CHAPTER 4

C++ Maps and Hash Tables

LESSON 4.1

Maps

Maps

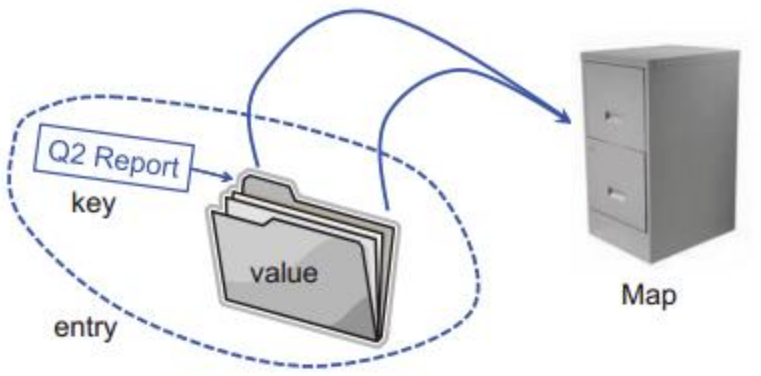


Figure 4.1.1 A conceptual illustration of the map ADT. Keys (labels) are assigned to values (folders) by a user. The resulting entries (labeled folders) are inserted into the map (file cabinet). The keys can be used later to retrieve or remove values.

A ***map*** allows us to store elements so they can be located quickly using keys. The motivation for such searches is that each element typically stores additional useful information besides its search key, but the only way to get at that information is to use the search key. Specifically, a map stores key-value pairs (k,v), which we call entries, where k is the key and v is its corresponding value. In addition, the map ADT requires that each key be unique, so the association of keys to values defines a mapping. In order to achieve the highest level of generality, we allow both the keys and the values stored in a map to be of any object type. (See Figure 4.1.1.) In a map storing student records (such as the student's name, address, and course grades), the key might be the student's ID number. In some applications, the key and the value may be the same. For example, if we had a map storing prime numbers, we could use each number itself as both a key and its value.

In either case, we use a key as a unique identifier that is assigned by an application or user to an associated value object. Thus, a map is most appropriate in situations where each key is to be viewed as a kind of unique index address for its value, that is, an object that serves as a kind of location for that value. For example, if we wish to store student records, we would probably want to use student ID objects as keys (and disallow two students having the same student

ID). In other words, the key associated with an object can be viewed as an “address” for that object. Indeed, maps are sometimes referred to as associative stores or associative containers, because the key associated with an object determines its “location” in the data structure.

Entries and Composition Pattern

As mentioned above, a map stores key-value pairs, called entries. An entry is actually an example of a more general object-oriented design pattern, the composition pattern, which defines a single object that is composed of other objects. A pair is the simplest composition, because it combines two objects into a single pair object.

To implement this concept, we define a class that stores two objects in its first and second member variables, respectively, and provides functions to access and update these variables. In Code Fragment below, we present such an implementation storing a single key-value pair. We define a class Entry, which is templated based on the key and value types. In addition to a constructor, it provides member functions that return references to the key and value. It also provides functions that allow us to set the key and value members.

```
template <typename K, typename V>
class Entry {                                   // a (key, value) pair
public:                                          // public functions
Entry(const K& k = K(), const V& v = V())       // constructor
: key(k), value(v) { }
const K& key() const { return key; }            // get key
const V& value() const { return value; }        // get value
void setKey(const K& k) { key = k; }            // set key
void setValue(const V& v) { value = v; }        // set value
private:                                        // private data
K key;                                          // key
V value;                                        // value
};
```

- A C++ class for an entry storing a key-value pair.

Map ADT

In this section, we describe a map ADT. Recall that a map is a collection of key value entries, with each value associated with a distinct key. We assume that a map provides a special pointer object, which permits us to reference entries of the map.

Such an object would normally be called a position. In order to be more consistent with the C++ Standard Template Library, we define a somewhat more general object called an iterator, which can both reference entries and navigate around the map. Given a map iterator p, the associated entry may be accessed by dereferencing the iterator, namely as *p. The individual key and value can be accessed using p->key() and p->value(), respectively.

