Ve 280

Programming and Introductory Data Structures

Standard Template Library:

Associative Containers

Outline

- Associative Containers
  - map
  - set

Review: Associative Containers

- Elements in an associative container are stored and retrieved by a **key**, in contrast to elements in a sequential container, which are stored and accessed sequentially by their position within the container

- Two primary associative container types: `map` and `set`
  - Elements in a `map` are (key, value) pairs
  - `set` contains only a key and supports efficient queries to whether a given key is present

Associative vs. Sequential Containers

- The associative container types define additional operations
- The big difference: for associative containers, elements are ordered by key
- There is one important consequence of the fact that elements are ordered by key:
  - When we iterate across an associative container, we are guaranteed that the elements are **accessed in key order**, irrespective of the order in which the elements were placed in the container.

Map

- map is also known as **associative array**
- It stores (key, value) pair
- To use, `#include <map>`
- Constructors
  - `map<k, v> m;` // Create an empty map named m // with key and value types k and v.
  - E.g., **`map<string,int> word_count`**
  - `map<k, v> m(m2);` // Create m as a copy of m2; // m and m2 must have the same key and value types
  - `map<k, v> m(b, e);` //Create m as a copy of the //elements from the range denoted by iterators b and e

Constraints on the Key Type

- Since elements in map are ordered by keys, we require that key type has an extra operation: **strict weak ordering**

Examples:
- < for int
- alphabetical order for string

- Strict weak ordering:
  - Think as less than (<)
- Technically
  - Yield false when we compare a key with itself
  - Given two keys, they cannot both be "less than" each other
  - Satisfy transitive property: if $k1<k2$ and $k2<k3$, then $k1<k3$
  - If we have two keys, neither of which is "less than" the other, then they are treated as equal

Preliminaries: the pair Type

- A simple companion type, holding two data values
- It is a template. Need to supply two type names

```
pair<string,string> spair; // hold two strings
```

- `pair<T1, T2> p1;`
  - Create a pair with two elements of types T1 and T2. The elements are value-initialized (use default constructor for class type; 0 for built-in type)
- `pair<T1, T2> p1(v1, v2);`
  - Create a pair with types T1 and T2. Initialize the first member from v1 and the second from v2.
  - **`pair<string, int> count("blue", 2);`**

Preliminaries: the pair Type

- We can access the two data members in the pair
  - `p.first //` return the **reference** to the first member
  - `p.second //` return the **reference** to the second member
  - They are **public**

- `make_pair(v1,v2)`
  - Create a new pair from the values v1 and v2. The type of the pair is inferred from the types of v1 and v2
    `pair<string,string> name = make_pair("John", "Adams");`

Map Iterator

- Dereferencing a map iterator yields a **pair** in which first member holds the **const key** and second member holds the **value**

```
map<string, int>::iterator it =
                          word_count.begin();
```

- `*it` is a reference to a `pair<const string, int>` object
  - It refers to neither the key nor the value
- To access key, use `it->first`

```
cout << it->first;
```

- However, first member is a **const key**, so we cannot change it

```
it->first = "new key"; // Error!
```

Map Iterator

```
map<string, int>::iterator it =
                                    word_count.begin();
```

- To access value, use **`it->second`**

```
cout << it->second;
```

- We can change value through iterator

```
it->second = 2;
```

Adding Elements to a map

- There are two ways:
  - Using the subscript operator
  - Using the insert member

Insert Using Subscripting

- If key `k` is not in the map `m`, you can insert `(k,v)` using

```
m[k] = v;
```

- Example

```
map <string, int> word_count; // empty map
// insert element with key "Anna";
// then assign 1 to its value
word_count["Anna"] = 1;
```

- You insert a pair **`("Anna", 1)`** into **`word_count`**.

Insert Using Subscripting

```
map <string, int> word_count; // empty map
// insert element with key "Anna";
// then assign 1 to its value
word_count["Anna"] = 1;
```

- What really happens is
  - **`word_count`** is searched for the element whose **`key`** is **`Anna`**. The element is not found.
  - A new (key, value) pair is inserted. key = "Anna". Value is value-initialized to 0.
  - The newly inserted element is fetched and is given the value 1.

Subscripting a map

- Subscripting a map behaves quite differently from subscripting an array or vector
  - Using an index (key) that **does not exist** adds an element with that index to the map
- If the key exists, the value associated with the key is returned. We can read and write to the value

```
cout << word_count["Anna"];
++word_count["Anna"]; // fetch the element
                      // and add one to it
```

- Subscripting a vector = dereferencing a vector iterator
- Subscripting a map ≠ dereferencing a map iterator

Use Subscript Behavior in a Smart Way

```
// count #times each word occurs from input
map<string, int> word_count;
// empty map from string to int
string word;
while (cin >> word)
  ++word_count[word];
```

Question: what's the behavior for the first time we encounter a word?

