Maps as associative arrays

The **Map** class overloads the square bracket operators used for array selection so that the statement

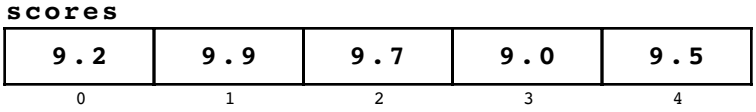
```
map[key] = value;
```

acts as a shorthand for

```
map.put(key, value);
```

and the expression **map[key]** returns the value from **map** associated with **key** in exactly the same way that **map.get(key)** does. While these shorthand forms of the **put** and **get** methods are certainly convenient, using array notation for maps is initially somewhat surprising, given that maps and arrays seem to be rather different in their structure. If you think about maps and arrays in the right way, however, they turn out to be more alike than you might at first suspect.

The insight necessary to unify these two seemingly different structures is that you can think of arrays as structures that map index positions to elements. For example, the array



used as an example in Chapter 2 maps the key 0 into the value 9.2, the key 1 into 9.9, the key 2 into 9.7, and so forth. Thus, an array is in some sense just a map with integer keys. Conversely, you can think of a map as an array that uses strings as index positions, which is precisely what the overloaded selection syntax for the **Map** class suggests.

Using array syntax to perform map-like operations is becoming increasingly common in programming languages beyond the C++ domain. Many popular scripting languages implement all arrays internally as maps, making it possible use index values that are not necessarily integers. Arrays implemented using maps as their underlying representation are called **associative arrays.**

4.6 The Lexicon class

A **lexicon** is conceptually a dictionary from which the definitions have been removed. Given a lexicon, you can only tell whether a word exists; there are no definitions or values associated with the individual words.

The structure of the Lexicon class

The **Lexicon** class offers two different forms of the constructor. The default constructor creates an empty lexicon to which you can add new words. In many applications, however, it is convenient to provide the constructor with the name of a data file that contains the words you want to include. For example, if the file **EnglishWords.dat** contains a list of all English words, you could use that file to create a lexicon using the followng declaration:

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

The implementation of the **Lexicon** class allows these data files to be in either of two formats:

1. A text file in which the words appear in any order, one word per line.

2. A precompiled data file that mirrors the internal representation of the lexicon. Using precompiled files (such as **`EnglishWords.dat`**) is more efficient, both in terms of space and time.

Unlike the classes presented earlier in this chapter, **`Lexicon`** does not require a type parameter, because a lexicon doesn't contain any values. It does, of course, contain a set of words, but the words are always strings.

The methods available in the **`Lexicon`** class appear in Table 4-6. The most commonly used method is **`containsWord`**, which checks to see if a word is in the lexicon. Assuming that you have initialized the variable **`english`** so that it contains a lexicon of all English words, you could see if a particular word exists by writing a test such as the following:

```
if (english.containsWord(word)) . . .
```

And because it is useful to make such tests in a variety of applications, you can also determine whether any English words begin with a particular substring by calling

```
if (english.containsPrefix(prefix)) . . .
```

A simple application of the `Lexicon` class

In many word games, such as the popular Scrabble™ crossword game, it is critical to memorize as many two letter words as you can, because knowing the two-letter words makes it easier to attach new words to the existing words on the board. Given that you have a lexicon containing English words, you could create such a list by generating all two-letter strings and then using the lexicon to check which of the resulting combinations are actually words. The code to do so appears in Figure 4-7.

As you will discover in section 4.8, it is also possible to solve this problem by going through the lexicon and printing out the words whose length is two. Given that there are more than 100,000 English words in the lexicon and only 676 (26 × 26) combinations of two letters, the strategy used in Figure 4-7 is probably more efficient.

Table 4-6 Methods exported by the `Lexicon` class

`size()`	Returns the number of words in the lexicon.
`isEmpty()`	Returns **`true`** if the lexicon is empty.
`add(`*word***`)`**	Adds a new word to the lexicon. If the word is already in the lexicon, this call has no effect; each word may appear only once. All words in a lexicon are stored in lower case.
`addWordsFromFile(`*name***`)`**	Adds all the words in the named file to the lexicon. The file must either be a text file, in which case the words are listed on separate lines, or a precompiled data file, in which the contents of the file match the internal structure of the lexicon. The **`addWordsFromFile`** method can read a precompiled file only if the lexicon is empty.
`containsWord(`*word***`)`**	Returns **`true`** if *word* is in the lexicon.
`containsPrefix(`*prefix***`)`**	Returns **`true`** if any of the words in the lexicon start with the specified prefix.
`clear()`	Removes all the elements from a lexicon.
`iterator()`	Returns an iterator that makes it easy to cycle through the words in the lexicon.

Figure 4-7 Program to display all two-letter English words

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

#include "genlib.h"
#include "lexicon.h"
#include <iostream>

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.containsWord(word)) {
            cout << word << endl;
         }
      }
   }
   return 0;
}
```

Why are lexicons useful if maps already exist

If you think about it, a lexicon is in many ways just a simplified version of a map in which you ignore the values altogether. It would therefore be easy enough to build most of the **Lexicon** class on top of the **Map** class. Adding a word to the lexicon corresponds to calling **put** using the word as the key; checking whether a word exists corresponds to calling **containsKey**.

Given that the **Map** class already provides most of the functionality of the **Lexicon**, it may seem odd that both classes are included in this book. When we designed the ADT library, we chose to include a separate **Lexicon** class for the following reasons:

- Not having to worry about the values in a map makes it possible to implement the lexicon in a more efficient way, particularly in terms of how much memory is required to store the data. Given the data structure used in the **Lexicon** implementation, the entire English dictionary requires approximately 350,000 bytes. If you were to use a **Map** to store the words, the storage requirements would be more than five times greater.
- The underlying representation used in the lexicon makes it possible to check not only whether a word exists in the lexicon, but also to find out whether any word in the lexicon starts with a particular set of letters (the **containsPrefix** method).
- The lexicon representation ensures that the words remain in alphabetical order.

Although these characteristics of the **Lexicon** class are clearly advantages, it is still somewhat surprising that these reasons focus on what seem to be implementation details, particularly in light of the emphasis this chapter places on ignoring such details. By choosing to include the **Lexicon** class, those details are not being revealed to the client. After reading the justification for the **Lexicon** class, you have no idea *why* it might be more efficient than the **Map** class for storing a list of words, but you might well choose to take advantage of that fact.