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
s1!=s2	等性判断对字母的大小写敏感
<, <=, >, >=	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。

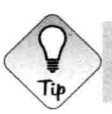
string 类及其他大多数标准库类型都定义了几种配套的类型。这些配套类型体现了标准库类型与机器无关的特性，类型 **size_type** 即是其中的一种。在具体使用的时候，通过作用域操作符来表明名字 size_type 是在类 string 中定义的。

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;
```

C++语言既有**类模板**（class template），也有函数模板，其中 vector 是一个类模板。只有对 C++有了相当深入的理解才能写出模板，事实上，我们直到第 16 章才会学习如何自定义模板。幸运的是，即使还不会创建模板，我们也可以先试着用用它。

模板本身不是类或函数，相反可以将模板看作为编译器生成类或函数编写的一份说明。编译器根据模板创建类或函数的过程称为**实例化**（instantiation），当使用模板时，需要指出编译器应把类或函数实例化成何种类型。

对于类模板来说，我们通过提供一些额外信息来指定模板到底实例化成什么样的类，需要提供哪些信息由模板决定。提供信息的方式总是这样：即在模板名字后面跟一对尖括号，在括号内放上信息。

以 vector 为例，提供的额外信息是 vector 内所存放对象的类型：

```
vector<int> ivec;                  // ivec 保存 int 类型的对象
vector<Sales_item> Sales_vec;      // 保存 Sales_item 类型的对象
vector<vector<string>> file;       // 该向量的元素是 vector 对象
```

97

在上面的例子中，编译器根据模板 vector 生成了三种不同的类型：vector<int>、vector<Sales_item>和 vector<vector<string>>。

vector 是模板而非类型，由 vector 生成的类型必须包含 vector 中元素的类型，例如 vector<int>。

vector 能容纳绝大多数类型的对象作为其元素，但是因为引用不是对象（参见 2.3.1 节，第 45 页），所以不存在包含引用的 vector。除此之外，其他大多数（非引用）内置类型和类类型都可以构成 vector 对象，甚至组成 vector 的元素也可以是 vector。

需要指出的是，在早期版本的 C++标准中如果 vector 的元素还是 vector（或者其他模板类型），则其定义的形式与现在的 C++11 新标准略有不同。过去，必须在外层 vector 对象的右尖括号和其元素类型之间添加一个空格，如应该写成 vector<vector<int> > 而非 vector<vector<int>>。

某些编译器可能仍需以老式的声明语句来处理元素为 vector 的 vector 对象，如 vector<vector<int> >。

3.3.1 定义和初始化 vector 对象

和任何一种类类型一样，vector 模板控制着定义和初始化向量的方法。表 3.4 列出了定义 vector 对象的常用方法。

表 3.4：初始化 vector 对象的方法

vector<T> v1	v1 是一个空 vector，它潜在的元素是 T 类型的，执行默认初始化
vector<T> v2(v1)	v2 中包含有 v1 所有元素的副本
vector<T> v2 = v1	等价于 v2(v1)，v2 中包含有 v1 所有元素的副本
vector<T> v3(n, val)	v3 包含了 n 个重复的元素，每个元素的值都是 val
vector<T> v4(n)	v4 包含了 n 个重复地执行了值初始化的对象
vector<T> v5{a,b,c...}	v5 包含了初始值个数的元素，每个元素被赋予相应的初始值
vector<T> v5={a,b,c...}	等价于 v5{a,b,c...}

可以默认初始化 vector 对象（参见 2.2.1 节，第 40 页），从而创建一个指定类型的空 vector：

```
vector<string> svec; //默认初始化，svec 不含任何元素
```

看起来空 vector 好像没什么用，但是很快我们就会知道程序在运行时可以很高效地往 vector 对象中添加元素。事实上，最常见的方式就是先定义一个空 vector，然后当运行时获取到元素的值后再逐一添加。

当然也可以在定义 vector 对象时指定元素的初始值。例如，允许把一个 vector 对象的元素拷贝给另外一个 vector 对象。此时，新 vector 对象的元素就是原 vector 对象对应元素的副本。注意两个 vector 对象的类型必须相同：

```
vector<int> ivec;                // 初始状态为空
// 在此处给 ivec 添加一些值
vector<int> ivec2(ivec);         // 把 ivec 的元素拷贝给 ivec2
vector<int> ivec3 = ivec;        // 把 ivec 的元素拷贝给 ivec3
vector<string> svec(ivec2);      // 错误：svec 的元素是 string 对象，不是 int
```

98 列表初始化 vector 对象

C++11 新标准还提供了另外一种为 vector 对象的元素赋初值的方法，即列表初始化（参见 2.2.1 节，第 39 页）。此时，用花括号括起来的 0 个或多个初始元素值被赋给 vector 对象：

```
vector<string> articles = {"a", "an", "the"};
```

上述 vector 对象包含三个元素：第一个是字符串"a"，第二个是字符串"an"，最后一个是字符串"the"。

之前已经讲过，C++语言提供了几种不同的初始化方式（参见 2.2.1 节，第 39 页）。在大多数情况下这些初始化方式可以相互等价地使用，不过也并非一直如此。目前已经介绍过的两种例外情况是：其一，使用拷贝初始化时（即使用=时）（参见 3.2.1 节，第 76 页），只能提供一个初始值；其二，如果提供的是一个类内初始值（参见 2.6.1 节，第 64 页），则只能使用拷贝初始化或使用花括号的形式初始化。第三种特殊的要求是，如果提供的是初始元素值的列表，则只能把初始值都放在花括号里进行列表初始化，而不能放在圆括号里：

```
vector<string> v1{"a", "an", "the"}; // 列表初始化
vector<string> v2("a", "an", "the"); // 错误
```

创建指定数量的元素

还可以用 vector 对象容纳的元素数量和所有元素的统一初始值来初始化 vector 对象：

```
vector<int> ivec(10, -1);     // 10 个 int 类型的元素，每个都被初始化为-1
vector<string> svec(10, "hi!"); // 10 个 string 类型的元素，
                                // 每个都被初始化为"hi!"
```

值初始化

通常情况下，可以只提供 vector 对象容纳的元素数量而不用略去初始值。此时库会创建一个**值初始化的**（value-initialized）元素初值，并把它赋给容器中的所有元素。这个初值由 vector 对象中元素的类型决定。

如果 vector 对象的元素是内置类型，比如 int，则元素初始值自动设为 0。如果元素是某种类类型，比如 string，则元素由类默认初始化：

```
vector<int> ivec(10);          // 10 个元素，每个都初始化为 0
vector<string> svec(10);       // 10 个元素，每个都是空 string 对象
```

对这种初始化的方式有两个特殊限制：其一，有些类要求必须明确地提供初始值（参见 2.2.1 节，第 40 页），如果 vector 对象中元素的类型不支持默认初始化，我们就必须提供初始的元素值。对这种类型的对象来说，只提供元素的数量而不设定初始值无法完成初始化工作。

其二，如果只提供了元素的数量而没有设定初始值，只能使用直接初始化：

```
vector<int> vi = 10; // 错误：必须使用直接初始化的形式指定向量大小
```

99

这里的 10 是用来说明如何初始化 vector 对象的，我们用它的本意是想创建含有 10 个值初始化了的元素的 vector 对象，而非把数字 10“拷贝”到 vector 中。因此，此时不宜使用拷贝初始化，7.5.4 节（第 265 页）将对这一点做更详细的介绍。

列表初始值还是元素数量？

在某些情况下，初始化的真实含义依赖于传递初始值时用的是花括号还是圆括号。例如，用一个整数来初始化 vector<int>时，整数的含义可能是 vector 对象的容量也可能是元素的值。类似的，用两个整数来初始化 vector<int>时，这两个整数可能一个是 vector 对象的容量，另一个是元素的初值，也可能它们是容量为 2 的 vector 对象中两个元素的初值。通过使用花括号或圆括号可以区分上述这些含义：

