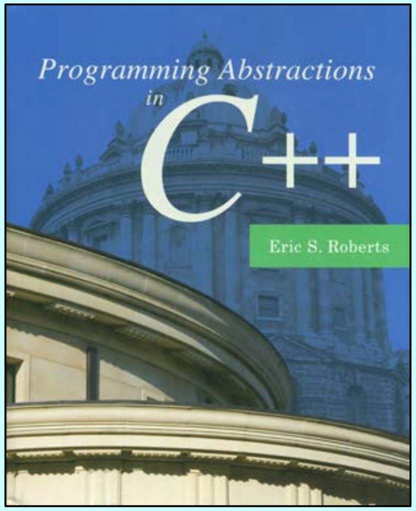
CHAPTER 5

Collections

In this way I have made quite a valuable collection.

—Mark Twain, *A Tramp Abroad*, 1880



The Collection Classes

- Chapter 5 describes a set of useful library classes that contain other objects and are therefore called ***container*** or ***collection classes***. These classes are provided by the Stanford C++ library:

`Vector`	`Grid`	`Stack`	`Queue`	`Map`	`Set`	`Lexicon`

- Here are some general guidelines for using these classes:
  – These classes represent ***abstract data types*** whose details are hidden.
  – Each class (except **`Lexicon`**) requires type parameters.
  – Declaring variables of these types always invokes a ***constructor***.
  – Any memory for these objects is freed when its declaration scope ends.
  – Assigning one value to another *copies* the entire structure.
  – To avoid copying, these structures are usually passed by reference.

C++ STL (standard template library) provides these container classes:

pair	priority queue	
vector	map	
list	multimap	
slist	hash_set	
stack	hash_multiset	
queue	hash_map	
deque	hash_multimap	
set	bitset	
multiset	valarray	(NOT grid or lexicon)

Template Classes

- The collection classes are implemented as ***template classes***, which make it possible for an entire family of classes to share the same code.
- Instead of using the class name alone, the collection classes require a type parameter that specifies the element type. For example, **`Vector<int>`** represents a vector of integers. Similarly, **`Grid<char>`** represents a two-dimensional array of characters.
- It is possible to nest classes, so that, for example, you could use the following definition to represent a list of chess positions:

```
Vector< Grid<char> > chessPositions;
```

Constructors for the **`Vector<`***type***`>`** Class

`Vector<type> vec;` Initializes an empty vector of the specified element type.
`Vector<type> vec(n);` Initializes a vector with **`n`** elements all set to the default value of the type.
`Vector<type> vec(n, value);` Initializes a vector with **`n`** elements all set to **`value`**.

The Stanford C++ library implementation of **`Vector`** includes a shorthand form for initializing an array given a list of values, as illustrated by the following example:

```
Vector<int> digits;
digits += 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
```

Methods in the **`Vector<`***type***`>`** Class

`vec.size()` Returns the number of elements in the vector.		
`vec.isEmpty()` Returns **`true`** if the vector is empty.		
`vec.get(i)` Returns the **`i`**th element of the vector.	*or*	**`vec[i]`**
`vec.set(i, value)` Sets the **`i`**th element of the vector to **`value`**.	*or*	**`vec[i] = value;`**
`vec.add(value)` Adds a new element to the end of the vector.	*or*	**`vec += value;`**
`vec.insertAt(index, value)` Inserts the value before the specified index position.		
`vec.removeAt(index)` Removes the element at the specified index.		
`vec.clear()` Removes all elements from the vector.		

