组是一样的，字符数组可以作为普通的一维数组来处理。

1. #include <stdio.h>
2. int main()
3. {
4. char ch[5] = {'H','e','l','l','o'};
5. for (int i = 0; i < 5; i++)
6. {
7. printf("%c",ch[i]);
8. }
9. getchar();
10. return 0;
11. }

程序6-3-1

程序6-3-1中，我们把ch当成一个普通的数组来对待，声明、初始化、访问（逐个输出ch数组中储存的字符）都看起来和其它数组没什么区别。

字符数组可以被当成普通数组来处理，同时它和普通数组又有一定的区别。

字符数组通常用来储存字符串（**string**），而一个字符串可能包含有很多个字符，如果我们在初始化这样的字符串的时候，用程序6-3-1的方式，效率就会非常低了。因此，C语言为字符数组提供了另外的初始化方式。

1. #include <stdio.h>
2. int main()
3. {
4. char ch1[10] = { "Hello" };
5. char ch2[10] = "World";
6. for (int i = 0; i < 5; i++)
7. {
8. printf("%c", ch1[i]);
9. }
10. for (int i = 0; i < 5; i++)
11. {
12. printf("%c", ch2[i]);
13. }
14. getchar();
15. return 0;
16. }

程序6-3-2

程序6-3-2展示了字符数组的两种初始化方式，直接在双引号里面添加目标字符串，然后作为数组的初始化值，在初始化的时候，花括号可以加也可以不加。要注意的是，字符串中字符的数量不能超过字符数组的长度，否则会被报错。

形如char ch1[10]={“Hello”}的写法是正确的，但是ch1={“Hello”}却是错误的，只有在初始化的时候才能直接把字符串“赋值”给数组。

char ch1[10]={“Hello”}; //正确，初始化的时候允许这样写。

ch1={“Hello”}; //错误，不能这样给数组赋值。

ch1=”Hello”; //错误，这样给数组赋值也是不允许的。

读者可能会有疑问，为什么程序6-3-1中的数组长度是5，而程序6-3-2中却是10呢？这和字符串的储存原理有关，C语言中，我们可以直接输出一个字符串，而不用像程序6-3-2那样用循环语句来输出。而事实上，程序在输出一个字符串的时候，也是用循环的方式来实现的（只是对于我们来说更方便了），那么程序怎么知道什么时候该停止输出呢？原来，程序之所以知道什么时候该停止输出了，是因为一个特殊的字符——0（不是’0’），这个字符的ASCII码就是0，我们可以称它为结束符。查阅ASCI码表，我们可以看出，0是一个控制字符，因此它是不能被显示的。如果我们用程序6-3-2展示的字符数组的初始化方式来初始化一个数组，那么0会被自动添加在字符串的结尾。

1. #include <stdio.h>
2. int main()
3. {
4. char ch[10] = {"Hello"};
5. printf("%d",ch[5]);
6. getchar();
7. return 0;
8. }

程序6-3-3

可以看到，输出的结果是0。这说明ch数组的第6个元素是结束符。

现在我们知道，一个字符数组在储存字符串的时候，需要留至少一个空间来储存结束符。现在我们应该了解如何输入和输出字符串了。

1. #include <stdio.h>
2. int main()
3. {
4. char str[100];
5. printf("请输入一个字符串：\n");
6. scanf("%s",str);
7. printf("你刚刚输入的字符串是：%s",str);
8. rewind(stdin);
9. getchar();
10. return 0;
11. }

程序6-3-4

程序6-3-4中用到了一个新的格式控制字符%s，这个格式控制字符的作用是输入或输出字符串。第6行的scanf语句中，我们直接用str作为参数，而没有取它的地址，这是因为str本身就表示数组的地址，所以我们可以直接把它作为参数来实现输入字符串。

从程序的运行结果来看，如果用户输入的字符串的长度没有超过100，那么第7行的printf语句就能顺利执行。而如果用户输入的字符串长度超过100，那么就有可能会造成越界访问，从而导致程序崩溃。

我们说，C语言在输出一个字符串的时候，遇到结束符0才会停止，那么如果没有结束符呢，会怎么样？

1. #include <stdio.h>
2. int main()
3. {
4. char str[5] = {'H','e','l','l','o'};
5. printf("%s",str);
6. rewind(stdin);
7. getchar();

1. return 0;
2. }

程序6-3-5

程序的某次运行结果如下：

Hello烫烫烫?Gy

很明显，程序在输出的时候，没有遇到结束符，于是一直不停地输出（实际上发生了越界访问），直到遇到一个结束符为止，而我们演示的运行结果中，y后面的字符正好是0，于是在输出y之后停止了输出，而由于中间的字符是随机的，所以看起来就是乱码了。

大多数时候，用scanf函数来输入字符串都是可行的，但是在某些时候，用scanf却达不到我们想要的结果。

我们再次看到程序6-3-4，如果我们在运行的时候，输入的字符串是这样的：first second，那么输出的结果就只有first而没有second。我们可以看到，first和second之间有一个空格，我们知道，空格对于scanf函数是分隔符，所以scanf在读取到空格之后，就会把空格前面的字符串储存到str数组中，而不再把空格后面的字符串也储存进去。

那么，如果我们希望输入含有空格的字符串，应该怎么办呢？C语言专门提供了一个这样的函数：gets\_s。看到gets\_s，很多读者会想起上一章提到过的scanf\_s，我们说过，scanf\_s比scanf更为安全，同样地，gets\_s也是一个比gets函数更安全的函数，它起初由微软公司提供，后来成为了标准库函数。我们不会介绍gets函数的用法，而是直接使用gets\_s。读者可能会有疑问，既然scanf和gets都是不安全的函数，那为什么我们还要使用scanf函数呢？事实上，虽然scanf是一个不安全的函数，但是它的使用却比scanf\_s更为简便，特别是在输入字符（串）的时候。而gets函数则不同，它并不比gets\_s方便很多。更重要的是，scanf依然算是一个标准库函数，它可以和scanf\_s同时存在，而gets函数却已经不再是一个C语言标准库函数了，C11标准明确规定，C语言标准库用gets\_s来取代gets。