```
vector<int> v1(10);        // v1 有 10 个元素，每个的值都是 0
vector<int> v2{10};        // v2 有 1 个元素，该元素的值是 10

vector<int> v3(10, 1);     // v3 有 10 个元素，每个的值都是 1
vector<int> v4{10, 1};     // v4 有 2 个元素，值分别是 10 和 1
```

如果用的是圆括号，可以说提供的值是用来构造（construct）vector 对象的。例如，v1 的初始值说明了 vector 对象的容量；v3 的两个初始值则分别说明了 vector 对象的容量和元素的初值。

如果用的是花括号，可以表述成我们想列表初始化（list initialize）该 vector 对象。也就是说，初始化过程会尽可能地把花括号内的值当成是元素初始值的列表来处理，只有在无法执行列表初始化时才会考虑其他初始化方式。在上例中，给 v2 和 v4 提供的初始值都能作为元素的值，所以它们都会执行列表初始化，vector 对象 v2 包含一个元素而 vector 对象 v4 包含两个元素。

另一方面，如果初始化时使用了花括号的形式但是提供的值又不能用来列表初始化，就要考虑用这样的值来构造 vector 对象了。例如，要想列表初始化一个含有 string 对象的 vector 对象，应该提供能赋给 string 对象的初值。此时不难区分到底是要列表初始化 vector 对象的元素还是用给定的容量值来构造 vector 对象：

100

```
vector<string> v5{"hi"}; // 列表初始化：v5 有一个元素
vector<string> v6("hi"); // 错误：不能使用字符串字面值构建 vector 对象
vector<string> v7{10};           // v7 有 10 个默认初始化的元素
vector<string> v8{10, "hi"};     // v8 有 10 个值为"hi"的元素
```

尽管在上面的例子中除了第二条语句之外都用了花括号，但其实只有 v5 是列表初始化。要想列表初始化 vector 对象，花括号里的值必须与元素类型相同。显然不能用 int 初始化 string 对象，所以 v7 和 v8 提供的值不能作为元素的初始值。确认无法执行列表初始化后，编译器会尝试用默认值初始化 vector 对象。

3.3.1 节练习

练习 3.12：下列 vector 对象的定义有不正确的吗？如果有，请指出来。对于正确的，描述其执行结果；对于不正确的，说明其错误的原因。

```
(a) vector<vector<int>> ivec;
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

练习 3.13：下列的 vector 对象各包含多少个元素？这些元素的值分别是多少？

```
(a) vector<int> v1;                    (b) vector<int> v2(10);
(c) vector<int> v3(10, 42);            (d) vector<int> v4{10};
(e) vector<int> v5{10, 42};            (f) vector<string> v6{10};
(g) vector<string> v7{10, "hi"};
```

3.3.2 向 vector 对象中添加元素

对 vector 对象来说，直接初始化的方式适用于三种情况：初始值已知且数量较少、初始值是另一个 vector 对象的副本、所有元素的初始值都一样。然而更常见的情况是：创建一个 vector 对象时并不清楚实际所需的元素个数，元素的值也经常无法确定。还有些时候即使元素的初值已知，但如果这些值总量较大而各不相同，那么在创建 vector 对象的时候执行初始化操作也会显得过于烦琐。

举个例子，如果想创建一个 vector 对象令其包含从 0 到 9 共 10 个元素，使用列表初始化的方法很容易做到这一点；但如果 vector 对象包含的元素是从 0 到 99 或者从 0 到 999 呢？这时通过列表初始化把所有元素都一一罗列出来就不太合适了。对于此例来说，更好的处理方法是先创建一个空 vector，然后在运行时再利用 vector 的成员函数 **push_back** 向其中添加元素。push_back 负责把一个值当成 vector 对象的尾元素“压到（push）”vector 对象的“尾端（back）”。例如：

101

```
vector<int> v2;          // 空 vector 对象
for (int i = 0; i != 100; ++i)
    v2.push_back(i); // 依次把整数值放到 v2 尾端
// 循环结束后 v2 有 100 个元素，值从 0 到 99
```

在上例中，尽管知道 vector 对象最后会包含 100 个元素，但在一开始还是把它声明成空 vector，在每次迭代时才顺序地把下一个整数作为 v2 的新元素添加给它。

同样的，如果直到运行时才能知道 vector 对象中元素的确切个数，也应该使用刚刚这种方法创建 vector 对象并为其赋值。例如，有时需要实时读入数据然后将其赋予 vector 对象：

```
// 从标准输入中读取单词，将其作为 vector 对象的元素存储
string word;
vector<string> text;          // 空 vector 对象
while (cin >> word) {
    text.push_back(word);     // 把 word 添加到 text 后面
}
```

和之前的例子一样，本例也是先创建一个空 vector，之后依次读入未知数量的值并保存到 text 中。

关键概念：vector 对象能高效增长

C++标准要求 vector 应该能在运行时高效快速地添加元素。因此既然 vector 对象能高效地增长，那么在定义 vector 对象的时候设定其大小也就没什么必要了，事实上如果这么做性能可能更差。只有一种例外情况，就是所有（all）元素的值都一样。一旦元素的值有所不同，更有效的办法是先定义一个空的 vector 对象，再在运行时向其中添加具体值。此外，9.4 节（第 317 页）将介绍，vector 还提供了方法，允许我们进一步提升动态添加元素的性能。

开始的时候创建空的 vector 对象，在运行时再动态添加元素，这一做法与 C 语言及其他大多数语言中内置数组类型的用法不同。特别是如果用惯了 C 或者 Java，可以预计在创建 vector 对象时顺便指定其容量是最好的。然而事实上，通常的情况是恰恰相反。

向 vector 对象添加元素蕴含的编程假定

由于能高效便捷地向 vector 对象中添加元素，很多编程工作被极大简化了。然而，这种简便性也伴随着一些对编写程序更高的要求：其中一条就是必须要确保所写的循环正确无误，特别是在循环有可能改变 vector 对象容量的时候。

随着对 vector 的更多使用，我们还会逐渐了解到其他一些隐含的要求，其中一条是现在就要指出的：如果循环体内部包含有向 vector 对象添加元素的语句，则不能使用范围 for 循环，具体原因将在 5.4.3 节（第 168 页）详细解释。

范围 for 语句体内不应改变其所遍历序列的大小。

3.3.2 节练习

102

练习 3.14：编写一段程序，用 cin 读入一组整数并把它们存入一个 vector 对象。

练习 3.15：改写上题的程序，不过这次读入的是字符串。

3.3.3 其他 vector 操作

除了 push_back 之外，vector 还提供了几种其他操作，大多数都和 string 的相关操作类似，表 3.5 列出了其中比较重要的一些。

表 3.5：vector 支持的操作

v.empty()	如果 v 不含有任何元素，返回真；否则返回假
v.size()	返回 v 中元素的个数
v.push_back(t)	向 v 的尾端添加一个值为 t 的元素
v[n]	返回 v 中第 n 个位置上元素的引用
v1 = v2	用 v2 中元素的拷贝替换 v1 中的元素
v1 = {a,b,c... }	用列表中元素的拷贝替换 v1 中的元素
v1 == v2	v1 和 v2 相等当且仅当它们的元素数量相同且对应位置的元素值都相同
v1 != v2	
<, <=, >, >=	顾名思义，以字典顺序进行比较

访问 vector 对象中元素的方法和访问 string 对象中字符的方法差不多，也是通过元素在 vector 对象中的位置。例如，可以使用范围 for 语句处理 vector 对象中的所有元素：

