3.5.4 C 风格字符串

尽管 C++支持 C 风格字符串，但在 C++程序中最好还是不要使用它们。这是因为 C 风格字符串不仅使用起来不太方便，而且极易引发程序漏洞，是诸多安全问题的根本原因。

122

字符串字面值是一种通用结构的实例，这种结构即是 C++由 C 继承而来的 **C 风格字符串**（C-style character string）。C 风格字符串不是一种类型，而是为了表达和使用字符串而形成的一种约定俗成的写法。按此习惯书写的字符串存放在字符数组中并以**空字符结束**（null terminated）。以空字符结束的意思是在字符串最后一个字符后面跟着一个空字符（'\0'）。一般利用指针来操作这些字符串。

C 标准库 String 函数

表 3.8 列举了 C 语言标准库提供的一组函数，这些函数可用于操作 C 风格字符串，它们定义在 cstring 头文件中，cstring 是 C 语言头文件 string.h 的 C++版本。

表 3.8：C 风格字符串的函数

strlen(p)	返回 p 的长度，空字符不计算在内
strcmp(p1, p2)	比较 p1 和 p2 的相等性。如果 p1==p2，返回 0；如果 p1>p2，返回一个正值；如果 p1<p2，返回一个负值
strcat(p1, p2)	将 p2 附加到 p1 之后，返回 p1
strcpy(p1, p2)	将 p2 拷贝给 p1，返回 p1

表 3.8 所列的函数不负责验证其字符串参数。

传入此类函数的指针必须指向以空字符作为结束的数组：

```
char ca[] = {'C', '+', '+'};       // 不以空字符结束
cout << strlen(ca) << endl;        // 严重错误：ca 没有以空字符结束
```

此例中，ca 虽然也是一个字符数组但它不是以空字符作为结束的，因此上述程序将产生未定义的结果。strlen 函数将有可能沿着 ca 在内存中的位置不断向前寻找，直到遇到空字符才停下来。

比较字符串

比较两个C风格字符串的方法和之前学习过的比较标准库 string 对象的方法大相径庭。比较标准库 string 对象的时候，用的是普通的关系运算符和相等性运算符：

```
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2) // false: s2 小于 s1
```

如果把这些运算符用在两个 C 风格字符串上，实际比较的将是指针而非字符串本身：

123

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2)   // 未定义的：试图比较两个无关地址
```

谨记之前介绍过的，当使用数组的时候其实真正用的是指向数组首元素的指针（参见 3.5.3 节，第 105 页）。因此，上面的 if 条件实际上比较的是两个 const char*的值。这两个

指针指向的并非同一对象，所以将得到未定义的结果。

要想比较两个 C 风格字符串需要调用 strcmp 函数，此时比较的就不再是指针了。如果两个字符串相等，strcmp 返回 0；如果前面的字符串较大，返回正值；如果后面的字符串较大，返回负值：

```
if (strcmp(ca1, ca2) < 0) // 和两个 string 对象的比较 s1 < s2 效果一样
```

目标字符串的大小由调用者指定

连接或拷贝 C 风格字符串也与标准库 string 对象的同类操作差别很大。例如，要想把刚刚定义的那两个 string 对象 s1 和 s2 连接起来，可以直接写成下面的形式：

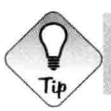
```
// 将 largeStr 初始化成 s1、一个空格和 s2 的连接
string largeStr = s1 + " " + s2;
```

同样的操作如果放到 ca1 和 ca2 这两个数组身上就会产生错误了。表达式 ca1 + ca2 试图将两个指针相加，显然这样的操作没什么意义，也肯定是非法的。

正确的方法是使用 strcat 函数和 strcpy 函数。不过要想使用这两个函数，还必须提供一个用于存放结果字符串的数组，该数组必须足够大以便容纳下结果字符串及末尾的空字符。下面的代码虽然很常见，但是充满了安全风险，极易引发严重错误：

```
// 如果我们计算错了 largeStr 的大小将引发严重错误
strcpy(largeStr, ca1);      // 把 ca1 拷贝给 largeStr
strcat(largeStr, " ");      // 在 largeStr 的末尾加上一个空格
strcat(largeStr, ca2);      // 把 ca2 连接到 largeStr 后面
```

一个潜在的问题是，我们在估算 largeStr 所需的空间时不容易估准，而且 largeStr 所存的内容一旦改变，就必须重新检查其空间是否足够。不幸的是，这样的代码到处都是，程序员根本没法照顾周全。这类代码充满了风险而且经常导致严重的安全泄漏。



对大多数应用来说，使用标准库 string 要比使用 C 风格字符串更安全、更高效。

124

3.5.4 节练习

练习 3.37：下面的程序是何含义，程序的输出结果是什么？

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

练习 3.38：在本节中我们提到，将两个指针相加不但是非法的，而且也没什么意义。请问为什么两个指针相加没什么意义？

练习 3.39：编写一段程序，比较两个 string 对象。再编写一段程序，比较两个 C 风格字符串的内容。

练习 3.40：编写一段程序，定义两个字符数组并用字符串字面值初始化它们；接着再定义一个字符数组存放前两个数组连接后的结果。使用 strcpy 和 strcat 把前两个数组的内容拷贝到第三个数组中。

3.5.5 与旧代码的接口

很多C++程序在标准库出现之前就已经写成了，它们肯定没用到 string 和 vector 类型。而且，有一些C++程序实际上是与C语言或其他语言的接口程序，当然也无法使用C++标准库。因此，现代的C++程序不得不与那些充满了数组和/或C风格字符串的代码衔接，为了使这一工作简单易行，C++专门提供了一组功能。

混用 string 对象和 C 风格字符串

3.2.1 节（第 76 页）介绍过允许使用字符串字面值来初始化 string 对象：

