本章介绍了C++的Vector、Stack、Queue、Map和Set类，它们一起代表了存储集合的一个强大的框架。目前，你只需要从用户的角度来看待这些类。在后续的章节中，你将有机会更深入地学习它们是如何实现的。鉴于当你完成本书的学习后，可能会实现这些集合类，虽然它们也提供了一些非常类似的方法集合，但这里介绍的类都是标准模板库中vector、stack、queue、map和 set类的某种程度上的简化。

本章重点包括：

- 根据其行为而不是其表示进行定义的数据结构被称为抽象数据类型（abstract datatype）。与基本数据类型相比，抽象数据类型有一些重要的优点。这些优点包括简单性、灵活性和安全性。
- 包含其他对象并作为一个完整集合的元素类被称为集合类（collection classe）。在C++中，集合类的定义使用了模板（template），即参数化类型（parameterized type）， 其中元素的类型名出现在集合类名字之后的尖括号中。例如，类`Vector<int>`表示一个包含元素值类型为int的Vector类。
- 矢量Vector类是一种抽象数据类型，它的行为与一个一维数组很像，但更加强大。和数组不一样的是，一个Vector对象可以随着元素的增加和减少其尺寸可动态地变化。Vector类也更加安全，因为它检查确保所有的索引都在其范围中。**尽管你可以使用Vector对象中包含多个Vector对象来创建一个二维结构，但是使用Stanford类库中的Grid类将更加简单。**
- 栈Stack类表示了一种对象的集合，这些对象的行为表现为从一个栈中删除元素的方向与向栈中添加元素的方向相反：即后进先出（LIFO）。Stack类的基本操作是push，也就是向栈中添加一个元素，另一个基本操作是pop，即删除并返回最近添加的元素值。
- 队列Queue类和栈Stack类相似，但有一点不同。从一个队列中删除元素和添加元素的顺序相同：即先进先出（FIFO）。一个队列的基本操作是enqueue，也就是在队列的末尾添加一个元素，另一个基本操作是dequeue，即从队列的开始删除一个元素并且返回该元素值。
- Map类在某种程度上实现了键（key）与值（value）的关联，以便能够高效地检索这些关联。一个Map对象的基本操作是put，也就是向Map对象中添加一个键–值对，另一个基本操作是get，即返回一个特定的键所关联的值。
- Set类表示一个集合，和数学中的集合一样，这个集合中的元素是无序的并且每个元素只能出现一次。一个Set对象的基本操作包含了add，也就是向Set对象中添加一个新的元素，同时也包含了contains，即检查一个元素是否已存在于Set对象中。
- **除了Stack和Queue类**，所有的集合类都支持foreach模式，它使循环遍历集合中的元素变得很简单。和在“迭代顺序”这一节中所描述的一样，每一个集合类都定义了自己的元素迭代顺序。
- 另外，对于Map类和Set类，Stanford类库都提供了非常相关联的HashMap和 HashSet 类。Map类与 HashMap类的唯一不同（或者Set与HashSet类之间）在于其基于范围的循环的迭代元素顺序不同。Map和Set以其元素类型值的升序来迭代元素，而HashMap和HashSet类更高效，它们似乎以随机顺序来迭代元素。

顺序容器 Vector

最有价值的集合类之一就是Vector类，**它提供了一种类似于你在早期编程中曾遇到过的具有数组功能的机制。**早期的编程大量使用数组。和大多数编程语言一样，C++也支持数组。然而，C++中的数组有若干缺点，主要包括：

1. 数组被分配具有固定大小的内存，以致于程序员不能在以后对其大小进行改变。
2. 即使数组有固定的大小，C++也不允许程序员获得这个大小。因此，典型的含有数组的程序需要一个附加的变量来记录数组元素的个数。
3. 传统的数组不支持插入和删除数组元素的操作。
4. C++对数组越界不做检查。例如：如果你创建了一个含有25个元素的数组，之后你试图挑选出索引为50的数组元素值，C++仅查看在索引为50内存地址中是否有数据存在。

Vector类通过以抽象数据类型的方式重新实现数组解决了上述问题。你可以在任何应用中使用Vector类代替传统的数组，通常在源代码中只需进行很少的修改和较小的删减就会产生出人意料的高效结果。实际上，一旦你有了Vector类，在很多场景中就不会再使用数组，除非你实际上必须实现和Vector一样的类，这个类使用数组作为它隐藏的数据结构。然而，作为一个Vector类的用户，你可能不会对其隐藏的数据结构感兴趣，而把相关的数组结构问题留给程序员，让他们去实现这个抽象数据类型。

作为一个Vector类的用户，你会面临一组不同的问题并且需要回答以下问题：

1. 如何指定包含在一个Vector中对象的类型？
2. 如何创建一个对象，它是一个vector类的实例？
3. 在Vector中存在什么样的方法来实现它的抽象行为？

表 5-1 Vector.h 接口中的条目

构造函数	
`Vector<type> ()`	创建一个空的 Vector 对象
`Vector<type> (n, value);`	创建一个含有 *n* 个元素的 Vector 对象，每个元素都被初始化为 *value*，如果 *value* 实参省略，则取 *value* 类型的默认值

（续）

方法	
`size()`	返回 Vector 对象中元素的个数
`isEmpty()`	若 Vector 对象为空，返回 **true**
`get(index)`	返回指定索引位置 *index* 的元素。尝试获取 Vector 对象边界外的元素会发生错误
`set(index, value)`	给指定位置 *index* 的元素设置新的值 *value*。尝试给 Vector 对象边界外的元素设置新值会发生错误
`add(value)`	在 Vector 对象的尾部添加一个新的元素 *value*
`insertAt(index, value)`	在指定位置 *index* 之前插入一个新值 *value*
`removeAt(index)`	删除指定位置 *index* 上的元素
`clear()`	删除 Vector 对象中的所有元素

操作符

操作符	
vec[*index*]	查找指定位置 *index* 上的元素。尝试获取 Vector 对象边界外的元素会发生错误
v_1 + v_2	连接 v_1 和 v_2，返回一个包含 v_1 和 v_2 中的所有元素的 Vector 对象
vec += e_1, e_2, ...	在 Vector 对象的尾部添加元素 e_1、e_2 等

