```cpp
/*
 *
 * Copyright (c) 2003
 * John Maddock
 *
 * Use, modification and distribution are subject to the
 * Boost Software License, Version 1.0. (See accompanying file
 * LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
 *
 */

 /*
  *   LOCATION:    see http://www.boost.org for most recent version.
  *   FILE         regex_token_iterator_example_1.cpp
  *   VERSION      see <boost/version.hpp>
  *   DESCRIPTION: regex_token_iterator example: split a string into tokens
  */


#include <boost/regex.hpp>

#include <iostream>
using namespace std;


#if defined(BOOST_MSVC) || (defined(__BORLANDC__) && (__BORLANDC__==0x550))
//
// problem with std::getline under MSVC6sp3
istream& getline(istream& is, std::string& s)
{
   s.erase();
   char c = static_cast<char>(is.get());
   while(c != '\n')
   {
      s.append(1, c);
      c = static_cast<char>(is.get());
   }
   return is;
}
#endif


// int main(int argc, const char*[])
// updated by JMH on December 5, 2012 at 12:21 PM to allow command line input
// note that if the command line input contains white space, it must be enclosed
//    in double quotes
int main(int argc, char** argv)
{
   string s;
   do{
      if ( argc == 1 )
      {
         cout << "Enter text to split (or \"quit\" to exit): ";
         getline(cin, s);
         if(s == "quit") break;
      }
      else {
        // s = "This is a string of tokens";
        s = argv[1] ;
        // cout << argc << " " << s << endl ;
      }

      boost::regex re("\\s+");

      // the -1 parameter below causes the token iterator to consider as tokens
      //    those parts of the input that do *not* match the regex
      // such an iterator, if dereferenced, returns a match_results corresponding to
```

```
68          //    the sequence of characters between the last match and the end of sequence
69          // see http://boost-sandbox.sourceforge.net/libs/xpressive/doc/html/boost_xpressive/user_s_guide/string_spl
70          // and http://en.cppreference.com/w/cpp/regex/regex_token_iterator
71          boost::sregex_token_iterator i(s.begin(), s.end(), re, -1);
72
73          // the default-constructed sregex_token_iterator is the end-of-sequence iterator
74          // see http://en.cppreference.com/w/cpp/regex/regex_token_iterator
75          boost::sregex_token_iterator j;
76
77          unsigned count = 0;
78          while(i != j)
79          // while(i != s.end())  yields compiler error: no match for operator!=
80          {
81             cout << *i++ << endl;
82             count++;
83          }
84          cout << "There were " << count << " tokens found." << endl;
85
86       }while(argc == 1);
87       return 0;
88    }
```

String Splitting and Tokenization

`regex_token_iterator<>` is the Ginsu knife of the text manipulation world. It slices! It dices! This section describes how to use the highly-configurable `regex_token_iterator<>` to chop up input sequences.

Overview

You initialize a `regex_token_iterator<>` with an input sequence, a regex, and some optional configuration parameters. The `regex_token_iterator<>` will use `regex_search()` to find the first place in the sequence that the regex matches. When dereferenced, the `regex_token_iterator<>` returns a *token* in the form of a `std::basic_string<>`. Which string it returns depends on the configuration parameters. By default it returns a string corresponding to the full match, but it could also return a string corresponding to a particular marked sub-expression, or even the part of the sequence that *didn't* match. When you increment the `regex_token_iterator<>`, it will move to the next token. Which token is next depends on the configuration parameters. It could simply be a different marked sub-expression in the current match, or it could be part or all of the next match. Or it could be the part that *didn't* match.

As you can see, `regex_token_iterator<>` can do a lot. That makes it hard to describe, but some examples should make it clear.

Example 1: Simple Tokenization

This example uses `regex_token_iterator<>` to chop a sequence into a series of tokens consisting of words.

```
std::string input("This is his face");
sregex re = +_w;                      // find a word

// iterate over all the words in the input
sregex_token_iterator begin( input.begin(), input.end(), re ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
This
is
his
face
```

Example 2: Simple Tokenization, Reloaded

This example also uses `regex_token_iterator<>` to chop a sequence into a series of tokens consisting of words, but it uses the regex as a delimiter. When we pass a `-1` as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens those parts of the input that *didn't* match the regex.

```
std::string input("This is his face");
sregex re = +_s;                      // find white space

// iterate over all non-white space in the input. Note the -1 below:
sregex_token_iterator begin( input.begin(), input.end(), re, -1 ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
This
is
his
face
```

Example 3: Simple Tokenization, Revolutions

This example also uses `regex_token_iterator<>` to chop a sequence containing a bunch of dates into a series of tokens consisting of just the years. When we pass a positive integer *N* as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens only the *N*-th marked sub-expression of each match.

