Chapter 8. Boost.Regex

Boost.Regex allows you to use *regular expressions* in C++. As the library is part of the standard library since C++11, you don’t depend on Boost.Regex if your development environment supports C++11. You can use identically named classes and functions in the namespace `std` if you include the header file `regex`.

The two most important classes in Boost.Regex are `boost::regex` and `boost::smatch`, both defined in `boost/regex.hpp`. The former defines a regular expression, and the latter saves the search results.

Boost.Regex provides three different functions to search for regular expressions.

Example 8.1. Comparing strings with `boost::regex_match()`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"\\w+\\s\\w+"};
  std::cout << std::boolalpha << boost::regex_match(s, expr) << '\n';
}
```

`boost::regex_match()` (see Example 8.1) compares a string with a regular expression. It will return `true` only if the expression matches the complete string.

`boost::regex_search()` searches a string for a regular expression.

Example 8.2. Searching strings with `boost::regex_search()`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  boost::smatch what;
  if (boost::regex_search(s, what, expr))
  {
    std::cout << what[0] << '\n';
    std::cout << what[1] << "_" << what[2] << '\n';
  }
}
```

`boost::regex_search()` expects a reference to an object of type `boost::smatch` as an additional parameter, which is used to store the results. `boost::regex_search()` only searches for groups. That’s why Example 8.2 returns two strings based on the two groups found in the regular expression.

The result storage class `boost::smatch` is a container holding elements of type `boost::sub_match`, which can be accessed through an interface similar to the one of `std::vector`. For example, elements can be accessed via `operator[]`.

The class `boost::sub_match` stores iterators to the specific positions in a string corresponding to the groups of a regular expression. Because `boost::sub_match` is derived from `std::pair`, the iterators that reference a particular substring can be accessed with **`first`** and **`second`**. However, to write a substring to the standard output stream, you don't have to access these iterators (see Example 8.2). Using the overloaded operator `operator<<`, the substring can be written directly to standard output.

Please note that because iterators are used to point to matched strings, `boost::sub_match` does not copy them. This implies that results are accessible only as long as the corresponding string, which is referenced by the iterators, exists.

Furthermore, please note that the first element of the container `boost::smatch` stores iterators referencing the string that matches the entire regular expression. The first substring that matches the first group is accessible at index 1.

The third function offered by Boost.Regex is `boost::regex_replace()` (see Example 8.3).

Example 8.3. Replacing characters in strings with `boost::regex_replace()`

```cpp
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = " Boost Libraries ";
  boost::regex expr{"\\s"};
  std::string fmt{"_"};
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

In addition to the search string and the regular expression, `boost::regex_replace()` needs a format that defines how substrings that match individual groups of the regular expression should be replaced. In case the regular expression does not contain any groups, the corresponding substrings are replaced one to one using the given format. Thus, Example 8.3 will output `_Boost_Libraries_`.

`boost::regex_replace()` always searches through the entire string for the regular expression. Thus, the program actually replaces all three spaces with underscores.

Example 8.4. Format with references to groups in regular expressions

```cpp
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  std::string fmt{"\\2 \\1"};
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

The format can access substrings returned by groups of the regular expression. Example 8.4 uses this technique to swap the first and last word, displaying `Libraries Boost` as a result.

There are different standards for regular expressions and formats. Each of the three functions takes an additional parameter that allows you to select a specific standard. You can also specify whether or not special characters should be interpreted in a specific format or whether the format should replace the complete string that matches the regular expression.

Example 8.5. Flags for formats

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  std::string fmt{"\\2 \\1"};
  std::cout << boost::regex_replace(s, expr, fmt,
    boost::regex_constants::format_literal) << '\n';
}
```

Example 8.5 passes the flag `boost::regex_constants::format_literal` as the fourth parameter to `boost::regex_replace()` to suppress handling of special characters in the format. Because the complete string that matches the regular expression is replaced with the format, the output of Example 8.5 is `\2 \1`.

Example 8.6. Iterating over strings with `boost::regex_token_iterator`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"\\w+"};
  boost::regex_token_iterator<std::string::iterator> it{s.begin(), s.end(),
    expr};
  boost::regex_token_iterator<std::string::iterator> end;
  while (it != end)
    std::cout << *it++ << '\n';
}
```

With `boost::regex_token_iterator`, Boost.Regex provides a class to iterate over a string with a regular expression. In Example 8.6 the iteration returns the two words in `s`. `it` is initialized with iterators to `s` and the regular expression “\w+”. The default constructor creates an end iterator.

Example 8.6 displays `Boost` and `Libraries`.

Example 8.7. Accessing groups with `boost::regex_token_iterator`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w)\\w+"};
  boost::regex_token_iterator<std::string::iterator> it{s.begin(), s.end(),
    expr, 1};
  boost::regex_token_iterator<std::string::iterator> end;
  while (it != end)
    std::cout << *it++ << '\n';
}
```

You can pass a number as an additional parameter to the constructor of `boost::regex_token_iterator`. If 1 is passed, as in Example 8.7, the iterator returns the first group in the regular expression. Because the regular expression “(\w)\w+” is used, Example 8.7 writes the initials B and L to standard output.

If -1 is passed to `boost::regex_token_iterator`, the regular expression is the delimiter. An iterator initialized with -1 returns substrings that do not match the regular expression.

Example 8.8. Linking a locale to a regular expression

```
#include <boost/regex.hpp>
#include <locale>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost k\xfct\xfcphaneleri";
  boost::basic_regex<char, boost::cpp_regex_traits<char>> expr;
  expr.imbue(std::locale{"Turkish"});
  expr = "\\w+\\s\\w+";
  std::cout << std::boolalpha << boost::regex_match(s, expr) << '\n';
}
```

Example 8.8 links a locale with `imbue()` to **expr**. This is done to apply the regular expression to the string “Boost kütüphaneleri,” which is the Turkish translation of “Boost Libraries.” If umlauts should be parsed as valid letters, the locale must be set – otherwise `boost::regex_match()` returns `false`.

To use a locale of type `std::locale`, **expr** must be based on a class instantiated with the type `boost::cpp_regex_traits`. That’s why Example 8.8 doesn’t use `boost::regex` but instead uses `boost::basic_regex<char, boost::cpp_regex_traits<char>>`. With the second template parameter of `boost::basic_regex`, the parameter for `imbue()` can be defined indirectly. Only with `boost::cpp_regex_traits` can a locale of type `std::locale` be passed to `imbue()`.

Note

If you want to run the example on a POSIX operating system, replace “Turkish” with “tr_TR”. Also make sure the locale for Turkish is installed.

Note that `boost::regex` is defined with a platform-dependent second template parameter. On Windows this parameter is `boost::w32_regex_traits`, which allows an LCID to be passed to `imbue()`. An LCID is a number that, on Windows, identifies a certain language and culture. If you want to write platform-independent code, you must use `boost::cpp_regex_traits` explicitly, as in Example 8.8. Alternatively, you can define the macro `BOOST_REGEX_USE_CPP_LOCALE`.