Chapter 9. Boost.Xpressive

Like Boost.Regex, Boost.Xpressive provides functions to search strings using *regular expressions*. However, Boost.Xpressive makes it possible to write down regular expressions as C++ code rather than strings. That makes it possible to check at compile time whether a regular expression is valid or not.

Only Boost.Regex was incorporated into C++11. The standard library doesn't provide any support for writing regular expressions as C++ code.

`boost/xpressive/xpressive.hpp` provides access to most library functions in Boost.Xpressive. For some functions, additional header files must be included. All definitions of the library can be found in the namespace `boost::xpressive`.

Example 9.1. Comparing strings with `boost::xpressive::regex_match`

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

using namespace boost::xpressive;

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

Boost.Xpressive basically provides the same functions as Boost.Regex, except they are defined in the namespace of Boost.Xpressive. `boost::xpressive::regex_match()` compares strings, `boost::xpressive::regex_search()` searches in strings, and `boost::xpressive::regex_replace()` replaces characters in strings. You can see this in Example 9.1, which uses the function `boost::xpressive::regex_match()`, and which looks similar to Example 8.1.

However, there is a fundamental difference between Boost.Xpressive and Boost.Regex. The type of the regular expression in Boost.Xpressive depends on the type of the string being searched. Because **s** is based on `std::string` in Example 9.1, the type of the regular expression must be `boost::xpressive::sregex`. Compare this with Example 9.2, where the regular expression is applied to a string of type `const char*`.

Example 9.2. `boost::xpressive::cregex` with strings of type `const char*`

```
#include <boost/xpressive/xpressive.hpp>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  const char *c = "Boost Libraries";
  cregex expr = cregex::compile("\\w+\\s\\w+");
  std::cout << std::boolalpha << regex_match(c, expr) << '\n';
}
```

For strings of type `const char*`, use the class `boost::xpressive::cregex`. If you use other string types, such as `std::wstring` or `const wchar_t*`, use `boost::xpressive::wsregex` or `boost::xpressive::wcregex`.

You must call the static member function `compile()` for regular expressions written as strings. The member function must be called on the type used for the regular expression.

Boost.Xpressive supports direct initialization of regular expressions that are written as C++ code. The regular expression has to be expressed in the notation supported by Boost.Xpressive (see Example 9.3).

Example 9.3. A regular expression with C++ code

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

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  sregex expr = +_w >> _s >> +_w;
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

The regular expression from Example 9.2, which was written as the string “\w+\s\w+”, is now expressed in Example 9.3 as `+_w >> _s >> +_w`. It is exactly the same regular expression. Both examples search for at least one alphanumeric character followed by one space followed by at least one alphanumeric character.

Boost.Xpressive makes it possible to write regular expressions with C++ code. The library provides objects for character groups. For example, the object **`_w`** is similar to “\w”. **`_s`** has the same meaning as “\s”.

While “\w” and “\s” can be written one after another in a string, objects like **`_w`** and **`_s`** must be concatenated with an operator. Otherwise, the result wouldn’t be valid C++ code. Boost.Xpressive provides the operator `operator>>`, which is used in Example 9.3.

To express that at least one alphanumeric character should be found, **`_w`** is prefixed with a plus sign. While the syntax of regular expressions expects that quantifiers are put behind character groups – like with “\w+” – the plus sign must be put in front of **`_w`**. The plus sign is an unary operator, which in C++ must be put in front of an object.

Boost.Xpressive emulates the rules of regular expressions as much as they can be emulated in C++. However, there are limits. For example, the question mark is a meta character in regular expressions to express that a preceding item is optional. Since the question mark isn’t a valid operator in C++, Boost.Xpressive replaces it with the exclamation mark. A notation like “\w?” becomes `!_w` with Boost.Xpressive because the exclamation mark must be prefixed.

Boost.Xpressive supports actions that can be linked to expressions – something Boost.Regex doesn't support.

Example 9.4. Linking actions to expressions

```
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
#include <string>
#include <iterator>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  std::ostream_iterator<std::string> it{std::cout, "\n"};
  sregex expr = (+_w)[*boost::xpressive::ref(it) = _] >> _s >> +_w;
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

Example 9.4 returns `true` for `boost::xpressive::regex_match()` and writes `Boost` to standard output.

You can link actions to expressions. An action is executed when the respective expression is found. In Example 9.4, the expression `+_w` is linked to the action `*boost::xpressive::ref(it) = _`. The action is a lambda function. The object _ refers to characters found by the expression – in this case the first word in **`s`**. The respective characters are assigned to the iterator **`it`**. Because **`it`** is an iterator of type `std::ostream_iterator`, which has been initialized with **`std::cout`**, `Boost` is written to standard output.

Please note that you must use the function `boost::xpressive::ref()` to wrap the iterator **`it`**. Only then it is possible to assign _ to the iterator. _ is an object provided by Boost.Xpressive in the namespace `boost::xpressive`, which normally couldn't be assigned to an iterator of type `std::ostream_iterator`. Because the assignment happens only when the string "Boost" has been found with `+_w`, `boost::xpressive::ref()` turns the assignment into a *lazy* operation. Although the code in square brackets attached to `+_w` is, according to C++ rules, immediately executed, the assignment to the iterator **`it`** can only occur when the regular expression is used. Thus, `*boost::xpressive::ref(it) = _` isn't executed immediately.

Example 9.4 includes the header file `boost/xpressive/regex_actions.hpp`. This is required because actions aren't available through `boost/xpressive/xpressive.hpp`.

Like Boost.Regex, Boost.Xpressive supports iterators to split a string with regular expressions. The classes `boost::xpressive::regex_token_iterator` and `boost::xpressive::regex_iterator` do this. It is also possible to link a locale to a regular expression to use a locale other than the global one.