保存二维结构Grid

尽管使用嵌套的 Vector 对象能够代表二维结构，但这种方法很不便利。为了简化那些需要二维结构的应用开发，Stanford 集合类库提供了一种名为 Grid 的类，尽管在标准模板库中并没有对应的类。grid.h 中的条目如表 5-2 所示。

表 5-2 grid.h 接口中的条目

构造函数	
Grid<*type***>()**	创建一个空的 grid 对象。那些使用默认构造函数的用户需要通过调用 **resize** 方法指定 grid 对象的维数
Grid<*type***>(***rows, cols***)**	创建一个指定具体行数和列数的 grid 对象。其中每个元素被初始化为元素类型的默认值
方法	
numRows()	返回 grid 对象中的水平行数
numCols()	返回 grid 对象中的垂直列数
inBounds(*row, col***)**	如果指定的行和列组成的坐标在 grid 对象中，返回 true
get(*row, col***)**	返回 grid 对象中指定的行和列所在的元素
set(*row, col, value***)**	给指定坐标位置上的元素设置新的值 value
resize(*rows, cols***)**	通过给定的行数 *rows* 以及列数 *cols* 来改变 grid 对象的尺寸。grid 对象中先前的内容都被丢弃
操作符	
grid[*row*][*col*]	在 grid 对象中查找指定行和列所在的元素

Stack与小型计算器

5.2.1 Stack 类结构

和 Vector 类以及 Grid 类一样，Stack 类也是一个需要指明元素类型的集合类。例如，Stack<int> 代表了一个元素类型为整型的栈，Stack<string> 代表了一个元素类型为字符串的栈。类似地，如果你定义了类 Plate 以及 Frame，你可以使用这些类的对象作为元素来创建 Stack<Plate> 和 Stack<Frame> 对象。stack.h 中的条目如表 5-3 所示。

表 5-3 stack.h 接口中的条目

构造函数	
Stack<*type*>()	创建一个空的可存储指定类型值的栈对象
方法	
size()	返回当前栈中的元素个数
isEmpty()	如果栈为空，则返回 **true**
push(*value*)	将值 value 压入栈顶
pop()	将栈顶元素出栈，并且将栈顶元素返回给调用者。在空栈中调用 **pop** 方法会产生错误
peek()	返回栈顶元素的值但并不出栈。在空栈中调用 **peek** 方法会产生错误
clear()	删除栈中的所有元素

小型计算机简介

在逆波兰表示法中，操作符在那些它们作用的操作数之后输入。例如，为了用逆波兰式计算器计算表达式

```
8.5 * 4.4 + 6.9 / 1.5
```

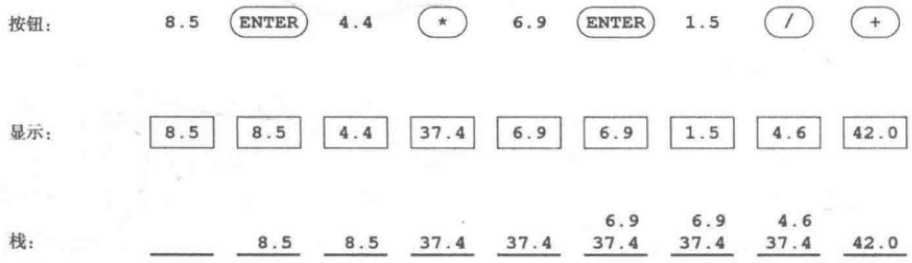
的结果，你将会按照下面的顺序输入操作数和操作符：

8.5 (ENTER) 4.4 (*) 6.9 (ENTER) 1.5 (/) (+)

当按下 ENTER 按钮时，计算器获得了先前的值并将该值压入栈。当按下操作符按钮时，计算器首先会检查用户是否已经输入了一个值，如果已经输入了，那么会自动地将其从栈中弹出。然后通过以下步骤计算应用操作符的结果：

- 从栈顶弹出两个值。
- 对这些值进行由按钮所指定的算术操作。
- 将结果压回到栈。

除了当用户在输入数值时，计算器总是显示栈顶的值。因此，图 5-3 显示了在计算中的每一时刻，计算器所显示的内容以及栈中所包含的值。



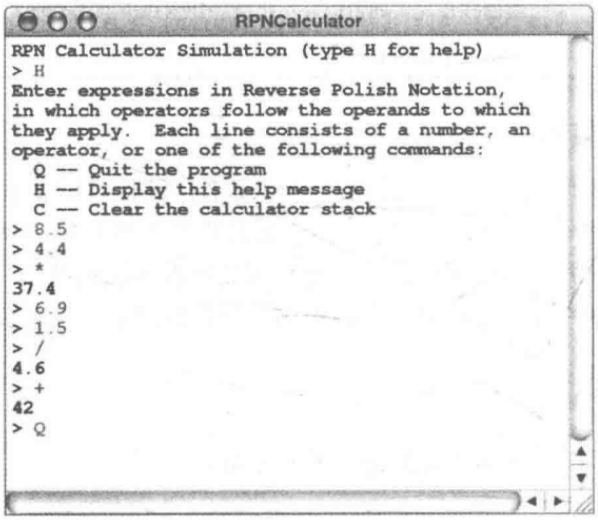
13

图 5-3 RPN 计算器的计算图

用 C++ 实现逆波兰式计算器需要在用户接口设计上做出一些改变。在一个真正的计算器中，数字和操作符都是出现在键盘上的。在这种实现方法下，我们可以很简单的想象用户在命令行中输入的每一行，并且这些行内容的形式有以下几种：

- 一个浮点数。
- 一个从 +、-、* 和 / 中选出的算术操作符。
- 字母 Q，它的作用是终止程序。
- 字母 H，它的作用是输出帮助消息。
- 字母 C，它的作用是消除当前栈中所有的值。

一个正在运行的计算器程序的例子如下图所示：



由于用户以 RETURN 键作为每一个数字输入的结束，并且它只是表明一个数字输入完成，因此没有必要对 RETURN 按钮做一个副本。当用户输入数字时，计算器程序仅仅是将这些数字添加到栈中。当计算器读到一个操作符时，它会弹出栈顶的两个元素，再基于操作符计算出结果，并且展示这个结果，最后再将结果压入栈。

```
/*
 * File: RPNCalculator.cpp
 * -----------------------
 * This program simulates an electronic calculator that uses
 * reverse Polish notation, in which the operators come after
 * the operands to which they apply.  Information for users
 * of this application appears in the helpCommand function.
 */