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto &i : v)          // 对于 v 中的每个元素（注意：i 是一个引用）
    i *= i;                // 求元素值的平方
for (auto i : v)           // 对于 v 中的每个元素
    cout << i << " ";      // 输出该元素
cout << endl;
```

第一个循环把控制变量 i 定义成引用类型，这样就能通过 i 给 v 的元素赋值，其中 i 的类型由 auto 关键字指定。这里用到了一种新的复合赋值运算符（参见 1.4.1 节，第 10 页）。如我们所知，+=把左侧运算对象和右侧运算对象相加，结果存入左侧运算对象；类似的，*=把左侧运算对象和右侧运算对象相乘，结果存入左侧运算对象。最后，第二个循环输出所有元素。

vector 的 empty 和 size 两个成员与 string 的同名成员（参见 3.2.2 节，第 78 页）功能完全一致：empty 检查 vector 对象是否包含元素然后返回一个布尔值；size 则返回 vector 对象中元素的个数，返回值的类型是由 vector 定义的 size_type 类型。

要使用 size_type，需首先指定它是由哪种类型定义的。vector 对象的类型总是包含着元素的类型（参见 3.3 节，第 87 页）：

```
vector<int>::size_type        // 正确
vector::size_type             // 错误
```

各个相等性运算符和关系运算符也与 string 的相应运算符（参见 3.2.2 节，第 79 页）功能一致。两个 vector 对象相等当且仅当它们所含的元素个数相同，而且对应位置的元素值也相同。关系运算符依照字典顺序进行比较：如果两个 vector 对象的容量不同，但是在相同位置上的元素值都一样，则元素较少的 vector 对象小于元素较多的 vector 对象；若元素的值有区别，则 vector 对象的大小关系由第一对相异的元素值的大小关系决定。

103

只有当元素的值可比较时，vector 对象才能被比较。一些类，如 string 等，确实定义了自己的相等性运算符和关系运算符；另外一些，如 Sales_item 类支持的运算已经全都罗列在 1.5.1 节（第 17 页）中了，显然并不支持相等性判断和关系运算等操作。因此，不能比较两个 vector<Sales_item>对象。

计算 vector 内对象的索引

使用下标运算符（参见 3.2.3 节，第 84 页）能获取到指定的元素。和 string 一样，vector 对象的下标也是从 0 开始计起，下标的类型是相应的 size_type 类型。只要 vector 对象不是一个常量，就能向下标运算符返回的元素赋值。此外，如 3.2.3 节（第 85 页）所述的那样，也能通过计算得到 vector 内对象的索引，然后直接获取索引位置上的元素。

举个例子，假设有一组成绩的集合，其中成绩的取值是从 0 到 100。以 10 分为一个分数段，要求统计各个分数段各有多少个成绩。显然，从 0 到 100 总共有 101 种可能的成绩取值，这些成绩分布在 11 个分数段上：每 10 个分数构成一个分数段，这样的分数段有 10 个，额外还有一个分数段表示满分 100 分。这样第一个分数段将统计成绩在 0 到 9 之间的数量；第二个分数段将统计成绩在 10 到 19 之间的数量，以此类推。最后一个分数段统计满分 100 分的数量。

按照上面的描述，如果输入的成绩如下：

```
42 65 95 100 39 67 95 76 88 76 83 92 76 93
```

则输出的结果应该是：

```
0 0 0 1 1 0 2 3 2 4 1
```

结果显示：成绩在 30 分以下的没有、30 分至 39 分有 1 个、40 分至 49 分有 1 个、50 分至 59 分没有、60 分至 69 分有 2 个、70 分至 79 分有 3 个、80 分至 89 分有 2 个、90 分至 99 分有 4 个，还有 1 个是满分。

在具体实现时使用一个含有 11 个元素的 vector 对象，每个元素分别用于统计各个分数段上出现的成绩个数。对于某个成绩来说，将其除以 10 就能得到对应的分数段索引。注意：两个整数相除，结果还是整数，余数部分被自动忽略掉了。例如，42/10=4、65/10=6、100/10=10 等。一旦计算得到了分数段索引，就能用它作为 vector 对象的下标，进而获取该分数段的计数值并加 1：

104

```
// 以 10 分为一个分数段统计成绩的数量：0～9, 10～19, ..., 90～99, 100
vector<unsigned> scores(11, 0);  // 11 个分数段，全都初始化为 0
unsigned grade;
while (cin >> grade) {           // 读取成绩
    if (grade <= 100)            // 只处理有效的成绩
        ++scores[grade/10];      // 将对应分数段的计数值加 1
}
```

在上面的程序中，首先定义了一个 vector 对象存放各个分数段上成绩的数量。此例中，由于初始状态下每个元素的值都相同，所以我们为 vector 对象申请了 11 个元素，并把所有元素的初始值都设为 0。while 语句的条件部分负责读入成绩，在循环体内部首先检查读入的成绩是否合法（即是否小于等于 100 分），如果合法，将成绩对应的分数段的计数值加 1。

执行计数值累加的那条语句很好地体现了 C++程序代码的简洁性。表达式

```
++scores[grade/10]; // 将当前分数段的计数值加 1
```

等价于

```
auto ind = grade/10;             // 得到分数段索引
scores[ind] = scores[ind] + 1;   // 将计数值加 1
```

上述语句的含义是：用 grade 除以 10 来计算成绩所在的分数段，然后将所得的结果作为变量 scores 的下标。通过运行下标运算获取该分数段对应的计数值，因为新出现了一个属于该分数段的成绩，所以将计数值加 1。

如前所述，使用下标的时候必须清楚地知道它是否在合理范围之内（参见 3.2.3 节，第 85 页）。在这个程序里，我们事先确认了输入的成绩确实在 0 到 100 之间，这样计算所得的下标就一定在 0 到 10 之间，属于 0 到 scores.size()-1 规定的有效范围，一定是合法的。

不能用下标形式添加元素

刚接触 C++语言的程序员也许会认为可以通过 vector 对象的下标形式来添加元素，事实并非如此。下面的代码试图为 vector 对象 ivec 添加 10 个元素：

```
vector<int> ivec; // 空 vector 对象
```

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec[ix] = ix; // 严重错误：ivec 不包含任何元素
```

然而，这段代码是错误的：ivec 是一个空 vector，根本不包含任何元素，当然也就不能通过下标去访问任何元素！如前所述，正确的方法是使用 push_back：

105

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // 正确：添加一个新元素，该元素的值是 ix
```

vector 对象（以及 string 对象）的下标运算符可用于访问已存在的元素，而不能用于添加元素。

提示：只能对确知已存在的元素执行下标操作！

关于下标必须明确的一点是：只能对确知已存在的元素执行下标操作。例如，

```
vector<int> ivec;          // 空 vector 对象
cout << ivec[0];           // 错误：ivec 不包含任何元素

vector<int> ivec2(10);     // 含有 10 个元素的 vector 对象
cout << ivec2[10];         // 错误：ivec2 元素的合法索引是从 0 到 9
```

试图用下标的形式去访问一个不存在的元素将引发错误，不过这种错误不会被编译器发现，而是在运行时产生一个不可预知的值。

不幸的是，这种通过下标访问不存在的元素的行为非常常见，而且会产生很严重的后果。所谓的缓冲区溢出（buffer overflow）指的就是这类错误，这也是导致 PC 及其他设备上应用程序出现安全问题的一个重要原因。

确保下标合法的一种有效手段就是尽可能使用范围 for 语句。

3.3.3 节练习

练习 3.16：编写一段程序，把练习 3.13 中 vector 对象的容量和具体内容输出出来。检验你之前的回答是否正确，如果不对，回过头重新学习 3.3.1 节（第 87 页）直到弄明白错在何处为止。

练习 3.17：从 cin 读入一组词并把它们存入一个 vector 对象，然后设法把所有词都改写为大写形式。输出改变后的结果，每个词占一行。

练习 3.18：下面的程序合法吗？如果不合法，你准备如何修改？

