Chapter 12. Boost.MultiIndex

Boost.MultiIndex makes it possible to define containers that support an arbitrary number of interfaces. While `std::vector` provides an interface that supports direct access to elements with an index and `std::set` provides an interface that sorts elements, Boost.MultiIndex lets you define containers that support both interfaces. Such a container could be used to access elements using an index and in a sorted fashion.

Boost.MultiIndex can be used if elements need to be accessed in different ways and would normally need to be stored in multiple containers. Instead of having to store elements in both a vector and a set and then synchronizing the containers continuously, you can define a container with Boost.MultiIndex that provides a vector interface and a set interface.

Boost.MultiIndex also makes sense if you need to access elements based on multiple different properties. In Example 12.1, animals are looked up by name and by number of legs. Without Boost.MultiIndex, two hash containers would be required – one to look up animals by name and the other to look them up by number of legs.

Example 12.1. Using `boost::multi_index::multi_index_container`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

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

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

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

  std::cout << animals.count("cat") << '\n';
  const animal_multi::nth_index<1>::type &legs_index = animals.get<1>();
  std::cout << legs_index.count(8) << '\n';
}
```

When you use Boost.MultiIndex, the first step is to define a new container. You have to decide which interfaces your new container should support and which element properties it should access.

The class `boost::multi_index::multi_index_container`, which is defined in `boost/multi_index_container.hpp`, is used for every container definition. This is a class template that requires at least two parameters. The first parameter is the type of elements the container should store – in Example 12.1 this is a user-defined class called `animal`. The second parameter is used to denote different indexes the container should provide.

The key advantage of containers based on Boost.MultiIndex is that you can access elements via different interfaces. When you define a new container, you can specify the number and type of interfaces. The container in Example 12.1 needs to support searching for animals by name or number of legs, so two interfaces are defined. Boost.MultiIndex calls these interfaces indexes – that's where the library's name comes from.

Interfaces are defined with the help of the class `boost::multi_index::indexed_by`. Each interface is passed as a template parameter. Two interfaces of type `boost::multi_index::hashed_non_unique`, which is defined in `boost/multi_index/hashed_index.hpp`, are used in Example 12.1. Using these interfaces makes the container behave like `std::unordered_set` and look up values using a hash value.

The class `boost::multi_index::hashed_non_unique` is a template as well and expects as its sole parameter a class that calculates hash values. Because both interfaces of the container need to look up animals, one interface calculates hash values for the name, while the other interface does so for the number of legs.

Boost.MultiIndex offers the helper class template `boost::multi_index::member`, which is defined in `boost/multi_index/member.hpp`, to access a member variable. As seen in Example 12.1, several parameters have been specified to let `boost::multi_index::member` know which member variable of `animal` should be accessed and which type the member variable has.

Even though the definition of `animal_multi` looks complicated at first, the class works like a map. The name and number of legs of an animal can be regarded as a key/value pair. The advantage of the container `animal_multi` over a map like `std::unordered_map` is that animals can be looked up by name or by number of legs. `animal_multi` supports two interfaces, one based on the name and one based on the number of legs. The interface determines which member variable is the key and which member variable is the value.

