第 3 章　数组和字符串

上一章讲述了 C++的基本数据类型，但仅仅靠这些数据类型无法满足实际编程的需求。比如表示多项同类型数据，就必须有相同个数的变量来存储这些数据，如果这样的数据很多，那么单单变量的个数就会让人抓狂。这时就需要有一种变量来表示这些同类型数据，这就是数组。与基本数据类型相比，数组是一种复合数据类型，它由一系列相同类型的数据项组成。

本章主要涉及的知识点有：

- 一维数组，如何使用一维数组及相关示例。
- 多维数组，了解多维数组原理，并学会使用多维数组。
- 字符串，理解字符串和数组的关系，并学会灵活使用字符串。

3.1　一 维 数 组

话说计划经济体制下，任何产品生产、价格、数量都是规定好的。比如生产一件衣服，决策者认为可能有 1000 个人会需要这种衣服，那么生产 1000 件衣服就好了，如果第 1001 个人想买这种衣服呢？对不起，这不在计划之内。引用计划经济这个例子的目的就是为说明数组也具有类似的性质。数组必须事先估计好同类型数据的规模，然后再进行相应的“生产”。下面就来看看一维数组是如何来表示这些数据的吧。

3.1.1　一维数组定义

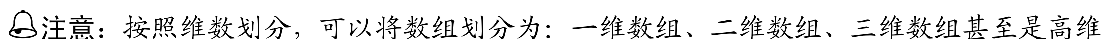
数组的本质是一组连续的内存位置，这些位置必须用来保存同类型的元素。这种同类型的元素可以是整型数据（int）、浮点类型数据（float、double）、字符类型数据（char）等。如何使用数组呢？使用之前必须对数组进行声明，其语法格式如下：

```
<数据类型>     数组名[<数组元素个数>];
```

与已经学过的基本数据类型声明相比，数组声明只不过多了一对方括号，在方括号中要指明该数组的元素个数。但这是必需的，否则编译器这个“工厂”就不知道到底要“生产”多少个元素了。

根据数组的语法格式，可以这么声明数组：

```
int intarray[10];                //声明元素个数为 10 的 int 型数组
double  darray[20];              //声明元素个数为 20 的 double 型数组
char    carray[5];               //声明元素个数为 5 的 char 型数组
```


注意：按照维数划分，可以将数组划分为：一维数组、二维数组、三维数组甚至是高维

数组，一维数组在默认情况下可以简称为数组，但其他维的数组则需全称。

按照上述声明方式，如果想保存一个班级某次 C++考试成绩，假设该班级有 50 个同学，那么就不用声明 50 个变量了，取而代之的是仅仅通过一个元素个数是 50 的数组就行了。这样看起来就省去不少麻烦。如下所示：

```
int socre[50];            //声明一个大小为 50 的 int 类型数组
```

那么这些数据在内存中是怎么进行存储的呢？具体可参见图 3.1。

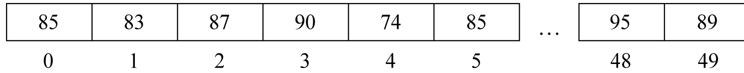


图 3.1　数组元素内存分布

从图 3.1 可以看出，数组元素在内存中是连续分布的。图中的编号是从 0～49，而不是平常认为的 1～50。C++中规定，数组的第一个元素都是零元素。应该特别注意“第一个元素”与“元素一”的区别，两者是不同的。“第一个元素”指的就是零元素（编号 0），而“元素一”指的是数组的第二元素（编号 1）。

3.1.2　一维数组的初始化

数组在声明的同时还可以进行初始化，声明并初始化的语法格式如下：

```
<数据类型>    数组名[数组元素个数]={初始值 1, 初始值 2, 初始值 3,….};
```

如以上语法格式所示，像其他类型变量初始化一样，数组变量初始化也是通过赋值语句来实现的。在进行初始化的时候应该注意以下这些问题：

- 初始化值放在一对大括号内，每个初始值之间是通过逗号进行分隔的。利用这种方式来实现元素编号和元素值一一对应。
- 该大括号内初始化值的个数不能大于声明的元素个数，也不能通过添加逗号的方式跳过。
- 初始值个数允许小于声明元素个数，那么没有对应初始值的元素，其初始值默认为 0。

比如以下这些初始化方式：

```
int  a1[4] = {1,2,3,4};            //正确
int  a2[4] = {1,2,,4};             //错误，不能用逗号跳过赋初值
int  a3[4] = {0,1,2,3,4};          //错误，元素个数小于初始值个数
int  a4[4] = {1,2};                //正确，后面两个元素会被置为 0
```

3.1.3　一维数组元素的引用

声明并初始化好数组之后，就可以引用该数组元素了，一维数组元素的引用的本质就是访问数组元素，比如上文记录一个班级 C++成绩的数组，想要访问该数组第 13 个元素，具体可以参见图 3.2。

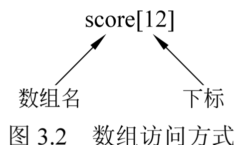


图 3.2　数组访问方式

如图 3.2 所示，访问数组由 3 个部分组成：数组名、下标及数组下标运算符。其中数组下标运算符具有最高优先级。方括号内的

编号其实应该称为下标，它表示距离数组起始处元素的个数。下标必须是一个整数或者是返回值是整数的表达式。如：

```
score[5];
score[5*3];                                          //下标是表达式
```

【例子 3.1】 具体介绍了如何使用数组，该例子实现了数组的声明并进行了初始化，最后计算数组元素的平均值。

--Example3.1.cpp--

```
01  #include<iostream>
02  using namespace std;
03  void main(void)
04  {
05      int iArray1[8] = {32,45,12,54,65,76,59,70};    //声明并初始化数组
06      int iArray2[] = {1,6,10,5,8};
07      int sum = 0;
08      for(int i=0;i<8;i++)
09      {
10          cout<<iArray1[i]<<" ";                     //打印输出数组元素
11          sum +=iArray1[i];                          //数组元素累加
12      }
13      cout<<endl;
14      cout<<sum*1.0/8<<endl;
15      sum = 0;
16      for(i=0;i<5;i++)                            //第二个数组只有 5 个元素
17      {
18          cout<<iArray2[i]<<"  ";
19          sum +=iArray2[i];
20      }
21      cout<<endl;
22      cout<<sum*1.0/5<<endl;
23  }
```

