82 第2章介绍的内置类型是由C++语言直接定义的。这些类型，比如数字和字符，体现了大多数计算机硬件本身具备的能力。标准库定义了另外一组具有更高级性质的类型，它们尚未直接实现到计算机硬件中。

本章将介绍两种最重要的标准库类型：string和vector。string表示可变长的字符序列，vector存放的是某种给定类型对象的可变长序列。本章还将介绍内置数组类型，和其他内置类型一样，数组的实现与硬件密切相关。因此相较于标准库类型string和vector，数组在灵活性上稍显不足。

在开始介绍标准库类型之前，先来学习一种访问库中名字的简单方法。

3.1 命名空间的using声明

目前为止，我们用到的库函数基本上都属于命名空间std，而程序也显式地将这一点标示了出来。例如，std::cin表示从标准输入中读取内容。此处使用作用域操作符（::）（参见1.2节，第7页）的含义是：编译器应从操作符左侧名字所示的作用域中寻找右侧那个名字。因此，std::cin的意思就是要使用命名空间std中的名字cin。

上面的方法显得比较烦琐，然而幸运的是，通过更简单的途径也能使用到命名空间中的成员。本节将学习其中一种最安全的方法，也就是使用**using声明**（using declaration），18.2.2节（第702页）会介绍另一种方法。

有了using声明就无须专门的前缀（形如命名空间::）也能使用所需的名字了。using声明具有如下的形式：

```
using namespace::name;
```

一旦声明了上述语句，就可以直接访问命名空间中的名字：

```
#include <iostream>
// using声明，当我们使用名字cin时，从命名空间std中获取它
using std::cin;

int main()
{
    int i;
    cin >> i;          // 正确：cin和std::cin含义相同
    cout << i;         // 错误：没有对应的using声明，必须使用完整的名字
    std::cout << i;    // 正确：显式地从std中使用cout
    return 0;
}
```

每个名字都需要独立的using声明

按照规定，每个using声明引入命名空间中的一个成员。例如，可以把要用到的标准库中的名字都以using声明的形式表示出来，重写1.2节（第5页）的程序如下：

83

```
#include <iostream>
// 通过下列using声明，我们可以使用标准库中的名字
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
```

```
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

在上述程序中，一开始就有对 cin、cout 和 endl 的 using 声明，这意味着我们不用再添加 std::形式的前缀就能直接使用它们。C++语言的形式比较自由，因此既可以一行只放一条 using 声明语句，也可以一行放上多条。不过要注意，用到的每个名字都必须有自己的声明语句，而且每句话都得以分号结束。

头文件不应包含 using 声明

位于头文件的代码（参见 2.6.3 节，第 67 页）一般来说不应该使用 using 声明。这是因为头文件的内容会拷贝到所有引用它的文件中去，如果头文件里有某个 using 声明，那么每个使用了该头文件的文件就都会有这个声明。对于某些程序来说，由于不经意间包含了一些名字，反而可能产生始料未及的名字冲突。

一点注意事项

经本节所述，后面的所有例子将假设，但凡用到的标准库中的名字都已经使用 using 语句声明过了。例如，我们将在代码中直接使用 cin，而不再使用 std::cin。

为了让书中的代码尽量简洁，今后将不会再把所有 using 声明和#include 指令一一标出。附录 A 中的表 A.1（第 766 页）列出了本书涉及的所有标准库中的名字及对应的头文件。

读者请注意：在编译及运行本书的示例前请为代码添加必要的#include 指令和 using 声明。

3.1 节练习

练习 3.1：使用恰当的 using 声明重做 1.4.1 节（第 11 页）和 2.6.2 节（第 67 页）的练习。

3.2 标准库类型 string

标准库类型 **string** 表示可变长的字符序列，使用 string 类型必须首先包含 string 头文件。作为标准库的一部分，string 定义在命名空间 std 中。接下来的示例都假定已包含了下述代码：

84

```
#include <string>
using std::string;
```

本节描述最常用的 string 操作，9.5 节（第 320 页）还将介绍另外一些。

C++标准一方面对库类型所提供的操作做了详细规定，另一方面也对库的实现者做出一些性能上的需求。因此，标准库类型对于一般应用场合来说有足够的效率。

3.2.1 定义和初始化 string 对象

如何初始化类的对象是由类本身决定的。一个类可以定义很多种初始化对象的方式，只不过这些方式之间必须有所区别：或者是初始值的数量不同，或者是初始值的类型不同。表 3.1 列出了初始化 string 对象最常用的一些方式，下面是几个例子：

