【C++】Day45(2)

Class	C++
Date	@January 25, 2022
Material	
Series Number	
Summary	

【Ch11】Associative Container

11.3 Operations on Associative Containers

The associative containers define the types listed below. These types represent the container's key and value types.

Table 11.3. Associative Container Additional Type Aliases

`key_type`	Type of the key for this container type
`mapped_type`	Type associated with each key; **map types only**
`value_type`	For `sets`, same as the `key_type` For `maps`, `pair<const key_type, mapped_type>`

For the set types, the key_type and the value_type are the same; the valeus held in a set are the keys.

In a map, the elements are key-value pairs.

```
set<string>::key_type v1; //v1 is a string
set<string>::val_type v2; //v2 is a string

map<string,int>::key_type v3; //v3 is a string
map<string, int>::mapped_type v4; //v4 is an int
map<string, int>::val_type v5; //v5 is an int
```

11.3.1 Associative Container Iterators

When we dereference an iterator, we get a reference to a value of the container's `value_type`.

In the case of map, the `value_type` is a pair in which the first holds the `const` key and second holds the value:

```
//get an iterator to an element in word_count
auto map_it = word_count.begin();

//*map_it is a reference to a pair<const string, size_t> object
cout << map_it->first; //prints the key for this element
cout << " " << map_it->second; //prints the value of the element
map_it->first = "new key"; //error: key is const
++map_it->second; //ok: we can change the value through an iterator
```

Note: It is essential to remember that the `value_type` of a map is a `pair` and that we can change the value but not the key member of that `pair`.

Iterators for sets Are const

Although the set types define both the `iterator` and `const_iterator` types, both types of iterators give us read-only access to the elements in the set.

We can use a set iterator to read, but not write, an element's value:

```
set<int> iset = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
set<int>::iteartor set_it = iset.begin();
if(set_it != iset.end()) {
  *set_it = 42; //error: keys in a set are read-only
  cout << *set_it << endl;  //ok: can read the key
}
```

Iterating across an Associative Container

The map and set types provide all the begin and end operations as usual.

Note: When we use an iterator to traverse a map, multimap, set, or multiset, the iterators yield elements in ascending key order.

Associative Containers and Algorithms

In general, we do not use the generic algorithms with the associative containers.

The fact that the keys are `const` means that we cannot pass associative container iterators to algorithms that write to the elements.

In practice, if we use a generic algorithm, we use an associative container with the algorithms either as the source sequence or as a destination. For example, we might use the generic copy algorithm to copy the elements from an associative container into another sequence.

Exercise

Exercise 11.16: Using a `map` iterator write an expression that assigns a value to an element.

Exercise 11.17: Assuming `c` is a `multiset` of `string`s and `v` is a `vector` of `string`s, explain the following calls. Indicate whether each call is legal:

ick here to view code image

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

11.3.2 Adding Elements

The `insert` members add one element or a range of elements. Because `map` and `set` contain unique keys, inserting an element that is already present has no effect:

```
vector<int> ivec = {2, 4, 6, 8, 2, 4, 6, 8};
set<int> set1; //empty set
set1.insert(ivec.cbegin(), ivec.cend()); //set2 has four elements
set1.insert({1, 3, 5, 7, 9});
```

Table 11.4. Associative Container insert Operations

`c.insert(v)` `c.emplace(args)`	`v` `value_type` object; *args* are used to construct an element. For `map` and `set`, the element is inserted (or constructed) only if an element with the given key is not already in `c`. Returns a `pair` containing an iterator referring to the element with the given key and a `bool` indicating whether the element was inserted. For `multimap` and `multiset`, inserts (or constructs) the given element and returns an iterator to the new element.
`c.insert(b, e)` `c.insert(il)`	`b` and `e` are iterators that denote a range of `c::value_type` values; `il` is a braced list of such values. Returns `void`. For `map` and `set`, inserts the elements with keys that are not already in `c`. For `multimap` and `multiset` inserts, each element in the range.
`c.insert(p, v)` `c.emplace(p, args)`	Like `insert(v)` (or `emplace(args)`), but uses iterator `p` as a hint for where to begin the search for where the new element should be stored. Returns an iterator to the element with the given key.

Adding Elements to a map

When we insert into a map, we must remember that the element type is a pair. Often, we don't have a pair object that we want to insert. Instead, we create a pair in the argument list to insert:

```
//four ways to add word to word_count
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

Testing the Return from insert

The value returned by `insert` (or `emplace`) depends on the container type and the parameters.

For the containers that have unique keys, the versions of insert and emplace that add a single element return a pair that lets us know whether the insertion happened.

The first member of the pair is an iterator to the element with the given key; the second is a bool indicating whether that element was inserted, or was already there.

If the key is already in the container, then `insert` does nothing, and the `bool` portion of the return value is false. If the key isn't present, then the element is inserted and the bool is true.

```
map<string, int> word_count;
string word;
while(cin >> word) {
  auto ret = word_count.insert({word, 1});
  if(!ret.second) //true if the insertion failed
    ++ret.first->second;
}
```