```
vector<int> ivec;
ivec[0] = 42;
```

练习 3.19：如果想定义一个含有 10 个元素的 vector 对象，所有元素的值都是 42，请列举出三种不同的实现方法。哪种方法更好呢？为什么？

练习 3.20：读入一组整数并把它们存入一个 vector 对象，将每对相邻整数的和输出出来。改写你的程序，这次要求先输出第 1 个和最后 1 个元素的和，接着输出第 2 个和倒数第 2 个元素的和，以此类推。

3.4 迭代器介绍

我们已经知道可以使用下标运算符来访问 string 对象的字符或 vector 对象的元素，还有另外一种更通用的机制也可以实现同样的目的，这就是**迭代器**（iterator）。在第 II 部分中将要介绍，除了 vector 之外，标准库还定义了其他几种容器。所有标准库容器都可以使用迭代器，但是其中只有少数几种才同时支持下标运算符。严格来说，string 对象不属于容器类型，但是 string 支持很多与容器类型类似的操作。vector 支持下标运算符，这点和 string 一样；string 支持迭代器，这也和 vector 是一样的。

106

类似于指针类型（参见 2.3.2 节，第 47 页），迭代器也提供了对对象的间接访问。就迭代器而言，其对象是容器中的元素或者 string 对象中的字符。使用迭代器可以访问某个元素，迭代器也能从一个元素移动到另外一个元素。迭代器有有效和无效之分，这一点和指针差不多。有效的迭代器或者指向某个元素，或者指向容器中尾元素的下一位置；其他所有情况都属于无效。

3.4.1 使用迭代器

和指针不一样的是，获取迭代器不是使用取地址符，有迭代器的类型同时拥有返回迭代器的成员。比如，这些类型都拥有名为 **begin** 和 **end** 的成员，其中 begin 成员负责返回指向第一个元素（或第一个字符）的迭代器。如有下述语句：

```
// 由编译器决定 b 和 e 的类型；参见 2.5.2 节（第 61 页）
// b 表示 v 的第一个元素，e 表示 v 尾元素的下一位置
auto b = v.begin(), e = v.end(); //b 和 e 的类型相同
```

end 成员则负责返回指向容器（或 string 对象）“尾元素的下一位置（one past the end）”的迭代器，也就是说，该迭代器指示的是容器的一个本不存在的“**尾后**（off the end）”元素。这样的迭代器没什么实际含义，仅是个标记而已，表示我们已经处理完了容器中的所有元素。end 成员返回的迭代器常被称作**尾后迭代器**（off-the-end iterator）或者简称为尾迭代器（end iterator）。特殊情况下如果容器为空，则 begin 和 end 返回的是同一个迭代器。

如果容器为空，则 begin 和 end 返回的是同一个迭代器，都是尾后迭代器。

一般来说，我们不清楚（不在意）迭代器准确的类型到底是什么。在上面的例子中，使用 auto 关键字定义变量 b 和 e（参见 2.5.2 节，第 61 页），这两个变量的类型也就是 begin 和 end 的返回值类型，第 97 页将对相关内容做更详细的介绍。

迭代器运算符

表 3.6 列举了迭代器支持的一些运算。使用==和!=来比较两个合法的迭代器是否相等，如果两个迭代器指向的元素相同或者都是同一个容器的尾后迭代器，则它们相等；否则就说这两个迭代器不相等。

表 3.6：标准容器迭代器的运算符	
*iter	返回迭代器 iter 所指元素的引用
iter->mem	解引用 iter 并获取该元素的名为 mem 的成员，等价于 (*iter).mem
++iter	令 iter 指示容器中的下一个元素
--iter	令 iter 指示容器中的上一个元素
iter1 == iter2 iter1 != iter2	判断两个迭代器是否相等（不相等），如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器，则相等；反之，不相等

107 和指针类似，也能通过解引用迭代器来获取它所指示的元素，执行解引用的迭代器必须合法并确实指示着某个元素（参见 2.3.2 节，第 48 页）。试图解引用一个非法迭代器或者尾后迭代器都是未被定义的行为。

举个例子，3.2.3 节（第 84 页）中的程序利用下标运算符把 string 对象的第一个字母改为了大写形式，下面利用迭代器实现同样的功能：

```
string s("some string");
if (s.begin() != s.end()) {  // 确保 s 非空
    auto it = s.begin();     // it 表示 s 的第一个字符
    *it = toupper(*it);      // 将当前字符改成大写形式
}
```

本例和原来的程序一样，首先检查 s 是否为空，显然通过检查 begin 和 end 返回的结果是否一致就能做到这一点。如果返回的结果一样，说明 s 为空；如果返回的结果不一样，说明 s 不为空，此时 s 中至少包含一个字符。

我们在 if 内部，声明了一个迭代器变量 it 并把 begin 返回的结果赋给它，这样就得到了指示 s 中第一个字符的迭代器，接下来通过解引用运算符将第一个字符更改为大写形式。和原来的程序一样，输出结果将是：

```
Some string
```

将迭代器从一个元素移动到另外一个元素

迭代器使用递增（++）运算符（参见 1.4.1 节，第 11 页）来从一个元素移动到下一个元素。从逻辑上来说，迭代器的递增和整数的递增类似，整数的递增是在整数值上“加 1”，迭代器的递增则是将迭代器“向前移动一个位置”。

因为 end 返回的迭代器并不实际指示某个元素，所以不能对其进行递增或解引用的操作。

之前有一个程序把 string 对象中第一个单词改写为大写形式，现在利用迭代器及其递增运算符可以实现相同的功能：

108
```
// 依次处理 s 的字符直至我们处理完全部字符或者遇到空白
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // 将当前字符改成大写形式
```

和 3.2.3 节（第 84 页）的那个程序一样，上面的循环也是遍历 s 的字符直到遇到空白字符为止，只不过之前的程序用的是下标运算符，现在这个程序用的是迭代器。

循环首先用 s.begin 的返回值来初始化 it，意味着 it 指示的是 s 中的第一个字符（如果有的话）。条件部分检查是否已到达 s 的尾部，如果尚未到达，则将 it 解引用的结

果传入 isspace 函数检查是否遇到了空白。每次迭代的最后，执行++it 令迭代器前移一个位置以访问 s 的下一个字符。

循环体内部和上一个程序 if 语句内的最后一句话一样，先解引用 it，然后将结果传入 toupper 函数得到该字母对应的大写形式，再把这个大写字母重新赋值给 it 所指示的字符。

关键概念：泛型编程

原来使用 C 或 Java 的程序员在转而使用 C++语言之后，会对 for 循环中使用!=而非<进行判断有点儿奇怪，比如上面的这个程序以及 85 页的那个。C++程序员习惯性地使用!=，其原因和他们更愿意使用迭代器而非下标的原因一样：因为这种编程风格在标准库提供的所有容器上都有效。

之前已经说过，只有 string 和 vector 等一些标准库类型有下标运算符，而并非全都如此。与之类似，所有标准库容器的迭代器都定义了==和!=，但是它们中的大多数都没有定义<运算符。因此，只要我们养成使用迭代器和!=的习惯，就不用太在意用的到底是哪种容器类型。

迭代器类型

就像不知道 string 和 vector 的 size_type 成员（参见 3.2.2 节，第 79 页）到底是什么类型一样，一般来说我们也不知道（其实是无须知道）迭代器的精确类型。而实际上，那些拥有迭代器的标准库类型使用 iterator 和 const_iterator 来表示迭代器的类型：

