Chapter 13. Boost.Bimap

The library Boost.Bimap is based on Boost.MultiIndex and provides a container that can be used immediately without being defined first. The container is similar to `std::map`, but supports looking up values from either side. Boost.Bimap allows you to create maps where either side can be the key, depending on how you access the map. When you access the left side as the key, the right side is the value, and vice versa.

Example 13.1. Using `boost::bimap`

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

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

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

  std::cout << animals.left.count("cat") << '\n';
  std::cout << animals.right.count(8) << '\n';
}
```

`boost::bimap` is defined in `boost/bimap.hpp` and provides two member variables, **`left`** and **`right`**, which can be used to access the two containers of type `std::map` that are unified by `boost::bimap`. In Example 13.1, **`left`** uses keys of type `std::string` to access the container, and **`right`** uses keys of type `int`.

Besides supporting access to individual records using a left or right container, `boost::bimap` allows you to view records as relations (see Example 13.2).

Example 13.2. Accessing relations

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

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

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

  for (auto it = animals.begin(); it != animals.end(); ++it)
    std::cout << it->left << " has " << it->right << " legs\n";
}
```

It is not necessary to access records using **`left`** or **`right`**. By iterating over records, the left and right parts of an individual record are made available through the iterator.

While `std::map` is accompanied by a container called `std::multimap`, which can store multiple records using the same key, there is no such equivalent for `boost::bimap`. However, this does not mean that storing multiple records with the same key inside a container of type `boost::bimap` is impossible. Strictly speaking, the two required template parameters specify container types for **`left`** and **`right`**, not the types of the elements to store. If no container type is specified, the container type `boost::bimaps::set_of` is used by default. This container, like `std::map`, only accepts records with unique keys.

Example 13.3. Using `boost::bimaps::set_of` explicitly

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

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>,
    boost::bimaps::set_of<int>> bimap;
  bimap animals;

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

  std::cout << animals.left.count("spider") << '\n';
  std::cout << animals.right.count(8) << '\n';
}
```

Example 13.3 specifies `boost::bimaps::set_of`.

Other container types besides `boost::bimaps::set_of` can be used to customize `boost::bimap`.

Example 13.4. Allowing duplicates with `boost::bimaps::multiset_of`

```
#include <boost/bimap.hpp>
#include <boost/bimap/multiset_of.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>,
    boost::bimaps::multiset_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"dog", 4});

  std::cout << animals.left.count("dog") << '\n';
  std::cout << animals.right.count(4) << '\n';
}
```

Example 13.4 uses the container type `boost::bimaps::multiset_of`, which is defined in `boost/bimap/multiset_of.hpp`. It works like `boost::bimaps::set_of`, except that keys don't need to be unique. Example 13.4 will successfully display `2` when searching for animals with four legs.

Because `boost::bimaps::set_of` is used by default for containers of type `boost::bimap`, the header file `boost/bimap/set_of.hpp` does not need to be included explicitly. However, when using other container types, the corresponding header files must be included.

In addition to the classes shown above, Boost.Bimap provides the following: `boost::bimaps::unordered_set_of`, `boost::bimaps::unordered_multiset_of`, `boost::bimaps::list_of`, `boost::bimaps::vector_of`, and `boost::bimaps::unconstrained_set_of`. Except for `boost::bimaps::unconstrained_set_of`, all of the other container types operate just like their counterparts from the standard library.

Example 13.5. Disabling one side with `boost::bimaps::unconstrained_set_of`

```
#include <boost/bimap.hpp>
#include <boost/bimap/unconstrained_set_of.hpp>
#include <boost/bimap/support/lambda.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string,
    boost::bimaps::unconstrained_set_of<int>> bimap;
  bimap animals;

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

  auto it = animals.left.find("cat");
  animals.left.modify_key(it, boost::bimaps::_key = "dog");

  std::cout << it->first << '\n';
}
```

`boost::bimaps::unconstrained_set_of` can be used to disable one side of `boost::bimap`. In Example 13.5, `boost::bimap` behaves like `std::map`. You can't access **`right`** to search for animals by legs.

Example 13.5 illustrates another reason why it can make sense to prefer `boost::bimap` over `std::map`. Since Boost.Bimap is based on Boost.MultiIndex, member functions from Boost.MultiIndex are available. Example 13.5 modifies a key using `modify_key()` – something that is not possible with `std::map`.

Note how the key is modified. A new value is assigned to the current key using **`boost::bimaps::_key`**, which is a placeholder that is defined in `boost/bimap/support/lambda.hpp`.

`boost/bimap/support/lambda.hpp` also defines **`boost::bimaps::_data`**. When calling the member function `modify_data()`, **`boost::bimaps::_data`** can be used to modify a value in a container of type `boost::bimap`.

Exercise

Implement the class `animals_container` with Boost.Bimap:

```
#include <boost/optional.hpp>
#include <string>
#include <vector>
#include <iostream>

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

    animal(std::string n, int l) : name(n), legs(l) {}
};

class animals_container
{
public:
    void add(animal a)
    {
        // TODO: Implement this member function.
    }

    boost::optional<animal> find_by_name(const std::string &name) const
    {
        // TODO: Implement this member function.
        return {};
    }

    std::vector<animal> find_by_legs(int from, int to) const
    {
        // TODO: Implement this member function.
        return {};
    }
};

int main()
{
    animals_container animals;
    animals.add({ "cat", 4 });
    animals.add({ "ant", 6 });
    animals.add({ "spider", 8 });
    animals.add({ "shark", 0 });

    auto shark = animals.find_by_name("shark");
    if (shark)
        std::cout << "shark has " << shark->legs << " legs\n";

    auto animals_with_4_to_6_legs = animals.find_by_legs(4, 7);
    for (auto animal : animals_with_4_to_6_legs)
        std::cout << animal.name << " has " << animal.legs << " legs\n";
}
```