```
string s1;              // 默认初始化，s1 是一个空字符串
string s2 = s1;         // s2 是 s1 的副本
string s3 = "hiya";     // s3 是该字符串字面值的副本
string s4(10, 'c');     // s4 的内容是 cccccccccc
```

可以通过默认的方式（参见 2.2.1 节，第 40 页）初始化一个 string 对象，这样就会得到一个空的 string，也就是说，该 string 对象中没有任何字符。如果提供了一个字符串字面值（参见 2.1.3 节，第 36 页），则该字面值中除了最后那个空字符外其他所有的字符都被拷贝到新创建的 string 对象中去。如果提供的是一个数字和一个字符，则 string 对象的内容是给定字符连续重复若干次后得到的序列。

表 3.1：初始化 string 对象的方式

`string s1`	默认初始化，s1 是一个空串
`string s2(s1)`	s2 是 s1 的副本
`string s2 = s1`	等价于 s2(s1)，s2 是 s1 的副本
`string s3("value")`	s3 是字面值"value"的副本，除了字面值最后的那个空字符外
`string s3 = "value"`	等价于 s3("value")，s3 是字面值"value"的副本
`string s4(n, 'c')`	把 s4 初始化为由连续 n 个字符 c 组成的串

直接初始化和拷贝初始化

由 2.2.1 节（第 39 页）的学习可知，C++语言有几种不同的初始化方式，通过 string 我们可以清楚地看到在这些初始化方式之间到底有什么区别和联系。如果使用等号（=）初始化一个变量，实际上执行的是**拷贝初始化**（copy initialization），编译器把等号右侧的初始值拷贝到新创建的对象中去。与之相反，如果不使用等号，则执行的是**直接初始化**（direct initialization）。

当初始值只有一个时，使用直接初始化或拷贝初始化都行。如果像上面的 s4 那样初始化要用到的值有多个，一般来说只能使用直接初始化的方式：

```
string s5 = "hiya";     // 拷贝初始化
string s6("hiya");      // 直接初始化
string s7(10, 'c');     // 直接初始化，s7 的内容是 cccccccccc
```

85 对于用多个值进行初始化的情况，非要用拷贝初始化的方式来处理也不是不可以，不过需要显式地创建一个（临时）对象用于拷贝：

```
string s8 = string(10, 'c'); // 拷贝初始化，s8 的内容是 cccccccccc
```

s8 的初始值是 string(10, 'c')，它实际上是用数字 10 和字符 c 两个参数创建出来的一个 string 对象，然后这个 string 对象又拷贝给了 s8。这条语句本质上等价于下面的两条语句：

```
string temp(10, 'c');   // temp 的内容是 cccccccccc
string s8 = temp;       // 将 temp 拷贝给 s8
```

其实我们可以看到，尽管初始化 s8 的语句合法，但和初始化 s7 的方式比较起来可读性较差，也没有任何补偿优势。

3.2.2 string 对象上的操作

一个类除了要规定初始化其对象的方式外，还要定义对象上所能执行的操作。其中，类既能定义通过函数名调用的操作，就像 Sales_item 类的 isbn 函数那样（参见 1.5.2 节，第20页），也能定义<<、+等各种运算符在该类对象上的新含义。表3.2中列举了 string 的大多数操作。

表 3.2： string 的操作

os<<s	将 s 写到输出流 os 当中，返回 os
is>>s	从 is 中读取字符串赋给 s，字符串以空白分隔，返回 is
getline(is, s)	从 is 中读取一行赋给 s，返回 is
s.empty()	s 为空返回 true，否则返回 false
s.size()	返回 s 中字符的个数
s[n]	返回 s 中第 n 个字符的引用，位置 n 从 0 计起
s1+s2	返回 s1 和 s2 连接后的结果
s1=s2	用 s2 的副本代替 s1 中原来的字符
s1==s2	如果 s1 和 s2 中所含的字符完全一样，则它们相等；string 对象的相等性判断对字母的大小写敏感
s1!=s2	
<, <=, >, >=	利用字符在字典中的顺序进行比较，且对字母的大小写敏感

读写 string 对象

第 1 章曾经介绍过，使用标准库中的 iostream 来读写 int、double 等内置类型的值。同样，也可以使用 IO 操作符读写 string 对象：