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // find a date

// iterate over all the years in the input. Note the 3 below, corresponding to the 3rd sub-expressior
sregex_token_iterator begin( input.begin(), input.end(), re, 3 ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
2003
1999
1981
```

Example 4: Not-So-Simple Tokenization

This example is like the previous one, except that instead of tokenizing just the years, this program turns the days, months and years into tokens. When we pass an array of integers *{I,J,...}* as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens the *I*-th, *J*-th, etc. marked sub-expression of each match.

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // find a date

// iterate over the days, months and years in the input
int const sub_matches[] = { 2, 1, 3 }; // day, month, year
sregex_token_iterator begin( input.begin(), input.end(), re, sub_matches ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
02
01
2003
23
04
1999
13
11
1981
```

The `sub_matches` array instructs the `regex_token_iterator<>` to first take the value of the 2nd sub-match, then the 1st sub-match, and finally the 3rd. Incrementing the iterator again instructs it to use `regex_search()` again to find the next match. At that point, the process repeats -- the token iterator takes the value of the 2nd sub-match, then the 1st, et cetera.

std::regex_token_iterator


```
template<

    class BidirIt,
    class CharT = typename std::iterator_traits<BidirIt>::value_type,     (since C++11)
    class Traits = std::regex_traits<CharT>

> class regex_token_iterator
```

`std::regex_token_iterator` is a read-only `ForwardIterator` that accesses the individual sub-matches of every match of a regular expression within the underlying character sequence. It can also be used to access the parts of the sequence that were not matched by the given regular expression (e.g. as a tokenizer).

On construction, it constructs an `std::regex_iterator` and on every increment it steps through the requested sub-matches from the current match_results, incrementing the underlying regex_iterator when incrementing away from the last submatch.

The default-constructed `std::regex_token_iterator` is the end-of-sequence iterator. When a valid `std::regex_token_iterator` is incremented after reaching the last submatch of the last match, it becomes equal to the end-of-sequence iterator. Dereferencing or incrementing it further invokes undefined behavior.

Just before becoming the end-of-sequence iterator, a `std::regex_token_iterator` may become a *suffix iterator*, if the index `-1` (non-matched fragment) appears in the list of the requested submatch indexes. Such iterator, if dereferenced, returns a match_results corresponding to the sequence of characters between the last match and the end of sequence.

A typical implementation of `std::regex_token_iterator` holds the underlying `std::regex_iterator`, a container (e.g. `std::vector<int>`) of the requested submatch indexes, the internal counter equal to the index of the submatch, a pointer to `std::match_results`, pointing at the current submatch of the current match, and a `std::match_results` object containing the last non-matched character sequence (used in tokenizer mode).

Type requirements

- `BidirIt` must meet the requirements of `BidirectionalIterator`.

Specializations

Several specializations for common character sequence types are defined:

Defined in header `<regex>`

Type	Definition
`cregex_token_iterator`	`regex_token_iterator<const char*>`
`wcregex_token_iterator`	`regex_token_iterator<const wchar_t*>`
`sregex_token_iterator`	`regex_token_iterator<std::string::const_iterator>`
`wsregex_token_iterator`	`regex_token_iterator<std::wstring::const_iterator>`

Member types

Member type	Definition
`value_type`	`std::sub_match<BidirIt>`
`difference_type`	`std::ptrdiff_t`
`pointer`	`const value_type*`
`reference`	`const value_type&`
`iterator_category`	`std::forward_iterator_tag`
`regex_type`	`basic_regex<CharT, Traits>`

Member functions

(constructor)	constructs a new regex_token_iterator (public member function)
(destructor) (implicitly declared)	destructs a regex_token_iterator, including the cached value (public member function)
operator=	replaces a regex_token_iterator (public member function)
operator== **operator!=**	compares two regex_token_iterators (public member function)
operator* **operator->**	accsses current submatch (public member function)
operator++ **operator++**`(int)`	advances the regex_token_iterator to the next submatch (public member function)

Notes

It is the programmer's responsibility to ensure that the `std::basic_regex` object passed to the iterator's constructor outlives the iterator. Because the iterator stores a `std::regex_iterator` which stores a pointer to the regex, incrementing the iterator after the regex was destroyed results in undefined behavior.

Example

```
#include <fstream>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <regex>
int main()
{
   std::string text = "Quick brown fox.";
   // tokenization (non-matched fragments)
   // Note that regex is matched only two times: when the third value is obtained
   // the iterator is a suffix iterator.
   std::regex ws_re("\\s+"); // whitespace
   std::copy( std::sregex_token_iterator(text.begin(), text.end(), ws_re, -1),
              std::sregex_token_iterator(),
              std::ostream_iterator<std::string>(std::cout, "\n"));

   // iterating the first submatches
   std::string html = "<p><a href=\"http://google.com\">google</a> "
                      "< a HREF =\"http://cppreference.com\">cppreference</a>\n</p>";
   std::regex url_re("<\\s*A\\s+[^>]*href\\s*=\\s*\"([^\"]*)\"", std::regex::icase);
   std::copy( std::sregex_token_iterator(html.begin(), html.end(), url_re, 1),
              std::sregex_token_iterator(),
              std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

Output:

```
Quick
brown
fox.
http://google.com
http://cppreference.com
```

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/regex/regex_token_iterator&oldid=41780"