In order to advance an iterator from its current position to the next, we overload the increment operator. Thus, ++p advances the iterator p to the next entry of the map. We can enumerate all the entries of a map M by initializing p to M.begin() and then repeatedly incrementing p as long as it is not equal to M.end().

In order to indicate that an object is not present in the map, we assume that there exists a special sentinel iterator called end. By convention, this sentinel refers to an imaginary element that lies just beyond the last element of the map.

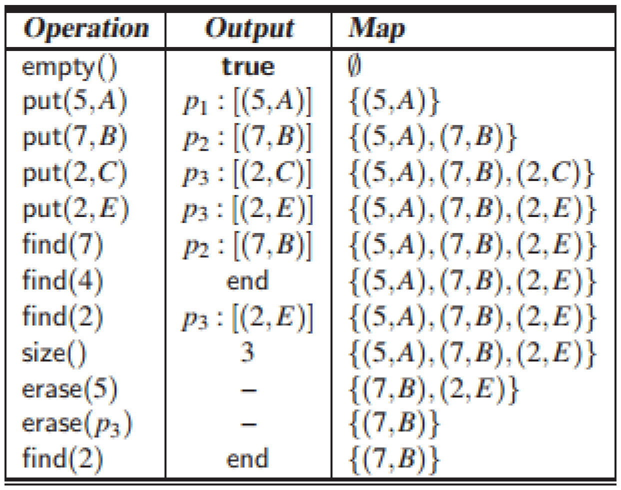
The map ADT consists of the following:

- size(): Return the number of entries in M.
- empty(): Return true if M is empty and false otherwise.
- find(k): If M contains an entry e = (k,v), with key equal to k, then
- return an iterator p referring to this entry, and otherwise
- return the special iterator end.
- put(k,v): If M does not have an entry with key equal to k, then
- add entry (k,v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.
- erase(k): Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.
- erase(p): Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.
- begin(): Return an iterator to the first entry of M.
- end(): Return an iterator to a position just beyond the end of M

We have provided two means of removing entries, one given a key and the other given an iterator. The key-based operation should be used only when it is known that the key is present in the map. Otherwise, it is necessary to first check that the key exists using the operation "p = M.find(k)," and if so, then apply the operation M.erase(p). The iterator-based removal operation has the advantage that it does not need to repeat the search for the key, and hence is more efficient.

The operation put, may either insert an entry or modify an existing entry. It is designed explicitly in this way, since we require that the keys be unique. Note that an iterator remains associated with an entry, even if its value is changed.

Example: In the following, we show the effect of a series of operations on an initially empty map storing entries with integer keys and single-character values. In the column "Output," we use the notation pi : [(k,v)] to mean that the operation returns an iterator denoted by pi that refers to the entry (k,v). The entries of the map are not listed in any particular order



Operation	*Output*	*Map*
empty()	**true**	$\emptyset$
put(5,A)	$p_1 : [(5,A)]$	$\{(5,A)\}$
put(7,B)	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
put(2,C)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
put(2,E)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,E)\}$
find(4)	end	$\{(5,A),(7,B),(2,E)\}$
find(2)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
size()	3	$\{(5,A),(7,B),(2,E)\}$
erase(5)	–	$\{(7,B),(2,E)\}$
erase(p_3)	–	$\{(7,B)\}$
find(2)	end	$\{(7,B)\}$

Figure 4.1.2

A C++ Map Interface

Before discussing specific implementations of the map ADT, we first define a C++ interface for a map in previous Code Fragment. It is not a complete C++ class, just a declaration of the public functions. The interface is templated by two types, the key type K, and the value type V.

```
template <typename K, typename V>
class Map {                               // map interface
public:
class Entry;                              // a (key,value) pair
class Iterator;                           // an iterator (and position)
int size() const;                         // number of entries in the map
bool empty() const;                       // is the map empty?
Iterator find(const K& k) const;          // find entry with key k
Iterator put(const K& k, const V& v);     // insert/replace pair (k,v)
void erase(const K& k)                    // remove entry with key k
throw(NonexistentElement);
void erase(const Iterator& p);            // erase entry at p
Iterator begin();                         // iterator to first entry
Iterator end();                           // iterator to end entry
```

};

- An informal C++ Map interface (not a complete class).

