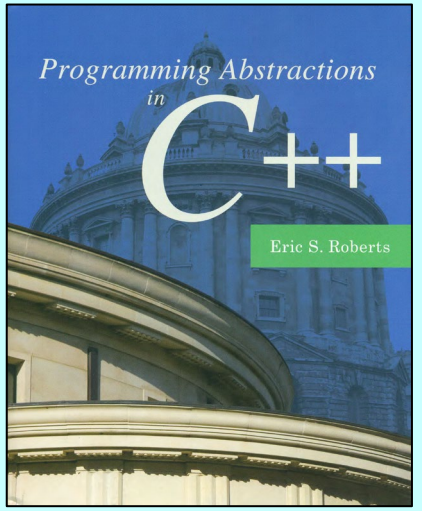

Tutorial 10

Implementing the HashMap class

Jiwei Chen
221019093@link.cuhk.edu.cn

Outline

- Quick Review of the ***StringMap*** mentioned in Chapter 15.
- Implementing the **HashMap** class
- Summary

Quick Review of the *StringMap*

The StringMap

Example: Suppose that you want to write a program that displays the name of a US state given its two-letter postal abbreviation.

FIGURE 15-4 USPS abbreviations for the 50 states

AK	Alaska	HI	Hawaii	ME	Maine	NJ	New Jersey	SD	South Dakota
AL	Alabama	IA	Iowa	MI	Michigan	NM	New Mexico	TN	Tennessee
AR	Arkansas	ID	Idaho	MN	Minnesota	NV	Nevada	TX	Texas
AZ	Arizona	IL	Illinois	MO	Missouri	NY	New York	UT	Utah
CA	California	IN	Indiana	MS	Mississippi	OH	Ohio	VA	Virginia
CO	Colorado	KS	Kansas	MT	Montana	OK	Oklahoma	VT	Vermont
CT	Connecticut	KY	Kentucky	NC	North Carolina	OR	Oregon	WA	Washington
DE	Delaware	LA	Louisiana	ND	North Dakota	PA	Pennsylvania	WI	Wisconsin
FL	Florida	MA	Massachusetts	NE	Nebraska	RI	Rhode Island	WV	West Virginia
GA	Georgia	MD	Maryland	NH	New Hampshire	SC	South Carolina	WY	Wyoming

The **StringMap**

To implement this program in C++, you need to perform the following steps, which are illustrated on the following slide:

1. Create a StringMap containing the key/value pairs.
2. Read in the two-letter abbreviation to translate.
3. Call get on the StringMap to find the state name.
4. Print out the name of the state.

Private Section of the `StringMap` Class

```
/* Private section */

private:

/* Type definition for cells in the bucket chain */

   struct Cell {
      std::string key;
      std::string value;
      Cell* link;
   };

/* Instance variables */

   Cell* *buckets;     /* Dynamic array of pointers to cells */
   int nBuckets;       /* The number of buckets in the array */
   int count;          /* The number of entries in the map   */

/* Private method prototypes */

   Cell* findCell(int bucket, std::string key);

/* Make copying illegal */

   StringMap(const StringMap & src) { }
   StringMap & operator=(const StringMap & src) { return *this; }
```

The `stringmap.cpp` Implementation

```
/*
 * File: stringmap.cpp
 * -------------------
 * This file implements the stringmap.h interface using a hash table
 * as the underlying representation.
 */

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

/*
 * Implementation notes: StringMap constructor and destructor
 * ----------------------------------------------------------
 * The constructor allocates the array of buckets and initializes each
 * bucket to the empty list.  The destructor frees the allocated cells.
 */
```

```
StringMap::StringMap() {
   nBuckets = INITIAL_BUCKET_COUNT;
   buckets = new Cell*[nBuckets];
   for (int i = 0; i < nBuckets; i++) {
      buckets[i] = NULL;
   }
}
```

```
StringMap::~StringMap() {
   for (int i = 0; i < nBuckets; i++) {
      Cell *cp = buckets[i];
      while (cp != NULL) {
         Cell *oldCell = cp;
         cp = cp->link;
         delete oldCell;
      }
   }
   delete [] buckets;
}
```

