【C++】Day44

Class	C++
Date	@January 24, 2022
Material	
Series Number	
Summary	

【Ch11】Associative Containers

Associative and sequential containers differ from one another in a fundamental way: Elements in an associative container are stored and retrieved by a key.

In contrast, elements in a sequential container are stored and accessed sequentially by their position in the container.

Associative containers support efficient lookup and retrieval by a key. The two primary associative-container types are `map` and `set`.

- The elements in a `map` are `key-value pairs`: The key serves as an index into the map, and the value represents the data associated with that index.
- A `set` element contains only a key; a set supports efficient queries as to whether a given key is present.

The library provides eight associative containers, listed below. These eight differ along three dimensions. Each container is

1. A `set` or a `map`
2. Requires unique keys or allows multiple keys
3. Stores the elements in order or not.

The containers that allow multiple keys include the word multi; those that do not keep their keys ordered start with the word unordered.

Table 11.1. Associative Container Types

Elements Ordered by Key	
`map`	Associative array; holds key–value pairs
`set`	Container in which the key is the value
`multimap`	`map` in which a key can appear multiple times
`multiset`	`set` in which a key can appear multiple times
Unordered Collections	
`unordered_map`	`map` organized by a hash function
`unordered_set`	`set` organized by a hash function
`unordered_multimap`	Hashed `map`; keys can appear multiple times
`unordered_multiset`	Hashed `set`; keys can appear multiple times

- The `map` and `multimap` types are defined in the `map` header
- The `set` and `multiset` types are in the `set` header
- The unordered containers are in the `unordered_map` and `unordered_set` headers.

11.1 Using an Associative Container

A `map` is a collection of key-value pairs.

For example, each pair might contain a person's name as a key and a phone number as its value. We speak of such a data structure as "mapping names to phone numbers."

The `map` type is often referred to as an associative array. An associative array is like a "normal" array except that its subscripts don't have to be integers.

Given a map of names to phone numbers, we'd use a person's name as a subscript to fetch that person's phone number.

In contrast, a `set` is simply a collection of keys. A `set` is most useful when we simply want to know whether a value is present.

For example, a business might define a set named `bad_checks` to hold the names of individuals who have written bad checks.

Using a map

A classic example that relies on associative arrays is a word-counting program:

```
//count the number of times each word occurs in the input
map<string, size_t> word_count; //empty map from string to size_t
string word;
while(cin >> wod)
  ++word_count[word];
for(const auto &w : word_count)
  coout << w.first << " occurs "
    << w.second << (w.second > 1 ? "times" : "time") << endl;
```

To define a map, we must specify both the key and value types.

In this program, the `map` stores elements in which the keys are strings and the values are `size_t.` When we subscript `word_count`, we use a string as the subscript, and we get back the `size_t` counter associated with that string.

When we fetch an element from a map, we get an object of type `pair`.

Briefly, a `pair` is a template type that holds two (public) data elements named `first` and `second`. The pairs used by map have a first member that is the key and a second member that is the corresponding value.

Using a set

A logical extension to our program is to ignore common words like “the”, “and”, “or”, and so on. We’ll use a set to hold the words we want to ignore and count only those words that are not in this set.

```
map<string, size_type> word_count;
set<string> exclude = {"And", "Or", "The", "and", "or", "the"};
string buffer;
while(cin >> buffer) {
  if(exclude.find(buffer) == exclude.end())
    ++word_count[buffer];
}
```

To define a `set`, we specify the type of its elements, which in this case are strings. As with the sequential containers, we can list initialize the elements of an associative container.

Exercise

Exercise 11.3: Write your own version of the word-counting program.

```
#include <iostream>
#include <string>
#include <map>


int main(int argc, char* argv[]) {
    std::map<std::string, std::string::size_type> word_count;
    std::string buffer;
    while(std::cin >> buffer)
        ++word_count[buffer];
    for(const auto &elem : word_count)
        std::cout << elem.first << " occurs " << elem.second << " times" << std::endl;
    return 0;
}
```

Exercise 11.4: Extend your program to ignore case and punctuation. For example, "example." "example," and "Example" should all increment the same counter.

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <algorithm>

int main(int argc, char* argv[]) {
    std::map<std::string, std::string::size_type> word_count;
    std::set<char> exclude {',', '!', '.', '?'};
    std::string buffer;
    while(std::cin >> buffer) {
        std::string word = buffer;
        for(auto &ch : word)
            ch = tolower(ch);
```

```
        word.erase(remove_if(word.begin(), word.end(), ispunct), word.end());
        ++word_count[word];

    }
    for(const auto &elem : word_count)
        std::cout << elem.first << " occurs " << elem.second << " times" << std::endl;
    return 0;
}
```