The **readEntireFile** Function

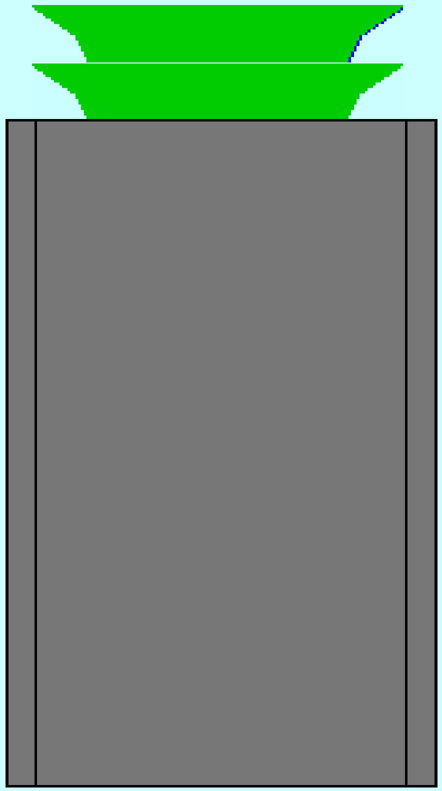
```
/*
 * Function: readEntireFile
 * Usage: readEntireFile(is, lines);
 * ---------------------------------
 * Reads the entire contents of the specified input stream
 * into the string vector lines.  The client is responsible
 * for opening and closing the stream
 */

void readEntireFile(istream & is, Vector<string> & lines) {
   lines.clear();
   while (true) {
      string line;
      getline(is, line);
      if (is.fail()) break;
      lines.add(line);
   }
}
```

Methods in the `Grid<`*type*`>` Class

`Grid<type> grid(nrows, ncols);` Constructs a grid with the specified dimensions.
`grid.numRows()` Returns the number of rows in the grid.
`grid.numCols()` Returns the number of columns in the grid.
`grid[i][j]` Selects the element in the **`i`**th row and **`j`**th column.
`resize(nrows, ncols)` Changes the dimensions of the grid and clears any previous contents.
`inBounds(row, col)` Returns **`true`** if the specified row and column position is within the grid.

The Stack Metaphor

- A ***stack*** is a data structure in which the elements are accessible only in a ***last-in/first-out*** order.
- The fundamental operations on a stack are **push**, which adds a new value to the top of the stack, and **pop**, which removes and returns the top value.
- One of the most common metaphors for the stack concept is a spring-loaded storage tray for dishes. Adding a new dish to the stack pushes any previous dishes downward. Taking the top dish away allows the dishes to pop back up.



Methods in the **`Stack<`*type*`>`** Class

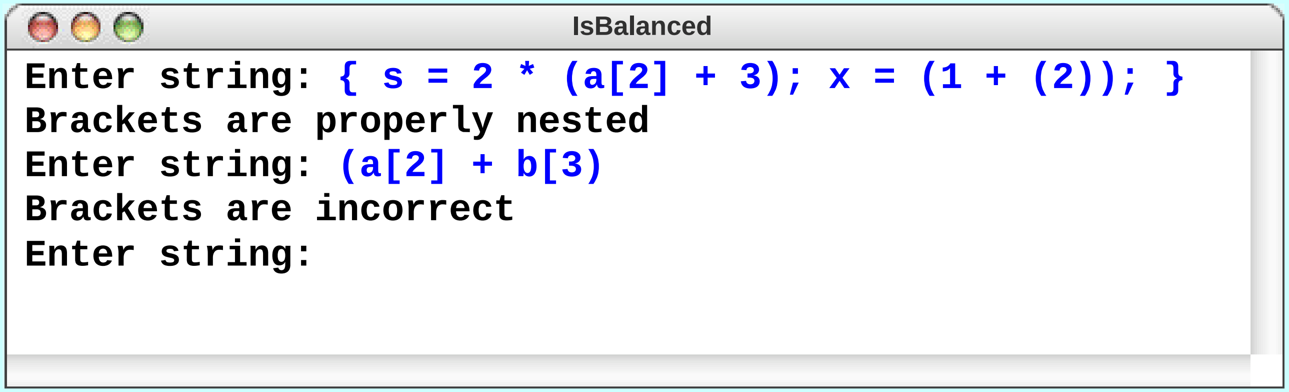
`stack.size()` Returns the number of values pushed onto the stack.
`stack.isEmpty()` Returns **`true`** if the stack is empty.
`stack.push(value)` Pushes a new value onto the stack.
`stack.pop()` Removes and returns the top value from the stack.
`stack.peek()` Returns the top value from the stack without removing it.
`stack.clear()` Removes all values from the stack.

