

```

1  This file was updated on Saturday, 2012-11-24 at 9:32 AM
2
3
4  =====
5  RegexTests.cpp
6  =====
7
8
9  /**
10 *   File:   /~heines/91.204/91.204-2012-13f/204-lecs/code/BoostRegexTests/RegExTests.cpp
11 *   Jesse M. Heines, UMass Lowell Computer Science, heines@cs.uml.edu
12 *   Copyright (c) 2012 by Jesse M. Heines. All rights reserved. May be freely
13 *   copied or excerpted for educational purposes with credit to the author.
14 *   updated by JMH on November 19, 2012 at 8:12 PM
15 *   updated by JMH on November 23, 2012 at 9:58 PM
16 */
17
18 #include <iostream>    // for cout and friends
19 #include <sstream>     // for string streams
20 #include <string>      // for the STL string class
21
22 #include <boost/regex.hpp>           // for regex_match
23 #include <boost/algorithm/string.hpp> // for iequal (case-insensitive match)
24 // the Boost string library contains many string manipulation functions not found in
25 // the STL library that you may be familiar with from Java or JavaScript, such as
26 // case-insensitive comparisons and trimming
27 // see http://stackoverflow.com/questions/11635/case-insensitive-string-comparison-in-c
28 // see http://www.boost.org/doc/libs/1_52_0/doc/html/string_algo/usage.html
29 // see http://www.boost.org/doc/libs/1_52_0/doc/html/string_algo/reference.html
30
31
32 using namespace std;    // to eliminate the need for std::
33 using namespace boost;  // to eliminate the need for boost::
34
35
36 // this function is modeled after code found in credit_card_example.cpp
37 void test1_BasicAndCaseInsensitiveMatches() {
38     // set up the strings to be tested
39     string str[ 10 ];
40     str[0] = "quit" ;
41     str[1] = "exit" ;
42     str[2] = "Quit" ;
43     str[3] = "quite" ;
44     str[4] = "unrequited" ;
45     str[5] = "unreQUITed" ;
46     int nStrings = 6 ;
47
48     // define a regular expression to test for "quit"
49     const boost::regex reQuit( "quit" ) ;
50     cout << "Results of case-sensitive searches:" << endl ;
51     // test each string against the regular expression
52     for ( int k = 0 ; k < nStrings ; k++ ) {
53         cout << "attempting to match \"" << str[k] << "\" to \"" << reQuit << "\" returned "
54              << boost::regex_match( str[k], reQuit ) << endl ;
55     }
56
57     // define a regular expression to test for "quit" anywhere in a string
58     const boost::regex reQuitA( ".*quit.*" ) ;
59     cout << "\nResults of anywhere searches:" << endl ;
60     // test each string against the regular expression
61     for ( int k = 0 ; k < nStrings ; k++ ) {
62         cout << "attempting to match \"" << str[k] << "\" to \"" << reQuitA << "\" returned "
63              << boost::regex_match( str[k], reQuitA ) << endl ;
64     }
65
66     // define a case-insensitive regular expression to test for "quit" anywhere in a string
67     const boost::regex reQuitI( ".*quit.*", boost::regex::icase ) ;
68     cout << "\nResults of case-insensitive anywhere searches:" << endl ;

