Chapter 15. Boost.Unordered

Boost.Unordered provides the classes `boost::unordered_set`, `boost::unordered_multiset`, `boost::unordered_map`, and `boost::unordered_multimap`. These classes are identical to the hash containers that were added to the standard library with C++11. Thus, you can ignore the containers from Boost.Unordered if you work with a development environment supporting C++11.

Example 15.1. Using `boost::unordered_set`

```
#include <boost/unordered_set.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::unordered_set<std::string> unordered_set;
  unordered_set set;

  set.emplace("cat");
  set.emplace("shark");
  set.emplace("spider");

  for (const std::string &s : set)
    std::cout << s << '\n';

  std::cout << set.size() << '\n';
  std::cout << set.max_size() << '\n';

  std::cout << std::boolalpha << (set.find("cat") != set.end()) << '\n';
  std::cout << set.count("shark") << '\n';
}
```

`boost::unordered_set` can be replaced with `std::unordered_set` in Example 15.1. `boost::unordered_set` doesn't differ from `std::unordered_set`.

Example 15.2. Using `boost::unordered_map`

```
#include <boost/unordered_map.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::unordered_map<std::string, int> unordered_map;
  unordered_map map;

  map.emplace("cat", 4);
  map.emplace("shark", 0);
  map.emplace("spider", 8);

  for (const auto &p : map)
    std::cout << p.first << ";" << p.second << '\n';

  std::cout << map.size() << '\n';
  std::cout << map.max_size() << '\n';

  std::cout << std::boolalpha << (map.find("cat") != map.end()) << '\n';
  std::cout << map.count("shark") << '\n';
}
```

Example 15.2 uses `boost::unordered_map` to store the names and the number of legs for several animals. Once again, `boost::unordered_map` could be replaced with `std::unordered_map`.

Example 15.3. User-defined type with Boost.Unordered

```
#include <boost/unordered_set.hpp>
#include <string>
#include <cstddef>

struct animal
{
  std::string name;
  int legs;
};

bool operator==(const animal &lhs, const animal &rhs)
{
  return lhs.name == rhs.name && lhs.legs == rhs.legs;
}

std::size_t hash_value(const animal &a)
{
  std::size_t seed = 0;
  boost::hash_combine(seed, a.name);
  boost::hash_combine(seed, a.legs);
  return seed;
}

int main()
{
  typedef boost::unordered_set<animal> unordered_set;
  unordered_set animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});
}
```

In Example 15.3 elements of type `animal` are stored in a container of type `boost::unordered_set`. Because the hash function of `boost::unordered_set` doesn't know the class `animal`, hash values can't be automatically calculated for elements of this type. That's why a hash function must be defined – otherwise the example can't be compiled.

The name of the hash function to define is `hash_value()`. It must expect as its sole parameter an object of the type the hash value should be calculated for. The type of the return value of `hash_value()` must be `std::size_t`.

The function `hash_value()` is automatically called when the hash value has to be calculated for an object. This function is defined for various types in the Boost libraries, including `std::string`. For user-defined types like `animal`, it must be defined by the developer.

Usually, the definition of `hash_value()` is rather simple: Hash values are created by accessing the member variables of an object one after another. This is done with the function `boost::hash_combine()`, which is provided by Boost.Hash and defined in `boost/functional/hash.hpp`. You don't have to include this header file if you use Boost.Unordered because all containers from this library access Boost.Hash to calculate hash values.

In addition to defining `hash_value()`, you need to make sure two objects can be compared using `==`. That's why the operator `operator==` is overloaded for `animal` in Example 15.3.

The hash containers from the C++11 standard library use a hash function from the header file `functional`. The hash containers from Boost.Unordered expect the hash function `hash_value()`. Whether you use Boost.Hash within `hash_value()` doesn't matter. Boost.Hash makes sense because functions like `boost::hash_combine()` make it easier to calculate hash values from multiple member variables step by step. However, this is only an implementation detail of `hash_value()`. Apart from the different hash functions used, the hash containers from Boost.Unordered and the standard library are basically equivalent.