Chapter 29. Boost.Algorithm

Boost.Algorithm provides algorithms that complement the algorithms from the standard library. Unlike Boost.Range, Boost.Algorithm doesn't introduce new concepts. The algorithms defined by Boost.Algorithm resemble the algorithms from the standard library.

Please note that there are numerous algorithms provided by other Boost libraries. For example, you will find algorithms to process strings in Boost.StringAlgorithms. The algorithms provided by Boost.Algorithm are not bound to particular classes, such as `std::string`. Like the algorithms from the standard library, they can be used with any container.

Example 29.1. Testing for exactly one value with `boost::algorithm::one_of_equal()`

```
#include <boost/algorithm/cxx11/one_of.hpp>
#include <array>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::array<int, 6> a{{0, 5, 2, 1, 4, 3}};
  auto predicate = [](int i){ return i == 4; };
  std::cout.setf(std::ios::boolalpha);
  std::cout << one_of(a.begin(), a.end(), predicate) << '\n';
  std::cout << one_of_equal(a.begin(), a.end(), 4) << '\n';
}
```

`boost::algorithm::one_of()` tests whether a condition is met exactly once. The condition to test is passed as a predicate. In Example 29.1 the call to `boost::algorithm::one_of()` returns `true` since the number 4 is stored exactly once in **a**.

To test elements in a container for equality, call `boost::algorithm::one_of_equal()`. You don't pass a predicate. Instead, you pass a value to compare to `boost::algorithm::one_of_equal()`. In Example 29.1 the call to `boost::algorithm::one_of_equal()` also returns `true`.

`boost::algorithm::one_of()` complements the algorithms `std::all_of()`, `std::any_of()`, and `std::none_of()`, which were added to the standard library with C++11. However, Boost.Algorithm provides the functions `boost::algorithm::all_of()`, `boost::algorithm::any_of()`, and `boost::algorithm::none_of()` for developers whose development environment doesn't support C++11. You will find these algorithms in the header files `boost/algorithm/cxx11/all_of.hpp`, `boost/algorithm/cxx11/any_of.hpp`, and `boost/algorithm/cxx11/none_of.hpp`.

Boost.Algorithm also defines the following functions: `boost::algorithm::all_of_equal()`, `boost::algorithm::any_of_equal()`, and `boost::algorithm::none_of_equal()`.

Boost.Algorithm provides more algorithms from the C++11 standard library. For example, you have access to `boost::algorithm::is_partitioned()`, `boost::algorithm::is_permutation()`, `boost::algorithm::copy_n()`,

`boost::algorithm::find_if_not()` and `boost::algorithm::iota()`. These functions work like the identically named functions from the C++11 standard library and are provided for developers who don't use C++11. However, Boost.Algorithm provides a few function variants that could be useful for C++11 developers, too.

Example 29.2. More variants of C++11 algorithms

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/algorithm/cxx11/is_sorted.hpp>
#include <boost/algorithm/cxx11/copy_if.hpp>
#include <vector>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v;
  iota_n(std::back_inserter(v), 10, 5);
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_increasing(v) << '\n';
  std::ostream_iterator<int> out{std::cout, ","};
  copy_until(v, out, [](int i){ return i > 12; });
}
```

Boost.Algorithm provides the C++11 algorithm `boost::algorithm::iota()` in the header file `boost/algorithm/cxx11/iota.hpp`. This function generates sequentially increasing numbers. It expects two iterators for the beginning and end of a container. The elements in the container are then overwritten with sequentially increasing numbers.

Instead of `boost::algorithm::iota()`, Example 29.2 uses `boost::algorithm::iota_n()`. This function expects one iterator to write the numbers to. The number of numbers to generate is passed as a third parameter to `boost::algorithm::iota_n()`.

`boost::algorithm::is_increasing()` and `boost::algorithm::is_sorted()` are defined in the header file `boost/algorithm/cxx11/is_sorted.hpp`. `boost::algorithm::is_increasing()` has the same function as `boost::algorithm::is_sorted()`, but the function name expresses more clearly that the function checks that values are in increasing order. The header file also defines the related function `boost::algorithm::is_decreasing()`.

In Example 29.2, **v** is passed directly to `boost::algorithm::is_increasing()`. All functions provided by Boost.Algorithm have a variant that operates based on ranges. Containers can be passed directly to these functions.

`boost::algorithm::copy_until()` is defined in `boost/algorithm/cxx11/copy_if.hpp`. This is another variant of `std::copy()`. Boost.Algorithm also provides `boost::algorithm::copy_while()`.

Example 29.2 displays `true` as a result from `boost::algorithm::is_increasing()`, and `boost::algorithm::copy_until()` writes the numbers `10`, `11`, and `12` to standard output.

Example 29.3. C++14 algorithms from Boost.Algorithm

