第 13 章　管理数组和字符串

在前面的章节中，读者声明单个 int 变量、char 变量或其他对象。经常需要声明一组对象，如 20 个 int 变量或一组 Cat 对象。

本章介绍以下内容：

- 什么是数组以及如何声明它们？
- 什么是字符串以及如何使用字符数组表示字符串？
- 数组与指针的关系。
- 如何使用指针算术？
- 什么是链表？

13.1　什么是数组

数组是一个顺序数据存储单元集合，其中每个存储单元存储相同类型的数据。存储单元被称为数组元素。要声明数组，可指定类型数组名和下标。下标用方括号括起，指定了数组包含多少个元素。例如：

```
long LongArray[25];
```

声明了一个名为 LongArray 数组，它包含 25 个 long 元素。编译器看到该声明时，分配足够的内存来存储这 25 个元素。由于每个 long 变量占用 4 字节，因此该声明分配 100 字节连续的内存，如图 13.1 所示。

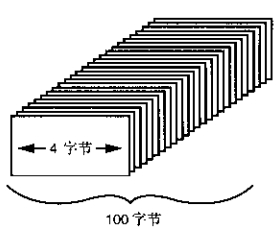

图 13.1　声明数组

13.1.1　访问数组元素

通过使用离数组开头的偏移量，可以访问数组元素。数组元素偏移量从 0 开始。因此第一个元素为 arrayName[0]。在 LongArray 范例中，LongArray[0]是第一个数组元素，LongArray[1]是第二个，依次类推。

这可能有些令人迷惑。数组 SomeArray[3]有三个元素，它们分别是 SomeArray[0]、SomeArray[1]和 SomeArray[2]。推而广之，SomeArray[*n*]有 *n* 个元素，分别是 SomeArray[0]到 SomeArray[*n*-1]。这是因为索引是偏移量，因此第一个元素位于从数组开头开始的第 0 个存储单元中，第二个元素位于第一个存储单元中，

依此类推。

因此，LongArray[25]的元素为 LongArray[0]到 LongArray[24]。程序清单 13.1 演示了如何声明一个包含 5 个元素的 int 数组并给每个元素赋值。

注意：从本章开始，程序清单中的行号将从零开始，旨在帮助读者牢记 C++中的数组索引从零开始。

程序清单 13.1 使用 int 数组

```
0:  //Listing 13.1 - Arrays
1:  #include <iostream>
2:
3:  int main()
4:  {
5:     int myArray[5];     // Array of 5 integers
6:     int i;
7:     for ( i=0; i<5; i++)  // 0-4
8:     {
9:        std::cout << "Value for myArray[" << i << "]: ";
10:       std::cin >> myArray[i];
11:    }
12:    for (i = 0; i<5; i++)
13:       std::cout << i << ": " << myArray[i] << std::endl;
14:    return 0;
15: }
```

输出：

```
Value for myArray[0]: 3
Value for myArray[1]: 6
Value for myArray[2]: 9
Value for myArray[3]: 12
Value for myArray[4]: 15
0: 3
1: 6
2: 9
3: 12
4: 15
```

分析：

程序清单 13.1 创建一个数组，要求用户输入每个元素的值，然后将这些值打印到控制台。第 5 行声明了一个名为 myArray 的 int 数组，这是因为 5 被放在方括号中，这意味着 myArray 能够存储 5 个整数，其中每个元素都可被视为一个 int 变量。

从第 7 行开始是一个 for 循环，它从 0 数到 4，这正好对应于包含 5 个元素的数组的索引。第 9 行提示用户输入一个值，第 10 行将这个值存储到数组的相应位置。

从第 10 行可知，每个元素都是使用数组名和用方括号括起的偏移量来访问的。然后，每个元素都可被视为一个变量，其类型为数组的类型。

第一个值存储在 myArray[0]中，第二个存储在 myArray[1]中，依次类推。在第 12 和 13 行，第二个 for 循环将每个值显示到屏幕上。

注意：数组索引从 0 而不是 1 开始。在 C++新手编写的程序中，这是导致很多错误的原因。将索引视为偏移量。第一个元素（如 ArrayName[0]）位于数组开头，因此偏移量为 0。因此，使用数组时，别忘了包含 10 个元素的数组的元素为 ArrayName[0]到 ArrayName[9]，使用元素 ArrayName[10]是错误的。

13.1.2　在数组末尾后写入数据

将值写入到数组元素中时，编译器根据每个元素的大小和下标来确定将这个值存储到哪里。假设你要求将一个值写入到 LongArray[5]（第 6 个元素）中，编译器将偏移量（5）和元素的大小（这里为 4）相乘。然后从数组开头向前移动相应数量（20）的字节，并将新值写入到这个位置。

如果你将输入写入到 LongArray[50]中，大多数编译器对“没有这样的元素”视而不见，而是计算从第一个元素开始需要移动多少个字节（200），然后将值写入到这个位置。这里存储的可能是任何数据，将新值写入到这里可能导致不可预料的结果。如果幸运的话，程序将立即崩溃；如果不那么幸运，在很久以后程序将产生奇怪的结果，且很难确定程序在什么地方出了问题。

编译器就像一个以步数测量距离的盲人。他从第一座房子 MainStreet[0]开始，当你要求他前往 Main Street 的第 6 座房子时，他自言自语地说：我必须再穿过 5 座房子，每座房子为 4 大步，因此还要走 20 步。如果你要他去 Mainsrteet[100]，而 Main Street 上只有 25 栋房子，他将向前走 400 步。远在到达目的地之前，他肯定地会撞向一辆卡车。因此，要他去什么地方之前，你一定要三思。

程序清单 13.2 在数据末尾后面写入数据。读者应编译该程序，看是否出现错误和警告消息。如果没有，在使用数组时一定要多加小心！

注意：千万别运行该程序，它可能导致系统崩溃。

程序清单 13.2　在数组末尾后写入数据

```
 0: //Listing 13.2 - Demonstrates what happens when you write
 1: // past the end of an array
 2: #include <iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    // sentinels
 8:    long sentinelOne[3];
 9:    long TargetArray[25]; // array to fill
10:    long sentinelTwo[3];
11:    int i;
12:    for (i=0; i<3; i++)
13:      {
14:        sentinelOne[i] = 0;
15:        sentinelTwo[i] = 0;
16:    }
17:    for (i=0; i<25; i++)
18:      TargetArray[i] = 10;
19:
20:    cout << "Test 1: \n";  // test current values (should be 0)
21:    cout << "TargetArray[0]: " << TargetArray[0] << endl;
22:    cout << "TargetArray[24]: " << TargetArray[24] << endl << endl;
23:
24:    for (i = 0; i<3; i++)
25:    {
26:       cout << "sentinelOne[" << i << "]: ";
27:       cout << sentinelOne[i] << endl;
28:       cout << "sentinelTwo[" << i << "]: ";
29:       cout << sentinelTwo[i]<< endl;
30:    }
31:
```

```
32:    cout << "\nAssigning...";
33:    for (i = 0; i<=25; i++)    // Going a little too far!
34:      TargetArray[i] = 20;
35:
36:    cout << "\nTest 2: \n";
37:    cout << "TargetArray[0]: " << TargetArray[0] << endl;
38:    cout << "TargetArray[24]: " << TargetArray[24] << endl;
39:    cout << "TargetArray[25]: " << TargetArray[25] << endl << endl;
40:    for (i = 0; i<3; i++)
41:    {
42:      cout << "sentinelOne[" << i << "]: ";
43:      cout << sentinelOne[i]<< endl;
44:      cout << "sentinelTwo[" << i << "]: ";
45:      cout << sentinelTwo[i]<< endl;
46:    }
47:
48:    return 0;
49: }
```

输出：

```
Test 1:
TargetArray[0]: 10
TargetArray[24]: 10

SentinelOne[0]: 0
SentinelTwo[0]: 0
SentinelOne[1]: 0
SentinelTwo[1]: 0
SentinelOne[2]: 0
SentinelTwo[2]: 0

Assigning...
Test 2:
TargetArray[0]: 20
TargetArray[24]: 20
TargetArray[25]: 20

SentinelOne[0]: 20
SentinelTwo[0]: 0
SentinelOne[1]: 0
SentinelTwo[1]: 0
SentinelOne[2]: 0
SentinelTwo[2]: 0
```