编译、运行上述程序得到如下结果：

```
32 45 12 54 65 76 59 70
51.625
1  6  10  5  8
6
```

分析程序，上述程序首先声明并初始化了两个数组。第一个数组声明方式在上一小节已经介绍了，但是第二种声明方式也是可以的，它并不指定数组元素的个数，数组元素个数由等式右边的初始值个数决定。因此第二种声明方式必须有初始值，否则就是非法的。

然后，程序通过循环逐个访问数组元素。由于数组下标是从 0 开始的，所以循环的控制变量需要从 0 开始。判断条件可以有两种写法：i<8 或者 i<=7。最后，程序输出了数组元素的平均值，变量 sum 需要乘以 1.0 的原因是为了转换为浮点数运算，提高精度。

3.1.4　一维数组示例

到此为止，读者应该对数组的声明、初始化有了一定的了解。数组其实也是一个变量，只不过它存储的值个数比别的变量多一些罢了。利用这种复合结构，可以实现更多丰富的应用。本小节利用上述介绍的知识来实现一个稍微复杂的程序，该程序功能描述如下：

（1）等待用户输入一组大于等于 0 的整数值，并且这组整数个数不大于 20 个。

（2）用户首选输入一个整数 n，该整数代表后续待输入 n 个整数，如果输入的是 0，则退出整个程序，否则依次输入 n 个整数。

（3）对输入的 n 个整数从小到大依次进行排序，排序过程为首先判断第 1 个元素和第 2 个元素大小，如果第 1 个元素大于第 2 个元素则相互交换位置。接着判断第 2 个元素和第 3 个元素大小，如果前者比后者大则交换位置。依此类推就可以找到最大值，且最大值在数组末尾，然后再从头开始进行下一轮判断，直到判断出所有元素。

（4）打印输出排序的结果，然后重复第一步。

程序难点分析：程序需要解决如何保存输入数据、编写排序算法、输出排序结果。保存输入数据最简单有效的办法就是利用数组。输出排序结果只需要利用循环打印输出数组元素就行了。编写排序算法是该程序的核心，如果对上述功能描述还是不太清楚，那么可以参考图 3.3。

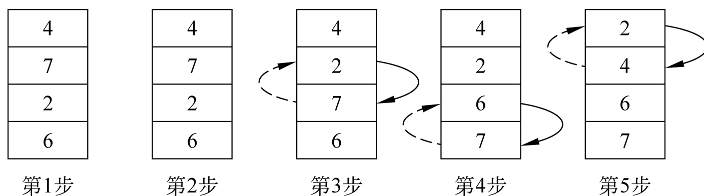


图 3.3　排序过程

从图 3.3 中可以看出，初始数组是（4,7,2,6），首先对 4 和 7 的值进行判断，4 小于 7 不用交换。继续往下判断，现在判断 7 和 2，7 是大于 2 的，那么需要进行交换，交换结果如第 3 步所示。接着判断 7 和 6，继续交换得到第 4 步的结果。数组已经遍历完了一遍，同时也确定了最大值，现在从头开始遍历到倒数第二个元素。4 大于 2，进行交换，如第 5 步所示。由于接下来的数据项都已经有序了，所以判断步骤在图中省略，但过程还是一样的。

【例子 3.2】 具体实现了上述程序的功能。程序代码如下所示：

--Example3.2.cpp--

```
01  #include<iostream>
02  using namespace std;
03  void main(void)
04  {
05      int num = 0;
06      int ia[20] = {0};
07      //输入数据个数并判断是否为 0
08      while(cout<<"请输入数据个数："&&cin>>num&&num>0)
09      {
10          for(int i=0;i<num;i++)                      //保存输入数据
11          {
12              cin>>ia[i];
13          }
14          for(i=0;i<num;i++)
15          {
16              for(int j=0;j<num-i-1;j++)              //防止数组越界
17              {
18                  if(ia[j]>ia[j+1])
```

```
19                  {
20                      int tmp = ia[j];            //数组元素交换
21                      ia[j] = ia[j+1];
22                      ia[j+1] = tmp;
23                  }
24              }
25          }
26          cout<<"排序结果如下:";
27          for(i=0;i<num;i++)                      //输出排序结果
28          {
29              cout<<ia[i]<<"  ";
30          }
31          cout<<endl;
32      }
33      cout<<"退出程序..."<<endl;
34  }
```

编译、运行上述程序得到如下输出结果：

```
请输入数据个数：4↙
4 7 2 6↙
排序结果如下:2  4  6  7
请输入数据个数：6↙
32 90 67 12 50 88↙
排序结果如下:12  32  50  67  88  90
请输入数据个数：0↙
退出程序...
```

分析程序，程序按照功能进行划分可分为 3 个部分，第 1 个部分是保存输入数据，对应的代码是 10～12 行。第 2 部分是实现排序功能，对应的代码是 14～25 行。第 3 部分是打印输出排序结果，对应的代码是 27～30 行。

应该特别注意第 2 部分的实现，该段代码利用嵌套循环来实现。第二个 for 循环的判断条件是：j<num-i-1，这样写的目的是不用再去判断已经筛选出来的前 i 个大值，再减去 1 的目的是为了防止数组访问越界，后面的判断条件是：ia[j]>ia[j+1]，如果不去 1 就会发生越界问题。

3.2　多 维 数 组

含有一个下标的数组叫做一维数组，那么如果有两个下标呢？那么这个数组就是二维数组，甚至还有三维、四维数组等，这些数组都是多维数组。但是，往往数组维数越多，其表示意义越复杂，也就越容易出错，而且在编程中一般接触的多维数组都是以二维为主。本节将以二维数组为主，详细介绍多维数组的用法。