To access a MultiIndex container, you need to select an interface. If you directly access the object **`animals`** using `insert()` or `count()`, the first interface is used. In Example 12.1, this is the hash container for the member variable **`name`**. If you need a different interface, you must explicitly select it.

Interfaces are numbered consecutively, starting at index 0 for the first interface. To access the second interface – as shown in Example 12.1 – call the member function `get()` and pass in the index of the desired interface as the template parameter.

The return value of `get()` looks complicated. It accesses a class of the MultiIndex container called `nth_index` which, again, is a template. The index of the interface to be used must be specified as a template parameter. This index must be the same as the one passed to `get()`. The final step is to access the type definition named `type` of `nth_index`. The value of `type` represents the type of the corresponding interface. The following examples use the keyword `auto` to simplify the code.

Although you do not need to know the specifics of an interface, since they are automatically derived from `nth_index` and `type`, you should still understand what kind of interface is accessed. Since interfaces are numbered consecutively in the container definition, this can be answered easily, since the index is passed to both `get()` and `nth_index`. Thus, **`legs_index`** is a hash interface that looks up animals by legs.

Because data such as names and legs can be keys of the MultiIndex container, they cannot be arbitrarily changed. If the number of legs is changed after an animal has been looked up by name, an interface using legs as a key would be unaware of the change and would not know that a new hash value needs to be calculated.

Just as the keys in a container of type `std::unordered_map` cannot be modified, neither can data stored within a MultiIndex container. Strictly speaking, all data stored in a MultiIndex container is constant. This includes member variables that aren't used by any interface. Even if no interface accesses **`legs`**, **`legs`** cannot be changed.

To avoid having to remove elements from a MultiIndex container and insert new ones, Boost.MultiIndex provides member functions to change values directly. Because these member functions operate on the MultiIndex container itself, and because no element in a container is modified directly, all interfaces will be notified and can calculate new hash values.

Example 12.2. Changing elements in a MultiIndex container with `modify()`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

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

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

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

  auto &legs_index = animals.get<1>();
  auto it = legs_index.find(4);
  legs_index.modify(it, [](animal &a){ a.name = "dog"; });

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

Every interface offered by Boost.MultiIndex provides the member function `modify()`, which operates directly on the container. The object to be modified is identified through an iterator passed as the first parameter to `modify()`. The second parameter is a function or function object that expects as its sole parameter an object of the type stored in the container. The function or function object can change the element as much as it wants. Example 12.2 illustrates how to use the member function `modify()` to change an element.

So far, only one interface has been introduced: `boost::multi_index::hashed_non_unique`, which calculates a hash value that does not have to be unique. In order to guarantee that no value is stored twice, use `boost::multi_index::hashed_unique`. Please note that values cannot be stored if they don’t satisfy the requirements of all interfaces of a particular container. If one interface does not allow you to store values multiple times, it does not matter whether another interface does allow it.

Example 12.3. A MultiIndex container with `boost::multi_index::hashed_unique`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

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

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

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

  auto &legs_index = animals.get<1>();
  std::cout << legs_index.count(4) << '\n';
}
```

