Chapter 50. Boost.Fusion

The standard library provides numerous containers that have one thing in common: They are homogeneous. That is, containers from the standard library can only store elements of one type. A vector of the type `std::vector<int>` can only store `int` values, and a vector of type `std::vector<std::string>` can only store strings.

Boost.Fusion makes it possible to create heterogeneous containers. For example, you can create a vector whose first element is an `int` and whose second element is a string. In addition, Boost.Fusion provides algorithms to process heterogeneous containers. You can think of Boost.Fusion as the standard library for heterogeneous containers.

Strictly speaking, since C++11, the standard library has provided a heterogeneous container, `std::tuple`. You can use different types for the values stored in a tuple. `boost:fusion::tuple` in Boost.Fusion is a similar type. While the standard library doesn't have much more to offer, tuples are just the starting place for Boost.Fusion.

Example 50.1. Processing Fusion tuples

```
#include <boost/fusion/tuple.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  std::cout << get<0>(t) << '\n';
  std::cout << get<1>(t) << '\n';
  std::cout << std::boolalpha << get<2>(t) << '\n';
  std::cout << get<3>(t) << '\n';
}
```

Example 50.1 defines a tuple consisting of an `int`, a `std::string`, a `bool`, and a `double`. The tuple is based on `boost:fusion::tuple`. In Example 50.1, the tuple is then instantiated, initialized, and the various elements are retrieved with `boost::fusion::get()` and written to standard output. The function `boost::fusion::get()` is similar to `std::get()`, which accesses elements in `std::tuple`.

Fusion tuples don't differ from tuples from the standard library. Thus it's no surprise that Boost.Fusion provides a function `boost::fusion::make_tuple()`, which works like `std::make_tuple()`. However, Boost.Fusion provides additional functions that go beyond what is offered in the standard library.

Example 50.2. Iterating over a tuple with `boost::fusion::for_each()`

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/algorithm.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;
```

```
struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  for_each(t, print{});
}
```

Example 50.2 introduces the algorithm `boost::fusion::for_each()`, which iterates over a Fusion container. The function is used here to write the values in the tuple **t** to standard output.

`boost::fusion::for_each()` is designed to work like `std::for_each()`. While `std::for_each()` only iterates over homogeneous containers, `boost::fusion::for_each()` works with heterogeneous containers. You pass a container, not an iterator, to `boost::fusion::for_each()`. If you don't want to iterate over all elements in a container, you can use a *view*.

Example 50.3. Filtering a Fusion container with `boost::fusion::filter_view`

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/view.hpp>
#include <boost/fusion/algorithm.hpp>
#include <boost/type_traits.hpp>
#include <boost/mpl/arg.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  filter_view<tuple_type, boost::is_integral<boost::mpl::arg<1>>> v{t};
  for_each(v, print{});
}
```

Boost.Fusion provides views, which act like containers but don't store data. With views, data in a container can be accessed differently. Views are similar to adaptors from Boost.Range. However, while adaptors from Boost.Range can be applied to only one container, views from Boost.Fusion can span data from multiple containers.

Example 50.3 uses the class `boost::fusion::filter_view` to filter the tuple **t**. The filter directs `boost::fusion::for_each()` to only write elements based on an integral type.

`boost::fusion::filter_view` expects as a first template parameter the type of the container to filter. The second template parameter must be a predicate to filter elements. The predicate must filter the elements based on their type.

The library is called Boost.Fusion because it combines two worlds: C++ programs process values at run time and types at compile time. For developers, values at run time are usually more important. Most tools from the standard library process values at run time. To process types at compile time, template meta programming is used. While values are processed depending on other values at run time, types are processed depending on other types at compile time. Boost.Fusion lets you process values depending on types.

The second template parameter passed to `boost::fusion::filter_view` is a predicate, which will be applied to every type in the tuple. The predicate expects a type as a parameter and returns `true` if the type should be part of the view. If `false` is returned, the type is filtered out.

Example 50.3 uses the class `boost::is_integral` from Boost.TypeTraits. `boost::is_integral` is a template that checks whether a type is integral. Because the template parameter has to be passed to `boost::fusion::filter_view`, a placeholder from Boost.MPL, `boost::mpl::arg<1>`, is used to create a lambda function. `boost::mpl::arg<1>` is similar to **`boost::phoenix::place_holders::arg1`** from Boost.Phoenix. In Example 50.3, the view **v** will contain only the `int` and `bool` elements from the tuple, and therefore, the example will write `10` and `true` to standard output.

Example 50.4. Accessing elements in Fusion containers with iterators

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/iterator.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  auto it = begin(t);
  std::cout << *it << '\n';
  auto it2 = advance<boost::mpl::int_<2>>(it);
  std::cout << std::boolalpha << *it2 << '\n';
}
```

After seeing `boost::fusion::tuple` and `boost::fusion::for_each()`, it shouldn't come as a surprise to find iterators in Example 50.4. Boost.Fusion provides several free-standing functions, such as `boost::fusion::begin()` and `boost::fusion::advance()`, that work like the identically named functions from the standard library.

The number of steps the iterator is to be incremented is passed to `boost::fusion::advance()` as a template parameter. The example again uses `boost::mpl::int_` from Boost.MPL.

`boost::fusion::advance()` returns an iterator of a different type from the one that was passed to the function. That's why Example 50.4 uses a second iterator **it2**. You can't assign the return value from `boost::fusion::advance()` to the first iterator **it**. Example 50.4 writes `10` and `true` to standard output.

In addition to the functions introduced in the example, Boost.Fusion provides other functions that work with iterators. These include the following: `boost::fusion::end()`, `boost::fusion::distance()`, `boost::fusion::next()` and `boost::fusion::prior()`.

Example 50.5. A heterogeneous vector with `boost::fusion::vector`

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef vector<int, std::string, bool, double> vector_type;
  vector_type v{10, "Boost", true, 3.14};
  std::cout << at<boost::mpl::int_<0>>(v) << '\n';

  auto v2 = push_back(v, 'X');
  std::cout << size(v) << '\n';
  std::cout << size(v2) << '\n';
  std::cout << back(v2) << '\n';
}
```

So far we've only seen one heterogeneous container, `boost::fusion::tuple`. Example 50.5 introduces another container, `boost::fusion::vector`.

