Chapter 66. Boost.Flyweight

Boost.Flyweight is a library that makes it easy to use the design pattern of the same name. Flyweight helps save memory when many objects share data. With this design pattern, instead of storing the same data multiple times in objects, shared data is kept in just one place, and all objects refer to that data. While you can implement this design pattern with, for example, pointers, it is easier to use Boost.Flyweight.

Example 66.1. A hundred thousand identical strings without Boost.Flyweight

```
#include <string>
#include <vector>

struct person
{
  int id_;
  std::string city_;
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Example 66.1 creates a hundred thousand objects of type `person`. `person` defines two member variables: **`id_`** identifies persons, and **`city_`** stores the city people live in. In this example, all people live in Berlin. That's why **`city_`** is set to "Berlin" in all hundred thousand objects. Thus, the example uses a hundred thousand strings all set to the same value. With Boost.Flyweight, one string – instead of thousands – can be used and memory consumption reduced.

Example 66.2. One string instead of a hundred thousand strings with Boost.Flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

To use Boost.Flyweight, include `boost/flyweight.hpp`, as in Example 66.2. Boost.Flyweight provides additional header files that only need to be included if you need to change the detailed

library settings.

All classes and functions are in the namespace `boost::flyweights`. Example 66.2 only uses the class `boost::flyweights::flyweight`, which is the most important class in this library. The member variable **`city_`** uses the type `flyweight<std::string>` rather than `std::string`. This is all you need to change to use this design pattern and reduce the memory requirements of the program.

Example 66.3. Using `boost::flyweights::flyweight` multiple times

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  flyweight<std::string> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Example 66.3 adds a second member variable, **`country_`**, to the class `person`. This member variable contains the names of the countries people live in. Since, in this example, all people live in Berlin, they all live in the same country. That's why `boost::flyweights::flyweight` is used in the definition of the member variable **`country_`**, too.

Boost.Flyweight uses an internal container to store objects. It makes sure there can't be multiple objects with same values. By default, Boost.Flyweight uses a hash container such as `std::unordered_set`. For different types, different hash containers are used. As in Example 66.3, both member variables **`city_`** and **`country_`** are strings; therefore, only one container is used. In this example, this is not a problem because the container only stores two strings: "Berlin" and "Germany." If many different cities and countries must be stored, it would be better to store cities in one container and countries in another.

Example 66.4. Using `boost::flyweights::flyweight` multiple times with tags

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct city {};
struct country {};
```

```
struct person
{
  int id_;
  flyweight<std::string, tag<city>> city_;
  flyweight<std::string, tag<country>> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Example 66.4 passes a second template parameter to `boost::flyweights::flyweight`. This is a *tag*. Tags are arbitrary types only used to differentiate the types on which **`city_`** and **`country_`** are based. Example 66.4 defines two empty structures `city` and `country`, which are used as tags. However, the example could have instead used `int`, `bool`, or any type.

The tags make **`city_`** and **`country_`** use different types. Now two hash containers are used by Boost.Flyweight – one stores cities, the other stores countries.

Example 66.5. Template parameters of `boost::flyweights::flyweight`

```
#include <boost/flyweight.hpp>
#include <boost/flyweight/set_factory.hpp>
#include <boost/flyweight/no_locking.hpp>
#include <boost/flyweight/no_tracking.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string, set_factory<>, no_locking, no_tracking> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Template parameters other than tags can be passed to `boost::flyweights::flyweight`. Example 66.5 passes `boost::flyweights::set_factory`, `boost::flyweights::no_locking`, and `boost::flyweights::no_tracking`. Additional header files are included to make use of these classes.

`boost::flyweights::set_factory` tells Boost.Flyweight to use a sorted container, such as `std::set`, rather than a hash container. With `boost::flyweights::no_locking`, support for multithreading, which is normally activated by default, is deactivated. `boost::flyweights::no_tracking` tells Boost.Flyweight to not track objects stored in internal

containers. By default, when objects are no longer used, Boost.Flyweight detects this and removes them from the containers. When `boost::flyweights::no_tracking` is set, the detection mechanism is disabled. This improves performance. However, containers can only grow and will never shrink.

Boost.Flyweight supports additional settings. Check the official documentation if you are interested in more details on tuning.

Exercise

Improve this program with Boost.Flyweight. Use Boost.Flyweight with disabled support for multithreading:

```
#include <string>
#include <vector>
#include <memory>

int main()
{
    std::vector<std::shared_ptr<std::string>> countries;
    auto germany = std::make_shared<std::string>("Germany");
    for (int i = 0; i < 500; ++i)
        countries.push_back(germany);
    auto netherlands = std::make_shared<std::string>("Netherlands");
    for (int i = 0; i < 500; ++i)
        countries.push_back(netherlands);
}
```