```
#include <boost/algorithm/cxx14/equal.hpp>
#include <boost/algorithm/cxx14/mismatch.hpp>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v{1, 2};
  std::vector<int> w{1, 2, 3};
  std::cout.setf(std::ios::boolalpha);
  std::cout << equal(v.begin(), v.end(), w.begin(), w.end()) << '\n';
  auto pair = mismatch(v.begin(), v.end(), w.begin(), w.end());
  if (pair.first != v.end())
    std::cout << *pair.first << '\n';
  if (pair.second != w.end())
    std::cout << *pair.second << '\n';
}
```

Besides the algorithms from the C++11 standard library, Boost.Algorithm also defines algorithms that will very likely be added to the standard library with C++14. Example 29.3 uses new variants of two of these functions, `boost::algorithm::equal()` and `boost::algorithm::mismatch()`. In contrast to the identically named functions that have been part of the standard library since C++98, four iterators, rather than three, are passed to these new functions. The algorithms in Example 29.3 don't expect the second sequence to contain as many elements as the first sequence.

While `boost::algorithm::equal()` returns a `bool`, `boost::algorithm::mismatch()` returns two iterators in a `std::pair`. **`first`** and **`second`** refer to the elements in the first and second sequence that are the first ones mismatching. These iterators may also refer to the end of a sequence.

Example 29.3 writes `false` and `3` to standard output. `false` is the return value of `boost::algorithm::equal()`, `3` the third element in **`w`**. Because the first two elements in **`v`** and **`w`** are equal, `boost::algorithm::mismatch()` returns, in **`first`**, an iterator to the end of **`v`** and, in **`second`**, an iterator to the third element of **`w`**. Because **`first`** refers to the end of **`v`**, the iterator isn't de-referenced, and there is no output.

Example 29.4. Using `boost::algorithm::hex()` and `boost::algorithm::unhex()`

```
#include <boost/algorithm/hex.hpp>
#include <vector>
#include <string>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<char> v{'C', '+', '+'};
  hex(v, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string s = "C++";
  std::cout << hex(s) << '\n';
```

```
  std::vector<char> w{'4', '3', '2', 'b', '2', 'b'};
  unhex(w, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string t = "432b2b";
  std::cout << unhex(t) << '\n';
}
```

Example 29.4 uses the two functions `boost::algorithm::hex()` and `boost::algorithm::unhex()`. These functions are designed after the identically named functions from the database system MySQL. They convert characters to hexadecimal values or hexadecimal values to characters.

Example 29.4 passes the vector **v** with the characters “C”, “+”, and “+” to `boost::algorithm::hex()`. This function expects an iterator as the second parameter to write the hexadecimal values to. The example writes `43` for “C” and `2B` (twice) for the two instances of “+” to standard output. The second call to `boost::algorithm::hex()` does the same thing except that “C++” is passed as a string and “432B2B” is returned as a string.

`boost::algorithm::unhex()` is the opposite of `boost::algorithm::hex()`. If the array **w** from Example 29.4 is passed with six hexadecimal values, each of the three pairs of values is interpreted as ASCII-Code. The same happens with the second call to `boost::algorithm::unhex()` when six hexadecimal values are passed as a string. In both cases `C++` is written to standard output.

Boost.Algorithm provides even more algorithms. For example, there are several string matching algorithms that search text efficiently. The documentation contains an overview of all available algorithms.

Exercise

Use a function from Boost.Algorithm to assign the numbers 51 to 56 in ascending order to an array with six elements. Interpret the numbers in the array as hexadecimal values, convert them to characters and write the result to standard output.