```

```
#include <iostream>
#include <cctype>
#include <string>
#include "error.h"
#include "simpio.h"
#include "stack.h"
#include "strlib.h"
using namespace std;

/* Function prototypes */

void applyOperator(char op, Stack<double> & operandStack);
void helpCommand();

/* Main program */

int main() {
   cout << "RPN Calculator Simulation (type H for help)" << endl;
   Stack<double> operandStack;
   while (true) {
      string line = getLine("> ");
      if (line.length() == 0) line = "Q";
      char ch = toupper(line[0]);
      if (ch == 'Q') {
         break;
      } else if (ch == 'C') {
         operandStack.clear();
      } else if (ch == 'H') {
         helpCommand();
      } else if (isdigit(ch)) {
         operandStack.push(stringToReal(line));
      } else {
         applyOperator(ch, operandStack);
      }
   }
   return 0;
}

/*
 * Function: applyOperator
 * Usage: applyOperator(op, operandStack);
 * ---------------------------------------
 * Applies the operator to the top two elements on the operand stack.
 * Because the elements on the stack are popped in reverse order,
 * the right operand is popped before the left operand.
 */

void applyOperator(char op, Stack<double> & operandStack) {
   double result;
   double rhs = operandStack.pop();
   double lhs = operandStack.pop();
   switch (op) {
```

```
    case '+': result = lhs + rhs; break;
    case '-': result = lhs - rhs; break;
    case '*': result = lhs * rhs; break;
    case '/': result = lhs / rhs; break;
    default:  error("Illegal operator");
   }
   cout << result << endl;
   operandStack.push(result);
}

/*
 * Function: helpCommand
 * Usage: helpCommand();
 * ---------------------
 * Generates a help message for the user.
 */

void helpCommand() {
   cout << "Enter expressions in Reverse Polish Notation," << endl;
   cout << "in which operators follow the operands to which" << endl;
   cout << "they apply.  Each line consists of a number, an" << endl;
   cout << "operator, or one of the following commands:" << endl;
   cout << "  Q -- Quit the program" << endl;
   cout << "  H -- Display this help message" << endl;
   cout << "  C -- Clear the calculator stack" << endl;
}
```

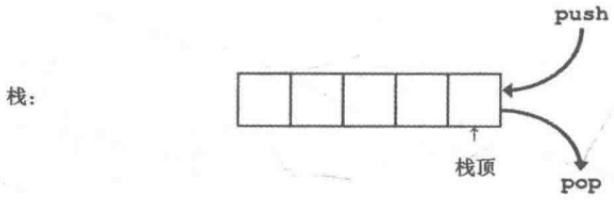
QUEUE

正如你在5.2节所学过的，栈的典型特点就是最后输入的总是最先输出。并强调了这种行为经常在计算机科学中作为LIFO被提到，**其中LIFO是短语“last in, first out”的首字母缩写。LIFO原则在编程环境中非常有用，因为它反映了函数的调用操作，最近调用的函数总是最先返回。**

然而，在现实社会中，“last in, first out”模型相对来说很少。实际上，在人类社会中，我们集体的公平的分配表示法被表达成为“先来先被服务（first come, first served）”。在编程中，这种顺序策略的短语是“先进先出（first in, first out）”，习惯上简写为FIFO。

一个使用FIFO原则存储数据的数据结构被称为队列（queue）。队列基本的操作（和栈的操作push以及pop相似）被称为入队（enqueue）和出队（dequeue）。入队操作在队列的最后添加一个新元素，通常被称之为队尾（tail），出队操作删除队列开头的元素，通常被称之为队首（head）。

栈和队列结构之间的差异可以用下图简单地阐明。在一个栈中，用户必须在数据结构的同一端（栈顶）添加和删除元素，如下图所示：



在队列中，用户在一端添加元素而在另一端删除元素，如下图所示：

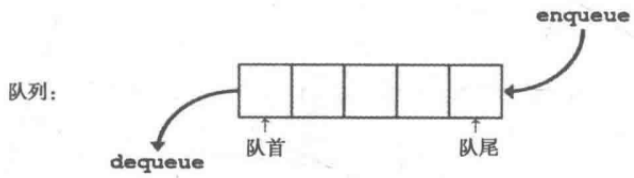


表 5-4　queue.h 接口中的条目

构造函数	
Queue<*type*>()	创建一个空的能存储指定类型值的队列对象

（续）

方法	
size()	返回当前队列中元素的个数
isEmpty()	若队列为空，则返回 true
enqueue(*value*)	将值 *value* 添加到队尾
dequeue()	删除队首元素并将此元素返回给调用者。在空队列中调用 dequeue 方法将产生错误
peek()	返回队首元素的值但并不将其从队列中删除。在空队列中调用 peek 方法将产生错误
clear()	删除队列中的所有元素