In addition to its member functions, the interface defines two types, Entry and Iterator. These two classes provide the types for the entry and iterator objects, respectively. Outside the class, these would be accessed with Map<K,V>::Entry and Map<K,V>::Iterator, respectively

We have not presented an interface for the iterator object, but its definition is similar to the STL iterator. It supports the operator "*", which returns a reference to the associated entry. The unary increment and decrement operators "++" and "– –" move an iterator forward and backwards, respectively. Also, two iterators can be compared for equality using "==".

A more sophisticated implementation would have also provided for a third type, namely a "const" iterator. Such an iterator provides a function for reading entries without modifying them.

The remainder of the interface follows from our earlier descriptions of the map operations. An error condition occurs if the function erase(k) is called with a key k that is not in the map. This is signaled by throwing an exception of type NonexistentElement.

STL Map Class

The C++ Standard Template Library (STL) provides an implementation of a map simply called map. As with many of the other STL classes we have seen, the STL map is an example of a container, and hence supports access by iterators.

In order to declare an object of type map, it is necessary to first include the definition file called "map." The map is part of the std namespace, and hence it is necessary either to use "std::map" or to provide an appropriate "using" statement.

The principal member functions of the STL map are given below. Let M be declared to be an STL map, let k be a key object, and let v be a value object for the class M. Let p be an iterator for M.

- size(): Return the number of elements in the map.
- empty(): Return true if the map is empty and false otherwise.
- find(k): Find the entry with key k and return an iterator to it; if no such key exists return end.
- operator[k]: Produce a reference to the value of key k; if no such key exists, create a new entry for key k.

- insert(pair(k,v)): Insert pair (k,v), returning an iterator to its position.
- erase(k): Remove the element with key k.
- erase(p): Remove the element referenced by iterator p.
- begin(): Return an iterator to the beginning of the map.
- end(): Return an iterator just past the end of the map.

Our map ADT is quite similar to the above functions. The insert function is a bit different. In our ADT, it is given two arguments. In the STL map, the argument is a composite object of type pair, whose first and second elements are the key and value, respectively.

The STL map provides a very convenient way to search, insert, and modify entries by overloading the subscript operator ("[ ]"). Given a map M, the assignment "M[k] = v" inserts the pair (k,v) if k is not already present, or modifies the value if it is. Thus, the subscript assignment behaves essentially the same as our ADT function put(k,v). Reading the value of M[k] is equivalent to performing find(k) and accessing the value part of the resulting iterator. An example of the use of the STL map is shown in Code Fragment below.

```
map<string, int> myMap;                         // a (string,int) map
map<string, int>::iterator p;                   // an iterator to the map
myMap.insert(pair<string, int>("Rob", 28));     // insert (“Rob”,28)
myMap["Joe"] = 38;                              // insert(“Joe”,38)
myMap["Joe"] = 50;                              // change to (“Joe”,50)
myMap["Sue"] = 75;                              // insert(“Sue”,75)
p = myMap.find("Joe");                          // *p = (“Joe”,50)
myMap.erase(p);                                 // remove (“Joe”,50)
myMap.erase("Sue");                             // remove (“Sue”,75)
p = myMap.find("Joe");
if (p == myMap.end()) cout << "nonexistent\n";  // outputs: “nonexistent”
for (p = myMap.begin(); p != myMap.end(); ++p) {        // print all entries
cout << "(" << p−>first << "," << p−>second << ")\n";
}
```

- Example of the usage of STL map

LESSON 4.2

Hash Tables

The keys associated with values in a map are typically thought of as “addresses” for those values. Examples of such applications include a compiler’s symbol table and a registry of environment variables. Both of these structures consist of a collection of symbolic names where each name serves as the “address” for properties about a variable’s type and value. One of the most efficient ways to implement a map in such circumstances is to use a hash table.