```

```

69 // test each string against the regular expression
70 for ( int k = 0 ; k < nStrings ; k++ ) {
71     cout << "attempting to match \"" << str[k] << "\" to \"" << reQuitI << "\" returned "
72         << boost::regex_match( str[k], reQuitI ) << endl ;
73 }
74 }
75
76
77 // this function is modeled after code found in regex_match_example.cpp
78 // note that this code demonstrates just one way to address the issue of parsing
79 // a command line using regular expressions, other approaches are not only
80 // possible, but perhaps even better
81 void test2_BasicCommandParsing_v1() {
82
83     string strCmd[10] ;
84     strCmd[0] = " add element root first one" ;
85     strCmd[1] = " add element root second" ;
86     strCmd[2] = " add attribute first attr1 attr1value" ;
87     strCmd[3] = " add attribute second attr2" ;
88     strCmd[4] = "print" ;
89     strCmd[5] = "quit" ;
90     strCmd[6] = "another command" ;
91     int nCmds = 7 ;
92
93     cmatch what;
94     // what[0] contains the entire matched string
95     // what[1] contains the first matched group
96     // what[2] contains the second matched group
97     // what[3] etc.
98
99     regex reBasicCmd( "^\\s*(add|print|quit).*", boost::regex::icase ) ;
100    regex reAddCmd( "^\\s*add\\s*(element|attribute)\\s(.+)\\s(.+)\\s*(.*)$", boost::regex::icase ) ;
101    regex reAddElementCmd( "^\\s*add\\s*element\\s(.+)\\s(.+)\\s*(.*)$", boost::regex::icase ) ;
102    regex reAddAttributeCmd( "^\\s*add\\s*attribute\\s(.+)\\s(.+)\\s*(.*)$", boost::regex::icase ) ;
103    regex reQuitCmd( "^\\s*quit", boost::regex::icase ) ;
104
105    // loop through all hard-coded command strings for testing purposes
106    for ( int n = 0 ; n < nCmds ; n++ ) {
107
108        // user entry point
109        cout << "\nYour command: " ;
110        // cin >> strCmd ;
111        cout << strCmd[n] << endl ;
112
113        // string version of a matched group
114        // for building a bridge between the cmatch type and an STL sting so that we can
115        // process matches with STL string functions
116        string strWhat ;
117
118        // test for a match of a basic command
119        if ( regex_match( strCmd[n].c_str(), what, reBasicCmd ) ) {
120            cout << " what.size() = " << what.size() << endl ;
121            for ( int k = 0 ; k < what.size() ; k++ ) {
122                strWhat = what[k] ;
123                cout << " what[" << k << "] = " << what[k] << " (" << strWhat.size() << " chars)" << endl ;
124            }
125
126            // handle an ADD command
127            if ( iequals( strWhat, "add" ) ) {
128                cout << " Command is ADD" << endl ;
129
130                // test for a match on the second word in the command
131                if ( regex_match( strCmd[n].c_str(), what, reAddCmd ) ) {
132                    for ( int k = 0 ; k < what.size() ; k++ ) {
133                        strWhat = what[k] ;
134                        cout << " what[" << k << "] = " << what[k] << " (" << strWhat.size() << " chars)" << endl ;
135                    }
136                    strWhat = what[1] ;
137

```

```

138         // handle an ADD ELEMENT command
139         if ( iequal( strWhat, "element" ) ) {
140             cout << " Command is ADD ELEMENT" << endl ;
141             cout << " Continue with adding an element here." << endl ;
142         }
143         // handle an ADD ATTRIBUTE command
144         else if ( iequal( strWhat, "attribute" ) ) {
145             cout << " Command is ADD ATTRIBUTE" << endl ;
146             cout << " Continue with adding an attribute here." << endl ;
147         }
148         // parsing error: ADD is followed by an invalid keyword
149         else {
150             cout << " Invalid ADD command: 2nd word must be 'element' or 'attribute'." << endl ;
151         }
152     }
153     // parsing error: ADD command syntax does not match the regular expression
154     else {
155         cout << " Invalid ADD command syntax." << endl ;
156     }
157 }
158
159 // handle a PRINT command
160 else if ( iequal( strWhat, "print" ) ) {
161     cout << " Command is PRINT" << endl ;
162     cout << " Call your print function here." << endl ;
163 }
164
165 // handle a QUIT command
166 else if ( iequal( strWhat, "quit" ) ) {
167     cout << " Command is QUIT" << endl ;
168     cout << " Goodbye." << endl ;
169     return ;
170 }
171
172 // parsing error: the first keyword is not ADD, PRINT, or QUIT
173 else {
174     cout << " Invalid command: 1st word must be 'add', 'print', or 'quit'." << endl ;
175 }
176 }
177 }
178 }
179
180
181
182
183 // this function is modeled after code found in regex_match_example.cpp
184 // note that this code demonstrates just one way to address the issue of parsing
185 // a command line using regular expressions, other approaches are not only
186 // possible, but perhaps even better
187 void test2_BasicCommandParsing_v2() {
188
189     string strCmd[10] ;
190     strCmd[0] = " add element root first one" ;
191     strCmd[1] = " add element root second" ;
192     strCmd[2] = " add attribute first attr1 attr1value" ;
193     strCmd[3] = " add attribute second attr2" ;
194     strCmd[4] = "print" ;
195     strCmd[5] = "a" ;
196     strCmd[6] = "ad" ;
197     strCmd[7] = "add" ;
198     strCmd[8] = "quit" ;
199     strCmd[9] = "another command" ;
200     int nCmds = 10 ;
201
202     cmatch what;
203     // what[0] contains the entire matched string
204     // what[1] contains the first matched group
205     // what[2] contains the second matched group
206     // what[3] etc.