分析：

第 8 行和第 10 行声明了两个包含 3 个元素的 long 数组，用作守卫 TargetArray 哨兵。第 12～16 行将这些哨兵数组的元素都初始化为 0。由于这两个数组分别是在 TargetArray 之前和之后声明的，因此，在内存中，它们很可能分别位于 TargetArray 的前面和后面。如果写入时，超过了 TargetArry 数组的末尾，被修改的很可能是哨兵数组，而不是未知的数据区域。有些编译器对内存编号时采用倒计数，有些采用正计数。因此，在 TargetArray 的前面和后面都放置了哨兵数组。

第 20～30 行打印哨兵数组的值以及数组 TargetArray 的第一个元素和最后一个元素，以确认它们正常。第 34 行给 TargetArray 的所有元素重新赋值，将它们从初始值 10 改为 20，但将偏移量数到了 25，而 TargetArray[25]不存在。

作为第二次测试，第 37～39 行打印数组 TargetArray 的值，以方便查看。注意，第 39 行的打印结果表明，TargetArray[25]的值为 20。然而，打印 SentinelOne 和 SentinelTwo 时，SentinelOne[0]的值被修改了。这是因为从 TargetArray[0]开始的第 25 个元素与 SentinelOne[0]使用的内存相同。存取不存在的 TargetArray[25]时，实际上存取的是 SentinelOne[0]。

注意：由于编译器使用内存的方式不同，因此读者的结果可能与此不同。读者可能发现，哨兵数组并没有被覆盖。在这种情况下，请尝试修改第 33 行，将其中的 25 改为 26。这将增大哨兵数组被覆盖的可能性。当然，也可能覆盖其他东西或导致系统崩溃。

这种令人讨厌的错误可能很难被发现，因为修改 SentinelOne[0]的值是在根本没有对 SentinelOne 数组执行写操作的代码中进行的。

13.1.3　护栏柱错误

由于越过数组末尾进行写操作的错误很常见，因此它有专用名称：护栏柱错误（fence post error）。它指的是这样一种问题：要建立一堵长 10 英尺的护栏，如果两个相邻护栏柱相隔 1 英尺，需要多少个护栏柱？大多数人会说 10 个，但实际上需要 11 个。图 13.2 说明了这一点。

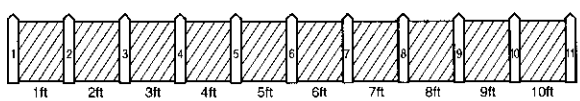

图 13.2　护栏柱错误

这种“少 1（off by one）”计数可能是 C++程序员的噩梦。然而，随着时间的推移，读者将习惯这样一种观念：包含 25 个元素的数组的最大索引为 24，一切都是 0 开始计数的。

注意：有些程序员将 ArrayName[0]称为第 0 个元素。养成这种习惯是种错误。如果 ArrayName[0]是第 0 个元素，ArrayName[1]又是什么呢？第一个元素吗？如果是的话，那么当你看到 ArrayName[24]时，认为它不是第 24 个而是第 25 个元素吗？更不会令人迷惑的说法是：ArrayName[0]的偏移量为零，是第一个元素。

13.1.4　初始化数组

可以在声明内置类型（如 int 和 char）的简单数组时对其进行初始化。为此，在数组名后加上等号（=）以及一组用大括号括起、用逗号分隔的值。例如：

```
int IntegerArray[5] = { 10, 20, 30, 40, 50 };
```

将 IntegerArray 声明为一个包含 5 个元素的 int 数组，并将 10 赋给 IntegerArray[0]，将 20 赋给 IntegerArray[1]等。

如果省略数组大小，将创建一个刚好能够存储初始值的数组。因此，如果这样编写代码：

```
int IntegerArray[] = { 10, 20, 30, 40, 50 };
```

将创建与前一个例子中相同的数组：一个包含 5 个元素的数组。

初始化的元素数不能超过为数组声明的元素数。因此，下面的代码将导致编译错误，因为你声明了一个包含 5 个元素的数组，却提供了 6 个初始值：

```
int IntegerArray[5] = { 10, 20, 30, 40, 50, 60};
```

然而，下面的写法是合法的：

```
int IntegerArray[5] = {10, 20};
```

在这个例子中，你声明了一个包含 5 个元素的数组，但只初始化前两个元素：IntegerArray[0]和

IntegerArray[1]。

应该：

对于被初始化的数组，应让编译器去设置其大小。

请牢记数组第一个元素的偏移量为 0。

不应该：

不能在超过数组末尾的地方执行写入操作。

不要使用怪异的数组名，数组名应该向其他变量名一样有意义。

13.1.5 声明数组

数组可以使用任意合法的变量名，但不能与其作用域中的其他变量或数组同名。因此不能同时有一个名为 myCats[5]的数组和一个名为 myCats 的变量。

另外，声明数组大小时，除使用字面量外，还可以使用常量或枚举值。实际上，使用常量或枚举值比使用字面量更好，因为它让你能够在一个地方控制元素数。在程序程序清单 13.2 中，使用的是字面量。如果要修改 TargetArray，使之存储 20 而不是 25 个元素，必须修改多行代码。如果使用常量来指定数组大小，则只需要修改该常量的值。

使用枚举量来指定元素数（数组长度）稍为有些不同，程序清单 13.3 说明了这一点，它创建一个数组来存储一周中每天的值。

程序清单 13.3 使用枚举量来指定数组大小

```
 0: // Listing 13.3
 1: // Dimensioning arrays with consts and enumerations
 2:
 3: #include <iostream>
 4: int main()
 5: {
 6:     enum WeekDays { Sun, Mon, Tue,
 7:             Wed, Thu, Fri, Sat, DaysInWeek };
 8:     int ArrayWeek[DaysInWeek] = { 10, 20, 30, 40, 50, 60, 70 };
 9:
10:    std::cout << "The value at Tuesday is: " << ArrayWeek[Tue];
11:    return 0;
12: }
```

输出：

```
The value at Tuesday is: 30
```

分析：

第 6 行创建了一个名为 WeekDays 的枚举类型，它有 8 个成员。Sun 的值为 0，DaysInweek 的值为 7。第 8 行声明了一个名为 ArrayWeek 的数组，它包含 DaysInWeek（7）个元素。

第 10 行将枚举常量 Tue 用作数组索引。由于 Tue 的值为 2，因此第 10 行返回并打印数组的第三个元素：ArrayWeek[2]。

数组

要声明数组，可指定其存储的对象的类型以及数组名和决定数组能存储多少个对象的下标。

范例 1

```
int MyIntegerArray[90];
```

范例 2

```
long * ArrayOfPointersToLongs[100];
```

要访问数组成员，可使用下标运算符。

范例 1

```
// assign ninth member of MyIntegerArray to theNinethInteger
int theNinethInteger = MyIntegerArray[8];
```

范例 2

```
// assign ninth member of ArrayOfPointersToLongs to pLong.
long * pLong = ArrayOfPointersToLongs[8];
```

数组索引从 0 开始。包含 n 个元素的数组的索引为 0 到 n–1。

13.2　使用对象数组

任何对象（无论是内置的，还是用户自定义的）都可存储在数组中。声明对象数组对，你告诉编译器要存储的对象的类以及为多少个对象分配内存。根据类声明，编译器知道每个对象需多大内存。类必须有一个不接受任何参数的默认构造函数，以便在定义数组时创建对象。

访问数组中对象的成员数据分两步。首先使用下标运算符（[]）指定数组元素，然后使用成员运算符（.）访问成员变量。程序清单 13.4 演示了如何创建一个包含 5 个 Cat 对象的数组。

程序清单 13.4　创建对象数组

```
 0: // Listing 13.4 - An array of objects
 1:
 2: #include <iostream>
 3: using namespace std;
 4:
 5: class Cat
 6: {
 7:   public:
 8:     Cat() { itsAge = 1; itsWeight=5; }
 9:     ~Cat() {}
10:    int GetAge() const { return itsAge; }
11:    int GetWeight() const { return itsWeight; }
12:    void SetAge(int age) { itsAge = age; }
13:
14:   private:
15:    int itsAge;
16:    int itsWeight;
17: };
18:
19: int main()
20: {
21:   Cat Litter[5];
22:   int i;
23:   for (i = 0; i < 5; i++)
24:     Litter[i].SetAge(2*i +1);
25:
26:   for (i = 0; i < 5; i++)
```

```
27:     {
28:         cout << "Cat #" << i+1<< ": ";
29:         cout << Litter[i].GetAge() << endl;
30:     }
31:     return 0;
32: }
```