Example: Stack Processing

We could use a stack to check whether the bracketing operators (parentheses, brackets, and curly braces) in a string are properly matched. As an example of proper matching, consider the string

```
{ s = 2 * (a[2] + 3); x = (1 + (2)); }
```

If you go through the string carefully, you discover that all the bracketing operators are correctly nested, with each open parenthesis matched by a close parenthesis, each open bracket matched by a close bracket, and so on.



IsBalanced

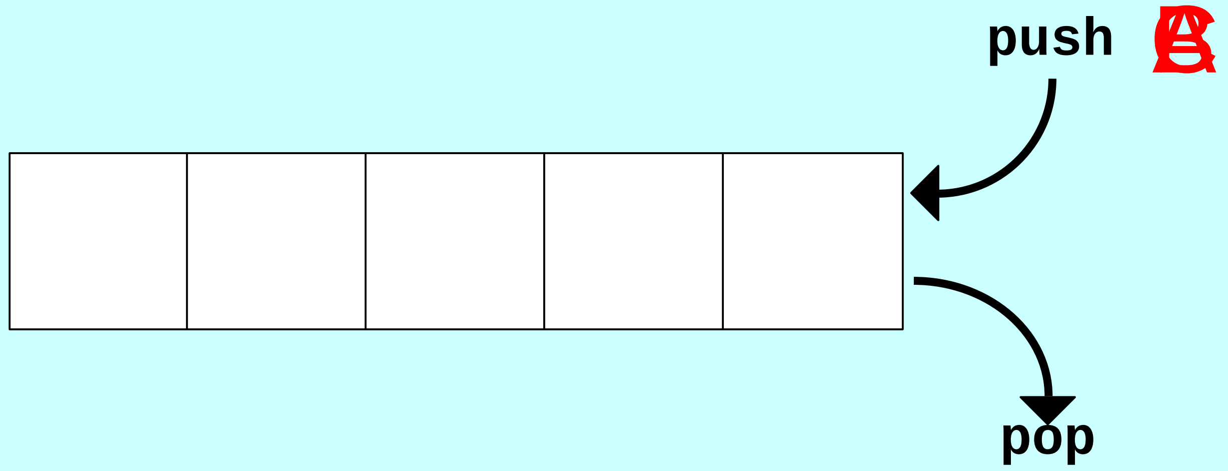
```
Enter string: { s = 2 * (a[2] + 3); x = (1 + (2)); }
Brackets are properly nested
Enter string: (a[2] + b[3)
Brackets are incorrect
Enter string:
```

Methods in the **`Queue<`***type***`>`** Class

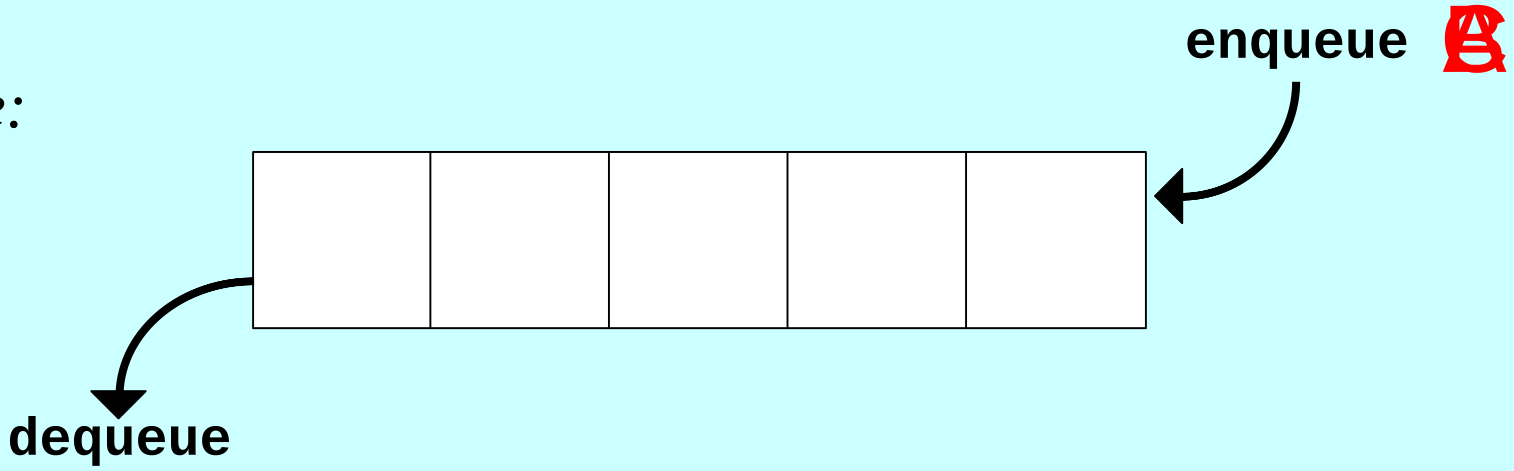
`queue.size()` Returns the number of values in the queue.
`queue.isEmpty()` Returns **`true`** if the queue is empty.
`queue.enqueue(value)` Adds a new value to the end of the queue (which is called its ***tail***).
`queue.dequeue()` Removes and returns the value at the front of the queue (which is called its ***head***).
`queue.peek()` Returns the value at the head of the queue without removing it.
`queue.clear()` Removes all values from the queue.

