Chapter 12
Implementing Maps

Yea, from the table of my memory
I'll wipe away all trivial fond records

— Shakespeare, *Hamlet*, 1602

Objectives

- To understand the technique of hashing and how it applies to maps.
- To appreciate that pointers to functions can be interpreted as data values in C++.
- To be able to use function pointers in the implementation of callback and mapping functions.

One of the most useful data structures introduced in Chapter 4 was the `Map` class, which provides an association between keys and values. The primary goal of this chapter is to show you how maps can be implemented extremely efficiently using a particularly clever representation called a *hash table*. Before doing so, however, it makes sense to start with a less efficient implementation that is not nearly so clever just to make sure that you understand what is involved in implementing the `map.h` interface. The following section defines an array-based implementation for the `Map` class. The rest of the chapter then looks at various strategies for improving on that simple design.

12.1 An array-based implementation of the map interface

Figure 12-1 shows a slightly simplified implementation of the `map.h` interface, which leaves out three features of the library version of the interface: deep copying, selection using square brackets, and the ability to iterate over the keys in a map. Even in its current form, however, the interface is quite useful, and it makes sense to investigate possible implementations of the fundamental operations before extending the interface.

When you are trying to learn how a particular data structure operates, it is often helpful to start with a specific example, understand how that example works, and then generalize from that example to get a sense of how the abstraction works as a whole. Suppose, for example, that you have been asked to implement (as you will be in exercise 1 at the end of this chapter) a program that translates Roman numerals into integers. As part of that program, you will need some way of encoding the following translation table:

`I`	→	1
`V`	→	5
`X`	→	10
`L`	→	50
`C`	→	100
`D`	→	500
`M`	→	1000

Figure 12-1 Preliminary version of the `map.h` interface

```
/*
 * File: map.h
 * -----------
 * This interface exports a slightly simplified version of the Map
 * template class.
 */

#ifndef _map_h
#define _map_h

#include "genlib.h"

/*
 * Class: Map
 * ----------
 * This interface defines a class template that stores a collection
 * of key-value pairs.  The keys are always strings, but the values
 * can be of any type.  This interface defines the value type using
 * the template facility in C++, which makes it possible to specify
 * the value type in angle brackets, as in Map<int> or Map<string>.
 */
```

```
template <typename ValueType>
class Map {

public:

/*
 * Constructor: Map
 * Usage: Map<int> map;
 * --------------------
 * The constructor initializes a new empty map.
 */

   Map();

/*
 * Destructor: ~Map
 * Usage: delete mp;
 * -----------------
 * The destructor frees any heap storage associated with this map.
 */

   ~Map();

/*
 * Method: size
 * Usage: nEntries = map.size();
 * -----------------------------
 * This method returns the number of entries in this map.
 */

   int size();

/*
 * Method: isEmpty
 * Usage: if (map.isEmpty())...
 * ----------------------------
 * This method returns true if this map contains no entries,
 * false otherwise.
 */

   bool isEmpty();

/*
 * Method: clear
 * Usage: map.clear();
 * -------------------
 * This method removes all entries from this map.
 */

   void clear();
```