```
vector<int>::iterator it;     // it 能读写 vector<int>的元素
string::iterator it2;         // it2 能读写 string 对象中的字符

vector<int>::const_iterator it3; // it3 只能读元素，不能写元素
string::const_iterator it4;      // it4 只能读字符，不能写字符
```

const_iterator 和常量指针（参见 2.4.2 节，第 56 页）差不多，能读取但不能修改它所指的元素值。相反，iterator 的对象可读可写。如果 vector 对象或 string 对象是一个常量，只能使用 const_iterator；如果 vector 对象或 string 对象不是常量，那么既能使用 iterator 也能使用 const_iterator。

术语：迭代器和迭代器类型

迭代器这个名词有三种不同的含义：可能是迭代器概念本身，也可能是指容器定义的迭代器类型，还可能是指某个迭代器对象。

重点是理解存在一组概念上相关的类型，我们认定某个类型是迭代器当且仅当它支持一套操作，这套操作使得我们能访问容器的元素或者从某个元素移动到另外一个元素。

每个容器类定义了一个名为 iterator 的类型，该类型支持迭代器概念所规定的一套操作。

109

begin 和 end 运算符

begin 和 end 返回的具体类型由对象是否是常量决定，如果对象是常量，begin 和 end 返回 const_iterator；如果对象不是常量，返回 iterator：

```
vector<int> v;
```

```
const vector<int> cv;
auto it1 = v.begin();     // it1 的类型是 vector<int>::iterator
auto it2 = cv.begin();    // it2 的类型是 vector<int>::const_iterator
```

有时候这种默认的行为并非我们所要。在 6.2.3 节（第 191 页）中将会看到，如果对象只需读操作而无须写操作的话最好使用常量类型（比如 const_iterator）。为了便于专门得到 const_iterator 类型的返回值，C++11 新标准引入了两个新函数，分别是 cbegin 和 cend：

C++11

```
auto it3 = v.cbegin(); // it3 的类型是 vector<int>::const_iterator
```

类似于 begin 和 end，上述两个新函数也分别返回指示容器第一个元素或最后元素下一位置的迭代器。有所不同的是，不论 vector 对象（或 string 对象）本身是否是常量，返回值都是 const_iterator。

结合解引用和成员访问操作

解引用迭代器可获得迭代器所指的对象，如果该对象的类型恰好是类，就有可能希望进一步访问它的成员。例如，对于一个由字符串组成的 vector 对象来说，要想检查其元素是否为空，令 it 是该 vector 对象的迭代器，只需检查 it 所指字符串是否为空就可以了，其代码如下所示：

```
(*it).empty()
```

注意，(*it).empty()中的圆括号必不可少，具体原因将在 4.1.2 节（第 121 页）介绍，该表达式的含义是先对 it 解引用，然后解引用的结果再执行点运算符（参见 1.5.2 节，第 20 页）。如果不加圆括号，点运算符将由 it 来执行，而非 it 解引用的结果：

```
(*it).empty()    // 解引用 it，然后调用结果对象的 empty 成员
*it.empty()      // 错误：试图访问 it 的名为 empty 的成员，但 it 是个迭代器，
                 // 没有 empty 成员
```

110

上面第二个表达式的含义是从名为 it 的对象中寻找其 empty 成员，显然 it 是一个迭代器，它没有哪个成员是叫 empty 的，所以第二个表达式将发生错误。

为了简化上述表达式，C++语言定义了**箭头运算符**（->）。箭头运算符把解引用和成员访问两个操作结合在一起，也就是说，it->mem 和(*it).mem 表达的意思相同。

例如，假设用一个名为 text 的字符串向量存放文本文件中的数据，其中的元素或者是一句话或者是一个用于表示段落分隔的空字符串。如果要输出 text 中第一段的内容，可以利用迭代器写一个循环令其遍历 text，直到遇到空字符串的元素为止：

```
// 依次输出 text 的每一行直至遇到第一个空白行为止
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

我们首先初始化 it 令其指向 text 的第一个元素，循环重复执行直至处理完了 text 的所有元素或者发现某个元素为空。每次迭代时只要发现还有元素并且尚未遇到空元素，就输出当前正在处理的元素。值得注意的是，因为循环从头到尾只是读取 text 的元素而未向其中写值，所以使用了 cbegin 和 cend 来控制整个迭代过程。

某些对 vector 对象的操作会使迭代器失效

3.3.2 节（第 90 页）曾经介绍过，虽然 vector 对象可以动态地增长，但是也会有一

些副作用。已知的一个限制是不能在范围 for 循环中向 vector 对象添加元素。另外一个限制是任何一种可能改变 vector 对象容量的操作，比如 push_back，都会使该 vector 对象的迭代器失效。9.3.6 节（第 315 页）将详细解释迭代器是如何失效的。

谨记，但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。

3.4.1 节练习

练习 3.21：请使用迭代器重做 3.3.3 节（第 94 页）的第一个练习。

练习 3.22：修改之前那个输出 text 第一段的程序，首先把 text 的第一段全都改成大写形式，然后再输出它。

练习 3.23：编写一段程序，创建一个含有 10 个整数的 vector 对象，然后使用迭代器将所有元素的值都变成原来的两倍。输出 vector 对象的内容，检验程序是否正确。

3.4.2 迭代器运算

迭代器的递增运算令迭代器每次移动一个元素，所有的标准库容器都有支持递增运算的迭代器。类似的，也能用==和!=对任意标准库类型的两个有效迭代器（参见 3.4 节，第 95 页）进行比较。

111

string 和 vector 的迭代器提供了更多额外的运算符，一方面可使得迭代器的每次移动跨过多个元素，另外也支持迭代器进行关系运算。所有这些运算被称作**迭代器运算**（iterator arithmetic），其细节由表 3.7 列出。

表 3.7：vector 和 string 迭代器支持的运算

iter + n	迭代器加上一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向前移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置
iter - n	迭代器减去一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向后移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置
iter1 += n	迭代器加法的复合赋值语句，将 iter1 加 n 的结果赋给 iter1
iter1 -= n	迭代器减法的复合赋值语句，将 iter1 减 n 的结果赋给 iter1
iter1 - iter2	两个迭代器相减的结果是它们之间的距离，也就是说，将运算符右侧的迭代器向前移动差值个元素后将得到左侧的迭代器。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置
>、 >=、 <、 <=	迭代器的关系运算符，如果某迭代器指向的容器位置在另一个迭代器所指位置之前，则说前者小于后者。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置

迭代器的算术运算

可以令迭代器和一个整数值相加（或相减），其返回值是向前（或向后）移动了若干个位置的迭代器。执行这样的操作时，结果迭代器或者指示原 vector 对象（或 string 对象）内的一个元素，或者指示原 vector 对象（或 string 对象）尾元素的下一位置。

举个例子，下面的代码得到一个迭代器，它指向某 vector 对象中间位置的元素：

```
// 计算得到最接近 vi 中间元素的一个迭代器
auto mid = vi.begin() + vi.size() / 2;
```

如果 vi 有 20 个元素，vi.size()/2 得 10，此例中即令 mid 等于 vi.begin()+10。已知下标从 0 开始，则迭代器所指的元素是 vi[10]，也就是从首元素开始向前相隔 10 个位置的那个元素。

对于 string 或 vector 的迭代器来说，除了判断是否相等，还能使用关系运算符(<、<=、>、>=)对其进行比较。参与比较的两个迭代器必须合法而且指向的是同一个容器的元素（或者尾元素的下一位置）。例如，假设 it 和 mid 是同一个 vector 对象的两个迭代器，可以用下面的代码来比较它们所指的位置孰前孰后：

```
if (it < mid)
    // 处理 vi 前半部分的元素
```

112 只要两个迭代器指向的是同一个容器中的元素或者尾元素的下一位置，就能将其相减，所得结果是两个迭代器的距离。所谓距离指的是右侧的迭代器向前移动多少位置就能追上左侧的迭代器，其类型是名为 **difference_type** 的带符号整型数。string 和 vector 都定义了 difference_type，因为这个距离可正可负，所以 difference_type 是带符号类型的。

使用迭代器运算

使用迭代器运算的一个经典算法是二分搜索。二分搜索从有序序列中寻找某个给定的值。二分搜索从序列中间的位置开始搜索，如果中间位置的元素正好就是要找的元素，搜索完成；如果不是，假如该元素小于要找的元素，则在序列的后半部分继续搜索；假如该元素大于要找的元素，则在序列的前半部分继续搜索。在缩小的范围中计算一个新的中间元素并重复之前的过程，直至最终找到目标或者没有元素可供继续搜索。