In general, a hash table consists of two major components, a *bucket array* and a *hash function*.

Bucket Arrays

A bucket array for a hash table is an array A of size N, where each cell of A is thought of as a “bucket” (that is, a collection of key-value pairs) and the integer N defines the capacity of the array. If the keys are integers well distributed in the range [0,N − 1], this bucket array is all that is needed. An entry e with key k is simply inserted into the bucket A[k]. (See Figure 4.2.1.)

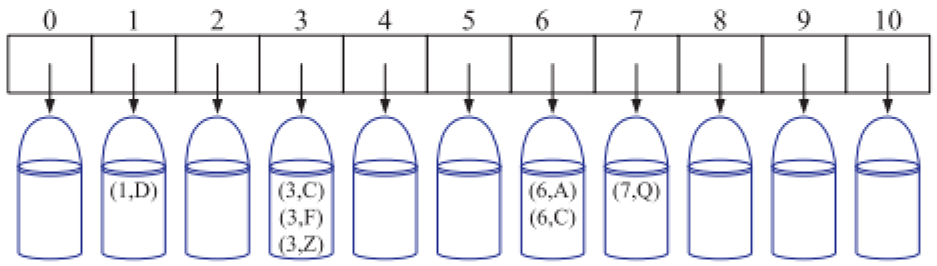


Figure 4.2.1 A bucket array of size 11 for the entries (1,D), (3,C), (3,F), (3,Z), (6,A), (6,C), and (7,Q).

If our keys are unique integers in the range [0,N − 1], then each bucket holds at most one entry. Thus, searches, insertions, and removals in the bucket array take O(1) time. This sounds like a great achievement, but it has two drawbacks. First, the space used is proportional to N. Thus, if N is much larger than the number of entries n actually present in the map, we have a waste of space. The second drawback is that keys are required to be integers in the range [0,N −1], which is often not the case. Because of these two drawbacks, we use the bucket array in conjunction with a “good” mapping from the keys to the integers in the range [0,N −1].

Hash Functions

The second part of a hash table structure is a function, h, called a hash function, that maps each key k in our map to an integer in the range [0,N − 1], where N is the capacity of the bucket array for this table. Equipped with such a hash function, h, we can apply the bucket array method to arbitrary keys. The main idea of this approach is to use the hash function value, h(k), as an index into our bucket array, A, instead of the key k (which is most likely inappropriate for use as a bucket array index). That is, we store the entry (k,v) in the bucket A[h(k)].

Of course, if there are two or more keys with the same hash value, then two different entries will be mapped to the same bucket in A. In this case, we say that a collision has occurred. Clearly, if each bucket of A can store only a single entry, then we cannot associate more than one entry with a single bucket, which is a problem in the case of collisions. To be sure, there are ways of dealing with collisions, which we discuss later, but the best strategy is to try to avoid them in the first place. We say that a hash function is "good" if it maps the keys in our map in such a way as to minimize collisions as much as possible. For practical reasons, we also would like a hash function to be fast and easy to compute. We view the evaluation of a hash function, h(k), as consisting of two actions—mapping the key k to an integer, called the hash code, and mapping the hash code to an integer within the range of indices ([0,N − 1]) of a bucket array, called the compression function. (See Figure 4.2.2.)

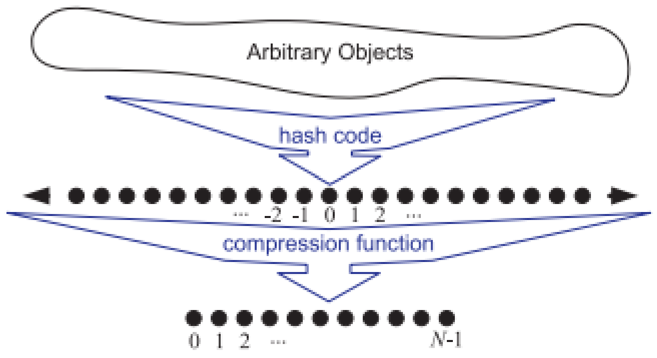


Figure 4.2.2 The two parts of a hash function: hash code and compression function.