The container in Example 12.3 uses `boost::multi_index::hashed_unique` as the second interface. That means no two animals with the same number of legs can be stored in the container because the hash values would be the same.

The example tries to store a dog, which has the same number of legs as the already stored cat. Because this violates the requirement of having unique hash values for the second interface, the dog will not be stored in the container. Therefore, when searching for animals with four legs, the program displays 1, because only the cat was stored and counted.

Example 12.4. The interfaces `sequenced`, `ordered_non_unique` and `random_access`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/random_access_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

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

typedef multi_index_container<
  animal,
  indexed_by<
    sequenced<>,
    ordered_non_unique<
      member<
        animal, int, &animal::legs
      >
    >,
    random_access<>
  >
> animal_multi;

int main()
{
  animal_multi animals;
  animals.push_back({"cat", 4});
  animals.push_back({"shark", 0});
  animals.push_back({"spider", 8});

  auto &legs_index = animals.get<1>();
  auto it = legs_index.lower_bound(4);
  auto end = legs_index.upper_bound(8);
  for (; it != end; ++it)
    std::cout << it->name << '\n';

  const auto &rand_index = animals.get<2>();
  std::cout << rand_index[0].name << '\n';
}
```

Example 12.4 introduces the last three interfaces of Boost.MultiIndex: `boost::multi_index::sequenced`, `boost::multi_index::ordered_non_unique`, and `boost::multi_index::random_access`.

The interface `boost::multi_index::sequenced` allows you to treat a MultiIndex container like a list of type `std::list`. Elements are stored in the given order.

With the interface `boost::multi_index::ordered_non_unique`, objects are automatically sorted. This interface requires that you specify a sorting criterion when defining the container. Example 12.4 sorts objects of type `animal` by the number of legs using the helper class `boost::multi_index::member`.

`boost::multi_index::ordered_non_unique` provides special member functions to find specific ranges within the sorted values. Using `lower_bound()` and `upper_bound()`, the program searches for animals that have at least four and no more than eight legs. Because they require elements to be sorted, these member functions are not provided by other interfaces.

The final interface introduced is `boost::multi_index::random_access`, which allows you to treat the MultiIndex container like a vector of type `std::vector`. The two most prominent member functions are `operator[]` and `at()`.

`boost::multi_index::random_access` includes `boost::multi_index::sequenced`. With `boost::multi_index::random_access`, all member functions of `boost::multi_index::sequenced` are available as well.

Now that we've covered the four interfaces of Boost.MultiIndex, the remainder of this chapter focuses on *key extractors*. One of the key extractors has already been introduced: `boost::multi_index::member`, which is defined in `boost/multi_index/member.hpp`. This helper class is called a key extractor because it allows you to specify which member variable of a class should be used as the key of an interface.

Example 12.5 introduces two more key extractors.

Example 12.5. The key extractors `identity` and `const_mem_fun`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/identity.hpp>
```

```
#include <boost/multi_index/mem_fun.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::multi_index;

class animal
{
public:
  animal(std::string name, int legs) : name_{std::move(name)},
    legs_(legs) {}
  bool operator<(const animal &a) const { return legs_ < a.legs_; }
  const std::string &name() const { return name_; }
private:
  std::string name_;
  int legs_;
};

typedef multi_index_container<
  animal,
  indexed_by<
    ordered_unique<
      identity<animal>
    >,
    hashed_unique<
      const_mem_fun<
        animal, const std::string&, &animal::name
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

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

  std::cout << animals.begin()->name() << '\n';

  const auto &name_index = animals.get<1>();
  std::cout << name_index.count("shark") << '\n';
}
```

The key extractor `boost::multi_index::identity`, defined in `boost/multi_index/identity.hpp`, uses elements stored in the container as keys. This requires the class `animal` to be sortable because objects of type `animal` will be used as the key for the interface `boost::multi_index::ordered_unique`. In Example 12.5, this is achieved through the overloaded `operator<`.

The header file `boost/multi_index/mem_fun.hpp` defines two key extractors – `boost::multi_index::const_mem_fun` and `boost::multi_index::mem_fun` – that use the return value of a member function as a key. In Example 12.5, the return value of `name()` is used that way. `boost::multi_index::const_mem_fun` is used for constant member functions, while `boost::multi_index::mem_fun` is used for non-constant member functions.

Boost.MultiIndex offers two more key extractors: `boost::multi_index::global_fun` and `boost::multi_index::composite_key`. The former can be used for free-standing or static member functions, and the latter allows you to design a key extractor made up of several other key extractors.

Exercise

Define the class `animals_container` with Boost.MultiIndex:

```cpp
#include <string>
#include <vector>
#include <iostream>

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

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

    const animal *find_by_name(const std::string &name) const
    {
        // TODO: Implement this member function.
        return nullptr;
    }

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

    std::vector<animal> find_by_tail(bool has_tail) const
    {
        // TODO: Implement this member function.
        return {};
    }
};

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

    const animal *a = animals.find_by_name("cat");
    if (a)
        std::cout << "cat has " << a->legs << " legs\n";

    auto animals_with_6_to_8_legs = animals.find_by_legs(6, 9);
    for (auto a : animals_with_6_to_8_legs)
        std::cout << a.name << " has " << a.legs << " legs\n";

    auto animals_without_tail = animals.find_by_tail(false);
    for (auto a : animals_without_tail)
        std::cout << a.name << " has no tail\n";
}
```