下面的程序使用迭代器完成了二分搜索：

```
// text 必须是有序的
// beg 和 end 表示我们搜索的范围
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // 初始状态下的中间点
// 当还有元素尚未检查并且我们还没有找到 sought 时执行循环
while (mid != end && *mid != sought) {
    if (sought < *mid)     // 我们要找的元素在前半部分吗?
        end = mid;         // 如果是，调整搜索范围使得忽略掉后半部分
    else                   // 我们要找的元素在后半部分
        beg = mid + 1;     // 在 mid 之后寻找
    mid = beg + (end - beg)/2;          // 新的中间点
}
```

程序的一开始定义了三个迭代器：beg 指向搜索范围内的第一个元素、end 指向尾元素的下一位置、mid 指向中间的那个元素。初始状态下，搜索范围是名为 text 的 vector<string>的全部范围。

循环部分先检查搜索范围是否为空，如果 mid 和 end 的当前值相等，说明已经找遍了所有元素。此时条件不满足，循环终止。当搜索范围不为空时，可知 mid 指向了某个元素，检查该元素是否就是我们所要搜索的，如果是，也终止循环。

当进入到循环体内部后，程序通过某种规则移动 beg 或者 end 来缩小搜索的范围。如果 mid 所指的元素比要找的元素 sought 大，可推测若 text 含有 sought，则必出现在 mid 所指元素的前面。此时，可以忽略 mid 后面的元素不再查找，并把 mid 赋给 end 即可。另一种情况，如果*mid 比 sought 小，则要找的元素必出现在 mid 所指元素的后面。此时，通过令 beg 指向 mid 的下一个位置即可改变搜索范围。因为已经验证过 mid 不是我们要找的对象，所以在接下来的搜索中不必考虑它。

循环过程终止时，mid 或者等于 end 或者指向要找的元素。如果 mid 等于 end，说明 text 中没有我们要找的元素。

3.4.2 节练习

113

练习 3.24：请使用迭代器重做 3.3.3 节（第 94 页）的最后一个练习。

练习 3.25：3.3.3 节（第 93 页）划分分数段的程序是使用下标运算符实现的，请利用迭代器改写该程序并实现完全相同的功能。

练习 3.26：在 100 页的二分搜索程序中，为什么用的是 `mid = beg + (end - beg) / 2`，而非 `mid = (beg + end) /2;`？

3.5 数组

数组是一种类似于标准库类型 vector（参见 3.3 节，第 86 页）的数据结构，但是在性能和灵活性的权衡上又与 vector 有所不同。与 vector 相似的地方是，数组也是存放类型相同的对象的容器，这些对象本身没有名字，需要通过其所在位置访问。与 vector 不同的地方是，数组的大小确定不变，不能随意向数组中增加元素。因为数组的大小固定，因此对某些特殊的应用来说程序的运行时性能较好，但是相应地也损失了一些灵活性。

如果不清楚元素的确切个数，请使用 vector。

3.5.1 定义和初始化内置数组

数组是一种复合类型（参见 2.3 节，第 45 页）。数组的声明形如 a[d]，其中 a 是数组的名字，d 是数组的维度。维度说明了数组中元素的个数，因此必须大于 0。数组中元素的个数也属于数组类型的一部分，编译的时候维度应该是已知的。也就是说，维度必须是一个常量表达式（参见 2.4.4 节，第 58 页）：

```
unsigned cnt = 42;          // 不是常量表达式
constexpr unsigned sz = 42; // 常量表达式，关于 constexpr，参见 2.4.4 节（第 59 页）
int arr[10];                // 含有 10 个整数的数组
int *parr[sz];              // 含有 42 个整型指针的数组
string bad[cnt];            // 错误：cnt 不是常量表达式
string strs[get_size()];    // 当 get_size 是 constexpr 时正确；否则错误
```

默认情况下，数组的元素被默认初始化（参见 2.2.1 节，第 40 页）。

和内置类型的变量一样，如果在函数内部定义了某种内置类型的数组，那么默认初始化会令数组含有未定义的值。

定义数组的时候必须指定数组的类型，不允许用 auto 关键字由初始值的列表推断类型。另外和 vector 一样，数组的元素应为对象，因此不存在引用的数组。

114

显式初始化数组元素

可以对数组的元素进行列表初始化（参见 3.3.1 节，第 88 页），此时允许忽略数组的维度。如果在声明时没有指明维度，编译器会根据初始值的数量计算并推测出来；相反，如果指明了维度，那么初始值的总数量不应该超出指定的大小。如果维度比提供的初始值数量大，则用提供的初始值初始化靠前的元素，剩下的元素被初始化成默认值（参见 3.3.1 节，第 88 页）：

```
const unsigned sz = 3;
int ia1[sz] = {0, 1, 2};          // 含有 3 个元素的数组,元素值分别是 0, 1, 2
int a2[] = {0, 1, 2};             // 维度是 3 的数组
int a3[5] = {0, 1, 2};            // 等价于 a3[] = {0, 1, 2, 0, 0}
string a4[3] = {"hi", "bye"};     // 等价于 a4[] = {"hi", "bye", ""}
int a5[2] = {0,1,2};              // 错误：初始值过多
```

字符数组的特殊性

字符数组有一种额外的初始化形式，我们可以用字符串字面值（参见 2.1.3 节，第 36 页）对此类数组初始化。当使用这种方式时，一定要注意字符串字面值的结尾处还有一个空字符，这个空字符也会像字符串的其他字符一样被拷贝到字符数组中去：

```
char a1[] = {'C', '+', '+'};        // 列表初始化，没有空字符
char a2[] = {'C', '+', '+', '\0'};  // 列表初始化，含有显式的空字符
char a3[] = "C++";                  // 自动添加表示字符串结束的空字符
const char a4[6] = "Daniel";        // 错误：没有空间可存放空字符!
```

a1 的维度是 3，a2 和 a3 的维度都是 4，a4 的定义是错误的。尽管字符串字面值"Daniel"看起来只有 6 个字符，但是数组的大小必须至少是 7，其中 6 个位置存放字面值的内容，另外 1 个存放结尾处的空字符。

不允许拷贝和赋值

不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值：

```
int a[] = {0, 1, 2};        // 含有 3 个整数的数组
int a2[] = a;               // 错误：不允许使用一个数组初始化另一个数组
a2 = a;                     // 错误：不能把一个数组直接赋值给另一个数组
```

一些编译器支持数组的赋值，这就是所谓的**编译器扩展**（compiler extension）。但一般来说，最好避免使用非标准特性，因为含有非标准特性的程序很可能在其他编译器上无法正常工作。

理解复杂的数组声明

和 vector 一样，数组能存放大多数类型的对象。例如，可以定义一个存放指针的数组。又因为数组本身就是对象，所以允许定义数组的指针及数组的引用。在这几种情况中，定义存放指针的数组比较简单和直接，但是定义数组的指针或数组的引用就稍微复杂一点了：

115

```
int *ptrs[10];              // ptrs 是含有 10 个整型指针的数组
int &refs[10] = /* ? */;    // 错误：不存在引用的数组
int (*Parray)[10] = &arr;   // Parray 指向一个含有 10 个整数的数组
int (&arrRef)[10] = arr;    // arrRef 引用一个含有 10 个整数的数组
```

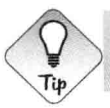
默认情况下，类型修饰符从右向左依次绑定。对于 ptrs 来说，从右向左（参见 2.3.3 节，第 52 页）理解其含义比较简单：首先知道我们定义的是一个大小为 10 的数组，它的名字是 ptrs，然后知道数组中存放的是指向 int 的指针。