The `hashCode` Function for Strings

```
const int HASH_SEED = 5381; /* Starting point for first cycle */
const int HASH_MULTIPLIER = 33; /* Multiplier for each cycle */
const int HASH_MASK = unsigned(-1) >> 1; /* Largest positive integer */

/*
 * Function: hashCode
 * Usage: int code = hashCode(key);
 * --------------------------------
 * This function takes a string key and uses it to derive a hash code,
 * which is nonnegative integer related to the key by a deterministic
 * function that distributes keys well across the space of integers.
 * The specific algorithm used here is called djb2 after the initials
 * of its inventor, Daniel J. Bernstein, Professor of Mathematics at
 * the University of Illinois at Chicago.
 */

int hashCode(const string & str) {
   unsigned hash = HASH_SEED;
   int nchars = str.length();
   for (int i = 0; i < nchars; i++) {
      hash = HASH_MULTIPLIER * hash + str[i];
   }
   return (hash & HASH_MASK);
}
```

Things that you only need to know that you don't know. I don't need you to memorize it for the exam.

The **stringmap.cpp** Implementation

```
/*
 * Implementation notes: get
 * -------------------------
 * The get method calls findCell to search the linked list for the
 * matching key.  If no key is found, get returns the empty string.
 */

string StringMap::get(const string & key) const {
   int bucket = hashCode(key) % nBuckets;
   Cell *cp = findCell(bucket, key);
   return (cp == NULL) ? "" : cp->value;
}
```

☞

FIGURE 15-8 Code for the hash table implementation of `StringMap` (continued)

```
/*
 * Implementation notes: put
 * -------------------------
 * The put method calls findCell to search the linked list for the
 * matching key.  If a cell already exists, put simply resets the
 * value field.  If no matching key is found, put adds a new cell
 * to the beginning of the list for that chain.
 */

void StringMap::put(const string & key, const string & value) {
   int bucket = hashCode(key) % nBuckets;
   Cell *cp = findCell(bucket, key);
   if (cp == NULL) {
      cp = new Cell;
      cp->key = key;
      cp->link = buckets[bucket];
      buckets[bucket] = cp;
   }
   cp->value = value;
}

/*
 * Private method: findCell
 * Usage: Cell *cp = findCell(bucket, key);
 * ----------------------------------------
 * Finds a cell in the chain for the specified bucket that matches key.
 * If a match is found, the return value is a pointer to the cell
 * containing the matching key.  If no match is found, findCell
 * returns NULL.
 */

StringMap::Cell *StringMap::findCell(int bucket, const string & key) const {
   Cell *cp = buckets[bucket];
   while (cp != NULL && key != cp->key) {
      cp = cp->link;
   }
   return cp;
}
```

The **Goal** of this tutorial

Once you understand how to implement the **`StringMap`** class using each of the possible representations, you can add the remaining methods and use the C++ template mechanism to generalize the key and value types, just like the extension from **`CharStack`** to **`Stack`**.

Up to now, the code examples in this chapter have implemented the StringMap interface rather than the more general HashMap interface introduced in Chapter 5.

StringMap stateMap;

HashMap<string,string> stateMap;

Implementing the HashMap class

Implementing the **HashMap** class

- To implement a more general **`HashMap`**, start from **`StringMap`**:
  - Adding the missing methods for maps.
  - Generalizing the key and value types using templates.
- The algorithm for implementing hash maps imposes several requirements on the type used to represent keys, as follows:
  - The key type must be assignable so that the code can store copies of the keys in the cells.
  - The key type must support the comparison operator **`==`** so that the code can tell whether two keys are identical.
  - At the time the template for **`HashMap`** is expanded for a specific key type defined by the client, other than the built-in types like **`string`** and **`int`**, a version of the **`hashCode`** function that produces a non-negative integer for every value of the key type must also be provided by the client.