Comparing Stacks and Queues

Stack:



Queue:



Methods in the Map Classes

- A ***map*** associates ***keys*** and ***values***. The Stanford library offers two flavors of maps—**`Map`** and **`HashMap`**—both of which implement the following methods:

`map.size()`	Returns the number of key/value pairs in the map.
`map.isEmpty()`	Returns **`true`** if the map is empty.
`map.put(key, value)` *or* **`map[key] = value;`**	Makes an association between **`key`** and **`value`**, discarding any existing one.
`map.get(key)` *or* **`map[key]`**	Returns the most recent value associated with **`key`**.
`map.containsKey(key)`	Returns **`true`** if there is a value associated with **`key`**.
`map.remove(key)`	Removes **`key`** from the map along with its associated value, if any.
`map.clear()`	Removes all key/value pairs from the map.

Using Maps in an Application

- Before going on to create new applications of maps, it seems worth going through the example from the text, which uses a map to associate three-letter airport codes with their locations.
- The association list is stored in a text file that looks like this:

```
ATL=Atlanta, GA, USA
ORD=Chicago, IL, USA
LHR=London, England, United Kingdom
HND=Tokyo, Japan
LAX=Los Angeles, CA, USA
CDG=Paris, France
DFW=Dallas/Ft Worth, TX, USA
FRA=Frankfurt, Germany
    ⋮
```

- The **`Airports.cpp`** program shows how to read this file into a **`Map<string,string>`**, where it can be more easily used.