Hash Codes

The first action that a hash function performs is to take an arbitrary key k in our map and assign it an integer value. The integer assigned to a key k is called the hash code for k. This integer value need not be in the range [0,N −1], and may even be negative, but we want the set of hash codes assigned to our keys to avoid collisions as much as possible. If the hash codes of our keys cause collisions, then there is no hope for our compression function to avoid them. In addition, to be consistent with all of our keys, the hash code we use for a key k should be the same as the hash code for any key that is equal to k.

Hash Codes in C++

The hash codes described below are based on the assumption that the number of bits of each type is known. This information is provided in the standard include file <limits>. This include file defines a templated class numeric limits. Given a base type T (such as char, int, or float), the number of bits in a variable of type T is given by “numeric limits<T>.digits.” Let us consider several common data types and some example functions for assigning hash codes to objects of these types.

Converting to an Integer

To begin, we note that, for any data type X that is represented using at most as many bits as our integer hash codes, we can simply take an integer interpretation of its bits as a hash code for X. Thus, for the C++ fundamental types char, short, and int, we can achieve a good hash code simply by casting this type to int.

On many machines, the type long has a bit representation that is twice as long as type int. One possible hash code for a long object is to simply cast it down to an integer and then apply the integer hash code. The problem is that such a hash code ignores half of the information present in the original value. If many of the keys in our map only differ in these bits, they will collide using this simple hash code. A better hash code, which takes all the original bits into consideration, sums an integer representation of the high-order bits with an integer representation of the low-order bits.

Indeed, the approach of summing components can be extended to any object x whose binary representation can be viewed as a k-tuple (x0,x1,...,xk−1) of integers, because we can then form a hash code for x as ∑(k−1 i=0) xi. For example, given any floating-point number, we can sum its mantissa and exponent as long integers, and then apply a hash code for long integers to the result.

Cyclic Shift Hash Codes

A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits. Such a function, applied to character strings in C++ could, for example, look like the following. We assume a 32-bit integer word length, and we assume access to a function hashCode(x) for integers. To achieve a 5-bit cyclic shift we form the “bitwise or” of a 5-bit left shift and a 27-bit right shift. As before, we use an unsigned integer so that right shifts fill with zeros.

```
int hashCode(const char* p, int len) {      // hash a character array

unsigned int h = 0;
for (int i = 0; i < len; i++) {
h = (h << 5) | (h >> 27);                   // 5-bit cyclic shift
h += (unsigned int) p[i];                   // add in next character
}
return hashCode(int(h));
}
```

As with the traditional polynomial hash code, using the cyclic-shift hash code requires some fine-tuning. In this case, we must wisely choose the amount to shift by for each new character.

Hashing Floating Point Quantities

On most machines, types int and float are both 32-bit quantities. Nonetheless, the approach of casting a float variable to type int would not produce a good hash function, since this would truncate the fractional part of the floating-point value. For the purposes of hashing, we do not really care about the number’s value. It is sufficient to treat the number as a sequence of bits. Assuming that a char is stored as an 8-bit byte, we could interpret a 32-bit float as a four-element character array, and a 64-bit double as an eight-element character array. C++ provides an operation called a reinterpret cast, to cast between such unrelated types. This cast treats quantities as a sequence of bits and makes no attempt to intelligently convert the meaning of one quantity to another.

For example, we could design a hash function for a float by first reinterpreting it as an array of characters and then applying the character-array hashCode function defined above. We use the operator sizeof, which returns the number of bytes in a type.

```
int hashCode(const float& x) {                                   // hash a float
int len = sizeof(x);
const char* p = reinterpret cast<const char*>(&x);
return hashCode(p, len);
}
```

Reinterpret casts are generally not portable operations, since the result depends on the particular machine's encoding of types as a pattern of bits. In our case, portability is not an issue since we are interested only in interpreting the floating point value as a sequence of bits. The only property that we require is that float variables with equal values must have the same bit sequence.