我们来看看gets\_s的用法。

1. #include <stdio.h>
2. int main()
3. {
4. char str[100];
5. printf("请输入一个字符串\n");
6. gets\_s(str, 100);
7. printf("你输入的字符串是：%s",str);
8. getchar();
9. return 0;
10. }

程序6-3-6

程序6-3-6的某次运行结果如下：

请输入一个字符串

**123 456**

你输入的字符串是：123 456

我们可以从运行结果看出，gets\_s可以接收字符串中的空格，而不会把它当成一个分隔符。gets\_s函数有两个参数，第一个参数表示接收字符串的数组，而第二个参数则是字符串的最大长度，这里我们把这个长度设定为100，也就是数组的大小。如果我们输入的字符串超过了这个长度，那么程序会自动报错，以此来防止造成更多的破坏。此外，我们可以看到，在第8行的getchar();前面没有rewind(stdin);，而在前面的程序中，只要程序使用到了scanf等输入函数，我们都会加上rewind(stdin);来清空输入缓冲区。在这个程序中我们却没有用到，原因是gets\_s是一个安全的输入函数，它会在输入结束后自动清空缓冲区。

在处理字符的输入输出的时候，我们经常用到getchar和putchar。同样地，既然有gets(gets\_s)，就应该有puts。puts相当于是printf的精简版，它能够很方便地实现字符串的输出，同时不能像printf函数那样实现格式控制。

1. #include <stdio.h>
2. int main()
3. {
4. char str[100];
5. printf("请输入一个字符串\n");
6. gets\_s(str, 100);
7. puts("你输入的字符串是：");
8. puts(str);
9. getchar();
10. return 0;
11. }

程序6-3-7

程序6-3-7演示了puts函数的用法，可以看出，puts函数只有一个参数，这个参数就是要输出的字符串，我们可以像第7行那样直接表示一个字符串，也可以像第8行那样用一个数组名来表示。

说完了gets\_s函数，我们再来说一说scanf\_s函数，感受一下scanf\_s、gets\_s、scanf函数之间的异同。

在输入整数的时候，scanf\_s和scanf是没有区别的。但是在输入字符或者输入字符串的时候，不仅需要给出用来储存字符（串）的变量（数组）的地址，还要给出相应的储存空间的长度。

1. #include <stdio.h>
2. int main()
3. {
4. char str[100],ch;
5. printf("请输入一个字符串和一个字符（用空格符隔开）：\n");
6. scanf\_s("%s %c",str,100,&ch,1);
7. printf("你输入的字符串是：%s，输入的字符是：%c",str,ch);
8. rewind(stdin);
9. getchar();
10. return 0;
11. }

程序6-3-8

程序的某次运行结果如下：

请输入一个字符串和一个字符（用空格符隔开）：

**abc d**

你输入的字符串是：abc，输入的字符是：d

程序6-3-8中，第6行使用了scanf\_s来输入一个字符串和一个字符，并且我们分别为它们指定了最大长度。注意到%s和%c之间的空格，我们在提示用户输入的时候告诉用户用空格符来分隔字符串和字符。当我们输入abc d的时候，abc被储存到了str数组中，空格符和d则留在了缓冲区里等待被ch接收，这个时候ch会优先接收空格符。让ch接收空格符显然不是我们想要的，那么怎么办呢？我们可以在格式控制字符上面改动一下，达到赋值抑制的效果，我们在%c前面加上一个空格，那么如果ch在接收数据的时候，遇到的第一个字符是空格，那么ch就会把这个空格跳过，转而接收下一个字符。

可以看出，scanf\_s虽然提高了安全性（限定了字符或字符串的长度，一定程度上防止越界访问），但是依然没有解决缓冲区数据残留的问题。此外，就程序6-3-8而言，如果我们输入的字符串长度超过100，程序会直接崩溃，而我们理想中最好的情况显然应该是只接收前100个字符，而忽略剩下的，并且保证程序继续正常运行。下一节我们将自己动手打造一个理想的字符（串）输入函数。

6.4字符数组与函数

上一节我们说到，gets\_s函数和scanf\_s函数都没有完美地解决字符（串）输入中的问题。gets\_s函数的缺点是如果输入的数据长度超过给定的长度，程序会崩溃，而scanf\_s的缺点更多，使用复杂，并且在接收数据之后不会清空缓冲区。接下来我们将自己打造一个函数用来接收字符串，这个函数将解决这些不完美的地方。

1. #include <stdio.h>
2. void getStr(char[], int);
3. int main()
4. {
5. char str[5];
6. printf("请输入一个字符串（少于5个字符）：\n");
7. getStr(str, 5);
8. printf("你输入的字符串是：");
9. puts(str);
10. char ch;
11. printf("请输入一个字符：\n");
12. getStr(&ch, 1);
13. printf("你输入的字符是：");
14. putchar(ch);
15. getchar();
16. return 0;
17. }
18. void getStr(char buffer[], int lenth)
19. {
20. rewind(stdin);
21. char temp=0;//这里必须初始化
22. if (lenth == 1)
23. {
24. buffer[0] = getchar();

1. }
2. else
3. {
4. int i;
5. for (i = 0; i < lenth - 1; i++)
6. {
7. temp = getchar();
8. if (temp == '\n')
9. {
10. break;
11. }
12. buffer[i] = temp;
13. }
14. buffer[i] = 0;
15. }
16. while (temp != '\n')
17. {
18. temp = getchar();
19. }
20. }