但是对于 Parray 来说，从右向左理解就不太合理了。因为数组的维度是紧跟着被声明的名字的，所以就数组而言，由内向外阅读要比从右向左好多了。由内向外的顺序可帮助我们更好地理解 Parray 的含义：首先是圆括号括起来的部分，*Parray 意味着 Parray 是个指针，接下来观察右边，可知道 Parray 是个指向大小为 10 的数组的指针，最后观察左边，知道数组中的元素是 int。这样最终的含义就明白无误了，Parray 是一个指针，它指向一个 int 数组，数组中包含 10 个元素。同理，(&arrRef) 表示 arrRef 是一个引用，它引用的对象是一个大小为 10 的数组，数组中元素的类型是 int。

当然，对修饰符的数量并没有特殊限制：

```
int *(&arry)[10] = ptrs; // arry 是数组的引用，该数组含有 10 个指针
```

按照由内向外的顺序阅读上述语句，首先知道 arry 是一个引用，然后观察右边知道，arry 引用的对象是一个大小为 10 的数组，最后观察左边知道，数组的元素类型是指向 int 的指针。这样，arry 就是一个含有 10 个 int 型指针的数组的引用。



要想理解数组声明的含义，最好的办法是从数组的名字开始按照由内向外的顺序阅读。

3.5.1 节练习

练习 3.27：假设 txt_size 是一个无参数的函数，它的返回值是 int。请回答下列哪个定义是非法的？为什么？

```
unsigned buf_size = 1024;
(a) int ia[buf_size];          (b) int ia[4 * 7 - 14];
(c) int ia[txt_size()];        (d) char st[11] = "fundamental";
```

练习 3.28：下列数组中元素的值是什么？

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int    ia2[10];
}
```

练习 3.29：相比于 vector 来说，数组有哪些缺点，请列举一些。

3.5.2 访问数组元素

116

与标准库类型 vector 和 string 一样，数组的元素也能使用范围 for 语句或下标运算符来访问。数组的索引从 0 开始，以一个包含 10 个元素的数组为例，它的索引从 0 到 9，而非从 1 到 10。

在使用数组下标的时候，通常将其定义为 **size_t** 类型。size_t 是一种机器相关的无符号类型，它被设计得足够大以便能表示内存中任意对象的大小。在 cstddef 头文件中定义了 size_t 类型，这个文件是 C 标准库 stddef.h 头文件的 C++语言版本。

数组除了大小固定这一特点外，其他用法与 vector 基本类似。例如，可以用数组来记录各分数段的成绩个数，从而实现与 3.3.3 节（第 93 页）的程序一样的功能：

```
// 以 10 分为一个分数段统计成绩的数量：  0～9, 10～19, ..., 90～99, 100
unsigned scores[11] = {};      // 11 个分数段，全部初始化为 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10];  // 将当前分数段的计数值加 1
}
```

与 93 页的程序相比，上面程序最大的不同是 scores 的声明。这里 scores 是一个含有 11 个无符号元素的数组。另外一处不太明显的区别是，本例所用的下标运算符是由 C++ 语言直接定义的，这个运算符能用在数组类型的运算对象上。93 页的那个程序所用的下标运算符是库模板 vector 定义的，只能用于 vector 类型的运算对象。

与 vector 和 string 一样，当需要遍历数组的所有元素时，最好的办法也是使用范围 for 语句。例如，下面的程序输出所有的 scores：

```
for (auto i : scores)          // 对于 scores 中的每个计数值
     cout << i << " ";         // 输出当前的计数值
cout << endl;
```

因为维度是数组类型的一部分，所以系统知道数组 scores 中有多少个元素，使用范围 for 语句可以减轻人为控制遍历过程的负担。

检查下标的值

与 vector 和 string 一样，数组的下标是否在合理范围之内由程序员负责检查，所谓合理就是说下标应该大于等于 0 而且小于数组的大小。要想防止数组下标越界，除了小心谨慎注意细节以及对代码进行彻底的测试之外，没有其他好办法。对于一个程序来说，即使顺利通过编译并执行，也不能肯定它不包含此类致命的错误。

大多数常见的安全问题都源于缓冲区溢出错误。当数组或其他类似数据结构的下标越界并试图访问非法内存区域时，就会产生此类错误。

117

3.5.2 节练习

练习 3.30：指出下面代码中的索引错误。

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
  ia[ix] = ix;
```

练习 3.31：编写一段程序，定义一个含有 10 个 int 的数组，令每个元素的值就是其下标值。

练习 3.32：将上一题刚刚创建的数组拷贝给另外一个数组。利用 vector 重写程序，实现类似的功能。

练习 3.33：对于 104 页的程序来说，如果不初始化 scores 将发生什么？

3.5.3 指针和数组

在C++语言中，指针和数组有非常紧密的联系。就如即将介绍的，使用数组的时候编译器一般会把它转换成指针。

通常情况下，使用取地址符（参见2.3.2节，第47页）来获取指向某个对象的指针，取地址符可以用于任何对象。数组的元素也是对象，对数组使用下标运算符得到该数组指定位置的元素。因此像其他对象一样，对数组的元素使用取地址符就能得到指向该元素的指针：

```
string nums[] = {"one", "two", "three"}; // 数组的元素是 string 对象
string *p = &nums[0];                    // p 指向 nums 的第一个元素
```

然而，数组还有一个特性：在很多用到数组名字的地方，编译器都会自动地将其替换为一个指向数组首元素的指针：

```
string *p2 = nums; // 等价于 p2 = &nums[0]
```

在大多数表达式中，使用数组类型的对象其实是使用一个指向该数组首元素的指针。

由上可知，在一些情况下数组的操作实际上是指针的操作，这一结论有很多隐含的意思。其中一层意思是当使用数组作为一个auto（参见2.5.2节，第61页）变量的初始值时，推断得到的类型是指针而非数组：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia 是一个含有 10 个整数的数组
auto ia2(ia);    // ia2 是一个整型指针，指向 ia 的第一个元素
ia2 = 42;        // 错误：ia2 是一个指针，不能用 int 值给指针赋值
```

尽管ia是由10个整数构成的数组，但当使用ia作为初始值时，编译器实际执行的初始化过程类似于下面的形式：

```
auto ia2(&ia[0]); //显然 ia2 的类型是 int*
```

118

必须指出的是，当使用decltype关键字（参见2.5.3节，第62页）时上述转换不会发生，decltype(ia)返回的类型是由10个整数构成的数组：

```
// ia3 是一个含有 10 个整数的数组
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p;      // 错误：不能用整型指针给数组赋值
ia3[4] = i; // 正确：把 i 的值赋给 ia3 的一个元素
```

指针也是迭代器

与2.3.2节（第47页）介绍的内容相比，指向数组元素的指针拥有更多功能。vector和string的迭代器（参见3.4节，第95页）支持的运算，数组的指针全都支持。例如，允许使用递增运算符将指向数组元素的指针向前移动到下一个位置上：

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr;     // p 指向 arr 的第一个元素
++p;              // p 指向 arr[1]
```

就像使用迭代器遍历vector对象中的元素一样，使用指针也能遍历数组中的元素。当然，这样做的前提是先得获取到指向数组第一个元素的指针和指向数组尾元素的下一位置的指针。之前已经介绍过，通过数组名字或者数组中首元素的地址都能得到指向首元素的指针；不过获取尾后指针就要用到数组的另外一个特殊性质了。我们可以设法获取数组

尾元素之后的那个并不存在的元素的地址：

```
int *e = &arr[10]; // 指向 arr 尾元素的下一位置的指针
```

这里显然使用下标运算符索引了一个不存在的元素，arr 有 10 个元素，尾元素所在位置的索引是 9，接下来那个不存在的元素唯一的用处就是提供其地址用于初始化 e。就像尾后迭代器（参见 3.4.1 节，第 95 页）一样，尾后指针也不指向具体的元素。因此，不能对尾后指针执行解引用或递增的操作。