Public Section of the **StringMap.h** Interface

```
public:

/*
 * Constructor: StringMap
 * Usage: StringMap map;
 * ---------------------
 * Initializes a new empty map that uses
strings as both keys and values.
 */

   StringMap();

/*
 * Destructor: ~StringMap
 * ----------------------
 * Frees any heap storage associated with
this map.
 */

   ~StringMap();
```

```
/*
 * Method: get
 * Usage: string value = map.get(key);
 * -----------------------------------
 * Returns the value for key or the empty
string, if key is unbound.
 */

   std::string get(const std::string & key)
const;

/*
 * Method: put
 * Usage: map.put(key, value);
 * ---------------------------
 * Associates key with value in this map.
 */

   void put(const std::string & key, const
std::string & value);
```

Implementing the **HashMap** class

- Adding the missing methods for maps

TABLE 5-5 Entries exported by the `map.h` interface

Constructors

`Map<`*key type*`,` *value type*`>()`	Creates an empty map associating keys and values.

Methods

`size()`	Returns the number of key/value pairs contained in the map.
`isEmpty()`	Returns `true` if the map is empty.
`put(`*key*`,` *value*`)`	Associates the specified key and value in the map. If *key* has no previous definition, a new entry is added; if a previous association exists, the old value is discarded and replaced by the new one.
`get(`*key*`)`	Returns the value currently associated with *key* in the map. If *key* is not defined, `get` returns the default value for the value type.
`remove(`*key*`)`	Removes *key* from the map along with any associated value. If *key* does not exist, this call leaves the map unchanged.
`containsKey(`*key*`)`	Checks to see whether *key* is associated with a value. If so, this method returns `true`; if not, it returns `false`.
`clear()`	Removes all the key/value pairs from the map.

Operators

map`[`*key*`]`	The `Map` class overloads the square bracket operator so that a map acts as an *associative array* indexed by the key value. If the key does not exist in the map, the square bracket operator creates a new entry and sets its value to the default for that type.

Implementing the **HashMap** class

- Adding the missing methods for maps

The HashMap.h Interface

```
public:

/*
 * Constructor: HashMap
 * Usage: HashMap<KeyType,ValueType> map;
 * ---------------------------------------
 * Initializes a new empty map that
associates keys and values of the
 * specified types.  The type used for the
key must define the == operator,
 * and there must be a free function with
the following signature:
 *
 *     int hashCode(KeyType key);
 *
 * that returns a positive integer
determined by the key.  This interface
 * exports hashCode functions for string and
the C++ primitive types.
 */

   HashMap();
```

```
/*
 * Destructor: ~HashMap
 * Usage: (usually implicit)
 * -------------------------
 * Frees any heap storage associated with this
map.
 */

  ~HashMap();
/*
 * Method: size
 * Usage: int nEntries = map.size();
 * ---------------------------------
 * Returns the number of entries in this map.
 */

   int size();
/*
 * Method: isEmpty
 * Usage: if (map.isEmpty()) . . .
 * -------------------------------
 * Returns true if this map contains no
entries.
 */

   bool isEmpty();
```

Implementing the **HashMap** class

- Adding the missing methods for maps

```
/*
 * Method: get
 * Usage: ValueType value = map.get(key);
 * --------------------------------------
 * Returns the value associated with key
in this map.  If key is not
 * found, the get method signals an error.
 */

   ValueType get(KeyType key);

/*
 * Method: put
 * Usage: map.put(key, value);
 * ---------------------------
 * Associates key with value in this map.
Any previous value associated
 * with key is replaced by the new value.
 */

   void put(KeyType key, ValueType value);
```

```
/*
 * Method: containsKey
 * Usage: if (map.containsKey(key)) . . .
 * --------------------------------------
 * Returns true if there is an entry for key
in this map.
 */

   bool containsKey(KeyType key);

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

   void clear();

#include "hashmappriv.h"
};
```

