Chapter 18. Boost.Intrusive

Boost.Intrusive is a library especially suited for use in high performance programs. The library provides tools to create *intrusive containers*. These containers replace the known containers from the standard library. Their disadvantage is that they can't be used as easily as, for example, `std::list` or `std::set`. But they have these advantages:

- Intrusive containers don't allocate memory dynamically. A call to `push_back()` doesn't lead to a dynamic allocation with `new`. This is a one reason why intrusive containers can improve performance.
- Intrusive containers store the original objects, not copies. After all, they don't allocate memory dynamically. This leads to another advantage: Member functions such as `push_back()` don't throw exceptions because they neither allocate memory nor copy objects.

The advantages are paid for with more complicated code because preconditions must be met to store objects in intrusive containers. You cannot store objects of arbitrary types in intrusive containers. For example, you cannot put strings of type `std::string` in an intrusive container; instead you must use containers from the standard library.

Example 18.1 prepares a class `animal` to allow objects of this type to be stored in an intrusive list.

Example 18.1. Using `boost::intrusive::list`

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal a3{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(a3);

  a1.name = "dog";

  for (const animal &a : animals)
```

```
    std::cout << a.name << '\n';
}
```

In a list, an element is always accessed from another element, usually using a pointer. If an intrusive list is to store objects of type `animal` without dynamic memory allocation, pointers must exist somewhere to concatenate elements.

To store objects of type `animal` in an intrusive list, the class must provide the variables required by the intrusive list to concatenate elements. Boost.Intrusive provides *hooks* – classes from which the required variables are inherited. To allow objects of the type `animal` to be stored in an intrusive list, `animal` must be derived from the class `boost::intrusive::list_base_hook`.

Hooks make it possible to ignore the implementation details. However, it's safe to assume that `boost::intrusive::list_base_hook` provides at least two pointers because `boost::intrusive::list` is a doubly linked list. Thanks to the base class `boost::intrusive::list_base_hook`, `animal` defines these two pointers to allow objects of this type to be concatenated.

Please note that `boost::intrusive::list_base_hook` is a template that comes with default template parameters. Thus, no types need to be passed explicitly.

Boost.Intrusive provides the class `boost::intrusive::list` to create an intrusive list. This class is defined in `boost/intrusive/list.hpp` and is used like `std::list`. Elements can be added using `push_back()`, and it's also possible to iterate over elements.

It is important to understand that intrusive containers do not store copies; they store the original objects. Example 18.1 writes `dog`, `shark`, and `spider` to standard output – not `cat`. The object **`a1`** is linked into the list. That's why the change of the name is visible when the program iterates over the elements in the list and displays the names.

Because intrusive containers don't store copies, you must remove objects from intrusive containers before you destroy them.

Example 18.2. Removing and destroying dynamically allocated objects

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};
```

```
  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  animals.pop_back();
  delete a3;

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Example 18.2 creates an object of type `animal` with `new` and inserts it to the list **`animals`**. If you want to destroy the object with `delete` when you don't need it anymore, you must remove it from the list. Make sure that you remove the object from the list before you destroy it – the order is important. Otherwise, the pointers in the elements of the intrusive container might refer to a memory location that no longer contains an object of type `animal`.

Because intrusive containers neither allocate nor free memory, objects stored in an intrusive container continue to exist when the intrusive container is destroyed.

Since removing elements from intrusive containers doesn't automatically destroy them, the containers provide non-standard extensions. `pop_back_and_dispose()` is one such member function.

Example 18.3. Removing and destroying with `pop_back_and_dispose()`

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  animals.pop_back_and_dispose([](animal *a){ delete a; });

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

`pop_back_and_dispose()` removes an element from a list and destroys it. Because intrusive containers don't know how an element should be destroyed, you need to pass to `pop_back_and_dispose()` a function or function object that does know how to destroy the element. `pop_back_and_dispose()` will remove the object from the list, then call the function or function object and pass it a pointer to the object to be destroyed. Example 18.3 passes a lambda function that calls `delete`.

In Example 18.3, only the third element in **`animals`** can be removed with `pop_back_and_dispose()`. The other elements in the list haven't been created with `new` and, thus, must not be destroyed with `delete`.

Boost.Intrusive supports another mechanism to link removing and destroying of elements.

Example 18.4. Removing and destroying with auto unlink mode

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

typedef link_mode<auto_unlink> mode;

struct animal : public list_base_hook<mode>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef constant_time_size<false> constant_time_size;
  typedef list<animal, constant_time_size> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  delete a3;

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Hooks support a parameter to set a link mode. The link mode is set with the class template `boost::intrusive::link_mode`. If `boost::intrusive::auto_unlink` is passed as a template parameter, the auto unlink mode is selected.

The auto unlink mode automatically removes an element from an intrusive container when it is destroyed. Example 18.4 writes only `cat` and `shark` to standard output.

The auto unlink mode can only be used if the member function `size()`, which is provided by all intrusive containers, has no *constant complexity*. By default, it has constant complexity, which means: the time it takes for `size()` to return the number of elements doesn't depend on how many elements are stored in a container. Switching constant complexity on or off is another option to optimize performance.

To change the complexity of `size()`, use the class template `boost::intrusive::constant_time_size`, which expects either `true` or `false` as a template parameter. `boost::intrusive::constant_time_size` can be passed as a second template parameter to intrusive containers, such as `boost::intrusive::list`, to set the complexity for `size()`.

Now that we've seen that intrusive containers support link mode and that there is an option to set the complexity for `size()`, it might seem as though there is still much more to discover, but there actually isn't. There are, for example, only three link modes supported, and auto unlink mode is the only one you need to know. The default mode used if you don't pick a link mode is good enough for all other use cases.

Furthermore, there are no options for other member functions. There are no other classes, other than `boost::intrusive::constant_time_size`, that you need to learn about.

Example 18.5 introduces a hook mechanism using another intrusive container: `boost::intrusive::set`.

Example 18.5. Defining a hook for `boost::intrusive::set` as a member variable