```
string s("Hello World"); // s 的内容是 Hello World
```

更一般的情况是，任何出现字符串字面值的地方都可以用以空字符结束的字符数组来替代：

- 允许使用以空字符结束的字符数组来初始化 string 对象或为 string 对象赋值。
- 在 string 对象的加法运算中允许使用以空字符结束的字符数组作为其中一个运算对象（不能两个运算对象都是）；在 string 对象的复合赋值运算中允许使用以空字符结束的字符数组作为右侧的运算对象。

上述性质反过来就不成立了：如果程序的某处需要一个 C 风格字符串，无法直接用 string 对象来代替它。例如，不能用 string 对象直接初始化指向字符的指针。为了完成该功能，string 专门提供了一个名为 c_str 的成员函数：

```
char *str = s; // 错误：不能用 string 对象初始化 char*
const char *str = s.c_str(); // 正确
```

顾名思义，c_str 函数的返回值是一个 C 风格的字符串。也就是说，函数的返回结果是一个指针，该指针指向一个以空字符结束的字符数组，而这个数组所存的数据恰好与那个 string 对象的一样。结果指针的类型是 const char*，从而确保我们不会改变字符数组的内容。 125

我们无法保证 c_str 函数返回的数组一直有效，事实上，如果后续的操作改变了 s 的值就可能让之前返回的数组失去效用。

如果执行完 c_str()函数后程序想一直都能使用其返回的数组，最好将该数组重新拷贝一份。

使用数组初始化 vector 对象

3.5.1 节（第 102 页）介绍过不允许使用一个数组为另一个内置类型的数组赋初值，也不允许使用 vector 对象初始化数组。相反的，允许使用数组来初始化 vector 对象。要实现这一目的，只需指明要拷贝区域的首元素地址和尾后地址就可以了：

```
int int_arr[] = {0, 1, 2, 3, 4, 5};
// ivec 有 6 个元素，分别是 int_arr 中对应元素的副本
vector<int> ivec(begin(int_arr), end(int_arr));
```

在上述代码中，用于创建 ivec 的两个指针实际上指明了用来初始化的值在数组 int_arr 中的位置，其中第二个指针应指向待拷贝区域尾元素的下一位置。此例中，使用标准库函数 begin 和 end（参见 3.5.3 节，第 106 页）来分别计算 int_arr 的首指针和尾后指针。在最终的结果中，ivec 将包含 6 个元素，它们的次序和值都与数组 int_arr 完全

一样。

用于初始化 vector 对象的值也可能仅是数组的一部分：

```
// 拷贝三个元素：int_arr[1]、int_arr[2]、int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
```

这条初始化语句用 3 个元素创建了对象 subVec，3 个元素的值分别来自 int_arr[1]、int_arr[2]和 int_arr[3]。

建议：尽量使用标准库类型而非数组

使用指针和数组很容易出错。一部分原因是概念上的问题：指针常用于底层操作，因此容易引发一些与烦琐细节有关的错误。其他问题则源于语法错误，特别是声明指针时的语法错误。

现代的 C++程序应当尽量使用 vector 和迭代器，避免使用内置数组和指针；应该尽量使用 string，避免使用 C 风格的基于数组的字符串。

3.5.5 节练习

练习 3.41：编写一段程序，用整型数组初始化一个 vector 对象。

练习 3.42：编写一段程序，将含有整数元素的 vector 对象拷贝给一个整型数组。

3.6 多维数组

严格来说，C++语言中没有多维数组，通常所说的多维数组其实是数组的数组。谨记
126 这一点，对今后理解和使用多维数组大有益处。

当一个数组的元素仍然是数组时，通常使用两个维度来定义它：一个维度表示数组本身大小，另外一个维度表示其元素（也是数组）大小：

```
int ia[3][4]; // 大小为 3 的数组，每个元素是含有 4 个整数的数组
// 大小为 10 的数组，它的每个元素都是大小为 20 的数组，
// 这些数组的元素是含有 30 个整数的数组
int arr[10][20][30] = {0}; // 将所有元素初始化为 0
```

如 3.5.1 节（第 103 页）所介绍的，按照由内而外的顺序阅读此类定义有助于更好地理解其真实含义。在第一条语句中，我们定义的名字是 ia，显然 ia 是一个含有 3 个元素的数组。接着观察右边发现，ia 的元素也有自己的维度，所以 ia 的元素本身又都是含有 4 个元素的数组。再观察左边知道，真正存储的元素是整数。因此最后可以明确第一条语句的含义：它定义了一个大小为 3 的数组，该数组的每个元素都是含有 4 个整数的数组。

使用同样的方式理解 arr 的定义。首先 arr 是一个大小为 10 的数组，它的每个元素都是大小为 20 的数组，这些数组的元素又都是含有 30 个整数的数组。实际上，定义数组时对下标运算符的数量并没有限制，因此只要愿意就可以定义这样一个数组：它的元素还是数组，下一级数组的元素还是数组，再下一级数组的元素还是数组，以此类推。

对于二维数组来说，常把第一个维度称作行，第二个维度称作列。