```
// 注意：要想编译下面的代码还需要适当的#include 语句和 using 声明
int main()
{
    string s;                  // 空字符串
    cin >> s;                  // 将 string 对象读入 s，遇到空白停止
    cout << s << endl;         // 输出 s
    return 0;
}
```

这段程序首先定义一个名为 s 的空 string，然后将标准输入的内容读取到 s 中。在执行读取操作时，string 对象会自动忽略开头的空白（即空格符、换行符、制表符等）并从第一个真正的字符开始读起，直到遇见下一处空白为止。 86

如上所述，如果程序的输入是“ **Hello World!** ”（注意开头和结尾处的空格），则输出将是“**Hello**”，输出结果中没有任何空格。

和内置类型的输入输出操作一样，string 对象的此类操作也是返回运算符左侧的运算对象作为其结果。因此，多个输入或者多个输出可以连写在一起：

```
string s1, s2;
cin >> s1 >> s2;           // 把第一个输入读到 s1 中，第二个输入读到 s2 中
cout << s1 << s2 << endl;  // 输出两个 string 对象
```

假设给上面这段程序输入与之前一样的内容“ **Hello World!** ”，输出将是“**HelloWorld!**”。

读取未知数量的 string 对象

1.4.3 节（第 13 页）的程序可以读入数量未知的整数，下面编写一个类似的程序用于读取 string 对象：

```
int main()
{
    string word;
    while (cin >> word)              // 反复读取，直至到达文件末尾
        cout << word << endl;        // 逐个输出单词，每个单词后面紧跟一个换行
    return 0;
}
```

在该程序中，读取的对象是 string 而非 int，但是 while 语句的条件部分和之前版本的程序是一样的。该条件负责在读取时检测流的情况，如果流有效，也就是说没遇到文件
87 结束标记或非法输入，那么执行 while 语句内部的操作。此时，循环体将输出刚刚从标准输入读取的内容。重复若干次之后，一旦遇到文件结束标记或非法输入循环也就结束了。

使用 getline 读取一整行

有时我们希望能在最终得到的字符串中保留输入时的空白符，这时应该用 **getline** 函数代替原来的>>运算符。getline 函数的参数是一个输入流和一个 string 对象，函数从给定的输入流中读入内容，直到遇到换行符为止（注意换行符也被读进来了），然后把所读的内容存入到那个 string 对象中去（注意不存换行符）。getline 只要一遇到换行符就结束读取操作并返回结果，哪怕输入的一开始就是换行符也是如此。如果输入真的一开始就是换行符，那么所得的结果是个空 string。

和输入运算符一样，getline 也会返回它的流参数。因此既然输入运算符能作为判断的条件（参见 1.4.3 节，第 13 页），我们也能用 getline 的结果作为条件。例如，可以通过改写之前的程序让它一次输出一整行，而不再是每行输出一个词了：

```
int main()
{
    string line;
    // 每次读入一整行，直至到达文件末尾
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

因为 line 中不包含换行符，所以我们手动地加上换行操作符。和往常一样，使用 endl 结束当前行并刷新显示缓冲区。

触发 getline 函数返回的那个换行符实际上被丢弃掉了，得到的 string 对象中并不包含该换行符。

string 的 empty 和 size 操作

顾名思义，**empty** 函数根据 string 对象是否为空返回一个对应的布尔值（参见第

2.1 节，30 页）。和 Sales_item 类（参见 1.5.2 节，第 20 页）的 isbn 成员一样，empty 也是 string 的一个成员函数。调用该函数的方法很简单，只要使用点操作符指明是哪个对象执行了 empty 函数就可以了。

通过改写之前的程序，可以做到只输出非空的行：

```
// 每次读入一整行，遇到空行直接跳过
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

在上面的程序中，if 语句的条件部分使用了**逻辑非运算符**（!），它返回与其运算对象相反的结果。此例中，如果 str 不为空则返回真。

size 函数返回 string 对象的长度(即 string 对象中字符的个数)，可以使用 size 函数只输出长度超过 80 个字符的行： 88