一个收银员服务的排队仿真

该仿真的核心是一个循环，它根据参数SIMULATION_TIME所指定的秒的数量来运行。每一秒，该仿真都会完成下面的操作：

1．检查是否有顾客到达，如果有的话，那么将该顾客加入到队列中。

2．如果收银员当前处于服务状态，提示收银员还需要为当前的顾客服务多长时间。最终，这个要求服务的时间结束，收银员变为空闲。

3．如果收银员是空闲的，将会为队列中的下一个顾客服务。

等待队列很自然地能被表示为一个队列。顾客到达队列的时间值被存储在队列中，它能用来确定顾客到达队首所花费的时间。

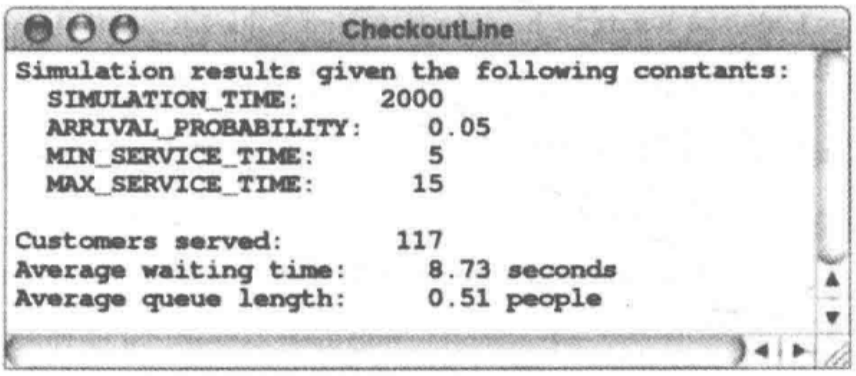
该仿真被以下常量所控制：

- SIMULATION_TIME：指定了仿真的持续时间。
- ARRIVAL_PROBABILITY：确定了在一个单位时间内一个新的顾客到达队列的概率。为了符合标准的统计学规定，这个概率被表示为一个在0到1之间的实数。
- MIN_SERVICE_TIME、MAX_SERVICE_TIME：这两个常量分别定义了顾客被服务时间的范围。对于任何一个顾客来说，收银员在他们身上所花费的时间的总和是在这个范围内(MIN_S_T ~ MAX_S_T)的一个随机整数。

当一个仿真结束后，程序会报告该仿真的常量值以及下面的结果：

- 被服务的总顾客数。
- 顾客在队列中平均等待时间。
- 队列的平均长度。

例如，下面的运行示例展现了给定常量值的仿真运行之后的结果：




```
/*
 * File: CheckoutLine.cpp
 * ----------------------
 * This program simulates a checkout line, such as one you
 * might encounter in a grocery store.  Customers arrive at
 * the checkout stand and get in line.  Those customers wait
 * in the line until the cashier is free, at which point
 * they are served and occupy the cashier for some period
```

```
 * of time.  After the service time is complete, the cashier
 * is free to serve the next customer in the line.
 *
 * In each unit of time, up to the constant SIMULATION_TIME,
 * the following operations are performed:
 *
 * 1. Determine whether a new customer has arrived.
 *    New customers arrive randomly, with a probability
 *    determined by the constant ARRIVAL_PROBABILITY.
 *
 * 2. If the cashier is busy, note that the cashier has
 *    spent another minute with that customer.  Eventually,
 *    the customer's time request is satisfied, which frees
 *    the cashier.
 *
 * 3. If the cashier is free, serve the next customer in line.
 *    The service time is taken to be a random period between
 *    MIN_SERVICE_TIME and MAX_SERVICE_TIME.
 *
 * At the end of the simulation, the program displays the
 * simulation constants and the following computed results:
 *
 * o  The number of customers served
 * o  The average time spent in line
 * o  The average number of people in line
 */

#include <iostream>
#include <iomanip>
#include "queue.h"
#include "random.h"
using namespace std;

/* Constants */

const double ARRIVAL_PROBABILITY = 0.05;
const int MIN_SERVICE_TIME =  5;
const int MAX_SERVICE_TIME = 15;
const int SIMULATION_TIME = 2000;

/* Function prototypes */

void runSimulation(int & nServed, int & totalWait, int & totalLength);
void printReport(int nServed, int totalWait, int totalLength);

/* Main program */

int main() {
   int nServed;
   int totalWait;
   int totalLength;
   runSimulation(nServed, totalWait, totalLength);
```

```
   printReport(nServed, totalWait, totalLength);
   return 0;
}

/*
 * Function: runSimulation
 * Usage: runSimulation();
 * -----------------------
 * Runs the actual simulation.  This function returns the results
 * of the simulation through the reference parameters, which record
 * the number of customers served, the total number of seconds that
 * customers were waiting in a queue, and the sum of the queue length
 * in each time step.
 */

void runSimulation(int & nServed, int & totalWait, int & totalLength) {
   Queue<int> queue;
   int timeRemaining = 0;
   nServed = 0;
   totalWait = 0;
   totalLength = 0;
   for (int t = 0; t < SIMULATION_TIME; t++) {
      if (randomChance(ARRIVAL_PROBABILITY)) {
         queue.enqueue(t);
      }
      if (timeRemaining > 0) {
         timeRemaining--;
      } else if (!queue.isEmpty()) {
         totalWait += t - queue.dequeue();
         nServed++;
         timeRemaining = randomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME);
      }
      totalLength += queue.size();
   }
}

/*
 * Function: printReport
 * Usage: printReport(nServed, totalWait, totalLength);
 * ----------------------------------------------------
 * Reports the results of the simulation in tabular format.
 */

void printReport(int nServed, int totalWait, int totalLength) {
   cout << "Simulation results given the following constants:"
        << endl;
   cout << fixed << setprecision(2);
   cout << "  SIMULATION_TIME:     " << setw(4)
        << SIMULATION_TIME << endl;
   cout << "  ARRIVAL_PROBABILITY: " << setw(7)
        << ARRIVAL_PROBABILITY << endl;
   cout << "  MIN_SERVICE_TIME:    " << setw(4)
```

```
        << MIN_SERVICE_TIME << endl;
    cout << "  MAX_SERVICE_TIME:    " << setw(4)
        << MAX_SERVICE_TIME << endl;
    cout << endl;
    cout << "Customers served:      " << setw(4) << nServed << endl;
    cout << "Average waiting time:  " << setw(7)
        << double(totalWait) / nServed << " seconds" << endl;
    cout << "Average queue length:  " << setw(7)
        << double(totalLength) / SIMULATION_TIME << " people" << endl;
}
```

关系容器：MAP

名为map的集合类，它和字典从概念上很相似。字典允许你查阅一个单词来了解它的含义。一个map是这个概念的概括，它提供了一个被称为键（key）的标签和一个相关联的被称为值（value）的值之间的联系，这可能是一个更大更为复杂的结构。在字典例子中，键就是你所要查找的单词，值就是这个词的具体定义。

Map在编程中有许多应用。例如，一种编程语言的解释器要能够给变量赋值，然后以该变量的名字作为引用。一个Map可以很简单地存储变量的名字和其所对应的值之间的联系。当它们在这种环境下使用时，Map经常被称为符号表（symbol table）。