Implementing the **HashMap** class

- Generalizing the key and value types

```
/*
 * Class: HashMap<KeyType,ValueType>
 * ---------------------------------
 * The HashMap class maintains an association between keys and values.
 * The types used for keys and values are specified as template parameters,
 * which makes it possible to use this structure with any data type.
 */

template <typename KeyType, typename ValueType>
```

```
ValueType get(KeyType key);

void put(KeyType key, ValueType value);
```

Implementing with the StringMap

Both the keys and values are always strings. The resulting class is called StringMap.

```
std::string get(const std::string & key) const;

void put(const std::string & key, const std::string & value);
```

Implementing the **HashMap** class

Notes1: The key type must be assignable so that the code can store copies of the keys in the cells.

```
/*
 * Class: HashMap<KeyType,ValueType>
 * ---------------------------------
 * The HashMap class maintains an association between keys
and values.
 * The types used for keys and values are specified as template
parameters,
 * which makes it possible to use this structure with any data
type.
 */

template <typename KeyType, typename ValueType>
```

```
ValueType get(KeyType key);

void put(KeyType key, ValueType value);
```

Implementing with the StringMap

Both the keys and values are always strings. The resulting class is called StringMap.

```
std::string get(const std::string
& key) const;

void put(const std::string &
key, const std::string & value);
```

Implementing the **HashMap** class

Notes2: The key type must support the == comparison operator so that the code can tell whether two keys are identical.

```
/*
 * Constructor: HashMap
 * Usage: HashMap<KeyType,ValueType> map;
 * ------------------------------------
 * Initializes a new empty map that
associates keys and values of the
 * specified types.  The type used for the
key must define the == operator,
 * and there must be a free function with
the following signature:
 *
 *     int hashCode(KeyType key);
 *
 * that returns a positive integer
determined by the key.  This interface
 * exports hashCode functions for string and
the C++ primitive types.
 */

   HashMap();
```

```
/*
 * Constructor: StringMap
 * Usage: StringMap map;
 * --------------------
 * Initializes a new empty map that uses strings as
both keys and values.
 */

   StringMap();
```

Public Section of the **HashMap** Class

Notes3: At the time the template for the **`HashMap`** class is expanded, the compiler must have access to a version of the **`hashCode`** function that produces a nonnegative integer for every value of the key type.

```
/*
 * Function: hashCode
 * Usage: int hash = hashCode(key);
 * --------------------------------
 * Returns a hash code for the specified key,
which is always a
 * nonnegative integer.  This function is
overloaded to support
 * all of the primitive types and the C++
<code>string</code> type.
 */

int hashCode(std::string key);
int hashCode(int key);
int hashCode(char key);
int hashCode(long key);
//int hashCode(double key);
```

```
/*
 * Free function: hashCode
 * Usage: int code = hashCode(key);
 * --------------------------------
 * Returns the hash code for key.
 */

int hashCode(const std::string & key);
```