3.2.1　多维数组定义

对于 C++语言而言，数组的维数是视具体编译程序决定的。多维数组的定义格式如下：

```
<数据类型>    数组名[元素个数 1][元素个数 2]…[元素个数 n]
```

同样，上述定义格式中的数据类型可以是基本数据类型如：int、float、double 等。数

组名需要满足标志符命名规则，n 对方括号来表示 n 维数组。按照这种定义格式，比如二维数组和三维数组可以按照如下方式声明：

```
int a1[6][5];                 //声明一个二维数组
int a2[5][6][7];              //声明一个三维数组
```

对于一维数组而言，声明时下标所表示的值就是数组的大小，那么二维数组大小是怎么计算的呢？比如上述例子中 a1[6][5]，表示的是 6+5 个元素吗？其实，计算多维数组元素个数需要将各个方括号内元素个数相乘，那么 a1 这个二维数组元素的个数就是 6*5=30 个。

既然如此，二维数组在内存中是如何分布的呢？是不是也和一维数组一样是一块连续区域？是的，无论是几维数组，它们的元素都是连续分布于内存之中。比如 A[5][7]这个二维数组内存分布，可以参见图 3.4。

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]	A[0][5]	A[0][6]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	A[1][6]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]	A[2][5]	A[2][6]
A[3][0]	A[3][1]	A[3][2]	A[3][3]	A[3][4]	A[3][5]	A[3][6]
A[4][0]	A[4][1]	A[4][2]	A[4][3]	A[4][4]	A[4][5]	A[4][6]

图 3.4　数组 a[5][7]的内存分布

注意：为了形象化说明二维数组在逻辑上元素分布情况，图 3.4 采用二维的形式来表示数组内存分布，但是实际上二维数组（高维数组）在内存分布上和一维数组一样，每个元素都是按照顺序排列的。

图 3.4 确切地说应该是一个二维数组逻辑分布图，其中第一个下标表示的是数组的行，第二个下标表示的是数组的列。那么想要访问任意一个数组元素时，只需要设置想要的行列数值就行了。比如 A[2][5]表示访问的是第 3 行、第 6 列元素。注意数组下标是从零开始的。

3.2.2　初始化多维数组

如何初始化多维数组？同样可以借鉴一维数组的初始化方式。一维数组的初始化是由一对花括号括起的若干数据项来表示的，这些数据项之间通过逗号进行分隔。同样多维数组就需要多对花括号包含数据项来表示，比如二维数组其初始化格式如下：

```
<数据类型>    数组名[元素个数 1][元素个数 2] = {{初始值表},{初始值表},…};
```

这里，初始值表指的是由若干个数据元素组成的表，并且该表中数据个数应该小于或者等于[元素个数 2]的值。而初始值表的个数应该小于或者等于[元素个数 1]的值。例如以下例子：

```
int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
                                //声明并初始化一个 int 类型二维数组
double  d[2][3] = {1.0,2.1,3.1,4.0,5.5,6,0};
                                //声明并初始化一个 double 类型二维数组
int a2[4][5] = {{1,2,3}};       //声明初始化二维数组
```

在上面进行初始化的 3 个例子中，前 2 个例子中数组每个元素都对应了初始值。在第

二个例子中只有一对花括号，这也是可以的，编译器会为数组元素存放顺序进行一一赋值。所以 d[0][0]的值是 1.0，d[0][1]的值是 2.1，d[0][2]的值是 3.1，依此类推。

第 3 个例子中，二维数组元素和初始值并没有一一对应，那么编译器会给没有对应的数组元素设置默认值 0。初始化后，a2[0][0]的值是 1，a[0][1]的值是 2，a[0][2]的值是 3。其余元素的值都是 0。

同理，三维数组的初始化格式也是类似的，只不过需要的花括号就要达到三层了，如：

```
int  a[2][3][2] = {{{1,2},{3,4},{5,6}},{{7,8},{9,10},{11,12}}};
```

3.2.3　多维数组应用举例

经过前几个小节的学习，读者应该已经基本了解了多维数组的逻辑结构及如何声明一个多维数组。

【例子 3.3】 详细介绍了使用二维数组，该例子实现了访问二维数组元素，并将二维数组的元素值打印输出，最后求该二维数组的平均值。

--Example3.3.cpp---

```
01  #include<iostream>
02  #include<iomanip>
03  using namespace std;
04  void main(void)
05  {
06      int a[4][6] = { {14 , 38 , 10 , 65 , 34 , 29},
                                                  //声明并初始化二维数组
07                    {12 , 54 , 9  , 23 , 17 , 99},
08                    {43 , -9 , -70, 40 , 33 , -20},
09                    {-11, 8  , 5  , 10 , 2  , 15}};
10
11      int sum = 0;
12      for(int i=0;i<4;i++)
13      {
14          for(int j=0;j<6;j++)
15          {
16              cout<<setiosflags(ios::left);      //设置左对齐
17              cout<<a[ I ][ j ]<<"\t";
18              sum +=a[i][j];                     //元素值累加
19          }
20          cout<<endl;
21      }
22      cout<<sum*1.0/(4*6)<<endl;                 //求平均值
23  }
```

编译、运行上述程序得到如下结果：

```
14      38      10      65      34      29
12      54      9       23      17      99
43      -9      -70     40      33      -20
-11     8       5       10      2       15
18.75
```

分析上述程序，程序首先声明了一个二维数组并进行初始化。接着，利用 for 的双重循环逐个访问数组元素（遍历），并进行打印输出。在访问数组元素的时候，可以先一行行

遍历，也可以一列列地遍历。比如上述程序实现的就是一行行遍历，如果想要实现一列列遍历，可以按照如下方式书写：

```
for(int i=0;i<6;i++)
{
    for(int j=0;j<4;j++)
    {
        cout<<a[ j ][ I ]<<"\t";                //注意下标的变化
        sum +=a[j][i];                          //元素值累加
    }
    cout<<endl;
}
```