利用上面得到的指针能重写之前的循环，令其输出 arr 的全部元素：

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // 输出 arr 的元素
```

标准库函数 begin 和 end

尽管能计算得到尾后指针，但这种用法极易出错。为了让指针的使用更简单、更安全，C++11 新标准引入了两个名为 begin 和 end 的函数。这两个函数与容器中的两个同名成员（参见 3.4.1 节，第 95 页）功能类似，不过数组毕竟不是类类型，因此这两个函数不是成员函数。正确的使用形式是将数组作为它们的参数：

C++11

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia 是一个含有 10 个整数的数组
int *beg = begin(ia);             // 指向 ia 首元素的指针
int *last = end(ia);              // 指向 arr 尾元素的下一位置的指针
```

119 begin 函数返回指向 ia 首元素的指针，end 函数返回指向 ia 尾元素下一位置的指针，这两个函数定义在 iterator 头文件中。

使用 begin 和 end 可以很容易地写出一个循环并处理数组中的元素。例如，假设 arr 是一个整型数组，下面的程序负责找到 arr 中的第一个负数：

```
// pbeg 指向 arr 的首元素，pend 指向 arr 尾元素的下一位置
int *pbeg = begin(arr), *pend = end(arr);
// 寻找第一个负值元素，如果已经检查完全部元素则结束循环
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

首先定义了两个名为 pbeg 和 pend 的整型指针，其中 pbeg 指向 arr 的第一个元素，pend 指向 arr 尾元素的下一位置。while 语句的条件部分通过比较 pbeg 和 pend 来确保可以安全地对 pbeg 解引用，如果 pbeg 确实指向了一个元素，将其解引用并检查元素值是否为负值。如果是，条件失效、退出循环；如果不是，将指针向前移动一位继续考查下一个元素。

一个指针如果指向了某种内置类型数组的尾元素的“下一位置”，则其具备与 vector 的 end 函数返回的与迭代器类似的功能。特别要注意，尾后指针不能执行解引用和递增操作。

指针运算

指向数组元素的指针可以执行表 3.6（第 96 页）和表 3.7（第 99 页）列出的所有迭代器运算。这些运算，包括解引用、递增、比较、与整数相加、两个指针相减等，用在指针和用在迭代器上意义完全一致。

给（从）一个指针加上（减去）某整数值，结果仍是指针。新指针指向的元素与原来

的指针相比前进了（后退了）该整数值个位置：

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr;        // 等价于 int *ip = &arr[0]
int *ip2 = ip + 4;    // ip2 指向 arr 的尾元素 arr[4]
```

ip 加上 4 所得的结果仍是一个指针，该指针所指的元素与 ip 原来所指的元素相比前进了 4 个位置。

给指针加上一个整数，得到的新指针仍需指向同一数组的其他元素，或者指向同一数组的尾元素的下一位置：

```
// 正确：arr 转换成指向它首元素的指针；p 指向 arr 尾元素的下一位置
int *p = arr + sz;        // 使用警告：不要解引用！
int *p2 = arr + 10;       // 错误：arr 只有 5 个元素，p2 的值未定义
```

当给 arr 加上 sz 时，编译器自动地将 arr 转换成指向数组 arr 中首元素的指针。执行加法后，指针从首元素开始向前移动了 sz（这里是 5）个位置，指向新位置的元素。也就是说，它指向了数组 arr 尾元素的下一位置。如果计算所得的指针超出了上述范围就将产生错误，而且这种错误编译器一般发现不了。 120

和迭代器一样，两个指针相减的结果是它们之间的距离。参与运算的两个指针必须指向同一个数组当中的元素：

```
auto n = end(arr) - begin(arr); // n 的值是 5，也就是 arr 中元素的数量
```

两个指针相减的结果的类型是一种名为 **ptrdiff_t** 的标准库类型，和 size_t 一样，ptrdiff_t 也是一种定义在 cstddef 头文件中的机器相关的类型。因为差值可能为负值，所以 ptrdiff_t 是一种带符号类型。

只要两个指针指向同一个数组的元素，或者指向该数组的尾元素的下一位置，就能利用关系运算符对其进行比较。例如，可以按照如下的方式遍历数组中的元素：

```
int *b = arr, *e = arr + sz;
while (b < e) {
    // 使用 *b
    ++b;
}
```

如果两个指针分别指向不相关的对象，则不能比较它们：

```
int i = 0, sz = 42;
int *p = &i, *e = &sz;
// 未定义的：p 和 e 无关，因此比较毫无意义！
while (p < e)
```

尽管作用可能不是特别明显，但必须说明的是，上述指针运算同样适用于空指针（参见 2.3.2 节，第 48 页）和所指对象并非数组的指针。在后一种情况下，两个指针必须指向同一个对象或该对象的下一位置。如果 p 是空指针，允许给 p 加上或减去一个值为 0 的整型常量表达式（参见 2.4.4 节，第 58 页）。两个空指针也允许彼此相减，结果当然是 0。

解引用和指针运算的交互

指针加上一个整数所得的结果还是一个指针。假设结果指针指向了一个元素，则允许解引用该结果指针：

```
int ia[] = {0,2,4,6,8};  // 含有 5 个整数的数组
int last = *(ia + 4);    // 正确：把 last 初始化成 8，也就是 ia[4]的值
```

表达式*(ia+4)计算 ia 前进 4 个元素后的新地址，解引用该结果指针的效果等价于表达式 ia[4]。

回忆一下在 3.4.1 节（第 98 页）中介绍过如果表达式含有解引用运算符和点运算符，最好在必要的地方加上圆括号。类似的，此例中指针加法的圆括号也不可缺少。如果写成下面的形式：

```
last = *ia + 4; // 正确： last =4 等价于 ia[0] +4
```

含义就与之前完全不同了，此时先解引用 ia，然后给解引用的结果再加上 4。4.1.2 节（第 121 页）将对这一问题做进一步分析。

下标和指针

121

如前所述，在很多情况下使用数组的名字其实用的是一个指向数组首元素的指针。一个典型的例子是当对数组使用下标运算符时，编译器会自动执行上述转换操作。给定

```
int ia[] = {0,2,4,6,8}; // 含有 5 个整数的数组
```

此时，ia[0]是一个使用了数组名字的表达式，对数组执行下标运算其实是对指向数组元素的指针执行下标运算：

```
int i = ia[2];        // ia 转换成指向数组首元素的指针
                      // ia[2]得到(ia + 2) 所指的元素
int *p = ia;          // p 指向 ia 的首元素
i = *(p + 2);         // 等价于 i = ia[2]
```

只要指针指向的是数组中的元素（或者数组中尾元素的下一位置），都可以执行下标运算：

```
int *p = &ia[2];      // p 指向索引为 2 的元素
int j = p[1];         // p[1]等价于 *(p + 1)，就是 ia[3]表示的那个元素
int k = p[-2];        // p[-2]是 ia[0]表示的那个元素
```

虽然标准库类型 string 和 vector 也能执行下标运算，但是数组与它们相比还是有所不同。标准库类型限定使用的下标必须是无符号类型，而内置的下标运算无此要求，上面的最后一个例子很好地说明了这一点。内置的下标运算符可以处理负值，当然，结果地址必须指向原来的指针所指同一数组中的元素（或是同一数组尾元素的下一位置）。

内置的下标运算符所用的索引值不是无符号类型，这一点与 vector 和 string 不一样。

3.5.3 节练习

练习 3.34：假定 p1 和 p2 指向同一个数组中的元素，则下面程序的功能是什么？什么情况下该程序是非法的？

```
p1 += p2 - p1;
```

练习 3.35：编写一段程序，利用指针将数组中的元素置为 0。

练习 3.36：编写一段程序，比较两个数组是否相等。再写一段程序，比较两个 vector 对象是否相等。

3.5.4 C 风格字符串

122

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。

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
```

123

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：

```
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。