A Comparison of the Private Section for StringMap and HashMap

```
/* ...*/

/*
 * Notes on the representation
 * ---------------------------
 * The HashMap class is represented using a hash table that keeps the
 * key/value pairs in an array of buckets, where each bucket is a
 * linked list of elements that share the same hash code.  If two or
 * more keys have the same hash code (which is called a "collision"),
 * each of those keys will be on the same list.
 */

private:

/* Type definition for cells in the bucket chain */

   struct Cell {
      KeyType key;
      ValueType value;
      Cell *link;
   };

/* Instance variables */

   Cell **buckets;     /* Dynamic array of pointers to cells */
   int nBuckets;       /* The number of buckets in the array */
   int count;          /* The number of entries in the map   */

/* Private methods */

/*
 * Private method: findCell
 * Usage: Cell *cp = findCell(bucket, key);
 * ----------------------------------------
 * Finds a cell in the chain for the specified bucket that matches key.
 * If a match is found, the return value is a pointer to the cell containing
 * the matching key.  If no match is found, the function returns NULL.
 * Given that this method is already embedded in a file marked private,
 * it makes sense to implement it here rather than doing so in the
 * hashmapimpl.cpp file.
 */

   Cell *findCell(int bucket, ValueType key) {
      Cell *cp = buckets[bucket];
      while (cp != NULL && key != cp->key) {
         cp = cp->link;
      }
      return cp;
   }
```

```
private:

/* Type definition for cells in the bucket chain */

   struct Cell {
      std::string key;
      std::string value;
      Cell *link;
   };

/* Constant definitions */

   static const int INITIAL_BUCKET_COUNT = 13;

/* Instance variables */

   Cell **buckets;              /* Dynamic array of pointers to cells */
   int nBuckets;                /* The number of buckets in the array */

/* Private methods */

   Cell *findCell(int bucket, const std::string & key) const;

/* Make copying illegal */

   StringMap(const StringMap & src) { }
   StringMap & operator=(const StringMap & src) { return *this; }

};
```

Postalookup.cpp

```
/* ...*/

#include <iostream>
#include "hashmap.h"
//include "simpio.h"
using namespace std;

/* Function prototypes */

void initStateMap(HashMap<string,string> & map);

/* Main program */

int main() {
   HashMap<string,string> stateMap;
   initStateMap(stateMap);
   while (true) {
      cout << "Enter two-letter state code: ";
      string code;
      getline(cin, code);
      if (code == "") break;
      if (stateMap.containsKey(code)) {
         cout << code << " = " << stateMap.get(code) << endl;
      } else {
         cout << code << " = ???" << endl;
      }
   }
   return 0;
}
```

```
/* ...*/

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

/* Function prototypes */

void initStateMap(StringMap & map);

/* Main program */

int main() {
   StringMap stateMap;
   initStateMap(stateMap);
   while (true) {
      cout << "Enter two-letter state code: ";
      string code;
      getline(cin, code);
      if (code == "") break;
      string name = stateMap.get(code);
      if (name == "") {
         cout << code << " is not a valid abbreviation" << endl;
      } else {
         cout << code << " = " << name << endl;
      }
   }
   return 0;
}
```

Postalookup.cpp

Example: Suppose that you want to write a program that displays the name of a US state given its two-letter postal abbreviation.

FIGURE 15-4 **USPS abbreviations for the 50 states**

AK	Alaska	HI	Hawaii	ME	Maine	NJ	New Jersey	SD	South Dakota
AL	Alabama	IA	Iowa	MI	Michigan	NM	New Mexico	TN	Tennessee
AR	Arkansas	ID	Idaho	MN	Minnesota	NV	Nevada	TX	Texas
AZ	Ari	IL	Illinois	MO	Missouri	NY	New York	UT	Utah
CA	Calif	IN	Indiana	MS	Mississippi	OH	Ohio	VA	Virginia
CO	Colorado	KS	Kansas	MT	Montana	OK	Oklahoma	VT	Vermont
CT	Connecticut	KY	Kentucky	NC	North Carolina	OR	Oregon	WA	Washington
DE	Delaware	LA	Louisiana	ND	North Dakota	PA	Pennsylvania	WI	Wisconsin
FL	Florida	MA	Massachusetts	NE	Nebraska	RI	Rhode Island	WV	West Virginia
GA	Georgia	MD	Maryland	NH	New Hampshire	SC	South Carolina	WY	Wyoming

More type.

Summary

1. Although templates offer considerable flexibility when you are designing a collection class, they also complicate both the interface and the implementation, making them harder to follow.

2. Some changes have to be made for implementing HashMap.

3. The Postalookup.cpp example.

Thank you!

Lots of thanks to Ming Li for providing previous slides