```

```

207
208     regex reAddCmd( "\\s*a(d|dd)?.*", boost::regex::icase ) ;
209     regex rePrintCmd( "\\s*p(r|ri|rin|rint)?.*", boost::regex::icase ) ;
210     regex reQuitCmd( "\\s*q(u|ui|uit)?.*", boost::regex::icase ) ;
211
212     // loop through all hard-coded command strings for testing purposes
213     for ( int n = 0 ; n < nCmds ; n++ ) {
214
215         // user entry point
216         cout << "\nYour command: " ;
217         // cin >> strCmd ;
218         cout << strCmd[n] << endl ;
219
220         // string version of a matched group
221         // for building a bridge between the cmatch type and an STL sting so that we can
222         // process matches with STL string functions
223         string strWhat ;
224
225         // test for a match of an ADD command
226         if ( regex_match( strCmd[n].c_str(), what, reAddCmd ) ) {
227             cout << " Command is ADD" << endl ;
228             cout << " Call a function to do your add command processing here." << endl ;
229         }
230
231         // test for a match of a PRINT command
232         else if ( regex_match( strCmd[n].c_str(), what, rePrintCmd ) ) {
233             cout << " Command is PRINT" << endl ;
234             cout << " Call your print function here." << endl ;
235         }
236
237         // handle a QUIT command
238         else if ( regex_match( strCmd[n].c_str(), what, reQuitCmd ) ) {
239             cout << " Command is QUIT" << endl ;
240             cout << " Goodbye." << endl ;
241             return ;
242         }
243
244         // parsing error: the first keyword is not ADD, PRINT, or QUIT
245         else {
246             cout << " Invalid command: 1st word must be 'add', 'print', or 'quit'." << endl ;
247         }
248     }
249 }
250
251 // standard C++ main function
252 int main( int argc, char* argv[] ) {
253     // test1_BasicAndCaseInsensitiveMatches() ;
254     // test2_BasicCommandParsing_v1() ;
255     test2_BasicCommandParsing_v2() ;
256
257     return 0 ;
258 }
259
260
261
262 =====

```

Anchors

<code>^</code>	Start of line +
<code>\A</code>	Start of string +
<code>\$</code>	End of line +
<code>\Z</code>	End of string +
<code>\b</code>	Word boundary +
<code>\B</code>	Not word boundary +
<code>\<</code>	Start of word
<code>\></code>	End of word

Character Classes

<code>\c</code>	Control character
<code>\s</code>	White space
<code>\S</code>	Not white space
<code>\d</code>	Digit
<code>\D</code>	Not digit
<code>\w</code>	Word
<code>\W</code>	Not word
<code>\xhh</code>	Hexadecimal character hh
<code>\Oxxx</code>	Octal character xxx

POSIX Character Classes

<code>[:upper:]</code>	Upper case letters
<code>[:lower:]</code>	Lower case letters
<code>[:alpha:]</code>	All letters
<code>[:alnum:]</code>	Digits and letters
<code>[:digit:]</code>	Digits
<code>[:xdigit:]</code>	Hexadecimal digits
<code>[:punct:]</code>	Punctuation
<code>[:blank:]</code>	Space and tab
<code>[:space:]</code>	Blank characters
<code>[:cntrl:]</code>	Control characters
<code>[:graph:]</code>	Printed characters
<code>[:print:]</code>	Printed characters and spaces
<code>[:word:]</code>	Digits, letters and underscore

Assertions

<code>?=</code>	Lookahead assertion +
<code>?!</code>	Negative lookahead +
<code>?<=</code>	Lookbehind assertion +
<code>?!= or ?<!</code>	Negative lookbehind +
<code>?></code>	Once-only Subexpression
<code>?()</code>	Condition [if then]
<code>?() </code>	Condition [if then else]
<code>?#</code>	Comment

Note

Items marked + should work in most regular expression implementations.

Sample Patterns

<code>([A-Za-z0-9-]+)</code>	Letters, numbers and hyphens
<code>(\d{1,2}\V\d{1,2}\V\d{4})</code>	Date (e.g. 21/3/2006)
<code>([^\s]+(?:=\.(jpg gif png)))\.\2)</code>	jpg, gif or png image
<code>(^[1-9]{1}\$ ^[1-4]{1}[0-9]{1}\$ ^[50]\$)</code>	Any number from 1 to 50 inclusive
<code>(#?([A-Fa-f0-9]){3}([A-Fa-f0-9]){3})?</code>	Valid hexadecimal colour code
<code>((?=[*\d])(?=[*a-z])(?=[*A-Z]).{8,15})</code>	8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
<code>(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})</code>	Email addresses
<code>(\<(/?[^\>]+)\>)</code>	HTML Tags

Note

These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.