Example: Symbol Tables

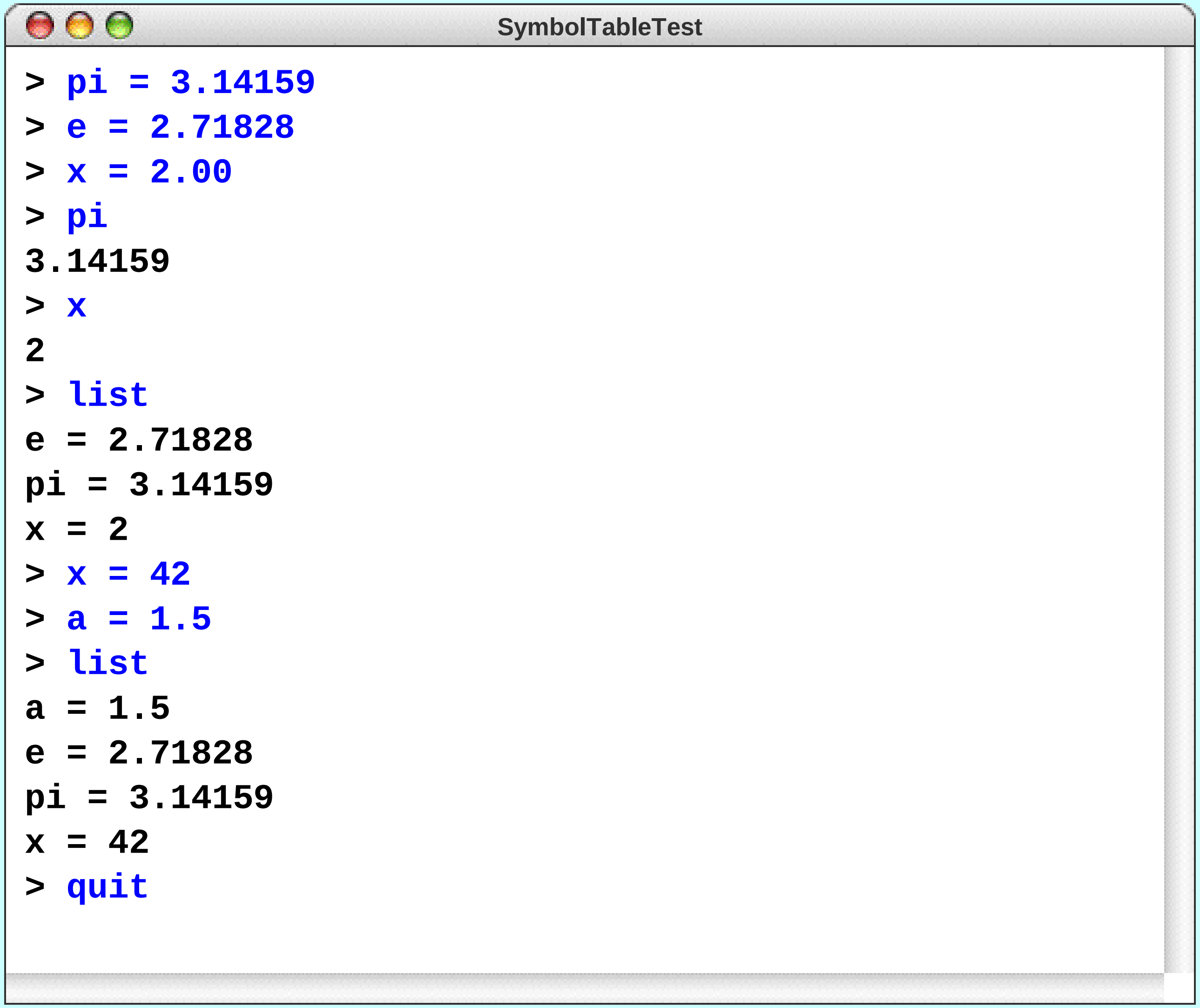
A map is often called a ***symbol table*** when it is used in the context of a programming language, because it is precisely the structure you need to store variables and their values. For example, if you are working in an application in which you need to assign floating-point values to variable names, you could do so using a map declared as follows:

```
Map<string,double> symbolTable;
```

We could write a C++ program that declares such a symbol table and then reads in command lines from the user, which must be in one of the following forms:

- A simple assignment statement of the form *var* **`=`** *number*.
- A variable alone on a line, which is a request to display its value.
- The command **`list`**, which lists all the variables.
- The command **`quit`**, which exits from the program.

Symbol Table Sample Run



```
> pi = 3.14159
> e = 2.71828
> x = 2.00
> pi
3.14159
> x
2
> list
e = 2.71828
pi = 3.14159
x = 2
> x = 42
> a = 1.5
> list
a = 1.5
e = 2.71828
pi = 3.14159
x = 42
> quit
```

The `foreach` Statement

- One of the common operations that clients need to perform when using a collection is to iterate through the elements.
- While it is easy to implement iteration for vectors and grids using **`for`** loops, it is less clear how you would do the same for other collection types. The modern approach to solving this problem is to use a general tool called an ***iterator*** that delivers the elements of the collection, one at a time.
- C++11 uses a ***range-based for statement*** to simplify iterators:

```
for (string key : map) {
        . . . code to process that key . . .
}
```

- The Stanford libraries implement the same idea like this:

```
foreach (string key in map) {
        . . . code to process that key . . .
}
```