输出：

```
Cat #1: 1
Cat #2: 2
Cat #3: 3
Cat #4: 4
Cat #5: 5
```

分析：

第 5～17 行声明了 Cat 类。Cat 类必须有一个默认构造函数，以便能够在声明数组时创建 Cat 对象。在这个例子中，默认构造函数是在第 8 行声明和定义的。对于每个 Cat 对象，年龄和体重分别被设置为默认值 1 和 5。本书前面指出过，如果你创建了任何构造函数，编译器将不会提供默认构造函数，你必须自己创建。

第一个 for 循环（第 23 和 24 行）设置数组中 5 个 Cat 对象的年龄。第二个 for 循环（第 26～30 行）访问数组的每个元素，并调用 GetAge()来显示每个 Cat 对象的年龄。

每个 Cat 对象的 GetAge()方法是通过访问数组元素来调用的：使用 Litter[i]、句点运算符（.）和成员函数。可以使用相同的方法来访问其他成员和方法。

13.2.1 声明多维数组

数组可以有多维，每维用一个下标表示，因此一维数组有两个下标，三维数组有一个下标，依次类推。虽然创建的数组通常为一维或二维，但数组可以有任意维数。

一个典型的二维数组是棋盘，其中一维代表 8 行，另一维代表 8 列，如图 13.3 所示。

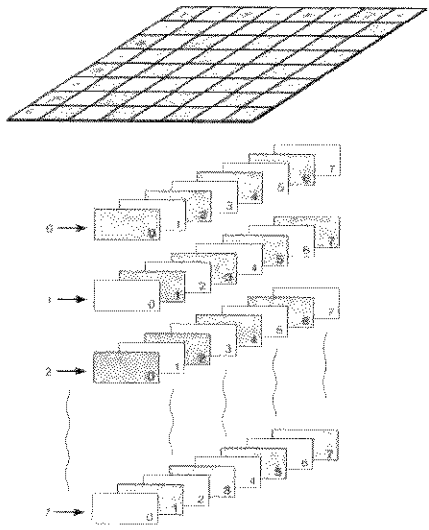

图 13.3 棋盘和二维数组

假设有一个名为 SQUARE 的类，声明一个名为 Board 的数组来表示棋盘的代码如下：

```
SQUARE Board[8,'8];
```

也可以用一个包含 64 个 SQUARE 对象的一维数组来表示棋盘，例如：

```
SQUARE Board[64];
```

然而，这不如二维数组那样更准确地对应现实世界中的物体。国际象棋开局时“王”位于第 1 行的第 4 个位置，假设数组的第一个下标代表行，第二个下标代表列，则这个位置对应于：

```
Board[0][3];
```

13.2.2　初始化多维数组

可以初始化多维数组。指定的初始值按如下顺序被赋给数组元素：最后（最右边）的数组下标从 0 递增，其他下标保持不变。因此，如果有下面这样的数组：

```
int theArray[5][3];
```

则前 3 个初始值将存储到 theArray[0]中；接下来的 3 个值存储到 theArray[1]中；依次类推。

可以这样来初始化该数组：

```
int theArray[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
```

出于清晰考虑，可以用大括号将初始值分组，例如：

```
int theArray[5][3] = {  {1,2,3},
      {4,5,6},
      {7,8,9},
      {10,11,12},
      {13,14,15} };
```

编译器将忽略里面的大括号，但它们确实使得更容易理解这些数字是如何被分配的。

初始化数组元素时，不管是否使用大括号，每个数都必须用逗号分开。全部初始值必须用大括号括起，并以分号结尾。

程序清单 13.5 创建了一个二维数组。第一维是由 0～4 组成，第二维元素是第一维对应元素的两倍。

程序清单 13.5　创建多维数组

```
 0: // Listing 13.5 - Creating a Multidimensional Array
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    int SomeArray[2][5] = { {0,2,3,4}, {0,2,4,6,8}};
 7:    for (int i = 0; i<2; i++)
 8:    {
 9:        for (int j=0; j<5; j++)
10:        {
11:          cout << "SomeArray[" << i << "][" << j << "]: ";
12:          cout << SomeArray[i][j]<< endl;
13:        }
14:    }
15:    return 0;
16: }
```

输出：

```
SomeArray[0][0]: 0
```

```
SomeArray[0][1]: 1
SomeArray[0][2]: 2
SomeArray[0][3]: 3
SomeArray[0][4]: 4
SomeArray[1][0]: 0
SomeArray[1][1]: 2
SomeArray[1][2]: 4
SomeArray[1][3]: 6
SomeArray[1][4]: 8
```

分析：

第 6 行将 SomeArray 声明为一个二维数组。第一个下标指出有两组，第二个下标指出每组包含 5 个 int 值。这相当于创建了一个 2×5 的网格，如图 13.4 所示。

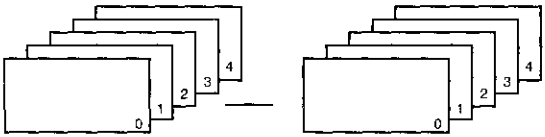

图 13.4　一个 2×5 数组

初始值被分成两组，第一组为 0～4，第二组为第一组的两倍。在这个程序清单中，直接设置数组元素的值，虽然也可以通过计算得到。第 7 行和第 9 行创建了一个嵌套 for 循环。外层 for 循环（从第 7 行开始）遍历第一个下标，对于其每个可能值，内层 for 循环（从第 9 行开始）遍历第二个下标，这与输出结果是一致的。SomeArray[0][0]的后面是 SomeArray[0][1]。仅当遍历完第二个下标的所有可能值后，才将第一个下标加 1，然后重新开始遍历第二个下标的所有可能值。

有关内存的说明

声明数组时，你告诉编译器，要在数组中存储多少个对象。编译器将为所有对象分配内存，即使从不使用它。在知道数组需要存储多少个对象后，这不是问题。例如，棋盘有 64 个方格，而猫产下 1～10 个小猫。然而，在不知道需要多少个对象时，必须使用更高级的数据结构。

本书将讨论指针数组、在自由存储区中创建的数组和其他各种集合。读者将看到一些高级数据结构，更详细的信息请参阅 Sams 公司出版的 *C++ Unleashed* 一书，也可参阅附录 E。

有关编程的两项重要内容是：总有其他的东西需要学习；总有其他的书籍需要参考。

13.3　指针数组

在之前讨论的所有数组中，其成员都存储在堆栈中。通常堆栈内存有限，而自由存储区大得多。可以在自由存储区声明每个对象，然后仅将指向对象的指针存储到数组中。这可极大地减少占用的堆栈内存量。程序清单 13.6 重写了程序清单 13.4 的数组，将所有对象存储在自由存储区。为了表明这样做可以使用更多的内存，数组元素由 5 个增加到了 500 个，名称由 Litter 改为 Family。

程序清单 13.6　在自由存储区存储数组

```
0:  // Listing 13.6 - An array of pointers to objects
1:
2:  #include <iostream>
3:  using namespace std;
4:
```

```
 5: class Cat
 6: {
 7:   public:
 8:     Cat() { itsAge = 1; itsWeight=5; }
 9:     ~Cat() {}                          // destructor
10:     int GetAge() const { return itsAge; }
11:     int GetWeight() const { return itsWeight; }
12:     void SetAge(int age) { itsAge = age; }
13:
14:   private:
15:     int itsAge;
16:     int itsWeight;
17: };
18:
19: int main()
20: {
21:    Cat * Family[500];
22:    int i;
23:    Cat * pCat;
24:    for (i = 0; i < 500; i++)
25:    {
26:       pCat = new Cat;
27:       pCat->SetAge(2*i +1);
28:       Family[i] = pCat;
29:    }
30:
31:    for (i = 0; i < 500; i++)
32:    {
33:       cout << "Cat #" << i+1 << ": ";
34:       cout << Family[i]->GetAge() << endl;
35:    }
36:    return 0;
37: }
```

输出：

```
Cat #1: 1
Cat #2: 3
Cat #3: 5
...
Cat #499: 997
Cat #500: 999
```

分析：

第 5～17 行声明的 Cat 类与程序清单 13.4 中相同，但这次在第 21 行声明的数组名为 Family，且能够存储 500 个元素，更重要的是，这 500 个元素是指向 Cat 对象的指针。