3.3　字符数组和字符串

字符数组在 C++中是一个重要的概念，因为实际应用中常常涉及对字符的处理。比如上课时经常要点名，那么在点名之前必须知道每个学生的姓名，而每个学生的姓名并不是一个整型、浮点型数据，其实它是一个字符串，那么保存字符串就要用到字符数组了。通过字符数组来管理每个学生的名字，在点名的时候就可以方便而清楚地知道哪些学生来上课，哪些学生逃课了、看来字符数组真的很重要啊。

3.3.1　字符数组的定义和赋值

字符数组的定义格式和前面介绍的一般数组的定义格式是一样的，只不过字符数组的数据类型是字符类型（char）而已。字符数组中每一个元素存放一个字符，它也可以分为一维、二维及多维数组。例如：

```
char     c[10]                  //声明一个大小为 10 的字符数组
char     c2[10][10]             //声明一个二维字符数组，其元素个数是 10*10
char  c3[2][3][4]               //声明一个三维字符数组，其元素个数是 2*3*4
```

字符数组声明方式和前面介绍的数组声明方式类似，其实字符数组的初始化方式也是类似的。比如以下一维字符数组初始化方式：

```
char  c1[15] = {'W','e','l','c','o','m','e','t','o','C','+','+'};
                                        //声明一个大小为 15 的字符串并初始化
char  c2[]={'H','e','l','l','O'};       //声明一个字符数组并初始化
char     c3[] =”Hello”;                 //效果和 c2 一样
```

上面有三种初始化方式，它们都是可以的。在进行字符数组初始化时应该注意以下几点问题：

- 花括号内元素个数不能大于数组长度，否则编译器将会提示错误信息。
- 如果花括号内的初始值个数小于预定的数组长度，那么就将这些元素赋给数组前面的元素，其余元素自动赋为空字符（'\0'）。如上述声明的 c1 字符数组，其内存分布参见图 3.5。
- 在初始化字符数组的时候，也可以省略数组长度，编译器会自动根据花括号内初始值个数来确定数组的长度，如 c2 这个字符数组，其内存可以参见图 3.5。

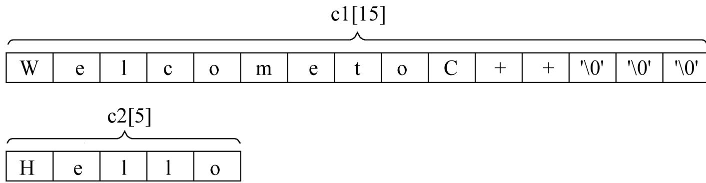


图 3.5　字符数组内存分布

字符数组除了在初始化时可以确定其数组元素外，还可以通过赋值运算来改变数组元素的值。但是和其他变量赋值操作不同，字符数组只能对字符元素赋值，而不能用赋值语句对整个数组赋值。如下面的代码：

```
char c[6];
c={'H','e','l','l','o'};                        //错误，不能对整个数组一次赋值
c[0] = 'H';c[1]='e';c[2]='l';c[3]='l';c[4]='o';c[5]='\0';      //正确
```

如果有两个字符数组，它们的长度是相同的，是不是就可以相互赋值了呢？答案是否定的，字符数组之间赋值也需要通过引用元素一个一个地赋值。如下面的代码：

```
char c1[6];
char c2[6]="Hello";
c1=c2;                                          //错误，不能整个数组一次性赋值
c1[0]=c2[0];c1[1]=c2[1];                        //正确。
```

【例子 3.4】 声明了 2 个字符数组并初始化，最后交换 2 个数组元素并打印输出。

--Example3.4.cpp--

```
01  #include<iostream>
02  using namespace std;
03  void main(void)
04  {
05      char c1[10] = {'H','e','l','l','o'};    //声明一个字符数组并初始化
06      char c2[10] = "Welcome";                //同上
07      cout<<"交换前："<<endl;
08      cout<<"c1 的内容是:"<<c1<<endl;          //输出字符数组内容
09      cout<<"c2 的内容是:"<<c2<<endl;
10      for(int i=0;i<10;i++)
11      {
12          char tmp;                           //交换
13          tmp = c1[i];
14          c1[i] = c2[i];
15          c2[i] = tmp;
16      }
17      cout<<"交换后："<<endl;
18      cout<<"c1 的内容是:"<<c1<<endl;
19      cout<<"c2 的内容是:"<<c2<<endl;
20  }
```

编译、运行上述程序得到如下结果：

```
交换前：
c1 的内容是:Hello
c2 的内容是:Welcome
交换后：
```

```
c1 的内容是:Welcome
c2 的内容是:Hello
```

分析上述程序，首先程序安装按照规则声明了 2 个字符数组并初始化。这 2 个数组的长度是一样的，然后通过一个 for 循环进行元素值的交换。

3.3.2　字符串的输入/输出

字符串在 C++中是一个十分重要的概念，常常用到。而且字符串的操作有很多（如字符串比较、字符串查找等），在这里只介绍字符串的输入/输出操作，字符串的其他操作会在后续章节进行介绍。字符串的输入/输出方式一般有两种：

（1）逐个字符的输入/输出，一般使用函数 getchar 和 putchar。如下面的程序段：

```
char c[10];                              //字符数组声明
for(int i=0;i<10;i++)
{
    c[i] = getchar();                    //一个个读入字符
}
for(i=0;i<10;i++)
{
    putchar(c[i]);                       //一个个输出字符
}
```

（2）将整个字符串一次性输入或者输出，这种方式一般只有 C++标志输入/输出函数 cin 或 cout。例如下面程序段：

```
char c[10];
cin>>c;                                  //一次性输入
cout<<c;                                 //一次性输出
```

比较两种输入方式，第二种看起来简洁明了，不过面向不同的应用需要选择不同的输入方式。如果想要在输入时就判断输入的是不是小写字母（'a'～'z'），显然采用单个输入的方式是最好的。那么如果想要通过输入字符串来查询名字，那么显然一次输入并进行查询更好。

