14.2. Dealing with Subexpressions

Consider the following example:

Click here to view code image

```
// regex/regex2.cpp

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

int main()
{
    string data = "XML tag: <tag-name>the value</tag-name>.";
    cout << "data:             " << data << "\n\n";

    smatch m;  // for returned details of the match
    bool found = regex_search (data,
                               m,
                               regex("<(.*)>(.*)</(\\1)>"));

    // print match details:
    cout << "m.empty():        " << boolalpha << m.empty() << endl;
    cout << "m.size():         " << m.size() << endl;
    if (found) {
        cout << "m.str():           " << m.str() << endl;
        cout << "m.length():        " << m.length() << endl;
        cout << "m.position():      " << m.position() << endl;
        cout << "m.prefix().str():  " << m.prefix().str() << endl;
        cout << "m.suffix().str():  " << m.suffix().str() << endl;
        cout << endl;

        // iterating over all matches (using the match index):
        for (int i=0; i<m.size(); ++i) {
            cout << "m[" << i << "].str():       " << m[i].str() <<
endl;
            cout << "m.str(" << i << "):         " << m.str(i) << endl;
            cout << "m.position(" << i << "):    " << m.position(i)
                 << endl;
        }
        cout << endl;

        // iterating over all matches (using iterators):
        cout << "matches:" << endl;
        for (auto pos = m.begin(); pos != m.end(); ++pos) {
            cout << " " << *pos << " ";
            cout << "(length: " << pos->length() << ")" << endl;
        }
    }
}
```

In this example, we can demonstrate the use of `match_results` objects, which can be passed to `regex_match()` and `regex_search()` to get details of matches. Class `std::match_results<>` is a template that has to get instantiated by the iterator type of the characters processed. The C++ standard library provides some predefined instantiations:

- `smatch` : for details of matches in `string` s
- `cmatch` : for details of matches in C-strings (`const char*`)
- `wsmatch` : for details of matches in `wstring` s
- `wcmatch` : for details of matches in wide C-strings (`const wchar_t*`)

Thus, if we call `regex_match()` or `regex_search()` for C++ strings, type `smatch` has to be used; for ordinary string literals, type `cmatch` has to be used.

What a `match_results` object yields is shown in detail by the example, where we search for the regular expression

```
<(.*)>(.*)</(\1)>
```

in the string `data`, initialized by the following character sequence:

```
"XML tag: <tag-name>the value</tag-name>."
```

After the call, the `match_results` object `m` has a state, which is visible in Figure 14.1 and provides the following interface:

- In general, the `match_results` object contains:
  - A `sub_match` object `m[0]` for all the matched characters
  - A `prefix()`, a `sub_match` object that represents all characters before the first matched character
  - A `suffix()`, a `sub_match` object that represents all characters after the last matched character
- In addition, for any capture group, you have access to a corresponding `sub_match` object `m[` *n* `]`. Because the regex specified here defines three capture groups, one for the introducing tag, one for the value, and one for the ending tag, these are available in `m[1]`, `m[2]`, and `m[3]`.
- `size()` yields the number of `sub_match` objects (including `m[0]`).
- All `sub_match` objects are derived from `pair<>` and have the position of the first character as member `first` and the position after the last character as member `second`. In addition, `str()` yields the characters as a string, `length()` yields the number of characters, operator `<<` writes the characters to a stream, and an implicit type conversion to a string is defined.
- In addition, the `match_results` object as a whole provides:
  - member function `str()` to yield the matched string as a whole (calling `str()` or `str(0)`) or the *n*th matched substring (calling `str(` *n* `)`), which is empty if no matched substring exists (thus, passing an *n* greater than `size()` is valid)
  - member function `length()` to yield the length of the matched string as a whole (calling `length()` or `length(0)`) or the length of the *n*th matched substring (calling `length(` *n* `)`), which is `0` if no matched substring exists (thus, passing an *n* greater than `size()` is valid)
  - member function `position()` to yield the position of the matched string as a whole (calling `position()` or `position(0)`) or the position of the *n*th matched substring (calling `length(` *n* `)`)
  - member functions `begin()`, `cbegin()`, `end()`, and `cend()` to iterate over the `sub_match` objects `m[0]` to `m[` *n* `]`

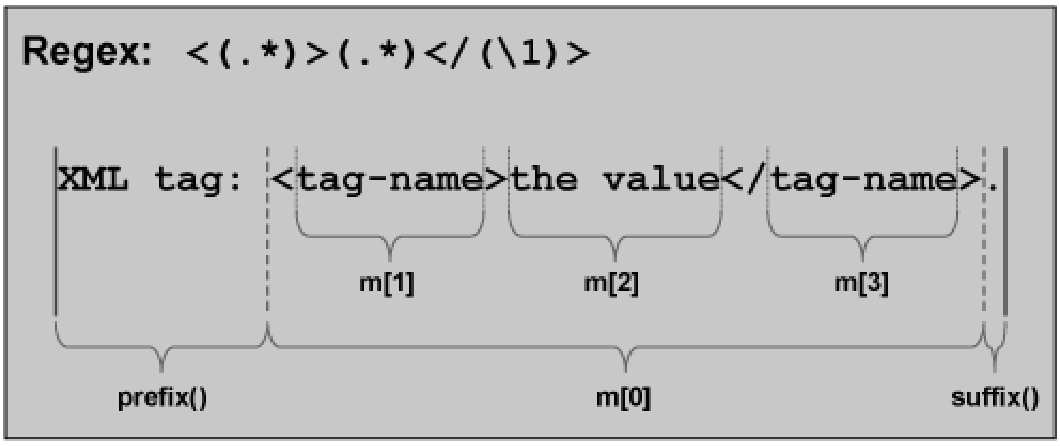

Figure 14.1. Regex Match Interface

For this reason, the program has the following output:

Click here to view code image

```
data:              XML tag: <tag-name>the value</tag-name>.

m.empty():         false
m.size():          4
m.str():           <tag-name>the value</tag-name>
m.length():        30
m.position():      9
m.prefix().str():  XML tag:
m.suffix().str():  .
```

```
m[0].str():       <tag-name>the value</tag-name>
m.str(0):         <tag-name>the value</tag-name>
m.position(0):    9
m[1].str():       tag-name
m.str(1):         tag-name
m.position(1):    10
m[2].str():       the value
m.str(2):         the value
m.position(2):    19
m[3].str():       tag-name
m.str(3):         tag-name
m.position(3):    30

matches:
 <tag-name>the value</tag-name> (length: 30)
 tag-name (length: 8)
 the value (length: 9)
 tag-name (length: 8)
```

In other words, you have four ways to yield the whole matched string in a `match_result<> m` :

Click here to view code image

```
m.str()           // yields whole matches string
m.str(0)          // ditto
m[0].str()        // ditto
*(m.begin())      // ditto
```

and three ways to yield the *n*th matches substring, if any:

Click here to view code image

```
m.str(1)            // yields first matched substring, if any, or "" otherwise
m[1].str()          // ditto
*(m.begin()+1)      // yields first matched substring, if any, invalid otherwise
```

If you call `regex_match()` instead of `regex_search()` , the `match_results` interface is the same. However, because `regex_match()` always matches the whole character sequence, prefix and suffix will always be empty.

Now we have all the information we need to find *all* matches of a regular expression, as the following program demonstrates:

Click here to view code image

```
// regex/regex3.cpp

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

int main()
{
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*)>(.*)</(\\1)>");

    // iterate over all matches
    auto pos=data.cbegin();
    auto end=data.cend();
    smatch m;
    for ( ; regex_search(pos,end,m,reg); pos=m.suffix().first) {
        cout << "match:  " << m.str() << endl;
        cout << " tag:   " << m.str(1) << endl;
        cout << " value: " << m.str(2) << endl;
    }
}
```

Here, we use the regular expression (the backslash has to get escaped in the C++ string literal)

```
<(.*)>(.*)</(\1)>
```

to search for:

<*anyNumberOfAnyChars1***>***anyNumberOfAnyChars2***</***anyNumberOfAnyChars1***>**

Thus, we search for XML tags (`\1` means: *the same as the first matched substring*).

In this example, we use this regular expression by a different interface that iterates over matched character sequences. For this reason, instead of passing the character sequence as a whole, we pass a range of the corresponding elements. We start with the range of all characters, using `cbegin()` and `cend()` of the string we search in:

```
auto pos=data.cbegin();
auto end=data.cend();
```

Then, after each match, we continue the search with the beginning of the remaining characters:

Click here to view code image

```
smatch m;
for ( ; regex_search(pos,end,m,reg); pos=m.suffix().first) {
    ...
}
```

So, because the string `data` we parse has the following value:

Click here to view code image

```
<person>
 <first>Nico</first>
 <last>Josuttis</last>
</person>
```

the program has the following output:

Click here to view code image

```
match:  <first>Nico</first>
 tag:   first
 value: Nico
match:  <last>Josuttis</last>
 tag:   last
 value: Josuttis
```

To reinitialize `pos` , we could also pass `m[0].second()` (the end of the matched characters) instead of the expression `m.suffix().first` . Note that in both cases we have to use `const_iterator` s. Thus, using `begin()` and `end()` to initialize `pos` and `end` would not compile here.

Note also that the output will be different if the tags in `data` were not separated by a newline character:

Click here to view code image

```
<person><first>Nico</first><last>Josuttis</last></person>
```

Then, the output would be:

Click here to view code image

```
match:  <person><first>Nico</first><last>Josuttis</last></person>
 tag:   person
 value: <first>Nico</first><last>Josuttis</last>
```

The reason is that regex functions try to operate in a *greedy* manner. That is, the longest match possible is returned. With newline characters, the tag opened with `<person>` could not match, because we were looking for " `.*` " as value, which means "any character except newline any times." Without newline characters, the whole tag opened with `<person>` now fulfills this pattern. To ensure that we still find the inner tags, we'd have to change the regular expression, for example, as follows:

```
"<(.*)>([^>]*)</(\\1)>"
```

For the value, we now look for " `[^>]*` ", which means "all but character `>` any times." Therefore, subtags do not fit any longer as part of a value.