第一个循环（第 24～29 行）在自由存储区创建 500 个新的 Cat 对象，并将每个对象的年龄设置为索引的 2 倍加 1，因此第一个 Cat 的年龄被设置为 1，第二个为 3，第 3 个为 5，依次类推。创建指针后，第 28 行将其赋给数组元素。由于数组被声明为存储指针，因此将指针而不是其指向的值赋给数组元素。

第 31～35 行的第二个循环打印每个值。在第 33 行，通过打印一个数字来指出当前打印的是哪个对象。由于索引从零开始，因此第 33 行加上 1 以便从 1 开始计数。第 34 行使用索引来访问指针（Family[i]），然后使用该地址来访问 GetAge()方法。

在这个例子中，数组 Family 及其所有元素都存储在堆栈中，但创建的 500 个 Cat 对象存储在自由存储区。

13.4 指针算术

第 8 章首次介绍了指针。继续讨论数组之前，有必要回过头来讨论一个有关指针的高级主题——指针算术。

可以对指针做一些数学运算。可以将两个指针相减。一个功能强大的技巧是，将两个指针指向同一个数组中不同的元素，然后将它们相减来确定它们之间相隔几个元素。在分析字符数组方面这种技巧很有用，程序清单 13.7 演示了这一点。

程序清单 13.7 提取字符串中的单词

```
0:  #include <iostream>
1:  #include <ctype.h>
2:  #include <string.h>
3:
4:  bool GetWord(char* theString,
5:           char* word, int& wordOffset);
6:
7:  // driver program
8:  int main()
9:  {
10:   const int bufferSize = 255;
11:   char buffer[bufferSize+1];   // hold the entire string
12:   char word[bufferSize+1];     // hold the word
13:   int wordOffset = 0;          // start at the beginning
14:
15:   std::cout << "Enter a string: ";
16:   std::cin.getline(buffer,bufferSize);
17:
18:   while (GetWord(buffer, word, wordOffset))
19:   {
20:     std::cout << "Got this word: " << word << std::endl;
21:   }
22:   return 0;
23: }
24:
25: // function to parse words from a string.
26: bool GetWord(char* theString, char* word, int& wordOffset)
27: {
28:   if (theString[wordOffset] == 0)  // end of string?
29:      return false;
30:
31:   char *p1, *p2;
32:   p1 = p2 = theString+wordOffset;  // point to the next word
33:
34:   // eat leading spaces
35:   for (int i = 0; i<(int)strlen(p1) && !isalnum(p1[0]); i++)
36:     p1++;
37:
```

```
38:    // see if you have a word
39:    if (!isalnum(p1[0]))
40:      return false;
41:
42:    // p1 now points to start of next word
43:    // point p2 there as well
44:    p2 = p1;
45:
46:    // march p2 to end of word
47:    while (isalnum(p2[0]))
48:      p2++;
49:
50:    // p2 is now at end of word
51:    // p1 is at beginning of word
52:    // length of word is the difference
53:    int len = int (p2 - p1);
54:
55:    // copy the word into the buffer
56:    strncpy (word,p1,len);
57:
58:    // null terminate it
59:    word[len] = '\0';
60:
61:    // now find the beginning of the next word
62:    for (int j = int(p2-theString); j<(int)strlen(theString)
63:      && !isalnum(p2[0]); j++)
64:    {
65:      p2++;
66:    }
67:
68:    wordOffset = int(p2-theString);
69:
70:    return true;
71:  }
```

输出：

```
Enter a string: this code first appeared in C++ Report
Got this word: this
Got this word: code
Got this word: first
Got this word: appeared
Got this word: in
Got this word: C
Got this word: Report
```

分析：

该程序让用户输入一个句子，然后提取该句子中的每个单词（每组字母数字字符）。第 15 行提示用户输入一个字符串（基本上是一个句子）。第 18 行将该字符串以及 buffer（用于存储提取的一个单词）和 int 变量 WordOffset（第 13 行已经将其初始化为 0）作为参数传递给函数 GetWord( )。

GetWord( )返回字符串中的单词，直到到达字符串末尾。第 20 行打印 GetWord( )返回的单词，直到它返回 false 为止。

每次调用 GetWord( )时都将跳到第 26 行处执行。第 28 行检查 theString[wordOffset]的值是否为零。到达或超越字符串末尾后，该条件将为真，此时 GetWord( )将返回 false。cin.getline( )确保输入的字符串以空字符结尾，即以值为零的字符（'\0'）结尾。

第31行声明了两个指针：p1 和p2；第32行将它们指向离字符串开头wordOffset的位置。一开始，wordOffset 的值为 0，因此这两个指针指向字符串开头。

第 35 和 36 行沿字符向前移，让 p1 指向第一个字母数字字符。第 39 和 40 行检查是否找到了字母数字字符，如果没有，则返回 false。

现在，p1 指向下一个单词的开头，第 44 行将 p2 也指向这个位置。

第 47 和 48 行遍历一个单词，导致 p2 指向下一个非字母数字字符。这样，p2 指向这样一个单词的末尾：开头位于 p1 指向的位置。第 53 行将 p2 与 p1 相减，并将结果强制转换为 int 值，从而得到该单词的长度。然后，将 p1 作为起点并将计算得到的差作为长度传递给标准库中的字符串复制函数，从而将该单词复制到缓冲区 word 中。

第 59 行添加一个空值字符以标记单词的末尾。然后对 p2 执行递增运算，使其指向下一个单词的开头，再将该单词的偏移量赋给 int 引用 wordOffset。最后返回 true，指出找到了单词。

这是一个经典的代码范例，理解它的最佳方式是，使用调试器以步进方式执行它。

在这个程序清单中，很多地方都使用了指针算术。从这个程序清单可知，通过将两个指针相减（第 53 行），可以确定它们之间相隔多少个元素。另外，第 55 行对指针执行递增运算使其指向数组中的下一个元素，而不是将其值加 1。使用指针和数组时，经常需要用到指针算术，但这也是一种危险的行为，使用时一定要小心。

13.5 在自由存储区声明数组

可以将整个数组放在自由存储区（也被称为堆）中。为此，可以创建一个数组指针，方法是使用 new 和下标运算符。这样将得到一个指向自由存储区中存储了数组的区域的指针。例如，下面的代码将 Family 声明为一个指向数组第一个元素的指针，该数组能够存储 500 个 Cat 对象。换句话说，Family 指向 Family[0]（地址与 Family[0]相同）。

```
Cat *Family = new Cat[500];
```

以这种方式声明 Family 的优点是，可以使用指针算术来访问 Family 的每个成员，例如，可以编写这样的代码：

```
Cat *Family = new Cat[500];
Cat *pCat = Family;             //pCat points to Family[0]
pCat->SetAge(10);               // set Family[0] to 10
pCat++;                         // advance to Family[1]
pCat->SetAge(20);               // set Family[1] to 20
```

上述代码声明了一个包含 500 个 Cat 对象的新数组，并声明了一个指向该数组开头的指针。然后使用该指针，对第一个 Cat 调用了函数 SetAge( )并将 10 传递给它。接下来对指针执行递增运算，使其指向数组中的下一个 Cat 对象，然后对第二个 Cat 对象调用函数 SetAge( )并将 20 传递给它。

13.5.1 数组指针和指针数组

请看下面 3 个声明：

```
1:  Cat   FamilyOne[500];
2:  Cat * FamilyTwo[500];
3:  Cat * FamilyThree = new Cat[500];
```

FamilyOne 是一个包含 500 个 Cat 对象的数组；FamilyTwo 是一个包含 500 个 Cat 对象的指针的数组；FamilyThree 是一个指针，指向一个包含 500 个 Cat 对象的数组。

这 3 行代码的区别将极大地影响这些数组的使用方式。更令人惊讶的是，FamilyThree 是 FamilyOne 的一个变种，但与 FamilyTwo 有天壤之别。

这提出了一个棘手的问题，指针与数组的关系如何。FamilyThree 是一个数组指针，也就是说，FamilyThree 指向的是数组中第一个元素的地址，FamilyOne 的情况与此相同。