表 5-5　map.h 接口中的条目

构造函数	
Map<*key type*, *value type*>()	创建一个空的关联键和值的 Map 对象

（续）

方法	
size()	返回 Map 对象中含有的键和值进行关联值对个数
isEmpty()	如果 Map 对象是空的，返回 **true**
put(*key*, *value*)	将特定的键和值进行关联。如果 *key* 在先前的 Map 对象中没有定义，则一个新的值被添加进来；如果 Map 对象中已存在该键 *key*，则旧值将被新值所替代
get(*key*)	返回在 Map 对象中当前与键 *key* 相关联的值。如果该键没有定义，则 **get** 方法返回值类型的默认值
remove(*key*)	从 Map 对象中删除键 *key* 及其所对应的值。如果该键不存在，调用不会对 Map 对象作出任何改变
containsKey(*key*)	检查键 *key* 是否存在与之相关联的值。若存在，则返回 **true**；反之，则返回 **false**
clear()	删除 Map 对象中所有的键 – 值对
操作符	
map[*key*]	和 **get** 方法功能一样，该操作符在 Map 对象中选出与键 *key* 相关联的值。如果 Map 对象中不存在该键，则该选择操作符会创建一个新的键 – 值对，且设置其对应的值为值类型的默认值。使用方括号查找和改变某一特定键的值的 Map 对象通常被称为关联数组（associative array）

227

在应用中使用MAP

如果你经常乘坐飞机的话，你可以快速地发现世界各地的每一个机场都有一个由国际飞机运输协会（International Air Transport Association，IATA）颁布的三字母代码。例 如，纽约市的约翰F·肯尼迪机场的代码为JFK。然而，其他的这些代码很难被分辨。大多数基于网络的交通运输系统提供了一些查阅这些代码的方法，来作为对它们顾客的一种服务。

假设你被要求编写一个简单的C++程序，用来从用户处读取一个机场的三字母代码，然后再向用户返回这个机场的位置。你需要的数据在一个名为AirportCodes.txt的文本文件中，这个文件包含了上千条由国际飞机运输协会已经颁布的飞机场的代码。文件中的每一行是由一个三字母代码、一个等号以及一个与之对应的机场位置所组成的。如果这个文件是按照2009年机场旅客流量的降序排列，并且由国际机场委员会编辑，那么这个文件将会如图5–6所示的一样。

```
1 /*
```

```
 * File: AirportCodes.cpp
 * ----------------------
 * This program looks up a three-letter airport code in a Map object.
 */

#include <iostream>
#include <fstream>
#include <string>
#include "error.h"
#include "map.h"
#include "strlib.h"
using namespace std;

/* Function prototypes */

void readCodeFile(string filename, Map<string,string> & map);

/* Main program */

int main() {
   Map<string,string> airportCodes;
   readCodeFile("AirportCodes.txt", airportCodes);
   while (true) {
      string line;
      cout << "Airport code: ";
      getline(cin, line);
      if (line == "") break;
      string code = toUpperCase(line);
      if (airportCodes.containsKey(code)) {
         cout << code << " is in " << airportCodes.get(code) << endl;
      } else {
         cout << "There is no such airport code" << endl;
      }
   }
   return 0;
}

/*
 * Function: readCodeFile
 * Usage: readCodeFile(filename, map);
 * -----------------------------------
 * Reads a data file representing airport codes and locations into the
 * map, which must be declared by the client.  Each line must consist of
 * a three-letter code, an equal sign, and the city name for that airport.
 */

void readCodeFile(string filename, Map<string,string> & map) {
   ifstream infile;
   infile.open(filename.c_str());
   if (infile.fail()) error("Can't read the data file");
   string line;
   while (getline(infile, line)) {
```

```
54          if (line.length() < 4 || line[3] != '=') {
55             error("Illegal data line: " + line);
56          }
57          string code = toUpperCase(line.substr(0, 3));
58          map.put(code, line.substr(4));
59       }
60       infile.close();
61    }
```

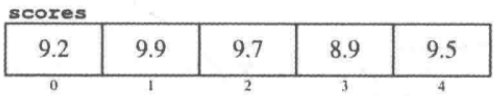
将MAP看成关联数组

Map类重载了用于数组查找的方括号操作符，因此代码`map[key] = value`扮演了代码`map.put(key, value)`的简写形式。

类似的`map[key]`也是`map.get(key)`的简写

类似地，与 map.get(key) 所做的一样，表达式 map[key] 返回了 Map 对象中与 key 相对应的值。毫无疑问，使用 put 以及 get 方法的简写方式非常方便，但是鉴于数组和 Map 类是两种完全不同的结构，在 Map 类中使用数组表示法可能令人惊讶。 然而，如果你从更抽象的角度考虑 Map 类和数组，则和你刚开始的怀疑相比，你会发现它们是很相似的。

要统一这两种表面上看起来不一样的结构，你可以把数组看成是一种将位置索引和元素值进行映射的结构。例如，假设你有一个数组，或者一个等价的 Vector 对象，其中包含了一系列由你记录的体育比赛的分数：



这个数组将键 0 映射为值 9.2，将键 1 映射为值 9.9，将键 2 映射为值 9.7 等等。因此，你可以将一个数组看成是一个用整数作为键的 Map 对象。相反地，你也可以把一个 Map 对象看成是一个使用键作为索引的数组，这也正是 Map 类重载选择语法所提倡的。

使用数组语法来完成 Map 类的操作在编程语言中正逐步流行，它甚至已超出了 C++ 的领域。许多流行的脚本语言都用 Map 类来实现数组，这可使数组使用索引值不一定为整数。用 Map 类作为其底层实现者的数组被称为**关联数组**（associative array）。

关系容器：SET

集合类中最有用的一个就是Set类，表5-6展示了其相关的条目。该类通常用于建模集合（set）的数学抽象，即它是一个集合，其中的元素是无序的且每个元素的值仅出现一次。Set类在某些算法应用中极为有用，因此值得花费单独的一节来介绍它。

表 5-6　set.h 接口中的条目

构造函数	
Set<*type*>()	创建一个包含特定类型值的空的 Set 对象
方法	
size()	返回 Set 对象中元素的个数
isEmpty()	如果 Set 对象为空则返回 **true**
add(*value*)	向 Set 对象中添加值 *value*。如果 Set 对象中已经存在该值，不产生任何错误，并且 Set 对象保持不变
remove(*value*)	从 Set 对象中删除该值 *value*。如果该值不存在，不产生任何错误，并且 Set 对象保持不变
contains(*value*)	如果 Set 对象中存在该值 *value*，则返回 **true**

