
14.1. The Regex Match and Search Interface

First, let's look at how we can check whether a sequence of characters matches or partially matches a specific regular expression:

Click here to view code image

```
// regex/regex1.cpp

#include <regex>
#include <iostream>
using namespace std;

void out (bool b)
{
    cout << ( b ? "found" : "not found") << endl;
}

int main()
{
    // find XML/HTML-tagged value (using default syntax):
    regex reg1("<.*>.*</.*>");
    bool found = regex_match ("<tag>value</tag>",    // data
                              reg1);                 // regular expression
    out(found);

    // find XML/HTML-tagged value (tags before and after the value must match):
    regex reg2("<(.*)>.*</\\1>");
    found = regex_match ("<tag>value</tag>",         // data
                         reg2);                      // regular expression
    out(found);

    // find XML/HTML-tagged value (using grep syntax):
    regex reg3("<\\(.*\\)>.*</\\1>",regex_constants::grep);
    found = regex_match ("<tag>value</tag>",         // data
                         reg3);                      // regular expression
    out(found);

    // use C-string as regular expression (needs explicit cast to regex):
    found = regex_match ("<tag>value</tag>",         // data
                         regex("<(.*)>.*</\\1>"));   // regular expression
    out(found);
    cout << endl;

    // regex_match() versus regex_search():
    found = regex_match ("XML tag: <tag>value</tag>",
                         regex("<(.*)>.*</\\1>"));          // fails to match
    out(found);
    found = regex_match ("XML tag: <tag>value</tag>",
                         regex(".*<(.*)>.*</\\1>.*"));      // matches
    out(found);
    found = regex_search ("XML tag: <tag>value</tag>",
                          regex("<(.*)>.*</\\1>"));         // matches
    out(found);
    found = regex_search ("XML tag: <tag>value</tag>",
                          regex(".*<(.*)>.*</\\1>.*"));     // matches
    out(found);
}
```

First, we include the necessary header file and global identifiers in namespace `std`:

```
#include <regex>
using namespace std;
```

Next, a first example demonstrates how a regular expression can be defined and used to check whether a character sequence matches a specific pattern. We declare and initialize `reg1` as a regular expression:

```
regex reg1("<.*>.*</.*>");
```

The type of the object representing the regular expression is `std::regex`. As with strings, this is a specialization of class

`std::basic_regex<>` for the character type `char`. For the character type `wchar_t`, class `std::wregex` is provided. `reg1` is initialized by the following regular expression:

```
<.*>.*</.*>
```

This regular expressions checks for "`<` *someChars* `>` *someChars* `</` *someChars* `>`" by using the syntax `.*`, where "`.`" stands for "any character except newline" and "`*`" stands for "zero or more times." Thus, we try to match the format of a tagged XML or HTML value. The character sequence `<tag>value</tag>` matches this pattern, so

Click here to view code image

```
regex_match ("<tag>value</tag>",      // data
             reg1);                   // regular expression
```

yields `true`.

We can even specify that the leading and the trailing tags have to be the same character sequence, which is what the next statements demonstrate:

Click here to view code image

```
regex reg2("<(.*)>.*</\\1>");
found = regex_match ("<tag>value</tag>",      // data
                     reg2);                   // regular expression
```

Again, `regex_match()` yields `true`.

Here, we use the concept of "grouping." We use "`(` ... `)`" to define a so-called *capture group*, to which we refer later on with the regular expression "`\1`". Note, however, that we specify the regular expression as an ordinary character sequence, so we have to specify the "character `\` followed by the character `1`" as "`\\1`". Alternatively, we could use a *raw string*, which was introduced with C++11 (see Section 3.1.6, page 23):

Click here to view code image

```
R"(<(.*)>.*</\1>)"      // equivalent to: "<(.*)>.*</\\1>"
```

Such a raw string allows you to define a character sequence by writing exactly its contents as a raw character sequence. It starts with "`R"(`" and ends with "`)"`". To be able to have "`)"`" inside the raw string, you can use a delimiter. Thus, the complete syntax of raw strings is `R"` *delim* `(` ... `)` *delim* `"`, where *delim* is a character sequence of at most 16 basic characters except the backslash, whitespaces, and parentheses.

What we introduce here as special characters for regular expressions is part of the grammar they have. Note that the C++ standard library supports various grammars. The default grammar is a "modified ECMAScript grammar," which is introduced in detail in Section 14.8, page 738. But the next statements show how a different grammar can be used:

Click here to view code image

```
regex reg3("<\\(.*\\)>.*</\\1>",regex_constants::grep);
found = regex_match ("<tag>value</tag>",      // data
                     reg3);                   // regular expression
```

Here, the optional second argument to the `regex` constructor `regex_constants::grep` specifies a grammar like the UNIX `grep` command, where, for example, you have to mask the grouping characters by additional backslashes (which have to be masked by backslashes in ordinary string literals). Section 14.9, page 739, discusses the differences of the various grammars supported.

All the previous examples used a separate object to specify the regular expression. This is not necessary; however, note that just passing a string or string literal as a regular expression is not enough. Although an implicit type conversion is declared, the resulting statement won't compile, because it is ambiguous. For example:

Click here to view code image

```
regex_match ("<tag>value</tag>",              // ERROR: ambiguous
             "<(.*)>.*</\\1>")
regex_match (string("<tag>value</tag>"),      // ERROR: ambiguous
             "<(.*)>.*</\\1>")
regex_match ("<tag>value</tag>",              // OK
             regex("<(.*)>.*</\\1>"))
```

Finally, we come to the difference of `regex_match()` and `regex_search`:

- `regex_match()` checks whether the *whole* character sequence matches a regular expression.
- `regex_search()` checks whether the character sequence *partially* matches a regular expression.

There is no other difference. Thus,

```
regex_search (data, regex(pattern))
```

is always equivalent to

Click here to view code image

```
regex_match (data, regex("(.|\n)*"+pattern+"(.|\n)*"))
```

where " `(.|\n)*` " stands for any number of any character (" `.` " stands for any character except the newline character and " `|` " stands for "or").

Now, you might say that these statements miss important information, at least for the function `regex_search()` : *where* a regular expression matches a given character sequence. For this and many more features, we have to introduce new versions of `regex_match()` and `regex_search()` , where a new parameter returns all necessary information about a match.