13.5.2　指针和数组名

在 C++中，数组名是一个常量指针，指向数组的第一个元素。因此，在下列声明中：

```
Cat Family[500];
```

Family 是一个指向数组 Family 的第一个元素 Family[0]的指针。

将数组名用作常量指针是合法的，反之亦然。因此，Family+4 是一种访问 Family[4]的合法方式。

当你对指针执行加法、递增、递减运算时，编译器将执行所有的算术运算。Family+4 指向的是 Family 中的第 4 个对象，而不是离 Family 开头 4 字节处。如果每个对象占 4 字节，则 Family+4 指向离数组开头 16 字节处。例如，如果每个对象都是 Cat，而后者包含 4 个 long 变量（每个占 4 字节）和两个 short 变量（每个占 2 字节），即每个 Cat 对象占 20 字节，则 Family+4 指向离数组开头 80 字节处。

程序清单 13.8 演示了如何在自由存储区中声明数组以及使用它。

程序清单 13.8　使用 new 创建数组

```
0:  // Listing 13.8 - An array on the free store
1:
2:  #include <iostream>
3:
4:  class Cat
5:  {
6:    public:
7:      Cat() { itsAge = 1; itsWeight=5; }
8:      ~Cat();
9:      int GetAge() const { return itsAge; }
10:     int GetWeight() const { return itsWeight; }
11:     void SetAge(int age) { itsAge = age; }
12:
13:   private:
14:     int itsAge;
15:     int itsWeight;
16: };
17:
18: Cat :: ~Cat()
19: {
20:   // std::cout << "Destructor called!\n";
21: }
22:
23: int main()
24: {
25:   Cat * Family = new Cat[500];
26:   int i;
27:
28:   for (i = 0; i < 500; i++)
29:   {
```

```
30:       Family[i].SetAge(2*i +1);
31:    }
32:
33:    for (i = 0; i < 500; i++)
34:    {
35:       std::cout << "Cat #" << i+1 << ": ";
36:       std::cout << Family[i].GetAge() << std::endl;
37:    }
38:
39:    delete [] Family;
40:
41:    return 0;
42: }
```

输出：

```
Cat #1: 1
Cat #2: 3
Cat #3: 5
...
Cat #499: 997
Cat #500: 999
```

分析：

第 25 行将 Family 声明为一个指向包含 500 个 Cat 对象的数组的指针，该数组是通过调用 new Cat[500] 在自由存储区中创建的。

从第 30 行可知，可以将索引运算符用于该指针，就像常规数组一样。第 36 行再次使用它来调用方法 GetAge( )。实际上，可以像使用数组名 Family 一样使用该指针。然而，你必须释放创建数组时分配的内存，这是在第 39 行通过调用 delete 来完成的。

13.5.3 删除自由存储区中的数组

当数组被删除时，给 Cat 对象分配的内存将发生什么情况呢？是否会出现内存泄漏？

如果使用 delete[ ]来删除数组 Family，将自动调还为它分配的所有内存。使用方括号时，编译器足够聪明，知道应销毁数组中的每个对象，将其占用的内存归还给自由存储区。

弄清这一点，可将第 25、28 和 33 行的数组大小从 500 改为 10，然后不将第 20 行的输出语句作为注释。当程序运行到第 39 行时数组被销毁，对每个 Cat 对象调用析构函数。

使用 new 在堆上创建对象时，务必使用 delete 来删除它以释放它占用的内存。同样，使用 new <class> [size] 创建数组时，应使用 delete[ ]来删除该数组以释放其占用的所有内存。方括号告诉编译器，要删除整个数组。

如果遗漏了方括号，将只删除数组中的第一个对象。可删除第 39 行的方括号来证明这一点。如果编辑第 20 行，以便显示析构函数被调用，将看到只销毁了一个 Cat 对象。祝贺你！你导致了内存泄漏。

13.5.4 在运行阶段调整数组大小

在堆上创建数组的最大优点是，可以在运行阶段确定数组的大小，然后为其分配内存。例如，如果让用户输入家庭成员的数量，并将其存储到变量 SizeOfFamily 中，便可以像下面这样声明一个 Cat 数组：

```
Cat *pFamily = new Cat[SizeOfFamily];
```

这声明了一个指向 Cat 对象数组的指针。然后可以创建一个指向第一个元素的指针，并使用该指针通过指针算术来遍历给数组：

```
Cat *pCurrentCat = pFamily[0];
for ( int Index = 0; Index < SizeOfFamily; Index++, pCurrentCat++ )
```

```
{
        pCurrentCat->SetAge(Index);
};
```

由于 C++将数组视为特殊的指针，因此可以不创建第二个指针，而直接使用标准数组索引表示法：

```
for (int Index = 0; Index < SizeOfFamily; Index++)
{
        pFamily[Index].SetAge(Index);
};
```

使用包含下标的方括号时，将自动对指针解除引用，编译器将指定合适的指针算术运算。

另一个优点是，在运行阶段当数组空间不够用时，可使用类似的技巧来调整数组的大小。程序清单 13.9 演示了如何为数组重新分配内存。

程序清单 13.9　在运行阶段给数组重新分配内存

```
 0: //Listing 13.9
 1:
 2: #include <iostream>
 3: using namespace std;
 4: int main()
 5: {
 6:    int  AllocationSize = 5;
 7:    int *pArrayOfNumbers = new int[AllocationSize];
 8:    int  ElementsUsedSoFar = 0;
 9:    int  MaximumElementsAllowed = AllocationSize;
10:    int InputNumber = -1;
11:
12:    cout << endl << "Next number = ";
13:    cin >> InputNumber;
14:
15:    while ( InputNumber > 0 )
16:    {
17:       pArrayOfNumbers[ElementsUsedSoFar++] = InputNumber;
18:
19:       if ( ElementsUsedSoFar == MaximumElementsAllowed )
20:       {
21:          int *pLargerArray =
22:             new int[MaximumElementsAllowed+AllocationSize];
23:
24:          for ( int CopyIndex = 0;
25:             CopyIndex < MaximumElementsAllowed;
26:             CopyIndex++ )
27:          {
28:             pLargerArray[CopyIndex] = pArrayOfNumbers[CopyIndex];
29:          };
30:
31:          delete [] pArrayOfNumbers;
32:          pArrayOfNumbers = pLargerArray;
33:          MaximumElementsAllowed+= AllocationSize;
34:       };
35:       cout << endl << "Next number = ";
36:       cin >> InputNumber;
```

```
37:    }
38:
39:    for (int Index = 0; Index < ElementsUsedSoFar; Index++)
40:    {
41:      cout << pArrayOfNumbers[Index] << endl;
42:    }
43:    return 0;
44:  }
```

输出：

```
Next number = 10

Next number = 20

Next number = 30

Next number = 40

Next number = 50

Next number = 60

Next number = 70

Next number = 0
10
20
30
40
50
60
70
```

分析：

在这个例子中，数字依次被输入并存储到一个数组中。用户输入的数字小于或等于 0 后，程序打印收集到的数组中的数字。

仔细阅读后读者将发现，第 6～9 行声明了一系列的变量。具体地说，一开始，第 6 行将数组的大小指定为 5，然后为数组分配内存，并将地址赋给 pArrayOfNumbers。

第 12～13 行从用户那里获取第一个数字，并将其存储到变量 InputNumber 中。第 15 行检查输入的数字是否大于 0，如果是则对其进行处理，否则程序跳到第 38 行执行。

第 17 行将 InputNumber 的值存储到数组中。这是安全的，因为你知道此时数组中还有空间。第 19 行检查数组是否至少还有存储一个元素的空间。如果有，则跳到第 35 行执行；否则执行 if 语句体（第 20～34 行）以增大数组。

第 21 行创建一个新的数组，该数组比当前数组多 5（AllocationSize）个元素。然后，第 24～29 行使用数组表示法（也可以使用指针算术）将原来数组中的元素复制到新数组中。

第 31 行删除原来的数组，第 32 行将指针指向新的数组，第 33 行根据新的最大长度更新 MaximumElementsAllowed。

第 39～42 行显示数组的内容。

应该：

请切记，包含 n 个元素的数组下标为 0 到 n–1。

对于指向数组的指针，一定要使用索引来访问数组元素。

对于在自由存储区中创建的数组，一定要使用 delete[ ]来删除整个数组。只使用 delete 而忽略[ ]将只删除第一个元素。

不应该：

不要在超出数组末尾的地方进行读写。

不要将指针数组和数组指针混为一谈。

别忘了释放使用 new 分配的内存。

13.6　字符数组和字符串

有一种数组需要特别注意，这就是以空字符结尾的字符数组。这种数组被称为“C-风格字符串”。到目前为止，读者见到的惟一的C-风格字符串是在 cout 语句中使用的未命名的C-风格字符串常量，例如：

```
cout << "hello world";
```

