【C++】Day49

Class	C++
Date	@February 1, 2022
Material	
Series Number	
Summary	

【Ch11】Associative Container

11.4 The Unordered Containers

The new standard defines four unordered associative containers.

Rather than using a comparison operation to organize their elements, these containers use a hash function and the key type's == operator.

An unordered container is most useful when we have a key type for which there is no obvious ordering relationship among the elements.

These containers are also useful for applications in which the cost of maintaining the elements in order is prohibitive.

Tip: Use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve.

Using an Unordered Container

The unordered containers provide the same operations as the ordered containers. As a result, we can usually use an unordered container in place of the corresponding ordered container, and vice versa.

For example, we can rewrite the word counting program in the previous exercise. The only difference in the program below is the type of the container:

```
unordered_map<string, int> word_count;
string word;
while(cin >> word)
  ++word_count[word];
for(auto &elem : word_count)
  cout << elem.first << " occurs " << elem.second << " times " << endl;
```

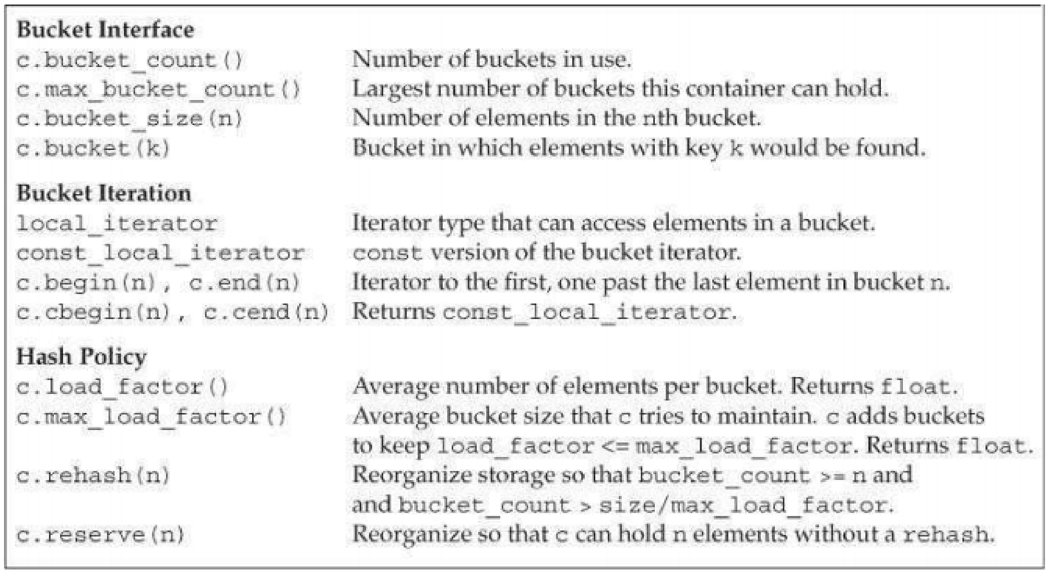
Managing the Buckets

The unordered containers are organized as a collection of buckets, each of which holds zero or more elements. These containers use a hash function to map elements to buckets.

To access an element, the container first computes the element's hash code, which tells which bucket to search. The container puts all of its elements with a given hash value into the same bucket.

If the container allows multiple elements with a given key, all the elements with the same key will be in the same bucket.
As a result, the performance of an unordered container depends on the quality of its hash function and on the number and size of tis buckets.

The unordered containers provide a set of functions, listed below that let us manage the buckets. These members let us inquire about the state of the container and force the container to reorganize itself as needed.



Bucket Interface	
`c.bucket_count()`	Number of buckets in use.
`c.max_bucket_count()`	Largest number of buckets this container can hold.
`c.bucket_size(n)`	Number of elements in the nth bucket.
`c.bucket(k)`	Bucket in which elements with key k would be found.
Bucket Iteration	
`local_iterator`	Iterator type that can access elements in a bucket.
`const_local_iterator`	`const` version of the bucket iterator.
`c.begin(n), c.end(n)`	Iterator to the first, one past the last element in bucket n.
`c.cbegin(n), c.cend(n)`	Returns `const_local_iterator`.
Hash Policy	
`c.load_factor()`	Average number of elements per bucket. Returns `float`.
`c.max_load_factor()`	Average bucket size that c tries to maintain. c adds buckets to keep `load_factor <= max_load_factor`. Returns `float`.
`c.rehash(n)`	Reorganize storage so that `bucket_count >= n` and and `bucket_count > size/max_load_factor`.
`c.reserve(n)`	Reorganize so that c can hold n elements without a `rehash`.

Requirements on Key Type for Unordered Containers

By default, the unordered containers use the `==` operator on the key type to compare elements.

They also use an object of type `hash<key_type>` to generate the hash code for each element.

However, we cannot directly define an unordered container that uses our own class types for its key type. Unlike the containers, we cannot use the hash template directly. Instead, we must supply our own version of the hash template.

We can supply functions to replace both the `==` operator and to calculate a hash code. We'll start by defining these functions:

```
size_t hasher(const Sales_data &sd) {
  return hash<string>()(sd.isbn());
}

bool eqOp(const Sales_data &lhs, const Sales_data &rhs) {
  return lhs.isbn() == rhs.isbn();
}

unordered_multiset<Sales_data, decltype(hasher)*, decltype(eqOp)*> bookstore(42, hasher, eqOp);
```