程序6-4-1

程序6-4-1中的getStr函数类似于程序6-2-7中的getNum函数。但是和getNum不同的是，getStr没有返回值，并且在清空缓冲区的时候，有一点不同。

getStr有两个参数，第一个参数是用来接收字符（串）的变量或数组，第二个参数是这个变量或数组的长度，如果长度为1，则表示是变量，否则是一个数组（我们可以认为变量是长度为1的数组）。

首先我们清空输入缓冲区，避免之前的输入造成干扰。

当lenth为1时，我们把buffer视为一个变量的地址，那么buffer[0]就表示这个变量本身。同时，由于buffer此时是一个变量的地址而不是数组的，所以我们只能按字符的方式来接受用户的输入，这时候，我们直接把getchar函数的返回值赋值给buffer[0]即可。

当lenth不为1的时候，buffer就是一个数组，这时候我们要接受的就是一个字符串，我们要考虑到两种情况。第一种，用户输入的字符串的长度小于lenth-1，这时候我们就需要把用户输入的所有字符都储存到数组里，并且在最后添加结束符0。第二种，用户输入的长度大于等于lenth-1，这时候，我们只接受前lenth-1个字符，并且保留数组的最后一个元素，用最后一个元素来储存结束符0。

在以前的程序中，当我们使用for循环的时候，通常把循环变量的声明放在for后面的括号里，比如：for(int i=0;i<10;i++)，这样做能把i的作用域限定在for循环内，大多数情况下，i在for语句之外是没有用处的，所以这样做就能让程序在i失去作用后自动回收它的空间，避免它一直占用空间但是却不发挥任何作用。而这一次，我们却把循环变量的声明放在了for语句之外，之所以这样做，是因为我们要在循环结束之后往数组里添加结束符，而i正好能用来标记应该把哪个元素设置为结束符。

我们知道，用户结束输入的方式是按回车键，只有在按下回车之后，getchar才会开始接收数据，循环结束的条件有两个，一个是遇到了回车符，一个是接收的长度达到限定的值。如果是前者，则说明用户输入的字符串的长度小于lenth-1，后者则说明用户输入的字符串长度大于等于lenth-1。而事实上，不管是哪一种情况，我们都可以在循环结束（或跳出）之后，把buffer[i]设为结束符。

在完成数据的处理之后，我们还需要再次清空缓冲区，直接用rewind(stdin)当然是合理的，但是这里我们给出了另外一种解决方案。

程序第40行开始了一个while循环，在while循环中，我们持续用temp来接受缓冲区中的残留数据，直到遇见回车符为止（回车符的出现标志着缓冲区已经被清空）。我们第21行的注释里说，temp一定要被初始化，因为这里我们是先判断temp的值，再用它来接受数据的。假如我们没有初始化temp，那么当lenth的长度为1的时候，就会出现temp没有被初始化就被访问的情况。

为什么在函数开始的时候，我们不用这种方式来清空缓冲区呢？因为如果用rewind(stdin)来清空缓冲区的话，不管缓冲区是否已经被清空，结果都是一样的。而如果用这种while循环来清空缓冲区，则缓冲区中必须有残留数据，否则第42行的getchar就会因为缓冲区中没有可接受的数据而等待用户输入。由于之前调用了至少一次getchar函数，所以我们可以认定，缓冲区中肯定有数据残留（至少残留着一个回车符）。

现在我们总算有了一个好用的字符（串）输入函数了，接下来我们将介绍一些C语言标准库所提供的字符串处理的相关函数，同时，由于这些库函数的原理较为简单，我们还将展示如何实现它们。

首先我们介绍strlen，这个函数用来求出一个字符串的长度，它由string.h头文件提供，这个函数的返回值是一个int型的数据，这个返回值就是字符串的长度。

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5. char str[10] = "Hello";
6. int lenth = strlen(str);
7. printf("%s的长度是%d",str,lenth);
8. getchar();
9. return 0;
10. }

程序6-4-2

要使用strlen函数，必须包含string.h头文件。strlen的参数就是字符数组的首地址，返回值是这个数组储存的字符串的长度（不包含结束符）。str中一共有5个字符，因此strlen的返回值也就是5。

这个函数的实现是非常简单的，我们可以从str[0]开始，逐个分析str数组中的元素，直到遇见结束符为止，而已经分析过的元素的数量就是字符串的长度。

1. #include <stdio.h>
2. int strLen(char[]);
3. int main()
4. {
5. char str[10] = "Hello";
6. int lenth = strLen(str);
7. printf("%s的长度是%d", str, lenth);
8. getchar();
9. return 0;
10. }
11. int strLen(char str[])
12. {
13. int i;
14. for (i = 0; str[i] != 0; i++);
15. return i;
16. }

程序6-4-3

程序6-4-3演示了自定义一个strlen函数的做法，为了和标准库中的strlen区分开来，我们将这个自定义的函数命名为strLen。strLen以一个字符数组作为参数，第13行的for循环的写法很有意思，没有花括号，并且在括号后面就有分号。这是因为这个for循环没有循环体，每次循环只需要判断str[i]是否为结束符，如果是则停止循环，循环结束后，i的值就是字符串的长度。

如果我们需要复制一个普通的变量的值，那么我们只需要把它赋值给另一个同类型的变量就行了。而如果我们想要复制一个字符串，就没这么简单了，直接把一个字符数组赋值给另外一个字符数组是不可行的。这时候我们就需要借助库函数strcpy，和strlen一样，strcpy也是由string.h提供的。

我们来看一看它的具体用法。

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5. char str1[10] = "Hello";
6. char str2[10];
7. strcpy(str2, str1);
8. printf("%s",str2);
9. getchar();
10. return 0;
11. }