可以像其他数组那样声明和初始化C-风格字符串，例如：

```
char Greeting[] =
{'H','e','l','l','o',' ','W','o','r','l','d','\0' };
```

在这个例子中，Greeting 被声明为一个字符数组，并使用一系列字符对其进行了初始化。最后一个字符'\0'是空字符，很多 C++函数都将其视为C-风格字符串的结束标记。虽然这种逐个字符初始化的方法可行，但输入时太麻烦且容易出错。C++提供了一种简便的方法来代替前面的代码，这就是：

```
char Greeting[] = "Hello World";
```

有关这种语法有两点需要注意：

- 使用双引号将C-风格字符串括起，没有逗号和大括号；而不是用由逗号将用单引号括起的字符分开，并用大括号将它们括起。
- 不必添加空字符，因为编译器会自动添加。

声明字符串时，应确保其大小能够满足需要。C-风格字符串的长度包括末尾的空字符在内。例如，字符串“Hello World”为 12 个字节，其中Hello 为 5 字节，空格为 1 字节，World 为 5 字节，空字符为 1 字节。

也可以创建没有被初始化的字符数组。像所有数组一样，必须确保存入的数组不超过其空间。程序清单 13.10 演示了如何使用未初始化的字符数组。

程序清单 13.10　填充数组

```
 0:  //Listing 13.10 char array buffers
 1:
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:      char buffer[80];
 7:      std::cout << "Enter the string: ";
 8:      std::cin >> buffer;
 9:      std::cout << "Here's the buffer:" << buffer << std::endl;
10:      return 0;
11:  }
```

输出：

```
Enter the string: Hello World
```

```
Here's the buffer: Hello
```

分析：

第 6 行声明了一个字符数组，以充当一个能存储 80 个字符的缓冲区。该数组最多可存储一个包含 79 个字符的 C-风格字符串和一个结束的空字符。

第 7 行提示用户输入一个 C-风格字符串，第 8 行将其存储到缓冲区中。cin 将字符串写入缓冲区后加上一个结束的空字符。

程序清单 13.10 中的程序存在两个问题。首先，如果用户输入的字符多于 79 个，cin 将在超出缓冲区末尾的地方写入；其次，如果用户输入了空格，cin 将认为这是字符串的结尾，从而停止向缓冲区写入。

为解决这些问题，必须对 cin 调用一个特殊的方法 get( )。cin.get( )接受 3 个参数：

- 待填充的缓冲区；
- 要读取的最大字符数；
- 终止输入的限定符。

默认限定符为换行符。程序清单 13.11 演示了 get( )的用法。

程序清单 13.11　填充数组

```
0:  //Listing 13.11 using cin.get()
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  int main()
6:  {
7:      char buffer[80];
8:      cout << "Enter the string: ";
9:      cin.get(buffer, 79);      // get up to 79 or newline
10:     cout << "Here's the buffer:" << buffer << endl;
11:     return 0;
12: }
```

输出：

```
Enter the string: Hello World
Here's the buffer: Hello World
```

分析：

第 9 行调用 cin 的 get( )方法。第 7 行声明的缓冲区被作为第一个参数传递给它，第二个参数是要获取的最大字符数。在这个例子中，它不能超过 79，这样才有空间存储结尾的空字符。由于默认值换行符足够了，因此不必提供结束字符。

如果用户输入了空格、制表符或其他空白字符，它们将被赋给字符串。换行符结束输入。输入 79 个字符后，也将结束输入。读者可以再次运行该程序清单，并输入一个超过 79 个字符的字符串来验证这一点。

13.7　使用方法 strcpy( )和 strncpy( )

C++库中有很多可用于处理 C-风格字符串的函数，其中的很多是从 C 语言那里继承而来的。在提供的很多函数中，有两个用于将一个字符串复制到另一个字符串中：strcpy( )和 strncpy( )。strcpy( )将整个字符串复制到指定的缓冲区中；strncpy( )将指定数目的字符从一个字符串复制到另一个字符串中。程序清单 13.12 演示了 strcpy( )的用法。

程序清单 13.12　使用 strcpy( )

```
0:  //Listing 13.12 Using strcpy()
1:
2:  #include <iostream>
3:  #include <string.h>
4:  using namespace std;
5:
6:  int main()
7:  {
8:     char String1[] = "No man is an island";
9:     char String2[80];
10:
11:    strcpy(String2,String1);
12:
13:    cout << "String1: " << String1 << endl;
14:    cout << "String2: " << String2 << endl;
15:    return 0;
16: }
```

输出：

```
String1: No man is an island
String2: No man is an island
```

分析：

这个程序清单比较简单，它将一个字符串中的数据复制到另一个字符串中。第 3 行包含了头文件 string.h，该文件包含函数 strcpy( )的原型。函数 strcpy( )接受两个字符数组，前者为源数组，后者为目标数组。第 11 行使用该函数将 String1 复制到 String2 中。

使用函数 strcpy( )时一定要小心。如果源数组比目标数组长，strcpy( )将覆盖缓冲区后面的内容。为防止这种情况发生，标准库还提供了 strncpy( )。该变体接受最多要复制的字符数作为参数。函数 strncpy( )将第一个空字符前的内容或指定的最大字符数复制到目标缓冲区中。程序清单 13.13 演示了 strncpy( )的用法。

程序清单 13.13　使用 strncpy( )

```
0:  //Listing 13.13 Using strncpy()
1:
2:  #include <iostream>
3:  #include <string.h>
4:
5:  int main()
6:  {
7:     const int MaxLength = 80;
8:     char String1[] = "No man is an island";
9:     char String2[MaxLength+1];
10:
11:    strncpy(String2,String1,MaxLength);
12:
13:    std::cout << "String1: " << String1 << std::endl;
14:    std::cout << "String2: " << String2 << std::endl;
15:    return 0;
16: }
```

输出：

```
String1: No man is an island
String2: No man is an island
```

分析：

同样，这个程序清单也很简单。和前一个程序清单一样，这里也只是将数据从一个字符串复制到另一个字符串中。在第 11 行调用了 strncpy( )而不是 strcpy( )，它接受第三个参数：要复制的最大字符数。缓冲区 String2 被声明为能存储 MaxLength+1 个字符；多出一个字符用于存储空字符，strcpy( )和 strncpy( )都会在字符串末尾自动添加一个空字符。

注意：和程序清单 13.9 中的 int 数组一样，也可以采用堆分配技术和逐元素复制的方法来调整字符数组的大小。提供给 C++程序员的最灵活的 string 类使用这种技术的变体来支持增大/缩小字符串以及在字符串中间插入或删除元素。

13.8 String 类

C++继承了 C 语言中以空字符结尾的 C-风格字符串以及包括 strcpy( )函数的函数库。但这些函数并没有集成到面向对象的框架中。标准库中包括一个 String 类，它提供了一套封装好的数据以及处理这些数据的函数，还提供了存取器函数以便对 String 类的客户隐蔽数据本身。