【例子 3.5】 通过基本字符串输入/输出操作来巩固上述知识点。

--Example3.5.cpp--

```
01  #include<iostream>
02  using namespace std;
03  void main(void)
04  {
05      char c[10];
06      for(int i=0;i<10;i++)
07      {
08          c[i] = getchar();            //单个字符输入
09      }
10      cout<<"c 的内容是:";
11      for(i=0;i<10;i++)
12      {
13          putchar(c[i]);               //单个字符输出
14      }
15      cout<<endl;
16      cin>>c;                          //一次输入
17      cout<<"c 的新内容是:"<<c<<endl;
```

```
18  }
```

编译、运行上述程序得到如下结果：

```
Hello C++↙
c 的内容是:Hello C++

welcom↙
c 的新内容是:welcom
```

分析上述程序，通过一个循环来一个个读入字符，程序中输入测试字符串“Hello C++”，这里只有 9 个字符，但程序需要读入 10 个字符才能跳出循环，所以再输入回车符（'↙'）后，刚好读到了 10 个字符。那么此时字符数组 c 的内存分布如图 3.6 所示。putchar 函数的功能就是将字符数组 c 里面的字符一个个地打印输出，当然也包括回车符。最后通过 cin 函数一次性输入“welcome”这个字符串，并打印输出。

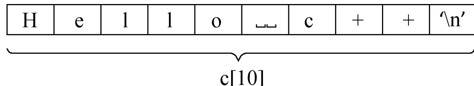


图 3.6　字符数组 c[10]的内存分布

3.3.3　字符串应用示例

现在已经介绍完了字符数组，通常用字符数组来存储字符串内容。下面通过字符串来实现一个较为全面的示例，通过这个示例使读者加深对字符串应用的印象。这个例子实现的核心功能就是通过输入的字符串来查找程序中是否有相应匹配的字符串。程序具体功能描述如下：

（1）等待用户输入字符串，如果整个字符串就是一个“0”，那么退出程序，否则进入步骤（2）。

（2）利用循环将内存中保存的字符串逐个和输入字符串进行匹配，如果匹配成功就输出“存在”，否则输出“不存在”。

（3）回到步骤（1）。

程序难点分析，在步骤（1）中只需要通过 cin 函数就可以实现输入字符串。字符串的匹配是一个难点，判断两个字符串相等的条件必须是字符串里面每个字符都是相等的。如：“abc”和“acb”不是相等的，“abc”和“abc”才相等。所以根据上述理解，可以利用循环来判断是不是相等。

【例子 3.6】 实现了上述描述的功能，具体代码如下：

--Example3.6.cpp--

```
01  #include<iostream>
02  using namespace std;
03  void main(void)
04  {
05      char str[][10] = {"Hello","Welcome","C++","function","world"};
                                                    //保存的字符串
06      char inputstr[10];
07      int i(0),j(0);
08      bool bsame;                                 //是否相同标志
09      while(cin>>inputstr&&inputstr[0]!='0')      //输入字符串
10      {
11          bsame = false;
```

```
12          for(i=0;i<5;i++)
13          {
14              for(j=0;j<10;j++)
15              {
16                  if(str[i][j]!=inputstr[j])
                                            //判断字符是否相等，不相等退出循环
17                      break;
18                  if(str[i][j]=='\0'&&inputstr[j]=='\0')
                                            //判断两个字符串是否结束
19                  {
20                      bsame = true;
21                      break;
22                  }
23              }
24              if(j>=10||bsame)
25                  break;
26          }
27          if(i<5)                         //说明字符串存在
28          {
29              cout<<inputstr<<"存在"<<endl;
30          }
31          else
32          {
33              cout<<inputstr<<"不存在"<<endl;
34          }
35      }
36      cout<<"退出程序"<<endl;
37  }
```

编译、运行上述程序得到如下结果：

```
Welcome
Welcome 存在
hello
hello 不存在
0
退出程序
```

分析上述程序，程序的核心是需要用到两个嵌套的 for 循环来实现，最里面的 for 是实现的关键，比如要判断输入字符串“Welcome”。

首先用 str 二维字符数组中存储的第一个字符串来判断。第一个字符串是“Hello”，那么进入第二个 for 循环以后，执行第 16 行这个判断，‘H’显然是不和‘W’字符相等的，所以执行 break 语句退出循环。

接着执行第二次循环，用字符串“Welcome”来判断，由于都相等，所以最后判断来到第 18 行代码，这行代码的作用是判断两个字符串是否已经达到末尾。显然成立，于是将标志变量 bsame 置为 true，然后跳出循环。此时，执行到了第 24 行代码这里，显然 bsame 是真的（true）的，那么等式成立，跳出第一个 for 循环。最后由于 i 的值是小于 5 的，所以输出该字符串存在。

3.4　本 章 小 结

数组概括地说就是数据有序的集合，这些数据必须是同一种类型的。数组是一个大小固定的复合数据类型，如果需要形象地理解数组，那么可以把数组比喻成一个水杯，水杯

的大小是固定的，倒入的水不能太多，否则就会溢出。

本章主要介绍了如何创建一般数据类型的数组（int、char、double 等），探讨了这些数组的特性。同时，本章也介绍了字符串这个特殊的数据，一般通过字符数组来存储字符串。第 18 章还会介绍用 string 来存储字符串。数组应用灵活，应该注意数组的一些操作，特别是数组操作是不能越界的，否则会引起意想不到的错误。

3.5　本 章 习 题

一、基础填空

1．数组元素之间的关系是它们必须具有________类型，常见的一般类型数组类型有________、________、________及________。

2．访问数组元素需要通过数组下标进行访问，数组的下标可以是________或者________。

3．已知一个数组：int ia[][4]={{2,10,7,8},{3,11,2},{1,0},{9}};这个数组是一个________维数组，该数组一共有________个元素，每个元素的类型都是________。数组元素 ia[0][0]的值是________，数组元素 ia[3][1]值是________，数组元素 ia[2][2]的值是________。

4．有一个字符数组：char c[][10]={"Hello","World","name"};这是一个________维的字符数组，该数组一共有________个元素，数组元素 c[0][0]代表的字符是________，数组元素 c[1][4]表示的字符是________，数组元素 c[2][5]表示的字符是________。

二、选择题

1．下列一维数组初始化正确的是________。

A．int　a[3]={1,2,3,4};　　B．int　a[]={0,1};