```
string line;
// 每次读入一整行，输出其中超过 80 个字符的行
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

string::size_type 类型

对于 size 函数来说，返回一个 int 或者如前面 2.1.1 节（第 31 页）所述的那样返回一个 unsigned 似乎都是合情合理的。但其实 size 函数返回的是一个 string::size_type 类型的值，下面就对这种新的类型稍作解释。

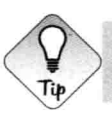
string 类及其他大多数标准库类型都定义了几种配套的类型。这些配套类型体现了标准库类型与机器无关的特性，类型 **size_type** 即是其中的一种。在具体使用的时候，通过作用域操作符来表明名字 size_type 是在类 string 中定义的。

尽管我们不太清楚 string::size_type 类型的细节，但有一点是肯定的：它是一个无符号类型的值（参见 2.1.1 节，第 30 页）而且能足够存放下任何 string 对象的大小。所有用于存放 string 类的 size 函数返回值的变量，都应该是 string::size_type 类型的。

过去，string::size_type 这种类型有点儿神秘，不太容易理解和使用。在 C++11 新标准中，允许编译器通过 auto 或者 decltype（参见 2.5.2 节，第 61 页）来推断变量的类型： 

```
auto len = line.size(); // len 的类型是 string::size_type
```

由于 size 函数返回的是一个无符号整型数，因此切记，如果在表达式中混用了带符号数和无符号数将可能产生意想不到的结果（参见 2.1.2 节，第 33 页）。例如，假设 n 是一个具有负值的 int，则表达式 s.size()<n 的判断结果几乎肯定是 true。这是因为负值 n 会自动地转换成一个比较大的无符号值。



如果一条表达式中已经有了 size()函数就不要再使用 int 了，这样可以避免混用 int 和 unsigned 可能带来的问题。

比较 string 对象

string 类定义了几种用于比较字符串的运算符。这些比较运算符逐一比较 string

对象中的字符，并且对大小写敏感，也就是说，在比较时同一个字母的大写形式和小写形式是不同的。

相等性运算符（==和!=）分别检验两个 string 对象相等或不相等，string 对象相等意味着它们的长度相同而且所包含的字符也全都相同。关系运算符<、<=、>、>=分别检验一个 string 对象是否小于、小于等于、大于、大于等于另外一个 string 对象。上述这些运算符都依照（大小写敏感的）字典顺序：

89

1. 如果两个 string 对象的长度不同，而且较短 string 对象的每个字符都与较长 string 对象对应位置上的字符相同，就说较短 string 对象小于较长 string 对象。
2. 如果两个 string 对象在某些对应的位置上不一致，则 string 对象比较的结果其实是 string 对象中第一对相异字符比较的结果。

下面是 string 对象比较的一个示例：

```
string str = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

根据规则 1 可判断，对象 str 小于对象 phrase；根据规则 2 可判断，对象 slang 既大于 str 也大于 phrase。

为 string 对象赋值

一般来说，在设计标准库类型时都力求在易用性上向内置类型看齐，因此大多数库类型都支持赋值操作。对于 string 类而言，允许把一个对象的值赋给另外一个对象：

```
string st1(10, 'c'), st2;// st1 的内容是 ccccccccc; st2 是一个空字符串
st1 = st2;               // 赋值：用 st2 的副本替换 st1 的内容
                         // 此时 st1 和 st2 都是空字符串
```

两个 string 对象相加

两个 string 对象相加得到一个新的 string 对象，其内容是把左侧的运算对象与右侧的运算对象串接而成。也就是说，对 string 对象使用加法运算符（+）的结果是一个新的 string 对象，它所包含的字符由两部分组成：前半部分是加号左侧 string 对象所含的字符、后半部分是加号右侧 string 对象所含的字符。另外，复合赋值运算符（+=）（参见 1.4.1 节，第 10 页）负责把右侧 string 对象的内容追加到左侧 string 对象的后面：