使用这个类之前先创建一个自定义的 String 类，让读者理解其中涉及的问题。该 String 类至少应克服字符数组的基本限制。

像所有数组一样，字符数组也是静态的。定义它们的大小后，即便根本不使用，它们也总是占据相应数量的内存。在超出数组末尾进行写操作将导致灾难性后果。

优秀的 String 类只分配所需的内存，且总是足以存储加入的东西。如果不能分配足够的内存，它将能够妥善地处理这种故障。

程序清单 13.14 是 String 类的近似版本。

注意：这个自定义的 String 类功能有限，根本谈不上完备或健壮，也不能用于商用目的。然而，标准库提供了一个完整的、健壮的 String 类。

程序清单 13.14　自定义的 String 类

```
 0: //Listing 13.14 Using a String class
 1:
 2: #include <iostream>
 3: #include <string.h>
 4: using namespace std;
 5:
 6: // Rudimentary string class
 7: class String
 8: {
 9:   public:
10:     // constructors
11:     String();
12:     String(const char *const);
13:     String(const String &);
14:     ~String();
15:
```

```
16:     // overloaded operators
17:     char & operator[](unsigned short offset);
18:     char operator[](unsigned short offset) const;
19:     String operator+(const String&);
20:     void operator+=(const String&);
21:     String & operator= (const String &);
22:
23:     // General accessors
24:     unsigned short GetLen()const { return itsLen; }
25:     const char * GetString() const { return itsString; }
26:
27:   private:
28:     String (unsigned short);        // private constructor
29:     char * itsString;
30:     unsigned short itsLen;
31: };
32:
33: // default constructor creates string of 0 bytes
34: String::String()
35: {
36:    itsString = new char[1];
37:    itsString[0] = '\0';
38:    itsLen=0;
39: }
40:
41: // private (helper) constructor, used only by
42: // class methods for creating a new string of
43: // required size. Null filled.
44: String::String(unsigned short len)
45: {
46:    itsString = new char[len+1];
47:    for (unsigned short i = 0; i<=len; i++)
48:      itsString[i] = '\0';
49:    itsLen=len;
50: }
51:
52: // Converts a character array to a String
53: String::String(const char * const cString)
54: {
55:    itsLen = strlen(cString);
56:    itsString = new char[itsLen+1];
57:    for (unsigned short i = 0; i<itsLen; i++)
58:      itsString[i] = cString[i];
59:    itsString[itsLen]='\0';
60: }
61:
62: // copy constructor
63: String::String (const String & rhs)
64: {
65:    itsLen=rhs.GetLen();
66:    itsString = new char[itsLen+1];
67:    for (unsigned short i = 0; i<itsLen;i++)
```

```
68:       itsString[i] = rhs[i];
69:     itsString[itsLen] = '\0';
70:   }
71:
72:   // destructor, frees allocated memory
73:   String::~String ()
74:   {
75:     delete [] itsString;
76:     itsLen = 0;
77:   }
78:
79:   // operator equals, frees existing memory
80:   // then copies string and size
81:   String& String::operator=(const String & rhs)
82:   {
83:     if (this == &rhs)
84:       return *this;
85:     delete [] itsString;
86:     itsLen=rhs.GetLen();
87:     itsString = new char[itsLen+1];
88:     for (unsigned short i = 0; i<itsLen;i++)
89:       itsString[i] = rhs[i];
90:     itsString[itsLen] = '\0';
91:     return *this;
92:   }
93:
94:   //nonconstant offset operator, returns
95:   // reference to character so it can be
96:   // changed!
97:   char & String::operator[](unsigned short offset)
98:   {
99:     if (offset > itsLen)
100:      return itsString[itsLen-1];
101:    else
102:      return itsString[offset];
103:  }
104:
105:  // constant offset operator for use
106:  // on const objects (see copy constructor!)
107:  char String::operator[](unsigned short offset) const
108:  {
109:    if (offset > itsLen)
110:      return itsString[itsLen-1];
111:    else
112:      return itsString[offset];
113:  }
114:
115:  // creates a new string by adding current
116:  // string to rhs
117:  String String::operator+(const String& rhs)
118:  {
119:    unsigned short  totalLen = itsLen + rhs.GetLen();
```

```
120:   String temp(totalLen);
121:   unsigned short i;
122:   for ( i= 0; i<itsLen; i++)
123:      temp[i] = itsString[i];
124:   for (unsigned short j   0; j<rhs.GetLen(); j++, i++)
125:      temp[i] = rhs[j];
126:   temp[totalLen] = '\0';
127:   return temp;
128: }
129:
130: // changes current string, returns nothing
131: void String::operator+=(const String& rhs)
132: {
133:   unsigned short rhsLen = rhs.GetLen();
134:   unsigned short totalLen = itsLen + rhsLen;
135:   String temp(totalLen);
136:   unsigned short i;
137:   for (i = 0; i<itsLen; i++)
138:      temp[i] = itsString[i];
139:   for (unsigned short j = 0; j<rhs.GetLen(); j++, i++)
140:      temp[i] = rhs[i-itsLen];
141:   temp[totalLen] = '\0';
142:   *this = temp;
143: }
144:
145: int main()
146: {
147:    String s1("initial test");
148:    cout << "S1:\t" << s1.GetString() << endl;
149:
150:    char * temp = "Hello World";
151:    s1 = temp;
152:    cout << "S1:\t" << s1.GetString() << endl;
153:
154:    char tempTwo[20];
155:    strcpy(tempTwo," nice to be here!");
156:    s1 += tempTwo;
157:    cout << "tempTwo:\t" << tempTwo << endl;
158:    cout << "S1:\t" << s1.GetString() << endl;
159:
160:    cout << "S1[4]:\t" << s1[4] << endl;
161:    s1[4] = 'x';
162:    cout << "S1:\t" << s1.GetString() << endl;
163:
164:    cout << "S1[999]:\t" << s1[999] << endl;
165:
166:    String s2(" Another string");
167:    String s3;
168:    s3 = s1+s2;
169:    cout << "S3:\t" << s3.GetString() << endl;
170:
171:    String s4;
```

```
172:    s4 = "Why does this work?";
173:    cout << "S4:\t" << s4.GetString() << endl;
174:    return 0;
175: }
```

输出：

```
S1:    initial test
S1:    Hello World
tempTwo:       ; nice to be here!
S1:    Hello World; nice to be here!
S1[4]: o
S1:    Hellx World; nice to be here!
S1[999]:       '
S3:    Hellx World; nice to be here! Another string
S4:    Why does this work?
```

分析：

第 7～31 行声明了 String 类。为增加灵活性，第 11～13 行包含 3 个构造函数：默认构造函数、复制构造函数和一个将以空字符结束（C-风格）的字符串作为参数的构造函数。

为让用户轻松地操作字符串，这个 String 类重载了多个运算符，其中包括下标运算符（[]）、加法运算符（+）和加等于运算符（+=）。下标运算符被重载了两次：一次为返回 char 的常量函数，另一次为返回 char 引用的非常量函数。

非常量版本用于下面这样的语句（第 161 行）中：

```
S1[4] = 'x';
```

这让用户可以直接访问字符串中的每个字符。返回字符的引用让调用函数能够操纵它。

访问常量 String 对象时使用常量函数，如从第 63 行开始的复制构造函数的实现。注意，这里访问的是 rhs[i]，而 rhs 被声明为 const String &，因此使用非常量成员函数来访问它是非法的。因此，下标运算符必须被重载为常量函数。

如果返回的对象很大，可能需要将返回值声明为常引用。但由于 char 变量仅占 1 字节，因此这么做没有意义。

默认构造函数的实现位于第 34～39 行。它创建一个长度为 0 的字符串。这个 String 类采用的约定是，报告其长度时不考虑结尾的空字符。这个默认字符串仅包含一个结尾的空字符。

复制构造函数的实现位于第 63～70 行。该构造函数将新字符串的长度设置为已有字符串的长度再加 1，用于存储结尾的空字符。它将已有字符串中的每个字符复制到新字符串中，然后以空字符结尾。别忘了，不同于赋值运算符，复制构造函数无需检查被复制的字符串是否是新对象本身，因为这种情况不可能发生。

第 53～60 行是接受一个 C-风格字符串作为参数的构造函数的实现。该构造函数与复制构造函数相似。C-风格字符串的长度是通过调用标准 String 库函数 strlen()来确定的。

第 28 行将另一个构造函数 String（unsigned short）声明为私有成员函数。这个类的设计者不允许客户类创建任意长度的 String。提供这个构造函数，只是为了必要时在内部创建 String，例如第 131 行的 operator+=。后面讨论 operator +=时将对此做深入介绍。

构造函数 String（unsigned short）用空字符（'\0'）填充其数组的每个成员。因此 for 循环检查的是 i<=len 而不是 i<len。

第 73～77 行实现的析构函数删除由 String 类维护的字符串。调用运算符 delete 时一定要包括方括号，以便删除数组的每个成员而不仅是第一个元素。

赋值运算符是在第 81～92 行重载的。它首先检查赋值语句的左边和右边是否相同。如果不是，则删除当前字符串，创建新的字符串并将其复制到正确位置。返回引用有助于下面这样的连续赋值：

```
String1 = String2 = String3;
```

另一个被重载的运算符是下标运算符。它被重载了两次，第一次位于第 97～103 行，第二次位于 107～113 行。两次都执行了初步边界检查，如果用户试图访问超越数组末尾的字符，将返回最后一个（第 len-1 个）字符。

