
14.4. Regex Token Iterators

A regex iterator helps to iterate over matched subsequences. However, sometimes you also want to process all the contents between matched expressions. That is especially the case if you want to split a string into separate tokens, separated by something, which might even be specified as a regular expression. Class `regex_token_iterator<>` having the usual instantiations for strings and character sequences with prefixes `S`, `C`, `WS`, or `WC` provides this functionality.

Again, to initialize it, you can pass the beginning and end of a character sequence and a regular expression. In addition, you can specify a list of integral values, which represent elements of a "tokenization":

- `-1` means that you are interested in all the subsequences between matched regular expressions (token separators).
- `0` means that you are interested in all the matched regular expressions (token separators).
- Any other value *n* means that you are interested in the matched *n*th subexpression inside the regular expressions.

Now consider the following example:

Click here to view code image

```
// regex/regextokeniter1.cpp

#include <string>
#include <regex>
#include <iostream>
#include <algorithm>
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 (using a regex_token_iterator):
    sregex_token_iterator pos(data.cbegin(),data.cend(),   // sequence
                              reg,                         // token separator
                              {0,2});      // 0: full match, 2: second
substring
    sregex_token_iterator end;
    for ( ; pos!=end ; ++pos ) {
        cout << "match:  " << pos->str() << endl;
    }
    cout << endl;

    string names = "nico, jim, helmut, paul, tim, john paul, rita";
    regex sep("[ \t\n]*[,;.][ \t\n]*");  // separated by , ; or . and spaces
    sregex_token_iterator p(names.cbegin(),names.cend(),   // sequence
                            sep,                           // separator
                            -1);         // -1: values between separators
    sregex_token_iterator e;
    for ( ; p!=e ; ++p ) {

        cout << "name:   " << *p << endl;
    }
}
```

The program has the following output:

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

name:   nico
name:   jim
name:   helmut
name:   paul
name:   tim
name:   john paul
```

```
name:   rita
```

Here, a regex token iterator for `string` s (prefix `s`) is initialized by the character sequence *data*, the regular expression `reg`, and a list of two indexes (`0` and `2`):

Click here to view code image

```
sregex_token_iterator pos(data.cbegin(),data.cend(),  // sequence
                          reg,                        // token separator
                          {0,2});       // 0: full match, 2: second substring
```

The list of indexes we are interested in defines that we are interested in all matches and the second substring of each match.

The usual application of such a regex token iterator demonstrates the next iteration. Here, we have a list of names:

Click here to view code image

```
string names = "nico, jim, helmut, paul, tim, john paul, rita";
```

Now a regular expression defines what separates these names. Here, it is a comma or a semicolon or a period with optional whitespaces (spaces, tabs, and newlines) around:

Click here to view code image

```
regex sep("[ \t\n]*[,;.][ \t\n]*"); // separated by ,; or . and spaces
```

Alternatively, we could use the following regular expression (see Section 14.8, page 738):

Click here to view code image

```
regex sep("[[:space:]]*[,;.][[:space:]]*"); // separated by ,; or . and
spaces
```

Because we are interested only in the values between these token separators, the program processes each name in this list (with spaces removed).

Note that the interface of `regex_token_iterator` allows you to specify the tokens of interest in various ways:

- You can pass a single integral value.
- You can pass an initializer list of integral values (see Section 3.1.3, page 15).
- You can pass a `vector` of integral values.
- You can pass an array of integral values.