（续）

clear()	删除 Set 对象中的所有元素
isSubsetOf(*set*)	如果 Set 对象是通过参数传递的 Set 对象的子集，则返回 **true**
first()	返回 Set 对象中由值类型确定顺序后的第一个元素
操作符	
s_1 + s_2	返回 s_1 和 s_2 的**并运算**（union）结果。其中包含的元素是两个原始集合中的所有元素
s_1 * s_2	返回 s_1 和 s_2 的**交运算**（intersection）结果。其中包含的元素是两个原始集合中共同的元素
s_1 - s_2	返回 s_1 和 s_2 的**差运算**（difference）结果。其中包含的元素是只出现在 s_1 而未出现在 s_2 中的元素
s_1 += s_2　s_1 -= s_2　s_1 *= s_2	就像数值运算中的 +、-、* 一样，这些操作符可以和赋值联系在一起。对于 += 和 -=，s_2 的值可以是一个集合、单个值，或者是用逗号隔开的一系列值

利用SET实现cctype

在第3章，你已经学习了`<cctype>`库，它导出了一些测试一个字符类型的判定函数。例如，调用`isdigit(ch)`将测试字符ch是否为一个数字字符。你可以通过测试ch是否超过了单个数字字符的值范围来实现函数`isdigit`，如下所示：

```
bool isdigit(ch) {
    return ch ≥ '0' && ch ≤ '9';
}
```

对于其他一些函数，情况可能变得更为复杂一些。**用同样的方式实现ispunct函数可能会有点困难，因为标点符号分布在ASCII范围的好几个间隔中。**如果你将所有的标点符号定义在一个集合中，事情就会变得很简单，在这种情况下，实现`ispunct(ch)`你所要做的就是检查字符ch是否出现在这个集合中。

图 5-8 展示了用集合实现 中的判定函数。**这段代码首先对于每一种字符类型都创建了一个Set，然后定义了判定函数，这样它们仅仅在适当的Set对象中调用contains就可以实现判定函数。**例如，为了实现 isdigit， cctype的实现定义了一个包含所有数字字符的 Set 对象，如下所示：

```
const Set<char> DIGIT_SET = setFromString("0123456789");
```

出现在图 5-8 中的 setFromString 函数仅仅是一个辅助函数，它通过向 Set 对象中依次添加参数字符串中的字符来创造一个 Set 对象。这个函数让定义 set 对象变得很简单，例如，对于定义标点符号字符的set对象，你只需要列出适合其描述的字符即可。

对于那些抽象和高级的操作来说，使用set类的好处之一就是让这些操作更易于思考。尽管在cctype.cpp中大多使用setFromString从实际字符中来创建Set对象，但是还有一些是使用+操作符， **+操作符在set类中被重载了，该操作可以返回两个Set对象的并运算结果。**例如，一旦你定义了Set对象LOWER_SET和UPPER_SET，使得它们可以包含小写字母和大写字母，你就可以通过编写如下代码来定义 ALPHA_SET：

```
const Set<char> ALPHA_SET = LOWER_SET + UPPER_SET;
```

基于集合的cctype的实现

```
/*
 * File: cctype.cpp
 * ----------------
 * This program simulates the <cctype> interface using sets of characters.
 */

#include <string>
#include "cctype.h"
#include "set.h"
using namespace std;

/* Function prototypes */

Set<char> setFromString(string str);

/*
 * Constant sets
 * -------------
 * These sets are initialized to contain the characters in the
 * corresponding character class.
 */

const Set<char> DIGIT_SET = setFromString("0123456789");
const Set<char> LOWER_SET = setFromString("abcdefghijklmnopqrstuvwxyz");
const Set<char> UPPER_SET = setFromString("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
const Set<char> PUNCT_SET = setFromString("!\"#$%&'()*+,-./:;⟸?@[\\]^_`{|}");
const Set<char> SPACE_SET = setFromString(" \t\v\f\n\r");
const Set<char> XDIGIT_SET = setFromString("0123456789ABCDEFabcdef");
const Set<char> ALPHA_SET = LOWER_SET + UPPER_SET;
const Set<char> ALNUM_SET = ALPHA_SET + DIGIT_SET;
const Set<char> PRINT_SET = ALNUM_SET + PUNCT_SET + SPACE_SET;
```

```
32
33  /* Exported functions */
34
35  bool isalnum(char ch) { return ALNUM_SET.contains(ch); }
36  bool isalpha(char ch) { return ALPHA_SET.contains(ch); }
37  bool isdigit(char ch) { return DIGIT_SET.contains(ch); }
38  bool islower(char ch) { return LOWER_SET.contains(ch); }
39  bool isprint(char ch) { return PRINT_SET.contains(ch); }
40  bool ispunct(char ch) { return PUNCT_SET.contains(ch); }
41  bool isspace(char ch) { return SPACE_SET.contains(ch); }
42  bool isupper(char ch) { return UPPER_SET.contains(ch); }
43  bool isxdigit(char ch) { return XDIGIT_SET.contains(ch); }
44
45  /* Helper function to create a set from a string of characters */
46
47  Set<char> setFromString(string str)  {
48     Set<char> set;
49     for (int i = 0; i < str.length(); i++) {
50        set.add(str[i]);
51     }
52     return set;
53  }
```

MAP与SET的差异

在本章前面对 Map类的讨论中，用于解释底层概念的示例之一就是：**一个字典中的键都是一个个独立的单词，并且其对应的值就是该单词的定义。**

在某些应用中，例如一个拼写检查程序或者Scrabble程序（一种拼字游戏），**你不需要知道一个词的定义**，你所要知道的就是一个字母的组合是否是一个合法的单词。**在那样的应用中，Set类是一个理想的工具。**与一个Map对象既包含单词又包含定义不一样，使用set类你所需要的就是包含所有合法单词的Set。如果单词包含在Set对象中，那么它就是合法的，反之，则是非法的。