C．int　a[3]={1,,2};　　D．int　a[3]={}

2．已知一个数组：double　d[5]={1,0.3,2,7.5,10.0};以及一个 int 类型变量 i=5，那么访问该数组 d[(i++)/2+i%4]的结果是________。

A．1　　B．7.5　　C．10.0　　D．0.3

3．以下程序段输出结果正确的是________。

```
char  c[10]={"Hel\0lo,\0i\n"};
cout<<c;
```

A．Hel　　B．Hello,　　C．Hello,I　　D．Hello,i\n

4．下列对字符数组描述错误的是________。

A．字符数组可以存放字符串

B．字符数组的字符串可以整体输入、输出

C．可以通过赋值语句对字符数组整体赋值

D．不可用关系运算符对字符数组中的字符串进行比较

5．有两个字符数组：char　a[]="ABC"和 char　b[]={'A','B','C'};则下列叙述正确的是________。

A．a 与 b 完全相同　　B．a 与 b 长度相同

C．a 与 b 都存放字符串　　　　D．a 数组比 b 数组长度长

三、编程实践

1．在程序中声明一个大小为 20 的字符数组，然后一次性输入一个不大于 20 的字符串，最后一次性输出所输入的字符串。

2．求平均值问题，要求程序求出输入一串整数的平均值，输入整数的个数不超过 10 个。首先程序输入一个整数 *n* 表示接下来要输入的整数个数，如果输入的 n 为 0，则退出程序，否则输入 *n* 个整数，然后求出输入整数的平均值，最后输出求出的平均值。输入格式举例如下：

```
4↙
2 4 5 7↙
平均值：4
0↙
退出程序
```

3．判断输入的整数是否是回文数，回文数是一个数字，这种数字的特点就是正着读和逆着读都是一样的。比如 121、22、111 这些都是回文数。那么现在程序的要求是，输入整数判断其是否是回文数。输入的格式：

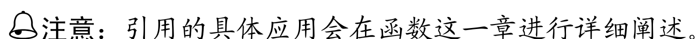
```
2↙
回文数
123↙
不是回文数
```

第 4 章　指针与引用

指针无疑是 C++编程语言中最强大的特性之一，它使程序更富有灵活性和高效性。然而，指针也是 C++语言中最难掌握的概念之一。要想熟练地掌握 C++编程语言，必须理解掌握指针概念，并且学会灵活运用。大道至简，本章力求通过一些通俗的比喻和简明的示例来一步步揭开指针和引用神秘的面纱。

本章主要涉及的知识点有：

- 声明指针，学会如何声明不同类型的指针。
- 指针和数组，学会使用指针访问数组，理解数组和指针的异同。
- 动态内存分配，学会如何动态申请内存空间及一些应该注意的问题。
- 引用，本章最后讲解了引用的基本概念以及和指针的关系。



4.1　市场经济——指针的定义及格式

指针到底是什么？市场经济体制。不是吗？相比于计划经济来说，市场经济下，每个人的需求都是按需进行分配的，买多少食物、吃什么东西都是根据个人意愿来购买，需求决定市场。指针跟市场经济有很多相似之处。本节将会详细地介绍指针的基本概念，理解并掌握这个概念是后续 C++语言学习的基础。了解指针概念后，通过进一步实践操作从而对指针有更深的认识。

4.1.1　什么是指针

指针是 C++中的一种重要的复合数据类型，与其他数据类型不同的是，指针是一种“用来存放地址值的”变量。计算机中，数据是存放在存储单元里面的，每个存储单元都有一个固定的地址，地址也是一个数。要想读出某个存储单元的数据，必须将这个地址告诉访存指令，才能找到这个单元。这个存储地址的单元就是指针。打个比方：快递公司要想把快件送到客户手中，必须知道客户的详细地址，而这个详细地址就类似于存储单元的地址。

但是，仅仅知道数据存储位置是不够的。如图 4.1 所示，有一个指向内存地址 0xA010:F020 的指针，指令根据该地址就可以读出 0xA010:F020 的数据。但是，这个数据代表什么意义却并不明确。因为不同的数据类型占用的内存空间是不一样的，比如：在 32 位计算机上，int 类型数据占 4 个字节，short 类型则占 2 个字节。指针必须指定需要访问的是哪种数据类型，如果是 int 类型则获取到的数据是 0xF0340012 这个值，如果是 short 类型数据值则为 0xF034。

因此，指针中所包含的不仅仅是存放地址值，更重要的是它还表示了数据项的类型。值得一提的是，为什么指针占用了 4 个字节的内存空间。Windows 采用的是一种叫虚拟内存的技术，每个应用程序在理论上有 4GB 的内存空间，空间中的每个地址都必须是唯一的，根据前面的知识可以知道 4（GB）=4*1024*1024*1024（B），那么内存地址就需要 32（bit）来表示。与现实生活中，如果地址不唯一，那么快递公司就不知道该往哪里寄包裹是同样的道理。

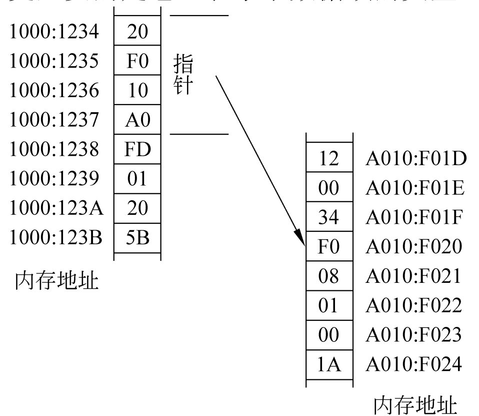


图 4.1　指针内存示意图

4.1.2　声明一个指针

使用指针之前需要对指针变量进行声明，指针的声明类似于一般的变量，区别在于指针类型名后面跟了一个星号(*)。

```
<类型> * <指针名>                //如 int *pInt;
```