程序6-4-4

strcpy是这样声明的：strcpy(char destination[],char source[])。后一个参数是待复制的字符数组，前一个则是接受复制结果的数组。strcpy在复制字符串的时候，会复制结束符。我们在使用strcpy的时候，需要保证destination数组足够容纳复制后的结果，否则就会因为发生越界访问而导致程序存在隐患。

接下来我们将自己实现strcpy函数，同样的，为了与库函数区分开来，我们将自定义的函数命名为strCpy。通过简单的分析，我们不难看出，实现strCpy的思路是用一个循环语句，在循环语句中不断地将source数组中的元素复制到destination数组中，直到遇见结束符0为止。

具体代码如下：

1. #include <stdio.h>
2. void strCpy(char[], char[]);
3. int main()
4. {
5. char str1[10] = "Hello";
6. char str2[10];
7. strCpy(str2, str1);
8. printf("%s",str2);
9. getchar();
10. return 0;
11. }
12. void strCpy(char destination[], char source[])
13. {
14. for (int i = 0;; i++)
15. {
16. if (source[i] == 0)
17. {
18. destination[i] = 0;
19. break;
20. }
21. destination[i] = source[i];
22. }
23. }

程序6-4-5

strCpy函数中，有效的代码只有一个for循环，在for循环中，我们首先检测source[i]是否已经为结束符（考虑到source中的字符串长度为0的情况），若为结束符，则在destination数组的最后补上一个结束符，然后退出循环，如果不是结束符，则正常复制destination数组中的元素，这个函数的实现依然是比较简单的。

需要注意的是，不同于strLen和strlen，strCpy和strcpy的差距更大，它们的的返回值是不同的，并且strCpy没有strcpy安全，当我们学习到更高级的知识——指针之后，我们将按照标准库的标准来重新实现strCpy函数。

除了字符串的复制，我们还可能用上字符串的连接，比如把”Hello ”和”World”连接成一个更长的字符串”Hello World”。连接字符串用到的函数是strcat，依然声明在string.h中。

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5. char str1[] = "tail";
6. char str2[20] = {"head "};
7. strcat(str2, str1);
8. printf("%s",str2);
9. getchar();
10. return 0;
11. }

程序6-4-6

strcat函数的声明是这样的：strcpy(char destination[],char source[])，储存最终结果的是destination数组。我们应该保证destination足够容纳连接之后的字符串。

实现strcat的思路同样不复杂，首先我们找到destination数组中储存结束符的元素，然后把source数组中的数据复制过来，具体的实现如下：

1. #include <stdio.h>
2. void strCat(char[], char[]);
3. int main()
4. {
5. char str1[20] = "head ";
6. char str2[10] = "tail";
7. strCat(str1, str2);
8. printf("%s", str1);
9. getchar();
10. return 0;
11. }
12. void strCat(char destination[], char source[])
13. {
14. int i;
15. for (i = 0;destination[i]!=0; i++);
16. for (int j = 0;; j++)
17. {
18. if (source[j] == 0)
19. {
20. destination[i] = 0;
21. break;
22. }
23. destination[i] = source[j];
24. i++;
25. }
26. }

程序6-4-7

在程序6-4-7中，第15行的作用是求出destination数组中的字符串的长度（参照前面的strLen函数），当第15行的for循环结束时，i就是destination中的结束符的下标。我们的目标是把两个字符串连接起来，因此必须把结束符覆盖掉。接下来的第16行的for循环和strCpy很类似，也就是将source中的内容逐个地复制到destination中，并在复制完成后加上结束符。

string.h中，处理字符串的函数还有很多，不能做到一一介绍，我们只介绍strlen、strcpy以及strcat这3个最常用的，其它的函数读者可以在需要用到的时候再查阅相关资料学习。

前面第4章，我们编写过一个打开网站的程序（程序4-2-4），当时我们说它存在一个巨大的缺陷，那就是不能打开用户指定的网页，我们还说会在第6章解决这个问题。而现在，我们已经有了足够的知识储备，可以解决这个问题了。

接下来我们将分析如何实现。首先，我们应该在原来的基础上增加一个菜单选项，除了打开指定的几个网站以外，增加一个“其它网站”选项，当选择了其它网站选项之后，提示用户输入需要打开的网站，输入完成后打开相应的网站，显示菜单并接受输入、打开网站这两个功能分给不同的函数来实现。网站的地址可以理解为一个字符串，现在我们已经知道如何输入一个字符串了，只不过这次，我们输入的就不是char型的字符串了。

事实上，字符型的数据除了有char以外，还有另外一种类型：wchar\_t。我们知道char占用一个字节，它的表示范围很小，而wchar\_t则占用两个字节，表示范围就远远大于char了。我们前面说到过，打开网页的函数ShellExecute所使用的字符与我们平时使用的不同，我们通常用char来表示字符串，而ShellExecute使用的则是wchar\_t。输入wchar\_t型的字符串的时候，用到的格式控制字符是%ws而不是%s。

1. #include <stdio.h>
2. #include <conio.h>
3. #include <windows.h>
4. char showMenu();
5. int main()
6. {
7. char choice;
8. wchar\_t webSite[256];
9. while (1)
10. {
11. choice = showMenu();
12. switch (choice)
13. {
14. case '1':
15. {
16. ShellExecute(NULL, L"open",
17. L"www.baidu.com", NULL, NULL, SW\_SHOWNORMAL);
18. break;
19. }
20. case '2':
21. {
22. ShellExecute(NULL, L"open",
23. L"www.weibo.com", NULL, NULL, SW\_SHOWNORMAL);
24. break;
25. }
26. case '3':
27. {
28. ShellExecute(NULL, L"open",
29. L"www.youku.com", NULL, NULL, SW\_SHOWNORMAL);
30. break;
31. }
32. case '4':
33. {
34. printf("请输入要打开的网站：");
35. scanf\_s("%ws", webSite, 256);
36. ShellExecute(NULL, L"open",
37. webSite, NULL, NULL, SW\_SHOWNORMAL);
38. break;
39. }
40. case '5':
41. {
42. printf("感谢使用\n");
43. \_getch();
44. exit(0);
45. break;
46. }
47. }
48. printf("网站打开成功，按任意键继续\n");
49. \_getch();
50. system("cls");
51. }
52. return 0;
53. }
54. char showMenu()
55. {
56. char choice;
57. printf("==========================\n");
58. printf("请选择打开哪个网站\n");
59. printf("1.百度(www.baidu.com)\n");
60. printf("2.微博(www.weibo.com)\n");
61. printf("3.优酷(www.youku.com)\n");
62. printf("4.其它网站\n");
63. printf("5.退出程序\n");
64. printf("==========================\n");
65. do
66. {
67. choice = \_getch();
68. } while ((choice<'1') || (choice>'5'));
69. return choice;
70. }

