Introduction

After sequential containers, let's enter the world of associative containers and learn about the principle of key/value pairs.

WE'LL COVER THE FOLLOWING

C++ has eight different associative containers. Four of them are associative containers with sorted keys: `std::set`, `std::map`, `std::multiset` and `std::multimap`. The other four are associative containers with unsorted keys: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` and `std::unordered_multimap`. The associative containers are special containers. That means they support all of the operations described in the chapter Interface of all containers.

Overview

All eight ordered and unordered containers have in common that they associate a key with a value. You can use the key to get the value. To get a classification of the associative containers, three simple questions need to be answered:

- Are the keys sorted?
- Does the key have an associated value?
- Can a key appear more than once?

The following table with 2^3= 8 rows gives the answers to the three questions. I answer a fourth question in the table. How fast is the access time of a key in the best case?

Associativ	Sorted	Associated	More	Access

e container		value	identical keys	time
`std::set`	yes	no	no	logarithmic
`std::unordered_set`	no	no	no	constant
`std::map`	yes	yes	no	logarithmic
`std::unordered_map`	no	yes	no	constant
`std::multiset`	yes	no	yes	logarithmic
`std::unordered_multiset`	no	no	yes	constant
`std::multimap`	yes	yes	yes	logarithmic
`std::unordered_multimap`	no	yes	yes	constant

Characteristics for associative containers

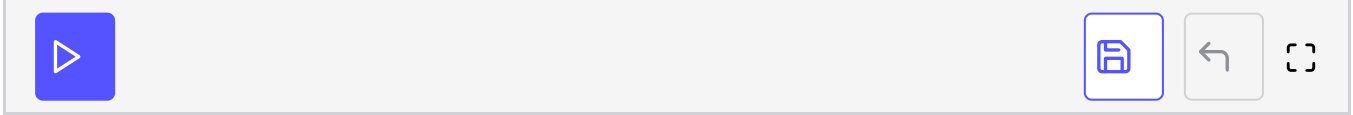
Since C++98, C++ has ordered associative containers; with C++11, C++ has in addition unordered associative containers. Both classes have a very similar interface. That's the reason that the following code sample is identical for `std::map` and `std::unordered_map`. To be more precise, the interface of `std::unordered_map` is a superset of the interface of `std::map`. The same holds for the remaining three unordered associative containers. So the porting of your code from the ordered to unordered containers is quite easy.

You can initialize the containers with an initialiser list and add new elements

with the index operator. To access the first element of the key/value pair `p`,

you have `p.first`, and for the second element, you have `p.second`. `p.first` is the key and `p.second` is the associated value of the pair.

```
// orderedUnorderedComparison.cpp
#include <iostream>
#include <map>
#include <unordered_map>

// std::map
int main(){
  std::map<std::string, int> m {{"Dijkstra", 1972}, {"Scott", 1976}};
  m["Ritchie"]= 1983;
  std::cout << m["Ritchie"];            // 1983
  std::cout<<"\n";

  for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
               // {Dijkstra,1972},{Ritchie,1983},{Scott,1976}
  std::cout<<"\n";

  m.erase("Scott");
  for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
               // {Dijkstra,1972},{Ritchie,1983}
  std::cout<<"\n";

  m.clear();
  std::cout << m.size() << std::endl;   // 0

  // std::unordered_map

  std::unordered_map<std::string, int> um {{"Dijkstra", 1972}, {"Scott", 1976}};
  um["Ritchie"]= 1983;
  std::cout << um["Ritchie"];          // 1983
  std::cout<<"\n";

  for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
                // {Ritchie,1983},{Scott,1976},{Dijkstra,1972}
  std::cout<<"\n";

  um.erase("Scott");
  for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
                // {Ritchie,1983},{Dijkstra,1972}
  std::cout<<"\n";

  um.clear();
  std::cout << um.size() << std::endl;  // 0

  return 0;
}
```



There is a subtle difference between the two program executions: The keys of the `std::map` are ordered, the keys of the `std::unordered_map` are unordered.

The question is: Why do we have such similar containers in C++? I already pointed it out in the table. The reason is so often the same: performance. The access time to the keys of an **unordered** associative container is constant and therefore independent of the size of the container. If the containers are big enough, the performance difference is significant. Have a look at the section about the performance.