```
#include <boost/intrusive/set.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal
{
  std::string name;
  int legs;
  set_member_hook<> set_hook;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal a3{"spider", 8};

  typedef member_hook<animal, set_member_hook<>, &animal::set_hook> hook;
  typedef set<animal, hook> animal_set;
  animal_set animals;

  animals.insert(a1);
  animals.insert(a2);
  animals.insert(a3);

  for (const animal &a : animals)
```

```
    std::cout << a.name << '\n';
}
```

There are two ways to add a hook to a class: either derive the class from a hook or define the hook as a member variable. While the previous examples derived a class from `boost::intrusive::list_base_hook`, Example 18.5 uses the class `boost::intrusive::set_member_hook` to define a member variable.

Please note that the name of the member variable doesn't matter. However, the hook class you use depends on the intrusive container. For example, to define a hook as a member variable for an intrusive list, use `boost::intrusive::list_member_hook` instead of `boost::intrusive::set_member_hook`.

Intrusive containers have different hooks because they have different requirements for elements. However, you can use different several hooks to allow objects to be stored in multiple intrusive containers. `boost::intrusive::any_base_hook` and `boost::intrusive::any_member_hook` let you store objects in any intrusive container. Thanks to these classes, you don't need to derive from multiple hooks or define multiple member variables as hooks.

Intrusive containers expect hooks to be defined in base classes by default. If a member variable is used as a hook, as in Example 18.5, the intrusive container has to be told which member variable to use. That's why both `animal` and the type `hook` are passed to `boost::intrusive::set`. `hook` is defined with `boost::intrusive::member_hook`, which is used whenever a member variable serves as a hook. `boost::intrusive::member_hook` expects the element type, the type of the hook, and a pointer to the member variable as template parameters.

Example 18.5 writes `shark`, `cat`, and `spider`, in that order, to standard output.

In addition to the classes `boost::intrusive::list` and `boost::intrusive::set` introduced in this chapter, Boost.Intrusive also provides, for example, `boost::intrusive::slist` for singly linked lists and `boost::intrusive::unordered_set` for hash containers.