程序6-4-8

程序6-4-8虽然看起来很长，但是总体来讲没有太大难度。showMenu函数的作用是显示菜单并接受输入，并在接收到合理的输入之后返回用户的选择。主函数中则根据用户的输入来进行不同的操作。和程序4-2-4不同的是，程序6-4-8中没有加入tchar.h头文件，我们说过，普通字符串转换成宽字符串需要用到tchar.h提供的\_T()，而事实上，我们也可以不用\_T()，而是像程序6-4-8里那样，用L来实现，只需要在字符串前面加上一个L，即可实现这个转换。

如果用户选择了打开自己指定的网站，那么程序运行第35行的scanf\_s语句，在这里，我们使用的格式控制字符是%ws，为了安全起见，我们使用scanf\_s而不是scanf。在接下来的ShellExecute中，我们直接把webSite数组当成第3个参数，而不用转换（L或\_T()都不能对变量进行转换）。

通过调用Windows的API，我们除了可以打开一个网页，还可以播放音乐，播放音乐用到的API是mciSendString。一般来说，要播放一段音乐，需要两个步骤，打开和播放，两个步骤需要用到不同的命令，这些命令由mciSendingString提供，假设现在C盘下有一个mp3文件a.mp3，我们现在通过C语言程序来播放这段音乐。

1. #include <stdio.h>
2. #include<windows.h>
3. #pragma comment(lib,"winmm.lib")
4. int main()
5. {
6. mciSendString(L"open c:/a.mp3 alias music", NULL, 0, NULL);
7. mciSendString(L"play music", NULL, 0, NULL);
8. printf("正在播放音乐，按回车键后退出");
9. getchar();
10. return 0;
11. }

程序6-4-9

程序6-4-9中，第3行的语句我们以前从未见过，事实上，这也是预编译指令的一种。前面第5章我们曾介绍过，C语言程序从高级语言变成可执行程序的过程中，除了需要编译，还需要连接。除了把程序的各个模块组合起来，还需要连接标准库中的文件。通常情况下，提供函数的头文件（如stdio.h）会自动为我们连接到对应的库文件，但是也有的头文件不会。播放音乐的功能需要用到winmm.lib这个库文件，头文件没有为我们自动连接，所以我们需要加上#pragma comment(lib,”winmm.lib”)来主动连接到这个库文件，否则程序将不能连接成功。

接着我们连续调用了两次mciSendString，可以看到，两次调用的时候，唯一的区别是第一个参数。后面3个参数分别为NULL、0、NULL，我们知道，通常情况下0和NULL是等价的，但是为什么这个地方要一会儿写成NULL，一会儿写成0呢，事实上，这只是一种习惯，两个NULL对应的是两个指针，而0则对应缓冲区的大小，对此仅作了解即可。我们不难看出，第一个参数是一个wchar\_t的字符串，里面的内容就是mciSendString的指令。第一次调用的时候，执行的操作是打开c:/a.mp3，并给这个文件起别名为music（起别名的意义在于方便之后的操作）。第二次调用执行的操作则是播放别名为music的文件（也就是c:/a.mp3），需要注意的是，音乐播放不会阻塞程序继续运行也就是说，我们可以一边播放音乐，一边执行其它的操作，比如输出信息或者接受用户的按键。

如果我们要实现播放用户指定路径的音乐应该怎样操作呢？这时候就需要用到字符串的连接了，wchar\_t类型的字符串依然是以0作为结束符的，我们可以仿照前面的strCat重写一个wchar\_t版的字符串连接函数，连接”open”、音乐路径、”alias music”这3个字符串，然后把结果当成mciSendString的第一个参数。不过在这里我们要介绍另外一种方法，即使用sprintf函数。

sprintf函数和printf函数的声明很类似，只不过要多一个参数：

sprintf(接收输出的缓冲区,“格式控制字符”,输出列表);

我们知道，printf函数的作用是按照一定格式向控制台屏幕输出信息。而sprintf则是把这些数据按照规定的格式输出到接受数据的缓冲区中，这个缓冲区通常是一个字符数组。

我们先来看看sprintf的具体用法：

1. #include <stdio.h>
2. int main()
3. {
4. int num = 10;
5. char ch ='a';
6. char buffer[10];
7. sprintf(buffer, "%d%c",num,ch);
8. printf("%d%c\n",num,ch);
9. printf("%s",buffer);
10. getchar();
11. return 0;
12. }

程序6-4-10

程序运行结果如下：

10a

10a

第7行中，将num和ch分别按照整数和字符的格式写入buffer，要注意的是，虽然num是按照整数写入buffer的，实际上还是写入的对应的字符串，也就是”10”。事实上，printf函数也是如此，我们最终在控制台屏幕上看到的其实是一个字符串而不是整数。