其中<类型>表示的是指针的类型，<指针名>是标识符。*是一个修饰符，表示声明了一个指针变量。除了例子给出的声明 int 类型指针外，还可以用如：long、double、float、char 等声明一个指针变量。

```
long*   pLong;                  //声明一个 long 类型的指针
double* pDouble;                //声明一个 double 类型指针
float*  pFloat;                 //声明一个 float 类型指针
char*   pChar;                  //声明一个 char 类型指针
```

注意：指针变量名通常在其前面加个 p 字符，来表示该变量是一个指针，利于后期程序维护，是一种比较好的编程风格。

从上面可以发现指针变量的声明是在类型旁边加星号，但这种表示并不是唯一的。还可以把星号放在指针名旁边。

```
int *pInt;
```

这两种声明方式都是正确的。虽然第一种方式比较常用，但是当混合使用一般变量和指针变量声明的时候容易产生混淆。比如下面的语句：

```
int*    pInt1, pInt2;
```

这种声明方式并不是声明了两个 int 类型的指针变量，只有 pInt1 才是 int 型指针变量，pInt2 是一个 int 型变量。但是如果按照第二种声明方式就不易产生混淆，然而会产生的问题是第二种方式含义不够清晰：

```
int *pInt1, pInt2;
```

因此，最好的表示方法还是分开表示，且利于程序对各个变量的注释。

```
int*    pInt1;                      //声明了一个 int 型指针变量
int pInt2;                          //声明一个 int 型变量
```

【例子 4.1】 声明 long、short 类型的指针和变量，并查看输出对应指针和变量的值和地址。对如何声明一个指针加深印象。

--Example4.1.cpp--

```
01  #include<iostream>
02  using namespace std;
03  void main(void)
04  {
05      long        L;          //声明一个长整型变量
06      long*       pL;         //声明一个长整型指针
07      short   S;              //声明一个短整型变量
08      short*  pS;             //声明一个短整型指针
09      pL  =   &L;             //&获取 L 变量的地址
10      pS  =   &S;
11      cout<<"L 内存地址是：\t"<<&L<<endl;
12      cout<<"pL 内存地址是：\t"<<&pL<<endl;
13      cout<<"S 内存地址是：\t"<<&S<<endl;
14      cout<<"pS 内存地址是：\t"<<&pS<<endl;
15      cout<<"pL 的值是：\t"<<pL<<endl;
16      cout<<"pS 的值是：\t"<<pS<<endl;
17      //获取 L、S 在内存所占字节数
18      cout<<"L 占内存字节数：\t"<<sizeof(L)<< "\tS 占内存字节数：
19      \t"<<sizeof(S)<<endl;   //获取 pL、pS 在内存所占字节数
20      cout<<"pL 占内存字节数：\t"<<sizeof(pL)<< "\tpS 占内存字节数：
        \t"<<sizeof(pS)<<endl;
21  }
```

编译、运行程序得到的输出结果为：

```
L 内存地址是：    0017F7A8
pL 内存地址是：   0017F79C
S 内存地址是：    0017F790
pS 内存地址是：   0017F784
pL 的值是：       0017F7A8
pS 的值是：       0017F790
L 占内存字节数：  4          S 占内存字节数：  2
pL 占内存字节数：  4         pS 占内存字节数：4
```

观察例子 4.1 运行结果，分析总结得到：

- ❑ 变量 L（或者 S）地址和指向变量的指针地址 pL（或者 pS）是没有对应直接关系的。
- ❑ 变量 L（或者 S）的地址和指向变量 pL（或者 pS）指针值是一致的。
- ❑ 无论哪种类型的指针，其所占内存空间在 32 位的 Windows 操作系统下都是 4 个字节，不会随指向类型不同而不同。

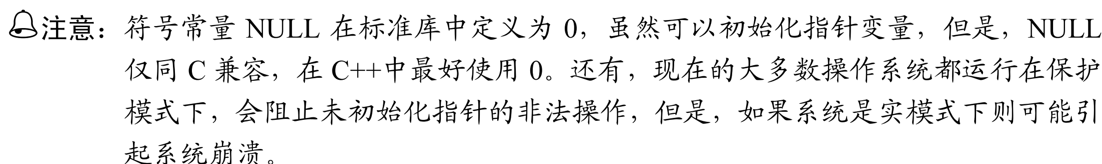
4.1.3　初始化指针变量

在 C++中，指针声明时并不会对其进行初始化，而是分配一个随机值，操作未初始化的指针是很危险的，这个值可能指向一个非常重要的数据，不当的操作可能会引起程序崩溃，甚至是系统的崩溃。所以把声明的指针初始化就好比要把不用的枪打开保险一样，枪

支走火可不是闹着玩的。指针“失控”也是相当危险的。因此，指针应该在声明或赋值语句中进行初始化。指针可以被初始化为 0、NULL（符号化常量）或地址，其中值为 0 或者 NULL 的指针不指向任何内容。变量的地址则是通过“&”符号把地址赋值给指针变量。如下列代码所示：

```
int *     pInt   =   0;                //或者用 *pInt    =  NULL;
long      A      =   10;
long*     pLong  =   &A;               //初始化 pLong 指针
```

上面的这些语句首先声明了一个 int 类型的指针变量，并初始化为空。接着又声明一个 long 类型的变量 A，并将 A 的地址赋值给一个 long 类型的指针变量 pLong。切记，在用另一个变量对指针进行初始化时，必须在该指针之前对变量进行声明，否则会引起编译错误。


注意：符号常量 NULL 在标准库中定义为 0，虽然可以初始化指针变量，但是，NULL 仅同 C 兼容，在 C++中最好使用 0。还有，现在的大多数操作系统都运行在保护模式下，会阻止未初始化指针的非法操作，但是，如果系统是实模式下则可能引起系统崩溃。

4.1.4　指向指针的指针

从前面的学习中可以知道，指针可以指向任何数据类型的对象，指针变量也是一种数据类型的对象，从而可以用另外一个指针指向它，这种指针称为指向指针的指针或二级指针。这种说法可能过于抽象，那么举个例子：今天上午要上 C++语言这门课程。于是，学生们查询他们的课表找到了上课地址是在教学楼 305 这个教室，可是等他们到达的时候发现这个教室黑板上写了一个通知，由于某些原因上课地址改为 407 这个教室，于是学生们找到 407 教室，终于赶上了重要的 C++课程。如果把 407 教室比作变量的地址的话，那么，305 黑板上的通知就是指向这个变量的一级指针，学生们的课表就是指向 305 教室的二级指针。