`boost::fusion::vector` is a vector: elements are accessed with indexes. Access is not implemented using the operator `operator[]`. Instead, it's implemented using `boost::fusion::at()`, a free-standing function. The index is passed as a template parameter wrapped with `boost::mpl::int_`.

This example adds a new element of type `char` to the vector. This is done with the free-standing function `boost::fusion::push_back()`. Two parameters are passed to `boost::fusion::push_back()`: the vector to add the element to and the value to add.

`boost::fusion::push_back()` returns a new vector. The vector **v** isn't changed. The new vector is a copy of the original vector with the added element.

This example gets the number of elements in the vectors **v** and **v2** with `boost::fusion::size()` and writes both values to standard output. The program displays `4` and `5`. It then calls `boost::fusion::back()` to get and write the last element in **v2** to standard output, in this case the value is `X`.

If you look more closely at Example 50.5, you will notice that there is no difference between `boost::fusion::tuple` and `boost::fusion::vector`; they are the same. Thus, Example 50.5 will also work with `boost::fusion::tuple`.

Boost.Fusion provides additional heterogeneous containers, including: `boost::fusion::deque`, `boost::fusion::list` and `boost::fusion::set`. Example 50.6 introduces `boost::fusion::map`, a container for key/value pairs.

Example 50.6. A heterogeneous map with `boost::fusion::map`

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/fusion/algorithm.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  auto m = make_map<int, std::string, bool, double>("Boost", 10, 3.14, true);
  if (has_key<std::string>(m))
    std::cout << at_key<std::string>(m) << '\n';
  auto m2 = erase_key<std::string>(m);
  auto m3 = push_back(m2, make_pair<float>('X'));
  std::cout << std::boolalpha << has_key<std::string>(m3) << '\n';
}
```

Example 50.6 creates a heterogeneous map with `boost::fusion::map()`. The map's type is `boost::fusion::map`, which isn't written out in the example thanks to the keyword `auto`.

A map of type `boost::fusion::map` stores key/value pairs like `std::map` does. However, the keys in the Fusion map are types. A key/value pair consists of a type and a value mapped to that type. The value may be a different type than the key. In Example 50.6, the string "Boost" is mapped to the key `int`.

After the map has been created, `boost::fusion::has_key()` is called to check whether a key `std::string` exists. Then, `boost::fusion::at_key()` is called to get the value mapped to that key. Because the number 10 is mapped to `std::string`, it is written to standard output.

The key/value pair is then erased with `boost::fusion::erase_key()`. This doesn't change the map **m**. `boost::fusion::erase_key()` returns a new map which is missing the erased key/value pair.

The call to `boost::fusion::push_back()` adds a new key/value pair to the map. The key is `float` and the value is "X". `boost::fusion::make_pair()` is called to create the new key/value pair. This function is similar to `std::make_pair()`.

Finally, `boost::fusion::has_key()` is called again to check whether the map has a key `std::string`. Because it was erased, `false` is returned.

Please note that you don't need to call `boost::fusion::has_key()` to check whether a key exists before you call `boost::fusion::at_key()`. If a key is passed to `boost::fusion::at_key()` that doesn't exist in the map, you get a compiler error.

Example 50.7. Fusion adaptors for structures

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
```

```
#include <boost/mpl/int.hpp>
#include <iostream>

struct strct
{
  int i;
  double d;
};

BOOST_FUSION_ADAPT_STRUCT(strct,
  (int, i)
  (double, d)
)

using namespace boost::fusion;

int main()
{
  strct s = {10, 3.14};
  std::cout << at<boost::mpl::int_<0>>(s) << '\n';
  std::cout << back(s) << '\n';
}
```

Boost.Fusion provides several macros that let you use structures as Fusion containers. This is possible because structures can act as heterogeneous containers. Example 50.7 defines a structure which can be used as a Fusion container thanks to the macro `BOOST_FUSION_ADAPT_STRUCT`. This makes it possible to use the structure with functions like `boost::fusion::at()` or `boost::fusion::back()`.

Example 50.8. Fusion support for `std::pair`

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <utility>
#include <iostream>

using namespace boost::fusion;

int main()
{
  auto p = std::make_pair(10, 3.14);
  std::cout << at<boost::mpl::int_<0>>(p) << '\n';
  std::cout << back(p) << '\n';
}
```

Boost.Fusion supports structures such as `std::pair` and `boost::tuple` without having to use macros. You just need to include the header file `boost/fusion/adapted.hpp` (see Example 50.8).

Exercise

Make `debug()` write the member variables of the structures used in the program to standard output:

```
#include <boost/math/constants/constants.hpp>
#include <iostream>

struct animal
{
```

```
    std::string name;
    int legs;
    bool has_tail;
};

struct important_numbers
{
    const float pi = boost::math::constants::pi<float>();
    const double e = boost::math::constants::e<double>();
};

template <class T>
void debug(const T &t)
{
    // TODO: Write member variables of t to standard output.
}

int main()
{
    animal a{ "cat", 4, true };
    debug(a);

    important_numbers in;
    debug(in);
}
```