一个只有单词没有与之对应的解释的集合我们称之为字典（lexicon）。如果你有一个包含了所有英文单词的名为 EnglishWords.txt 的文本文件，并且每个单词只占一行，你可以通过下面的这段代码创建一个英文字典：

```
1  Set<string> lexicon;
2  ifstream infile;
3  infile.open("EnglishWords.txt");
4  if (infile.fail()) error("Can't open EnglishWords.txt");
5  string word;
6  while (getline(infile, word)){
7      lexicon.add(word);
8  }
9  infile.close();
```

Lexicon库

表 5-7 lexicon.h 接口导出的条目

构造函数	
Lexicon()	创建一个空的 Lexicon 对象
Lexicon(*file*)	通过从文件 file 中读取数据来初始化一个 Lexicon 对象
方法	
size()	返回 Lexicon 对象中单词的总数
isEmpty()	如果 Lexicon 对象为空，返回 **true**
add(*word*)	如果该单词在 Lexicon 对象中不存在的话，则向其中添加一个新的单词 word。所有的单词都以小写字母形式存储在 Lexicon 对象中
addWordsFromFile(*file*)	将参数名为 file 文件中的所有单词添加到 lexicon 对象中。file 要么是一个其单词为分行存储的文本文件，要么是为 lexicon 对象而特别设计其格式的已编译的数据文件向 Lexicon 对象中添加 file 中所有的单词
contains(*word*)	如果单词 word 存在于 Lexicon 对象中，则返回 **true**
containsPrefix(*prefix*)	如果 Lexicon 对象中的任何一个单词都是以特定的前缀 prefix 开头，则返回 **true**
clear()	删除 Lexicon 对象中的所有的单词

尽管对于一个字典来说，用Set类作为其底层表示表现得相当好，但它并不是很有效率。因为一个高效的字典表示法可能会给编程项目带来许多令人激动的事情。Stanford类库中包含了一个名为Lexicon的类，该类是Set类中一个用于存储单词集合的优化的定制版本。表5-7展示了由Lexicon类导出的条目。正如你所看到的一样，它们和Set类中的大多数条目是一样的。

这个库中同时还包含了一个名为Englishwords.dat的数据文件，这是一个已编译的包含了所有英文单词的字典。使用英文字典的程序通常都使用以下声明语句对其进行初始化：

```
Lexicon english("EnglishWords.dat");
```

在像Scrabble一样的文字游戏中，尽可能多地记住两个字母的单词是很有用的，因为知道两个字母的单词能让你更容易地知道字典中以这两个字母为基础的新单词。**假设你有一个包含英文单词的字典，你可通过生成所有的两个字母的字符串来创建这样一个列表，然后再用这个字典检查上述两个字母的字符串的组合是否为一个单词。**

```
/*
* File: TwoLetterWords.cpp
* ------------------------
* This program generates a list of the two-letter English words.
*/

#include <iostream>
#include "lexicon.h"
using namespace std;

int main() {
    Lexicon english("EnglishWords.dat");
    string word="xx";
    for (char c0 = 'a'; c0 ≤ 'z'; c0++) {
```

```
        word[0] = c0;
        for (char c1 = 'a', c1 ⩽ 'z'; c1++) {
            word[1] = c1;
            if (english.contains(word)) {
                cout << word << endl;
            }
        }
    }

    return 0;
}
```

对关系容器的迭代

TwoLetterWords程序通过生成两个字母的所有可能组合，然后再查阅字典，检查这些两个字母的组合是否出现在英文字典中的方式，产生了一个由两个字母组成的单词的清单。**另一种达到同样效果的策略是浏览字典中的每一个单词，然后再将长度为2的单词显示出来。要做到这一点，你所要做的就是以某种方式每次一个单词地遍历Lexicon对象中的每个单词。**

但是这些集合类-包括标准模板库里的集合类和本章的简化版本-支持一种新的被称为**基于范围的循环（range-based for loop）**的控制模式，如下所示：

```
for (type variable : collection){
    body of the loop
}
```

例如，如果你想要迭代英语字典中所有的单词，并且挑选出只含有两个字母的单词，你可以这样编写代码，如下所示：

```
for (string word : english) {
    if (word.length() == 2) {
        cout << word << endl;
    }
}
```

基于范围的循环是C++11的新特性，C++11是2011年发布的C++的新版本。因为这个版本是最近发布的，因此，C++11所扩展的特性还没有被完全的合并到所有C++编程环境中，包括了几个主要的特性。如果你使用的是一个旧的编译器，你将不能够使用基于范围的循环的标准形式。但是也不必绝望，标准C++库中包含了一个名为`foreach.h`的接口，它使用了C++预处理，用一个很熟悉的方式定义了一个名为`foreach`的宏：

```
foreach (type variable in collection){
    body of the loop
}
```

foreach宏与基于范围的循环的唯一不同在于关键字的名字，foreach中使用关键词in而不是一个冒号。和基于范围的循环一样，foreach对于Stanford类库和标准模板库中实现的集合类都起作用。

迭代的顺序

当你使用基于范围的循环时，有时候理解迭代处理元素的顺序是很有用的。这没有什么通用的规则。对于迭代顺序，基于效率上的考虑，每个集合类都定义了关于其自身的迭代顺序策略。你之前见过的类，对关于元素值的顺序都进行了下面的保证：

- 当你对一个Vector类中的元素进行遍历时，基于范围的循环以索引位置为序来对元素进行遍历，因此，索引位置为0的元素首先被遍历，紧接着是索引位置为1的元素，直到这个Vector类中的最后一个元素被遍历。因此，迭代的顺序与传统的for循环模式顺序是一样的：

```
for (int i =0; i < vec.size();i++){
    code to process vec[i]
}
```

- 当你迭代一个Grid类中的元素时，基于范围的循环首先依次浏览行数为0的各元素，然后再浏览行数为1的各元素，依此类推。Grid类的迭代策略和下面使用的for循环相类似：

```
for (int row =0; row < grid.numRows(); row++){
    for (int col =0; col < grid.numCols(); co1++){
        code to process grid[row][col]
    }
}
```