第 117～127 行将运算符（+）重载为拼接运算符，以便能够编写这样的代码：

```
String3 = String1 + String2;
```

这样，String3 将是另外两个字符串的拼接结果。为此，相加运算符（+）函数计算两个字符串合并后的长度，并创建了一个临时字符串 temp。这是调用私有构造函数来实现的，该构造函数接受一个 int 参量，并创建一个用空字符填充的字符串。然后，用两个字符串中的内容代替这些空字符。先复制左边的字符串（*this），然后复制右边的字符串（rhs）。第一个 for 循环遍历左边的字符串并将每个字符添加到新创建的字符串中；第二个 for 循环遍历右边的字符串。

在第 127 行，相加运算符（+）按值返回 temp 字符串，后者被赋给赋值运算符左边的字符串（string1）。在第 131～143 行，operator+=处理已有的字符串，即语句 string1 += string2 的左边。除第 142 行将临时字符串 temp 赋给当前字符串（*this = temp）外，它和相加运算符的工作原理相同。

main()函数（第 145～175 行）用于测试这个类。第 147 行使用将以空字符结尾的 C-风格字符串作为参数的构造函数，创建了一个 String 对象。第 148 行使用存取器函数 GetString()来打印它的内容。第 150 行创建另外一个 C-风格字符串，第 151 行将其赋给原来的字符串 s1。第 152 行打印这种复制的结果，以表明对赋值运算符的重载确实可行。

第 154 行创建了第三个 C-风格字符串 tempTwo。第 155 行调用 strcpy()，用字符串“; nice to be here!”来填充该缓冲区。第 156 行调用重载的 operator+=，将 tempTwo 与字符串 s1 合并起来。第 158 行打印合并结果。

第 160 行使用重载的下标运算符来访问和打印 s1 中的第 5 个字符。第 161 行将新值（x）赋给字符串中的这个字符，这调用了非常量下标运算符（[]）。第 162 行打印结果，以表明实际值确实被修改了。

第 164 行试图访问超出数组末尾的字符。从打印出的信息可知，按设计的那样返回了数组的最后一个字符。

第 166 和 167 行又创建了两个 String 对象，第 168 行调用加法运算符，第 169 行打印结果。

第 171 行创建了一个新的 String 对象 s4。第 172 行调用重载的赋值运算符，将一个字面 C-风格字符串赋给 s4。第 173 行打印结果。读者可能会想：第 21 行将赋值运算符定义为接受一个常量 String 引用，而在这里传递的是一个 C-风格字符串，为什么这样做是合法的？”

答案是这样的：编译器期望一个 String，但提供的是一个字符数组。因此，它检查是否它可以根据提供的字符数组创建一个 String。第 12 行声明了一个可根据字符数组创建 String 的构造函数，因此编译器根据提供的字符数组创建一个临时 String，并将其传递给赋值运算符。这被称为隐式强制转换或升格（promotion）。如果没有声明并实现接受字符数组作为参数的构造函数，这种赋值将导致编译错误。

仔细阅读程序清单 13.14 将发现，你设计的 String 类已经非常健壮了。你还将认识到，要实现完整的功能，代码将比该程序清单长得多。幸运的是，C++标准库提供了一个更健壮的 String 类，你可以通过包含<string>库来使用它。

13.9　链表和其他结构

数组很像家用塑料制品。它们是很不错的容器，但大小是固定的。如果选择的容器太大，将浪费存储空间；如果选择的容器太小，里面的东西将溢出，带来大麻烦。

程序清单 13.9 提供了一种解决这种问题的方法。然而，当你使用大型数组或需要移动、删除或插入数组元素时，分配和释放内存的开销可能非常高。

另一种解决方法是使用链表。链表是一种数据结构，由小型容器组成，这些容器被设计成必要时能够链

接起来。其思想是，编写一个存储数据对象的类（如 Cat 或 Rectangle），它能够指向下一个容器。为需要存储的每个对象创建一个容器，然后在需要时将它们链接起来。

链表被视为高级主题，更详细的信息请参阅附录 E。

13.10 创建数组类

相比于使用内置数组，编写自己的数组类有很多优点。对于初学者来说，可以防止数组泛滥成灾。你也许还应考虑让你的数组类能动态调整大小：创建时可能只有一个成员，但在程序执行过程中根据需要不断扩大。

你可能想对数组成员进行排序。你可能需要下面这些功能强大的数组变体：

- **有序集合**：所有成员按顺序排列。
- **集（set）**：每个成员最多出现一次。
- **词典**：使用成对值，其中一个值充当关键字，用于检索另一个值。
- **稀疏数组**：下标的取值范围很大，但只有被添加到数组中的值才占用内存。你可以声明 SparseArray[5] 或 SparseArray[200]，但可能只为几个元素分配了内存。
- **布袋**：无序集合，按随机顺序添加和检索。

通过重载下标运算符（[]），可将链表变成有序集合。通过排除重复内容，可将集合变成一个集（set）。如果链表中的每个对象都有一对匹配的值，可使用链表来构建词典或稀疏数组。

注意：与使用内置的类相比，编写自己的类有很多优点；而使用标准库中类似类的实现通常优于编写自己的类。

13.11 小 结

本章介绍如何创建数组。数组是大小固定的相同类型的对象集合。

数组执行边界检查。因此，在超出数组末尾的地方进行读写是合法的，虽然其后果是灾难性的。数组索引从 0 开始。一种常见的错误是，将下标 *n* 用于包含 *n* 个元素的数组。

数组可以是一维或多维的。无论是一维还是多维的，只要数组包含的元素为内置类型（如 int）或有默认构造函数的类，就可以被初始化。

数组及其元素可以位于自由存储区中，也可以位于堆栈中。删除自由存储区中的数组时，别忘了在调用 delete 时加上方括号。

数组名是指向数组第一个元素的常量指针。可以对指针和数组使用指针算术来找到数组中的下一个元素。

字符串是字符数组。C++为管理字符数组提供了专用的功能，其中包括使用引号括起字符串来初始化它们。

13.12 问 与 答

问：未被初始化的数组元素将包含什么内容？

答：分配内存时包含在内存中的内容。使用未初始化的数组元素的结果是不可预测的。如果编译器遵循了 C++标准，静态的非局部数组元素将被初始化为零。

问：可以合并数组吗？

答：可以。对于简单数组，可使用指针将它们合并成一个更大的新数组；对于字符串，可使用一些内置

函数（如 strcat）来合并。

问：如果数组管用为什么还要创建链表呢？

答：数组的大小是固定的，而链表可在运行阶段动态地调整大小。附录 E 提供了有关创建链表的更详细信息。

问：既然可以创建更好的数组类，为什么还要使用内置的数组呢？

答：内置的数组速度快且易于使用。创建更好的数组类时通常需要用到内置的数组。

问：有比数组更好的结构吗？

答：第 19 章将介绍模板和标准模板库。这个库中包含数组模板，它们提供了你通常需要的所有功能。与创建自己的模板相比，使用这些模板更安全。

问：字符串类一定要用 char* 来存储字符串的内容吗？

答：不一定。可以使用设计人员认为最合适的方式。

13.13　作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

13.13.1　测验

1. SomeArray[25]中第一个元素和最后一个元素分别是什么？
2. 如何声明多维数组？
3. 初始化数组 SomeArray[2][3][2]的元素。
4. 数组 SomeArray[10][5][20]包含多少个元素？
5. 链表和数组之间的区别何在？
6. 字符串“Jesse knows C++”中存储了多少个字符？
7. 字符串“Brad is a nice guy”的最后一个字符是什么？

13.13.2　练习

1. 声明一个二维数组来表示井字游戏棋盘。
2. 编写将练习 1 中声明的数组中所有元素都初始化为 0 的代码。
3. 编写一个使用了 4 个数组的程序，其中的 3 个数组分别存储了你的姓、中名和名。使用本章介绍的字符串函数将这些字符串复制到存储全名的第 4 个数组中。
4. 查错：下面的代码段有什么错误？

```
unsigned short SomeArray[5][4];
for (int i = 0; i<4; i++)
   for (int j = 0; j<5; j++)
      SomeArray[i][j] = i+j;
```

5. 查错：下面的代码段有什么错误？

```
unsigned short SomeArray[5][4];
for (int i = 0; i<=5; i++)
   for (int j = 0; j<=4; j++)
      SomeArray[i][j] = 0;
```