```
string s1 = "hello, ", s2 = "world\n";
string s3 = s1 + s2;     // s3 的内容是 hello, world\n
s1 += s2;                // 等价于 s1 = s1 + s2
```

字面值和 string 对象相加

如 2.1.2 节（第 33 页）所讲的，即使一种类型并非所需，我们也可以使用它，不过前提是该种类型可以自动转换成所需的类型。因为标准库允许把字符字面值和字符串字面值（参见 2.1.3 节，第 36 页）转换成 string 对象，所以在需要 string 对象的地方就可以使用这两种字面值来替代。利用这一点将之前的程序改写为如下形式：

```
string s1 = "hello", s2 = "world"; // 在 s1 和 s2 中都没有标点符号
string s3 = s1 + ", " + s2 + '\n';
```

当把 string 对象和字符字面值及字符串字面值混在一条语句中使用时，必须确保每个加法运算符（+）的两侧的运算对象至少有一个是 string：

```
string s4 = s1 + ", ";          // 正确：把一个 string 对象和一个字面值相加
string s5 = "hello" + ", ";     // 错误：两个运算对象都不是 string
// 正确：每个加法运算符都有一个运算对象是 string
string s6 = s1 + ", " + "world";
string s7 = "hello" + ", " + s2; // 错误：不能把字面值直接相加
```

90

s4 和 s5 初始化时只用到了一个加法运算符，因此很容易判断是否合法。s6 的初始化形式之前没有出现过，但其实它的工作机理和连续输入连续输出（参见 1.2 节，第 6 页）是一样的，可以用如下的形式分组：

```
string s6 = (s1 + ", ") + "world";
```

其中子表达式 s1 + ","的结果是一个 string 对象，它同时作为第二个加法运算符的左侧运算对象，因此上述语句和下面的两个语句是等价的：

```
string tmp = s1 + ", ";  // 正确：加法运算符有一个运算对象是 string
s6 = tmp + "world";      // 正确：加法运算符有一个运算对象是 string
```

另一方面，s7 的初始化是非法的，根据其语义加上括号后就成了下面的形式：

```
string s7 = ("hello" + ", ") + s2; // 错误：不能把字面值直接相加
```

很容易看到，括号内的子表达式试图把两个字符串字面值加在一起，而编译器根本没法做到这一点，所以这条语句是错误的。

因为某些历史原因，也为了与 C 兼容，所以 C++语言中的字符串字面值并不是标准库类型 string 的对象。切记，字符串字面值与 string 是不同的类型。

3.2.2 节练习

练习 3.2：编写一段程序从标准输入中一次读入一整行，然后修改该程序使其一次读入一个词。

练习 3.3：请说明 string 类的输入运算符和 getline 函数分别是如何处理空白字符的。

练习 3.4：编写一段程序读入两个字符串，比较其是否相等并输出结果。如果不相等，输出较大的那个字符串。改写上述程序，比较输入的两个字符串是否等长，如果不等长，输出长度较大的那个字符串。

练习 3.5：编写一段程序从标准输入中读入多个字符串并将它们连接在一起，输出连接成的大字符串。然后修改上述程序，用空格把输入的多个字符串分隔开来。

3.2.3 处理 string 对象中的字符

我们经常需要单独处理 string 对象中的字符，比如检查一个 string 对象是否包含空白，或者把 string 对象中的字母改成小写，再或者查看某个特定的字符是否出现等。

这类处理的一个关键问题是如何获取字符本身。有时需要处理 string 对象中的每一个字符，另外一些时候则只需处理某个特定的字符，还有些时候遇到某个条件处理就要停

91

下来。以往的经验告诉我们，处理这些情况常常要涉及语言和库的很多方面。

另一个关键问题是要知道能改变某个字符的特性。在 cctype 头文件中定义了一组标准库函数处理这部分工作，表 3.3 列出了主要的函数名及其含义。

表 3.3：cctype 头文件中的函数

`isalnum(c)`	当 c 是字母或数字时为真
`isalpha(c)`	当 c 是字母时为真
`iscntrl(c)`	当 c 是控制字符时为真
`isdigit(c)`	当 c 是数字时为真
`isgraph(c)`	当 c 不是空格但可打印时为真
`islower(c)`	当 c 是小写字母时为真
`isprint(c)`	当 c 是可打印字符时为真（即 c 是空格或 c 具有可视形式）
`ispunct(c)`	当 c 是标点符号时为真（即 c 不是控制字符、数字、字母、可打印空白中的一种）
`isspace(c)`	当 c 是空白时为真（即 c 是空格、横向制表符、纵向制表符、回车符、换行符、进纸符中的一种）
`isupper(c)`	当 c 是大写字母时为真
`isxdigit(c)`	当 c 是十六进制数字时为真
`tolower(c)`	如果 c 是大写字母，输出对应的小写字母；否则原样输出 c
`toupper(c)`	如果 c 是小写字母，输出对应的大写字母；否则原样输出 c

建议：使用 C++版本的 C 标准库头文件

C++标准库中除了定义 C++语言特有的功能外，也兼容了 C 语言的标准库。C 语言的头文件形如 *name*.h，C++则将这些文件命名为 c*name*。也就是去掉了 .h 后缀，而在文件名 *name* 之前添加了字母 c，这里的 c 表示这是一个属于 C 语言标准库的头文件。

因此，cctype 头文件和 ctype.h 头文件的内容是一样的，只不过从命名规范上来讲更符合 C++语言的要求。特别的，在名为 c*name* 的头文件中定义的名字从属于命名空间 std，而定义在名为 .h 的头文件中的则不然。

一般来说，C++程序应该使用名为 c*name* 的头文件而不使用 *name*.h 的形式，标准库中的名字总能在命名空间 std 中找到。如果使用 .h 形式的头文件，程序员就不得不时刻牢记哪些是从 C 语言那儿继承过来的，哪些又是 C++语言所独有的。

处理每个字符？使用基于范围的 for 语句

C++11

如果想对 string 对象中的每个字符做点儿什么操作，目前最好的办法是使用 C++11 新标准提供的一种语句：**范围 for**（range for）语句。这种语句遍历给定序列中的每个元素并对序列中的每个值执行某种操作，其语法形式是：