Methods in the **`Set<`*`type`*`>`** Class

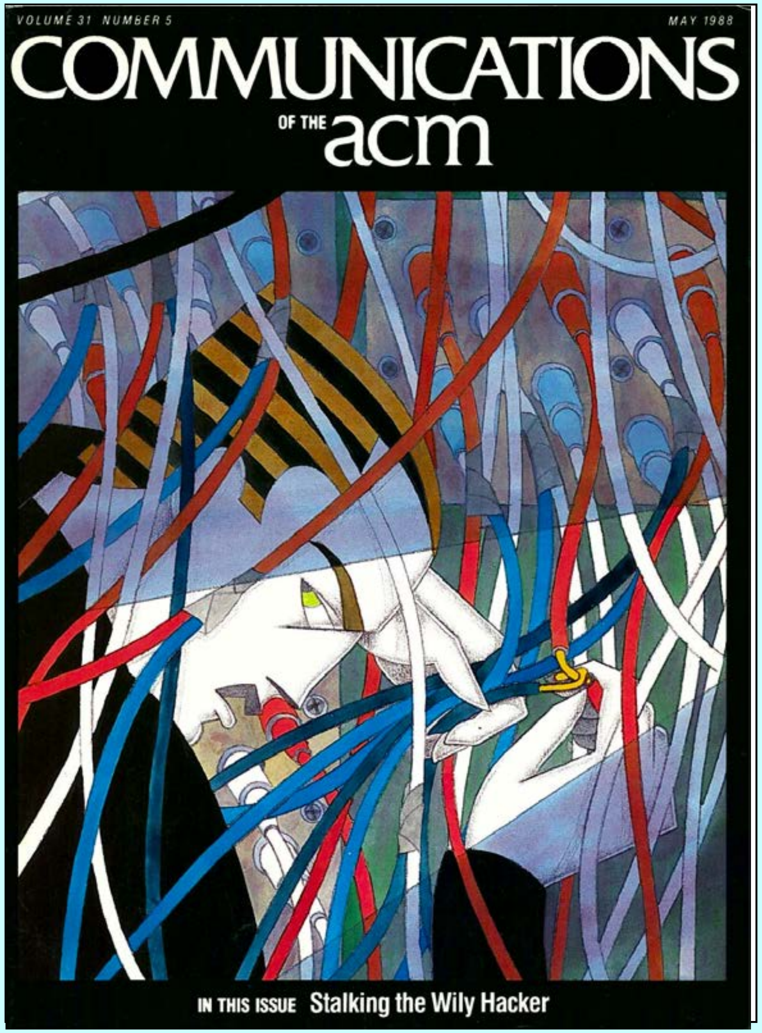
`set.size()` Returns the number of elements in the set.
`set.isEmpty()` Returns **`true`** if the set is empty.
`set.add(value)` Adds **`value`** to the set.
`set.remove(value)` Removes **`value`** from the set.
`set.contains(value)` Returns **`true`** if the set contains the specified value.
`set.clear()` Removes all words from the set.
`s1.isSubsetOf(s2)` Returns **`true`** if **`s1`** is a subset of **`s2`**.
`set.first()` Returns the first element of the set in the ordering specified by the value type.

Methods in the `Lexicon` Class

`lexicon.size()` Returns the number of words in the lexicon.
`lexicon.isEmpty()` Returns **`true`** if the lexicon is empty.
`lexicon.add(word)` Adds **`word`** to the lexicon, always in lowercase.
`lexicon.addWordsFromFile(filename)` Adds all the words in the specified file to the lexicon.
`lexicon.contains(word)` Returns **`true`** if the lexicon contains the specified word.
`lexicon.containsPrefix(prefix)` Returns **`true`** if the lexicon contains any word beginning with **`prefix`**.
`lexicon.clear()` Removes all words from the lexicon.

Why Do Both **`Lexicon`** and **`Set`** Exist?

- The **`Lexicon`** representation is extremely space-efficient. The data structure used in the library implementation stores the full English dictionary in 350,000 bytes, which is shorter than a text file containing those words.
- The underlying representation makes it possible to implement a **`containsPrefix`** method that is useful in many applications.
- The representation makes it easy for iterators to process the words in alphabetical order.