二级指针可以这样声明：

```
int **    pInt;                      //声明了一个 int 类型的二级指针
```

因此，可以通过“**”来声明一个二级指针。那么可以这样初始化一个二级指针：

```
int *   pInt    =   0;
int** pPint =   &pInt;               //或者： int** pPint = 0;
```

观察图 4.2 二级指针的内存布局，来加深对二级指针的理解。二级指针是一个十分常用的概念，经常和二维数组在一起使用，具体应用会在后文详细讲述。

【例子 4.2】 主要通过查看二级指针、指针及变量之间的关系，加深对二级指针使用的理解。

---Example4.2.cpp---

```
01  #include<iostream>
02  using namespace std;
```

```
03  void main(void)
04  {
05      int i   =   10;
06      int*    pInt    =   &i;             //初始化 pInt 指针
07      int**   pPint   =   &pInt;          //初始化 pPint 二级指针
08      cout<<"i 的内存地址是：\t"<<&i<<endl;
09      cout<<"pInt 值是：\t"<<pInt<<endl;
10      cout<<"pInt 的地址是：\t"<<&pInt<<endl;
11      cout<<"pPint 值是：\t"<<pPint<<endl;
12      cout<<"pPint 的地址是：\t"<<&pPint<<endl;
13      cout<<"pPint 访问 i 的结果：\t"<<**pPint<<endl;
14  }
```

编译、运行输出结果为：

```
i 的内存地址是：  0028F830
pInt 值是：       0028F830
pInt 的地址是：   0028F824
pPint 值是：      0028F824
pPint 的地址是：  0028F818
pPint 访问 i 的结果：     10
```

上面这段程序声明了一个 pInt 指针、pPint 二级指针，并都进行初始化，且依次打印输出了变量 i、指针 pInt 及指针 pPint 的地址和值。从输出结果上看 pInt 的值是 0x0028F830，就是 i 的内存地址 0x0028F830，pInt 的地址 0x0028F824 则是 pPint 的值 0x0028F824。可以看出指针地址和它所指向变量的值是没有对应联系的，如：i 的值是 10，而指向它的指针地址是 0x0028F824。

如果有一个指针指向二级指针，那么这个指针就叫三级指针，甚至还有更高阶的指针，但这些指针都很少使用，使用时也很容易引起误解。

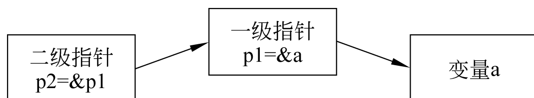


图 4.2　二级指针内存布局

4.2　指针的运算

由于指针是一种特殊的数据类型，所以对指针进行的操作就有一定的限制。除了赋值运算外，指针只能进行与整数相加减、同类型指针之间的比较运算，以及同类型指针之间的相减运算。本节将详细讲述这些运算的规则。

4.2.1　指针的赋值运算和增减值运算

1．指针的赋值运算

指针在定义或者声明的时候可以被赋初值，指针变量也可以被赋值。指针赋值运算在 C++编程中是经常用到的。

指针在进行赋值运算时应遵守一定的规则：只能将类型相同、级别一致的变量或者对

象的地址赋值给指针。例如下面声明的变量和指针：

```
int *   pInt;                //声明一个指针
int     a = 10;
double  d = 1.0;
```

对比下面的这个赋值操作：

```
pInt    = &a;                //正确赋值操作
pInt    = &d;                //编译出错
```

因为指针的类型是 int 型的，而变量是个 double 类型，违反了类型相同原则。

又比如下面的例子：

```
int a[10] , b[2][10] , *pInt;
pInt    =   a;               //正确赋值操作
pInt    =   b;               //错误赋值操作
```

数组 a 代表的是一个常量指针，它是这个数组的首地址值，所以是可以赋值的。b 是一个二级常量指针，而 p 是一级指针，违反了级别一致原则。

指针变量的赋值如：

```
int *   pInt = 0;            //初始化置为空指针，以免发生致命错误
int     a(5);                //与 a=5；效果一致
pInt    =   &a;              //用变量 a 给 pInt 指针赋值
*pInt   =  10;
```

调试一下查看运行结果，程序执行到第一步时，查看 pInt 变量值是 0，也就是这是一个空指针。接着执行第二步，a 的值是 5，执行第三步后 pInt 值变成了 a 的地址（0x0028F71A，不同计算机可能不一样）。此时变量 a 的值还是 5，执行完第四步后 pInt 值没变，而 a 的值变成了 10。这说明通过指针也可以修改指向变量的内存空间。

还有一点值得注意的是，不要被“int *p=NULL”和“*p=NULL”迷惑，这两句代码只是“穿的衣服比较相似而已”。第一句是声明一个指针 p 并将它指向一个内存特殊位置 0x00000000。而第二句话则是将 p 所指内存空间值置为 0。如果 p 指向的是合法地址，那么该空间值变成了 0，如果地址不合法，则系统会提示非法操作。这也就是为什么建议选择用‘0’来初始化一个指针而不用 NULL 的原因。

2．增值和减值运算

指针可以和一个 int 型变量进行加减运算（当然 short 类型变量也是可以的），++和– –运算符也适用于指针运算。由于指针值是一个内存地址值，加减运算相当于改变了地址值，也就是移动了指针在内存所指向的位置。注意，保存指针值的地址不变。

例如下面的代码：

```
short   a[5];                //声明一个 short 类型的数组
short   *ps = a;             //初始化 ps 指针，使其指向 a 数组
ps++;                        //ps 做增值运算
```

如图 4.3 所示为指针 ps 指针位置变化情况。图中虚线代表的是 ps 指针在没有进行自增操作时指向了内存的 0x0022F9CC 这个位置，实线代表 ps 完成自增操作之后指针位置。