```
for (declaration : expression)
    statement
```

其中，*expression* 部分是一个对象，用于表示一个序列。*declaration* 部分负责定义一个变量，该变量将被用于访问序列中的基础元素。每次迭代，*declaration* 部分的变量会被初始化为 *expression* 部分的下一个元素值。

一个 string 对象表示一个字符的序列，因此 string 对象可以作为范围 for 语句

中的 *expression* 部分。举一个简单的例子，我们可以使用范围 for 语句把 string 对象中的字符每行一个输出出来：

```
string str("some string");
// 每行输出 str 中的一个字符。
for (auto c : str)                  // 对于 str 中的每个字符
    cout << c << endl;              // 输出当前字符，后面紧跟一个换行符
```

for 循环把变量 c 和 str 联系了起来，其中我们定义循环控制变量的方式与定义任意一个普通变量是一样的。此例中，通过使用 auto 关键字（参见 2.5.2 节，第 61 页）让编译器来决定变量 c 的类型，这里 c 的类型是 char。每次迭代，str 的下一个字符被拷贝给 c，因此该循环可以读作“对于字符串 str 中的每个字符 c，”执行某某操作。此例中的“某某操作”即输出一个字符，然后换行。 92

举个稍微复杂一点的例子，使用范围 for 语句和 ispunct 函数来统计 string 对象中标点符号的个数：

```
string s("Hello World!!!");
// punct_cnt 的类型和 s.size 的返回类型一样；参见 2.5.3 节（第 62 页）
decltype(s.size()) punct_cnt = 0;
//统计 s 中标点符号的数量
for (auto c : s)                    // 对于 s 中的每个字符
    if (ispunct(c))                 // 如果该字符是标点符号
        ++punct_cnt;                // 将标点符号的计数值加 1
cout << punct_cnt
     << " punctuation characters in " << s << endl;
```

程序的输出结果将是：