Quantifiers

<code>*</code>	0 or more +
<code>*?</code>	0 or more, ungreedy +
<code>+</code>	1 or more +
<code>+?</code>	1 or more, ungreedy +
<code>?</code>	0 or 1 +
<code>??</code>	0 or 1, ungreedy +
<code>{3}</code>	Exactly 3 +
<code>{3,}</code>	3 or more +
<code>{3,5}</code>	3, 4 or 5 +
<code>{3,5}?</code>	3, 4 or 5, ungreedy +

Special Characters

<code>\</code>	Escape Character +
<code>\n</code>	New line +
<code>\r</code>	Carriage return +
<code>\t</code>	Tab +
<code>\v</code>	Vertical tab +
<code>\f</code>	Form feed +
<code>\a</code>	Alarm
<code>[\b]</code>	Backspace
<code>\e</code>	Escape
<code>\N{name}</code>	Named Character

String Replacement (Backreferences)

<code>\$n</code>	nth non-passive group
<code>\$2</code>	"xyz" in <code>/^(abc(xyz))\$/</code>
<code>\$1</code>	"xyz" in <code>/^(?:abc)(xyz)\$/</code>
<code>\$`</code>	Before matched string
<code>\$'</code>	After matched string
<code>\$+</code>	Last matched string
<code>\$&</code>	Entire matched string
<code>\$_</code>	Entire input string
<code>\$\$</code>	Literal "\$"

Ranges

<code>.</code>	Any character except new line (<code>\n</code>) +
<code>(a b)</code>	a or b +
<code>(...)</code>	Group +
<code>(?:...)</code>	Passive Group +
<code>[abc]</code>	Range (a or b or c) +
<code>[^abc]</code>	Not a or b or c +
<code>[a-q]</code>	Letter between a and q +
<code>[A-Q]</code>	Upper case letter + between A and Q +
<code>[0-7]</code>	Digit between 0 and 7 +
<code>\n</code>	nth group/subpattern +

Note

Ranges are inclusive.

Pattern Modifiers

<code>g</code>	Global match
<code>i</code>	Case-insensitive
<code>m</code>	Multiple lines
<code>s</code>	Treat string as single line
<code>x</code>	Allow comments and white space in pattern
<code>e</code>	Evaluate replacement
<code>U</code>	Ungreedy pattern

Metacharacters (must be escaped)

<code>^</code>	<code>[</code>	<code>.</code>
<code>\$</code>	<code>{</code>	<code>*</code>
<code>(</code>	<code>\</code>	<code>+</code>
<code>)</code>	<code> </code>	<code>?</code>
<code><</code>	<code>></code>	



"...one of the most highly regarded and expertly designed C++ library projects in the world."
— [Herb Sutter](#) and [Andrei Alexandrescu](#), [C++ Coding Standards](#)

Boost.Regex

John Maddock

Copyright © 1998-2010 John Maddock

Distributed under 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)

Table of Contents

Configuration

- Compiler Setup
- Locale and traits class selection
- Linkage Options
- Algorithm Selection
- Algorithm Tuning

Building and Installing the Library

Introduction and Overview

Unicode and Boost.Regex

Understanding Marked Sub-Expressions and Captures

Partial Matches

Regular Expression Syntax

- Perl Regular Expression Syntax
- POSIX Extended Regular Expression Syntax
- POSIX Basic Regular Expression Syntax
- Character Class Names
 - Character Classes that are Always Supported
 - Character classes that are supported by Unicode Regular Expressions

Collating Names

- Digraphs
- POSIX Symbolic Names
- Named Unicode Characters

The Leftmost Longest Rule

Search and Replace Format String Syntax

- Sed Format String Syntax
- Perl Format String Syntax
- Boost-Extended Format String Syntax

Reference

- basic_regex
- match_results
- sub_match
- regex_match
- regex_search
- regex_replace
- regex_iterator
- regex_token_iterator
- bad_expression
- syntax_option_type
 - syntax_option_type Synopsis
 - Overview of syntax_option_type
 - Options for Perl Regular Expressions
 - Options for POSIX Extended Regular Expressions
 - Options for POSIX Basic Regular Expressions
 - Options for Literal Strings
- match_flag_type
- error_type