A C++ Hash Table Implementation

In Code Fragments below, we present a C++ implementation of the map ADT, called HashMap, which is based on hashing with separate chaining. The class is templated with the key type K, the value type V, and the hash comparator type H. The hash comparator defines a function, hash(k), which maps a key into an integer index. As with less-than comparators, a hash comparator class does this by overriding the "()" operator.

We present the general class structure in Code Fragment below. The definition begins with the public types required by the map interface, the entry type Entry, and the iterator type Iterator. This is followed by the declarations of the public member functions. We then give the private member data, which consists of the number of entries n, the hash comparator function hash, and the bucket array B. The first is a declaration of some utility types and functions and the second is the declaration of the map's iterator class.

```
template <typename K, typename V, typename H>
class HashMap {
public:                                     // public types
typedef Entry<const K,V> Entry;             // a (key,value) pair
class Iterator;                             // a iterator/position
public:                                     // public functions
HashMap(int capacity = 100);                // constructor
int size() const;                           // number of entries
bool empty() const;                         // is the map empty?
Iterator find(const K& k);                  // find entry with key k
Iterator put(const K& k, const V& v);       // insert/replace (k,v)
void erase(const K& k);                     // remove entry with key k
void erase(const Iterator& p);              // erase entry at p
Iterator begin();                           // iterator to first entry
Iterator end();                             // iterator to end entry
protected:                                  // protected types
typedef std::list<Entry> Bucket;            // a bucket of entries
typedef std::vector<Bucket> BktArray;       // a bucket array
        // . . .insert HashMap utilities here
private:
int n;                                      // number of entries
H hash;                                     // the hash comparator
BktArray B;                                 // bucket array
public:                                     // public types
        // . . .insert Iterator class declaration here
};
```

- The class HashMap, which implements the map ADT.

We have defined the key part of Entry to be "const K," rather than "K." This prevents a user from inadvertently modifying a key. The class makes use of two major data types. The first is an STL list of entries, called a Bucket, each storing a single bucket. The other is an STL vector of buckets, called BktArray.

Before describing the main elements of the class, we introduce a few local (protected) utilities in Code Fragment below. We declare three helper functions, finder, inserter, and eraser, which, respectively, handle the low-level details of finding, inserting, and removing entries. For convenience, we define two iterator types, one called BItor for iterating over the buckets of the bucket array, and one called EItor, for iterating over the entries of a bucket. We also give two utility functions, nextBkt and endOfBkt, which are used to iterate through the entries of a single bucket.

```
Iterator finder(const K& k);                        // find utility
Iterator inserter(const Iterator& p, const Entry& e); // insert utility
void eraser(const Iterator& p);                     // remove utility
typedef typename BktArray::iterator BItor;          // bucket iterator
typedef typename Bucket::iterator EItor;            // entry iterator
static void nextEntry(Iterator& p)                  // bucket's next entry
{ ++p.ent; }
static bool endOfBkt(const Iterator& p)             // end of bucket?
{ return p.ent == p.bkt−>end(); }
```

- Declarations of utilities to be inserted into HashMap.

We present the class Iterator in Code Fragment below. An iterator needs to store enough information about the position of an entry to allow it to navigate. The members ent, bkt, and ba store, respectively, an iterator to the current entry, the bucket containing this entry, and the bucket array containing the bucket. The first two are of types EItor and BItor, respectively, and the third is a pointer. Our implementation is minimal. In addition to a constructor, we provide operators for dereferencing ("*"), testing equality ("=="), and advancing through the map ("++").

```
class Iterator {                                  // an iterator (& position)
private:
EItor ent;                                        // which entry
BItor bkt;                                        // which bucket
const BktArray* ba;                               // which bucket array
public:
Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())
: ent(q), bkt(b), ba(&a) { }
Entry& operator*() const;                         // get entry
bool operator==(const Iterator& p) const; // are iterators equal?
Iterator& operator++();                           // advance to next entry
friend class HashMap;                             // give HashMap access
};
```

- Declaration of the Iterator class for HashMap