```
3 punctuation characters in Hello World!!!
```

这里我们使用 decltype 关键字（参见 2.5.3 节，第 62 页）声明计数变量 punct_cnt，它的类型是 s.size 函数返回值的类型，也就是 string::size_type。使用范围 for 语句处理 string 对象中的每个字符并检查其是否是标点符号。如果是，使用递增运算符（参见 1.4.1 节，第 10 页）给计数变量加 1。最后，待范围 for 语句结束后输出统计结果。

使用范围 for 语句改变字符串中的字符 93

如果想要改变 string 对象中字符的值，必须把循环变量定义成引用类型（参见 2.3.1 节，第 45 页）。记住，所谓引用只是给定对象的一个别名，因此当使用引用作为循环控制变量时，这个变量实际上被依次绑定到了序列的每个元素上。使用这个引用，我们就能改变它绑定的字符。

新的例子不再是统计标点符号的个数了，假设我们想要把字符串改写为大写字母的形式。为了做到这一点可以使用标准库函数 toupper，该函数接收一个字符，然后输出其对应的大写形式。这样，为了把整个 string 对象转换成大写，只要对其中的每个字符调用 toupper 函数并将结果再赋给原字符就可以了：

```
string s("Hello World!!!");
// 转换成大写形式。
for (auto &c : s)           // 对于 s 中的每个字符（注意：c 是引用）
    c = toupper(c);         // c 是一个引用，因此赋值语句将改变 s 中字符的值
cout << s << endl;
```

上述代码的输出结果将是：

```
HELLO WORLD!!!
```

每次迭代时，变量 c 引用 string 对象 s 的下一个字符，赋值给 c 也就是在改变 s 中对应字符的值。因此当执行下面的语句时，

```
c = toupper(c); // c是一个引用，因此赋值语句将改变s中字符的值
```

实际上改变了 c 绑定的字符的值。整个循环结束后，str 中的所有字符都变成了大写形式。

只处理一部分字符？

如果要处理 string 对象中的每一个字符，使用范围 for 语句是个好主意。然而，有时我们需要访问的只是其中一个字符，或者访问多个字符但遇到某个条件就要停下来。例如，同样是将字符改为大写形式，不过新的要求不再是对整个字符串都这样做，而仅仅把 string 对象中的第一个字母或第一个单词大写化。

要想访问 string 对象中的单个字符有两种方式：一种是使用下标，另外一种是使用迭代器，其中关于迭代器的内容将在 3.4 节（第 95 页）和第 9 章中介绍。

下标运算符（[ ]）接收的输入参数是 string::size_type 类型的值（参见 3.2.2 节，第 79 页），这个参数表示要访问的字符的位置；返回值是该位置上字符的引用。

string 对象的下标从 0 计起。如果 string 对象 s 至少包含两个字符，则 s[0]是第 1 个字符、s[1]是第 2 个字符、s[s.size()-1]是最后一个字符。

string 对象的下标必须大于等于 0 而小于 s.size()。

使用超出此范围的下标将引发不可预知的结果，以此推断，使用下标访问空 string 也会引发不可预知的结果。

94 下标的值称作“下标”或“**索引**”，任何表达式只要它的值是一个整型值就能作为索引。不过，如果某个索引是带符号类型的值将自动转换成由 string::size_type（参见 2.1.2 节，第 33 页）表达的无符号类型。

下面的程序使用下标运算符输出 string 对象中的第一个字符：

```
if (!s.empty())                 // 确保确实有字符需要输出
    cout << s[0] << endl;       // 输出s的第一个字符
```

在访问指定字符之前，首先检查 s 是否为空。其实不管什么时候只要对 string 对象使用了下标，都要确认在那个位置上确实有值。如果 s 为空，则 s[0]的结果将是未定义的。

只要字符串不是常量（参见 2.4 节，第 53 页），就能为下标运算符返回的字符赋新值。例如，下面的程序将字符串的首字符改成了大写形式：

```
string s("some string");
if (!s.empty())                 // 确保s[0]的位置确实有字符
    s[0] = toupper(s[0]);       // 为s的第一个字符赋一个新值
```

程序的输出结果将是：

```
Some string
```

使用下标执行迭代

另一个例子是把 s 的第一个词改成大写形式：

```
// 依次处理 s 中的字符直至我们处理完全部字符或者遇到一个空白
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
         s[index] = toupper(s[index]); //将当前字符改成大写形式
```

程序的输出结果将是：