在我们往buffer中写入了相应的数据之后，再按字符串来输出它，得到的结果就和直接用printf输出一样了。

现在我们知道具体的思路了，首先接受用户的输入，然后把指令字符串连同用户输入的信息一起写入缓冲区，然后用缓冲区作为参数。只不过有些不同的是，如果缓冲区是wchar\_t类型的话，我们需要使用wsprintf而不是sprintf，wsprintf和sprintf函数的参数是没有区别的。

1. #include <stdio.h>
2. #include<windows.h>
3. #pragma comment(lib, "winmm.lib")
4. int main()
5. {
6. wchar\_t buffer[256];
7. wchar\_t temp[200];
8. printf("请输入要播放的音乐的路径(如c:/a.mp3)\n");
9. scanf\_s("%ws", temp,200);
10. wsprintf(buffer, L"open %s alias music",temp);
11. mciSendString(buffer, NULL, 0, NULL);
12. mciSendString(L"play music", NULL, 0, NULL);
13. printf("音乐正在播放，按回车键后退出\n");
14. rewind(stdin);
15. getchar();
16. return 0;
17. }

程序6-4-11

我们依然用scanf\_s来接受用户的输入，接着用wsprintf来将命令写入buffer中。由于wsprintf本身就是针对wchar\_t的，所以我们的格式控制字符可以直接用%s而不用%ws（用%ws也可以）。同时，open、alias等字符串也都需要转换成wchar\_t类型，所以我们需要将格式控制字符串转换为wchar\_t类型，也就是加上L。接着我们把buffer作为mciSendString的参数，这样就能顺利播放我们指定路径的音乐了。

关于字符数组，我们最后再说一说中文在数组中的储存。在C语言的学习过程中，中文一直是一个被无意或刻意回避的地方，大多数情况下，我们都试图避免和中文打交道，这也就是为什么，大多数的C语言教程中的代码在输出提示信息的时候，都尽量采用英文。然而，对于实际项目而言，中文是没有办法逃避的，客户可不喜欢纯英文的东西。我们将对C语言中的中文进行一定的分析，让读者对中文在C语言中的各种操作有一定的了解，这样，当读者真正从事软件开发的时，不至于因为中文而束手无策。

仅用52个字符（26个字母的大写和小写状态）加上少量的标点符号就能表示一切英文字符串，一个char类型的变量（或常量）足以表示一切构成英文的元素。而中文则不同，常用的汉字就有数千个，而《现代汉语词典》第5版中更是收录了6.9万个汉字。显然，用一个char型变量来表示中文是不够的。在学习计算机的基础操作的时候，我们被告知，一个英文字母用一个字节表示，而一个汉字则需要两个字节。而在C语言中，表示中文需要用到一对char型变量。我们知道，char型变量的表示范围是-128~127，而ASCII码表中，只有0~127。而负数部分，正好就可以用来表示中文，所以中文实际上是由一对负数来表示的。

我们知道，在用printf输出char型的变量的时候除了可以用%c来正常输出以外，还可以用%d来输出它的ASCII码，为了验证中文是由一对负数来表示的，我们可以初始化一个储存中文字符串的字符数组，然后依次输出它的每个值。

1. #include <stdio.h>
2. int main()
3. {
4. char str[] = "中文字符串";
5. for (int i=0; str[i] != 0; i++)
6. {
7. printf("%d ",str[i]);
8. }
9. getchar();
10. return 0;
11. }

程序6-4-12

程序的运行结果如下：

-42 -48 -50 -60 -41 -42 -73 -5 -76 -82

一共5个中文字符，最后输出的结果是10个负数，结果能够验证，一个中文字符需要占用两个字节，并且中文是由一对负数来表示的。

我们记下这些数据，然后用它们来初始化一个字符数组，最后再输出这个字符数组，看看结果是不是“中文字符串”。

1. #include <stdio.h>
2. int main()
3. {
4. char str[] = { -42 ,- 48 ,- 50 ,- 60, - 41 ,- 42,- 73,- 5,- 76,- 82,0};
5. printf("%s",str);
6. getchar();
7. return 0;
8. }

程序6-4-13

程序运行结果如下：

中文字符串

程序6-4-13的运行结果再次验证，一个中文字符和一对负数之间是相互对应的关系。

6.5二维数组

有两个下标的数组就是二维数组。二维数组的声明方式如下：

数据类型 数组名[表达式1][表达式2];

和一维数组类似地，C语言标准规定表达式1或表达式2都可以含有变量，但是Visual Studio暂时不支持表达式1或表达式2中含有变量的声明方式。

假如我们声明了这样一个二维数组：int a[3][4];。则数组a的第一维的大小为3，第二维的大小为4，数组a中一共有3\*4=12个元素。访问数组a中的元素需要两个下标。所有元素的清单如下：

a[0][0], a[0][1], a[0][2], a[0][3]

a[1][0], a[1][1], a[1][2], a[1][3]

a[2][0], a[2][1], a[2][2], a[2][3]

可以看出，二维数组很像是一个表格。第一维和第二维的下标同样都是从0开始的。对于某个元素来讲，它的第一维下标表示它所在的行，而第二维下标则表示它所在的列。

假设我们要用储存一个班30个同学的3门课程的成绩，如果用一维数组来实现的话，我们将需要3个数组，而如果我们用二维数组来实现，则只需要一个数组，显然用二维数组更有利于数据的维护。我们不妨把所有成绩做成一个成绩表格，可以看出，这个表格一共有30行3列，每一行的3个元素分别表示某位同学的3门课程的成绩。

1. #include <stdio.h>
2. #include <time.h>
3. #include <stdlib.h>
4. int main()
5. {
6. int grades[30][3];
7. srand((unsigned) time(NULL));
8. for (int i = 0; i < 30; i++)
9. {
10. for (int j = 0; j < 3; j++)
11. {
12. grades[i][j] = rand() % 101;
13. }
14. }
15. for (int i = 0; i < 30; i++)
16. {
17. printf("第%2d位同学的3门课程的成绩：",i+1);
18. for (int j = 0; j < 3; j++)
19. {
20. printf("%3d ",grades[i][j]);
21. }
22. printf("\n");
23. }
24. getchar();
25. return 0;
26. }