- The first time we encounter a word, a new element indexed by word is created and inserted into map
  - Its value is initialized with zero
- Then, the value of that element is immediately incremented. So, the count is the (correct) value of one
- If word is already in the map, then its value is incremented.

insert()

- `m.insert(e)`
  - `e` is a (key, value) pair. If the key is not in `m`, insert the pair. If the key is in `m`, then `m` is unchanged

```
word_count.insert(make_pair("Anna", 1));
```

insert()

- `m.insert(e)`
  - Returns a pair of (map iterator, bool)
    - map iterator refers to the element with key
    - bool indicates whether the element was inserted or not.

```
map<string, int> word_count;
while (cin >> word) {
  pair<map<string, int>::iterator, bool> ret =
      word_count.insert(make_pair(word, 1));
  if (!ret.second) // word already in word_count
      ++ret.first->second; // increment count
}
```

Finding and Retrieving a map Element

- The subscript operator provides the simplest method of retrieving a value
- But, it has a side effect. What is it?
  - If that key is not already in the map, then subscript inserts an element with that key.
- How can we determine if a key is present without causing it to be inserted?
  - **`m.find(k)`**

find()

- `m.find(k)`
  - Returns an iterator to the element indexed by key `k`, if there is one
  - Otherwise, returns an off-the-end iterator (i.e., end()) if the key is not present

```
int occurs = 0;
map<string,int>::iterator it =
  word_count.find("foobar");
if (it != word_count.end())
  occurs = it->second;
```

This code only looks for the element once

erase()

- `m.erase(iter)`
  - Removes element referred to by the iterator `iter` from `m`. `iter` must refer to an actual element in `m`; it must not be equal to `m.end()`.
  - Returns void.
- `m.erase(k)`
  - Removes the element with key `k` from `m` if it exists
  - Otherwise, do nothing
  - Returns the number of elements removed. For map, this is either 0 or 1

```
if (word_count.erase(rm_word)) // rm_word is a key
  cout << "ok: " << rm_word << “removed\n”;
else cout << rm_word << " not found!\n";
```

Iterate across a map

- map has `begin()` and `end()`, with which we can traverse the map
- Example: print all the elements in `word_count`

```
map<string, int>::iterator it;
for(it=word_count.begin();
    it!=word_count.end(); ++it)
    cout << it->first << " occurs "
      << it->second << " times";
```

- The output prints the words in **alphabetical order**.
  - **Note**: When we use an iterator to traverse a map, the iterators yield elements in **ascending key order**.

Outline

- Associative Containers
  - map
  - set

Set

- Set is simply a collection of keys
  - If we only want to know whether a value is present, use set
- The operations supported by set are almost same as map except there is only key but no value for set
- **One major difference**: no subscript operator []
- Constructors
  - `set<k> s;` // create an empty set
  - E.g., `set<string> str_set;`
  - `set<k> s(s1);` // copy constructor
  - `set<k> s(b, e);` // Create `s` as a copy of the // elements from the range denoted by iterators b and e

Keys in Set are const

- If we have an iterator to an element of the set, all we can do is read it; we cannot write through it.

```
set<int>::iterator it = iset.begin(); //int set
*it = 11; // Error: keys in a set are read-only
cout << *it << endl; // OK: can read the key
```

Set Operations

- `s.insert(key)`
  - Return value is like map: a pair of (set iterator, bool)
    - set iterator refers to the element with key
    - bool indicates whether the element was inserted or not.
- `s.find(key)`
  - Returns iterator to key if found; otherwise, return end()
- `s.erase(iter)`
  - Removes element referred to by the iterator `iter` from `s`.
- `s.erase(key)`
  - Removes the element with key `key` from `s` if it exists
  - Otherwise, do nothing

The Order on Elements in set

- As map, the items in the set are ordered in **ascending order of the key**

```
set<int> IntSet;
// insert items into IntSet
set<int>::iterator it;
for(it=IntSet.begin(); it!=IntSet.end(); ++it)
    cout << *it << " ";
```

- Question: how do we change the order to descending order of keys?

Define a set with a Comparator Type

- Specify a comparator type in <> and supply a object of that type in declaration
- **`set<T, COMP> s(cmpObj);`**
  - **`COMP`** is a type. Usually, either a function pointer or a function object type.
  - **`cmpObj`** is an object of **`COMP`**
  - If **`cmpObj(a,b)`** returns true, then **`a`** goes before **`b`** in the set.
- Example:

```
bool fcmp(int a, int b) {return a>b;}
set<int, bool(*)(int, int)> s(fcmp);
```

  - **`fcmp`** is an object of **`bool(*)(int a, int b)`**
  - Now larger ints are put first

Define a set with a Comparator Type

```
set<T, COMP> s(compObj);
```

- **`COMP`** can also be a function object class
- In this case, **`compObj`**'s default value is a default object of **`COMP`** type and hence, can be omitted.
- Example

```
class classcomp {
public:
  bool operator() (const int &a,
    const int &b) { return a>b; }
};
set<int, classcomp> s; // Equivalent to:
  // classcomp c;
  // set<int, classcomp> s(c);
```

Reference

- **C++ Primer (4th Edision)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
  - Chapter 10 Associative Containers