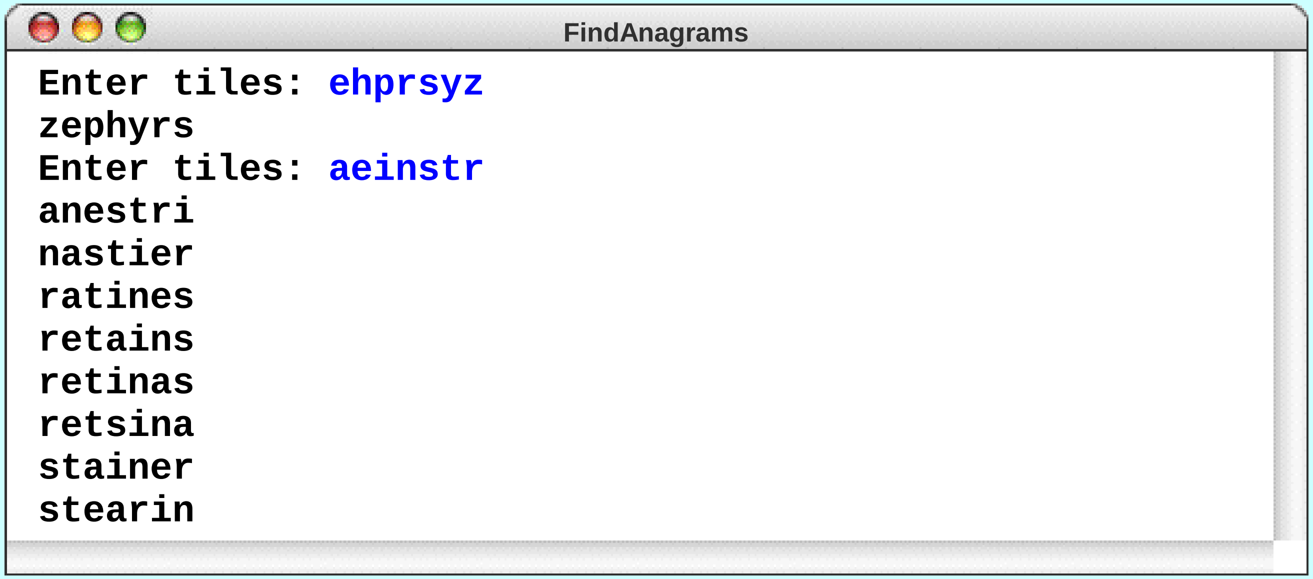


Iterator Order

- When you look at the documentation for an iterator, one of the important things to determine is whether the collection class specifies the order in which elements are generated. The Stanford C++ libraries make the following guarantees:
  - Iterators for arrays operate in index order.
  - Iterators for grids operate in ***row-major order***, which means that the iterator runs through every element in row 0, then every element in row 1, and so on.
  - Iterators for the **`Map`** class deliver the keys in the order imposed by the standard comparison function for the key type; iterators for the **`HashMap`** class return keys in a seemingly random order.
  - Iterators for the **`Set`** class deliver the elements in the order imposed by the standard comparison function for the value type; the **`HashSet`** class is unordered.
  - Iterators for lexicons always deliver words in alphabetical order.

Example: Finding Anagrams

- How can a program read in a set of letters and see whether any anagrams of that set of letters are themselves words?



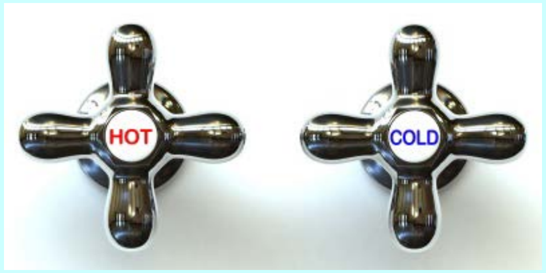
FindAnagrams

```
Enter tiles: ehprsyz
zephyrs
Enter tiles: aeinstr
anestri
nastier
ratines
retains
retinas
retsina
stainer
stearin
```

- Generating all anagrams of a word is not a simple task. Most solutions require some tricky recursion, but can you think of another way to solve this problem? Hint: What if you had a function that sorts the letters in a word. Would that help?

Example: Finding “S” Hooks

- In Scrabble, one of the most important strategic principles is to conserve your S tiles so that you can hook longer words (ideally, the high-scoring seven-letter plays called ***bingos***) onto existing words.
- Consider a hotel where the shower taps are prominently labeled with HOT and COLD:



- A Scrabble player might notice that each of these words takes an S on *either* end, making them ideally flexible for Scrabble plays.
- How can we write a C++ program that finds all such words?

The End