程序6-5-1

程序6-5-1中，我们声明了一个二维数组grades，它的大小是30\*3。第8行到第14行的作用是用随机数来为grades中的元素赋值。这里用到了一个双重循环，我们希望从数组的第0行一直处理到第29行，所以第一层循环的条件是i<30。而在每一行的处理过程中，我们又需要对这一行的3个元素分别进行处理，所以有了第2层循环，在第2层循环中，我们用rand()%101来产生0~100之间的随机数作为3门课程的成绩。

第15行到第23行的循环的作用是输出所有的信息，%2d和%3d的作用是控制输出宽度，使结果显得更为美观。

在访问一维数组中的所有元素的时候，我们通常会使用单层的循环，而通过程序6-5-1，我们可以看出，访问二维数组中的所有元素需要双重循环。

和一维数组类似地，声明二维数组的时候也可以对它进行初始化。初始化分为很多种形式，我们先来看如何按行初始化。

1. #include <stdio.h>
2. int main()
3. {
4. int num[3][2] = { {1,2},{3} };
5. for (int i = 0; i < 3; i++)
6. {
7. for (int j = 0; j < 2; j++)
8. {
9. printf("%d ",num[i][j]);
10. }
11. }
12. getchar();
13. return 0;
14. }

程序6-5-2

程序运行结果如下：

1 2 3 0 0 0

数组num一共有3行，每行2个元素。我们在第4行对num进行了初始化，我们选择的初始化方式是按行初始化，并且只初始化了前两行，同时，第2行我们只初始化了1个元素。二维数组可以看成若干个一维数组的集合，而这种按行初始化的方式相当于是对组成二维数组的每个一维数组分别进行初始化。

除了按行初始化，我们还可以直接初始化数组。

1. #include <stdio.h>
2. int main()
3. {
4. int num[3][2] = { 1,2,3 };
5. for (int i = 0; i < 3; i++)
6. {
7. for (int j = 0; j < 2; j++)
8. {
9. printf("%d ",num[i][j]);
10. }
11. }
12. getchar();
13. return 0;
14. }

程序6-5-3

程序6-5-3和程序6-5-2的运行结果是一样的。虽然我们说二维数组相当于一个表，但是这并不代表二维数组在内存中的表示形式就是二维的。相反，由于二维数组是若干个一维数组的集合，所以二维数组和一维数组的储存形式是一样的，关于这一点，我们将在讲完数组的初始化之后详细介绍。

我们知道，一维数组在初始化的时候，可以省略数组的大小。同样地，二维数组在声明的时候也可以省略一些东西，只是不同于一维数组，二维数组仅能够省略第一维的大小，而不能省略第二维。初始化二维数组的时候如果省略第一维的大小，程序会自动判断。

1. #include <stdio.h>
2. int main()
3. {
4. int num1[][2] = { {1} ,{1} ,{1} };
5. printf("%d ",sizeof(num1)/sizeof(int));
6. int num2[][2] = {1,2,3};
7. printf("%d", sizeof(num2) / sizeof(int));
8. getchar();
9. return 0;
10. }

程序6-5-4

程序运行结果如下：

6 4

从运行结果中可以看出，num1中有6个元素，这说明num1的第1维的大小是3，同样地，num2的第一维是2。

在介绍一维数组的时候，我们曾通过分析各个元素的地址的方式来证明一维数组中的各个元素在内存中是紧挨着的。现在我们可以用同样的方式来分析二维数组。

1. #include <stdio.h>
2. int main()
3. {
4. int data[2][2];
5. for (int i = 0; i < 2; i++)
6. {
7. for (int j = 0; j < 2; j++)
8. {
9. printf("%p\n",&data[i][j]);
10. }
11. }
12. getchar();
13. return 0;
14. }

程序6-5-5

程序6-5-5中，我们声明了一个2\*2的数组，然后依次输出里面每个元素的地址。某次运行结果如下：

002DFBDC

002DFBE0

002DFBE4

002DFBE8

可以看到，data[0][0]和data[0][1]之间相差4个字节，而data[0][1]和data[1][0]之间也是相差4个字节，这说明了data[0][0]到data[2][2]的4个元素都是紧邻的。

从程序6-5-5中，我们验证了，data[0][1]和data[1][0]是相邻的。我们在介绍一维数组的时候，曾说过形如num[5]（假设num是int型数组）这样的表达式表示在num[0]的地址基础上增加5个单位的地址长度之后的地址中储存的值。同样地，在二维数组中，也有这样的现象。我们说过，二维数组可以看成是若干个一维数组的集合。以程序6-5-5为例，data数组一共由2个一维数组组成，data[0]和data[1]分别表示这两个一维数组的地址。在地址的基础上增加一个下标即表示对应的元素。如num表示一维数组的地址，num[n]就可以表示一维数组的元素，而data[0]表示二维数组中第一行的地址，则data[0][n]就可以表示第一行的第n个元素。

一维数组中，可以发生越界访问，越界的时候会访问一维数组周围的不属于该数组的内存空间。同样地，二维数组中的一维数组也可以发生越界访问。

1. #include <stdio.h>
2. int main()
3. {
4. int data[2][2];
5. for (int i = 0; i < 2 \* 2; i++)
6. {
7. data[0][i] = i + 1;
8. }
9. for (int i = 0; i < 2; i++)
10. {
11. for (int j = 0; j < 2; j++)
12. {
13. printf("%d ", data[i][j]);
14. }
15. }
16. getchar();
17. return 0;
18. }