```
SOME string
```

在上述程序中，for 循环使用变量 index 作为 s 的下标，index 的类型是由 decltype 关键字决定的。首先把 index 初始化为 0，这样第一次迭代就会从 s 的首字符开始；之后每次迭代将 index 加 1 以得到 s 的下一个字符。循环体负责将当前的字母改写为大写形式。

for 语句的条件部分涉及一点新知识，该条件使用了**逻辑与运算符**（&&）。如果参与运算的两个运算对象都为真，则逻辑与结果为真；否则结果为假。对这个运算符来说最重要的一点是，C++语言规定只有当左侧运算对象为真时才会检查右侧运算对象的情况。如此例所示，这条规定确保了只有当下标取值在合理范围之内时才会真的用此下标去访问字符串。也就是说，只有在 index 达到 s.size()之前才会执行 s[index]。随着 index 的增加，它永远也不可能超过 s.size()的值，所以可以确保 index 比 s.size()小。

提示：注意检查下标的合法性 95

使用下标时必须确保其在合理范围之内，也就是说，下标必须大于等于 0 而小于字符串的 size()的值。一种简便易行的方法是，总是设下标的类型为 string::size_type，因为此类型是无符号数，可以确保下标不会小于 0。此时，代码只需保证下标小于 size()的值就可以了。

C++标准并不要求标准库检测下标是否合法。一旦使用了一个超出范围的下标，就会产生不可预知的结果。

使用下标执行随机访问

在之前的示例中，我们让字符串的下标每次加 1 从而按顺序把所有字符改写成了大写形式。其实也能通过计算得到某个下标值，然后直接获取对应位置的字符，并不是每次都得从前往后依次访问。

例如，想要编写一个程序把 0 到 15 之间的十进制数转换成对应的十六进制形式，只需初始化一个字符串令其存放 16 个十六进制“数字”：

```
const string hexdigits = "0123456789ABCDEF"; // 可能的十六进制数字
cout << "Enter a series of numbers between 0 and 15"
     << " separated by spaces. Hit ENTER when finished: "
     << endl;
string result;           // 用于保存十六进制的字符串
string::size_type n;     // 用于保存从输入流读取的数
while (cin >> n)
    if (n < hexdigits.size())     // 忽略无效输入
        result += hexdigits[n];  // 得到对应的十六进制数字
```

```
cout << "Your hex number is: " << result << endl;
```

假设输入的内容如下：

```
12 0 5 15 8 15
```

程序的输出结果将是：

```
Your hex number is: C05F8F
```

上述程序的执行过程是这样的：首先初始化变量 hexdigits 令其存放从 0 到 F 的十六进制数字，注意我们把 hexdigits 声明成了常量（参见 2.4 节，第 53 页），这是因为在后面的程序中不打算再改变它的值。在循环内部使用输入值 n 作为 hexdigits 的下标，hexdigits[n]的值就是 hexdigits 内位置 n 处的字符。例如，如果 n 是 15，则结果是 F；如果 n 是 12，则结果是 C，以此类推。把得到的十六进制数字添加到 result 内，最后一并输出。

无论何时用到字符串的下标，都应该注意检查其合法性。在上面的程序中，下标 n 是 string::size_type 类型，也就是无符号类型，所以 n 可以确保大于或等于 0。在实际使用时，还需检查 n 是否小于 hexdigits 的长度。

96

3.2.3 节练习

练习 3.6：编写一段程序，使用范围 for 语句将字符串内的所有字符用 X 代替。

练习 3.7：就上一题完成的程序而言，如果将循环控制变量的类型设为 char 将发生什么？先估计一下结果，然后实际编程进行验证。

练习 3.8：分别用 while 循环和传统的 for 循环重写第一题的程序，你觉得哪种形式更好呢？为什么？

练习 3.9：下面的程序有何作用？它合法吗？如果不合法，为什么？

```
string s;
cout << s[0] << endl;
```

练习 3.10：编写一段程序，读入一个包含标点符号的字符串，将标点符号去除后输出字符串剩余的部分。

练习 3.11：下面的范围 for 语句合法吗？如果合法，c 的类型是什么？

```
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
```

3.3 标准库类型 vector

标准库类型 **vector** 表示对象的集合，其中所有对象的类型都相同。集合中的每个对象都有一个与之对应的索引，索引用于访问对象。因为 vector“容纳着”其他对象，所以它也常被称作**容器**（container）。第 II 部将对容器进行更为详细的介绍。

要想使用 vector，必须包含适当的头文件。在后续的例子中，都将假定做了如下 using 声明：

```
#include <vector>
using std::vector;
```