这种外循环出现行下标，它的遍历顺序被称为行优先次序（row-major order）。

- 当你对Map类中的元素进行遍历时，基于范围的循环返回以键为序的且由其类型决定的所有键值。例如，一个键类型为整型的Map对象将会按照数字升序的顺序排列其键值。一个键类型为字符串类型的Map对象将会按照字典序（lexicographic order）排列其键值，字典序是通过比较其内部的ASCII码值来决定其顺序的。
- 当你对一个Set类或者 Lexicon类中的元素进行遍历时，基于范围的循环返回的元素的顺序总是由其值的类型所确定的。在Lexicon类中，基于范围的循环返回小写字母的所有单词。
- 你**不能使用**基于范围的循环去遍历Stack类和Queue类。当只有一个元素（这个元素是栈顶元素或者是队首元素）是可见的时候，允许自由地访问这些结构将会违背栈和队列的读取原则。

再论儿童黑话

像3.2节所描述的一样，当你将英语转化成儿童黑话时，大多数单词将转变成某种与传统的英语截然不同的，听起来模糊的类拉丁文语言。然而，有几个**单词在它们转化后恰好与英文单词相同**。例如，trash的儿童黑话是ashtray，entry的儿童黑话是entryway。但这些单词并不是一种普遍情况，存储在EnglishWords. dat文件中的字典，有超过100000个英语单词，但仅有27个单词符合上述情况。**采用基于范围的循环和第3章PigLatin程序的 translateWord函数，可以容易地编写出图5-10中列出所有这些单词的程序。**

```
/*
 * File: PigEnglish.cpp
 * --------------------
 * This program finds all English words that remain words when
 * you convert them to Pig Latin, such as "trash" (which becomes
 * "ashtray") and "entry" (which becomes "entryway").  The code
 * ignores words containing no vowels (mostly Welsh-derived
 * words like "cwm"), which don't change form under the Pig Latin
 * rules introduced in Chapter 3.
 */

#include <iostream>
#include <string>
#include <cctype>
```

```
#include "lexicon.h"
using namespace std;

/* Function prototypes */

string wordToPigLatin(string word);
int findFirstVowel(string word);
bool isVowel(char ch);

/* Main program */

int main() {
   cout << "This program finds words that remain words"
        << " when translated to Pig Latin." << endl;
   Lexicon english("EnglishWords.dat");
   foreach (string word in english) {
      string pig = wordToPigLatin(word);
      if (pig != word && english.contains(pig)) {
         cout << word << " -> " << pig << endl;
      }
   }
   return 0;
}

/*
 * Function: wordToPigLatin
 * Usage: string translation = wordToPigLatin(word);
 * -------------------------------------------------
 * This function translates a word from English to Pig Latin using
 * the rules specified in the text.  The translated word is
 * returned as the value of the function.
 */

string wordToPigLatin(string word) {
   int vp = findFirstVowel(word);
   if (vp == -1) {
      return word;
   } else if (vp == 0) {
      return word + "way";
   } else {
      string head = word.substr(0, vp);
      string tail = word.substr(vp);
      return tail + head + "ay";
   }
}

/*
 * Function: findFirstVowel
 * Usage: int k = findFirstVowel(word);
 * ------------------------------------
 * Returns the index position of the first vowel in word.  If
 * word does not contain a vowel, findFirstVowel returns -1.
```

```
 */

int findFirstVowel(string word) {
   for (int i = 0; i < word.length(); i++) {
      if (isVowel(word[i])) return i;
   }
   return -1;
}

/*
 * Function: isVowel
 * Usage: if (isVowel(ch)) . . .
 * -----------------------------
 * Returns true if the character ch is a vowel.
 */

bool isVowel(char ch) {
   switch (ch) {
    case 'A': case 'E': case 'I': case 'O': case 'U':
    case 'a': case 'e': case 'i': case 'o': case 'u':
      return true;
    default:
      return false;
   }
}
```

计算单词频率

WordFrequency程序是另一个迭代起重要作用的应用程序。采用之前示例中你已经用过的机制，则必要的程序代码将相当简单。将一行划分成一个个单词的实现策略与你在第3章中已经见过的PigLatin程序很类似。为了记录每个单词以及它出现的频率值，你明显需要一个`Map<string, int>`对象。

```
/*
 * File: WordFrequency.cpp
 * -----------------------
 * This program computes the frequency of words in a text file.
 */

#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <cctype>
#include "filelib.h"
#include "map.h"
#include "strlib.h"
#include "vector.h"
using namespace std;

/* Function prototypes */

```

```
void countWords(istream & stream, Map<string,int> & wordCounts);
void displayWordCounts(Map<string,int> & wordCounts);
void extractWords(string line, Vector<string> & words);

/* Main program */

int main() {
   ifstream infile;
   Map<string,int> wordCounts;
   promptUserForFile(infile, "Input file: ");
   countWords(infile, wordCounts);
   infile.close();
   displayWordCounts(wordCounts);
   return 0;
}

/*
 * Function: countWords
 * Usage: countWords(stream, wordCounts);
 * --------------------------------------
 * Counts words in the input stream, storing the results in wordCounts.
 */

void countWords(istream & stream, Map<string,int> & wordCounts) {
   Vector<string> lines, words;
   readEntireFile(stream, lines);
   foreach (string line in lines) {
      extractWords(line, words);
      foreach (string word in words) {
         wordCounts[toLowerCase(word)]++;
      }
   }
}

/*
 * Function: displayWordCounts
 * Usage: displayWordCounts(wordCount);
 * ------------------------------------
 * Displays the count associated with each word in the wordCount map.
 */

void displayWordCounts(Map<string,int> & wordCounts) {
   foreach (string word in wordCounts) {
      cout << left << setw(15) << word
           << right << setw(5) << wordCounts[word] << endl;
   }
}

/*
 * Function: extractWords
 * Usage: extractWords(line, words);
 * ---------------------------------
```

```
 * Extracts words from the line into the string vector words.
 */

void extractWords(string line, Vector<string> & words) {
   words.clear();
   int start = -1;
   for (int i = 0; i < line.length(); i++) {
      if (isalpha(line[i])) {
         if (start == -1) start = i;
      } else {
         if (start >= 0) {
            words.add(line.substr(start, i - start));
            start = -1;
         }
      }
   }
   if (start >= 0) words.add(line.substr(start));
}
```