程序6-5-6

程序运行结果如下：

1 2 3 4

在程序6-5-6中，我们对data[0][0]~data[0][3]分别进行赋值，而我们知道，data[0][2]和data[0][3]不算是合理的下标，也就是说，这样做相当于是越界访问。而data[0][2]和data[0][3]就是data[1][0]和data[1][1]，对这两个元素的访问是合法的，所以程序6-5-6中的越界访问不会对数组外的信息造成影响，破坏程度较低。事实上，在某些情况下，采用这样的越界访问是完全合理的，因为这样可以把二维数组转化为一个一维数组，减少了for循环的层数，能增加代码的可读性。

6.6二维数组与函数

在前面几节中，我们介绍了如何用一维数组和字符数组作为函数参数。这一节我们将讲解二维数组作为函数参数的用法。

在作为函数参数的时候，一维数组通常会省略数组的大小，而二维数组则只能省略第一维的大小。

先来看一个简单的例子，我们编写一个函数用来输出一个二维数组中的元素。我们需要2个参数，一个是待访问的二维数组，一个是二维数组的行数（列数是固定并且已知的）。由于在用二维数组作为参数传递的的时候，被调函数只知道该数组的列数，所以我们需要另外给出一个参数来指定数组的行数，这样被调函数才能正确地访问该数组。

具体代码如下：

1. #include <stdio.h>
2. void display(int [][3], int);
3. int main()
4. {
5. int num[3][3] = { {1,2,3},{4,5,6},{7,8,9} };
6. display(num,3);
7. getchar();
8. return 0;
9. }
10. void display(int num[][3], int row)
11. {
12. for (int i = 0; i < row;i++)
13. {
14. for (int j = 0; j < 3; j++)
15. {
16. printf("%d",num[i][j]);
17. }
18. printf("\n");
19. }
20. }

程序6-6-1

程序运行结果如下：

1 2 3

4 5 6

7 8 9

程序6-6-1中，我们首先在主函数中声明并初始化了一个二维数组num，num是一个3\*3的数组。display函数的第一个参数是一个二维数组，这个二维数组的第二维的大小是固定的，和num数组一样。在主函数中，我们调用了一次display函数，给出的参数分别是num本身以及它的行数（第一维的大小）。而display函数中则用了一个双重循环来访问num数组，num数组的第一维的大小也就是第一层循环的次数。

程序6-6-1中的数组看起来很像是一个3\*3的矩阵。而事实上，二维数组是可以理解成矩阵的。矩阵有转置运算，我们可以用数组来模拟矩阵转置。假设有一个m\*n的矩阵，那么它转置之后的矩阵就是n\*m的。假设表示矩阵的两个数组分别是A和B，那么还有一个关系是A[i][j]=B[j][i]。根据这些信息，我们可以编写程序来实现矩阵的转置。

1. #include <stdio.h>
2. void tran(int [][3], int [][4]);
3. int main()
4. {
5. int A[4][3] = { {1,2,3},{4,5,6},{7,8,9},{10,11,12} };
6. int B[3][4];
7. tran(A, B);
8. for (int i = 0; i < 3; i++)
9. {
10. for (int j = 0; j < 4; j++)
11. {
12. printf("%d ",B[i][j]);
13. }
14. printf("\n");
15. }
16. getchar();
17. return 0;
18. }
19. void tran(int A[][3], int B[][4])
20. {
21. for (int i = 0; i < 4; i++)
22. {
23. for (int j = 0; j < 3; j++)
24. {
25. B[j][i] = A[i][j];//保证A[i][j]的值和B[j][i]相等。
26. }
27. }
28. }

程序6-6-2

程序6-6-2中，我们在调用tran函数将矩阵A转置之后，输出了转置后的矩阵B。程序运行结果如下：

1 4 7 10

2 5 8 11

3 6 9 12

从运行结果可以看出，矩阵B和矩阵A之间满足A[i][j]=B[j][i]的关系。

6.7多维数组

我们知道，一维数组还是二维数组在内存中的储存方式没有本质区别。二维数组可以看成若干个一维数组的集合，也正因为如此，C语言对数组的维数没有限制。当数组的维数大于或等于3的时候，这样的数组就是多维数组。

一维数组有1个下标，二维数组有2个下标，三维数组则有3个下标。我们来看一个3维数组的简单应用。

1. #include <stdio.h>
2. int main()
3. {
4. int num[2][2][2] = {1,2,3,4,5,6,7,8};
5. for (int i = 0; i < 2;i++)
6. {
7. for (int j= 0; j < 2; j++)
8. {
9. for (int k = 0; k < 2; k++)
10. {
11. printf("%d ",num[i][j][k]);
12. }
13. }
14. }
15. getchar();
16. return 0;
17. }

程序6-7-1

从程序6-7-1中可以看出，我们可以像初始化一维数组那样初始化三维数组。同时访问三维数组中的元素需要3个下标，访问三维数组中的所有元素需要3层循环。

和二维数组一样，三维数组在初始化和作为函数参数的时候可以省略第一维的大小。

事实上，三维以及更高维的数组在C语言中几乎没有使用价值。随着数组维数的增加，它所占用的空间也将很快地增加，而数组是存在于栈空间中的，使用高维数组会大量占用栈空间中有限的资源，所以我们通常用动态数组或其它的数据结构来代替直接声明的高维数组。

练习题

1.改写程序6-2-5中的sort函数，把排序方式由递增改成递减，要求使用冒泡排序。

2.改写程序6-2-6中的sort函数，把排序方式由递增改成递减，要求使用选择排序。

3.为了保证信息的安全性，人们通常会对一些关键数据进行加密，加密的算法多种多样。有一种最简单的加密方式是，把字符串中的每个字母都向后顺移一个如’a’变成’b’，’z’变成’a’，’Z’变成’A’ ，而其它的字符不变。编写程序，接受用户输入的字符串，输出加密后的字符串。