- regex_traits
- Interfacing With Non-Standard String Types
 - Working With Unicode and ICU String Types
 - Introduction to using Regex with ICU
 - Unicode regular expression types
 - Unicode Regular Expression Algorithms
 - Unicode Aware Regex Iterators
 - Using Boost Regex With MFC Strings
 - Introduction to Boost.Regex and MFC Strings
 - Regex Types Used With MFC Strings
 - Regular Expression Creation From an MFC String
 - Overloaded Algorithms For MFC String Types
 - Iterating Over the Matches Within An MFC String
- POSIX Compatible C API's
- Concepts
 - charT Requirements
 - Traits Class Requirements
 - Iterator Requirements
- Deprecated Interfaces
 - regex_format (Deprecated)
 - regex_grep (Deprecated)
 - regex_split (deprecated)
 - High Level Class RegEx (Deprecated)
- Internal Details
 - Unicode Iterators
- Background Information
 - Headers
 - Localization
 - Thread Safety
 - Test and Example Programs
 - References and Further Information
 - FAQ
 - Performance
 - Standards Conformance
 - Redistributables
 - Acknowledgements
 - History

A printer-friendly PDF version of this manual is also available.

Last revised: March 28, 2012 at 11:56:40 GMT



"...one of the most highly regarded and expertly designed C++ library projects in the world."
 — [Herb Sutter](#) and [Andrei Alexandrescu](#), [C++ Coding Standards](#)

Chapter 27. Boost String Algorithms Library

Pavol Droba

Copyright © 2002-2004 Pavol Droba

Use, modification and distribution is 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)

Table of Contents

Introduction

Release Notes

Usage

First Example

Case conversion

Predicates and Classification

Trimming

Find algorithms

Replace Algorithms

Find Iterator

Split

Quick Reference

Algorithms

Finders and Formatters

Iterators

Classification

Design Topics

String Representation

Sequence Traits

Find Algorithms

Replace Algorithms

Find Iterators & Split Algorithms

Exception Safety

Concepts

Definitions

Finder Concept

Formatter concept

Reference

Header `<boost/algorithm/string.hpp>`

Header `<boost/algorithm/string/case_conv.hpp>`

Header `<boost/algorithm/string/classification.hpp>`

Header `<boost/algorithm/string/compare.hpp>`

Header `<boost/algorithm/string/concept.hpp>`

Header `<boost/algorithm/string/constants.hpp>`

Header `<boost/algorithm/string/erase.hpp>`

Header `<boost/algorithm/string/find.hpp>`

Header `<boost/algorithm/string/find_format.hpp>`

Header `<boost/algorithm/string/find_iterator.hpp>`

Header `<boost/algorithm/string/finder.hpp>`

Header `<boost/algorithm/string/formatter.hpp>`

Header `<boost/algorithm/string/iter_find.hpp>`

Header `<boost/algorithm/string/join.hpp>`

Header `<boost/algorithm/string/predicate.hpp>`

Header `<boost/algorithm/string/regex.hpp>`

Header `<boost/algorithm/string/regex_find_format.hpp>`

Header `<boost/algorithm/string/replace.hpp>`

Header `<boost/algorithm/string/sequence_traits.hpp>`

```
Header <boost/algorithm/string/split.hpp>
Header <boost/algorithm/string/std_containers_traits.hpp>
Header <boost/algorithm/string/trim.hpp>
Header <boost/algorithm/string/trim_all.hpp>
Header <boost/algorithm/string_regex.hpp>
Rationale
  Locales
  Regular Expressions
Environment
  Build
  Examples
  Tests
  Portability
Credits
  Acknowledgments
```

Introduction

The String Algorithm Library provides a generic implementation of string-related algorithms which are missing in STL. It is an extension to the algorithms library of STL and it includes trimming, case conversion, predicates and find/replace functions. All of them come in different variants so it is easier to choose the best fit for a particular need.

The implementation is not restricted to work with a particular container (like `std::basic_string`), rather it is as generic as possible. This generalization is not compromising the performance since algorithms are using container specific features when it means a performance gain.

Important note: In this documentation we use term *string* to designate a sequence of *characters* stored in an arbitrary container. A *string* is not restricted to `std::basic_string` and *character* does not have to be `char` or `wchar_t`, although these are most common candidates. Consult the design chapter to see precise specification of supported string types.

The library interface functions and classes are defined in namespace `boost::algorithm`, and they are lifted into namespace `boost` via using declaration.

The documentation is divided into several sections. For a quick start read the Usage section followed by Quick Reference. The Design Topics, Concepts and Rationale provide some explanation about the library design and structure and explain how it should be used. See the Reference for the complete list of provided utilities and algorithms. Functions and classes in the reference are organized by the headers in which they are defined. The reference contains links to the detailed description for every entity in the library.

Last revised: July 10, 2010 at 21:29:03 +0100
