```
This file was updated on Saturday, 2012-11-24 at 9:32 AM


======================================================================
RegexTests.cpp
======================================================================


/**
 *  File:  /~heines/91.204/91.204-2012-13f/204-lecs/code/BoostRegexTests/RegExTests.cpp
 *  Jesse M. Heines, UMass Lowell Computer Science, heines@cs.uml.edu
 *  Copyright (c) 2012 by Jesse M. Heines.  All rights reserved.  May be freely
 *    copied or excerpted for educational purposes with credit to the author.
 *  updated by JMH on November 19, 2012 at 8:12 PM
 *  updated by JMH on November 23, 2012 at 9:58 PM
 */

#include <iostream>  // for cout and friends
#include <sstream>   // for string streams
#include <string>    // for the STL string class

#include <boost/regex.hpp>              // for regex_match
#include <boost/algorithm/string.hpp>   // for iequals (case-insensitive match)
  // the Boost string library contains many string manipulation functions not found in
  //    the STL library that you may be familiar with from Java or JavaScript, such as
  //    case-insensitive comparisons and trimming
  // see http://stackoverflow.com/questions/11635/case-insensitive-string-comparison-in-c
  // see http://www.boost.org/doc/libs/1_52_0/doc/html/string_algo/usage.html
  // see http://www.boost.org/doc/libs/1_52_0/doc/html/string_algo/reference.html


using namespace std ;     // to eliminate the need for std::
using namespace boost ;   // to eliminate the need for boost::


// this function is modeled after code found in credit_card_example.cpp
void test1_BasicAndCaseInsensitiveMatches() {
  // set up the strings to be tested
  string str[ 10 ] ;
  str[0] = "quit" ;
  str[1] = "exit" ;
  str[2] = "Quit" ;
  str[3] = "quite" ;
  str[4] = "unrequited" ;
  str[5] = "unreQUITed" ;
  int nStrings = 6 ;

  // define a regular expression to test for "quit"
  const boost::regex reQuit( "quit" ) ;
  cout << "Results of case-sensitive searches:" << endl ;
  // test each string against the regular expression
  for ( int k = 0 ; k < nStrings ; k++ ) {
    cout << "attempting to match \"" << str[k] << "\" to \"" << reQuit << "\" returned "
        << boost::regex_match( str[k], reQuit ) << endl ;
  }

  // define a regular expression to test for "quit" anywhere in a string
  const boost::regex reQuitA( ".*quit.*" ) ;
  cout << "\nResults of anywhere searches:" << endl ;
  // test each string against the regular expression
  for ( int k = 0 ; k < nStrings ; k++ ) {
    cout << "attempting to match \"" << str[k] << "\" to \"" << reQuit << "\" returned "
        << boost::regex_match( str[k], reQuitA ) << endl ;
  }

  // define a case-insensitive regular expression to test for "quit" anywhere in a string
  const boost::regex reQuitI( ".*quit.*", boost::regex::icase ) ;
  cout << "\nResults of case-insensitive anywhere searches:" << endl ;
  // test each string against the regular expression
  for ( int k = 0 ; k < nStrings ; k++ ) {
    cout << "attempting to match \"" << str[k] << "\" to \"" << reQuitI << "\" returned "
        << boost::regex_match( str[k], reQuitI ) << endl ;
  }
}


// this function is modeled after code found in regex_match_example.cpp
// note that this code demonstrates just one way to address the issue of parsing
//    a command line using regular expressions, other approaches are not only
//    possible, but perhaps even better
void test2_BasicCommandParsing_v1() {

  string strCmd[10] ;
  strCmd[0] = " add element root first one" ;
  strCmd[1] = "  add element root second" ;
  strCmd[2] = "  add  attribute  first   attr1 attr1value" ;
  strCmd[3] = "  add  attribute  second  attr2" ;
  strCmd[4] = "print" ;
  strCmd[5] = "quit" ;
  strCmd[6] = "another command" ;
  int nCmds = 7 ;

  cmatch what;
  // what[0] contains the entire matched string
  // what[1] contains the first matched group
  // what[2] contains the second matched group
  // what[3] etc.

  regex reBasicCmd( "^\\s*(add|print|quit).*", boost::regex::icase ) ;
  regex reAddCmd( "^\\s*add\\s*(element|attribute)\\s(.+)\\s(.+)\\s*(.*)$", boost::regex::icase ) ;
  regex reAddElementCmd( "^\\s*add\\s*element\\s(.+)\\s(.+)\\s*(.*)$", boost::regex::icase ) ;
  regex reAddAttributeCmd( "^\\s*add\\s*attribute\\s(.+)\\s(.+)\\s*(.*)$", boost::regex::icase ) ;
  regex reQuitCmd( "^\\s*quit", boost::regex::icase ) ;

  // loop through all hard-coded command strings for testing purposes
  for ( int n = 0 ; n < nCmds ; n++ ) {

    // user entry point
    cout << "\nYour command: " ;
    // cin >> strCmd ;
    cout << strCmd[n] << endl ;

    // string version of a matched group
    // for building a bridge between the cmatch type and an STL sting so that we can
    //    process matches with STL string functions
    string strWhat ;

    // test for a match of a basic command
    if ( regex_match( strCmd[n].c_str(), what, reBasicCmd ) ) {
      cout << "  what.size() = " << what.size() << endl ;
      for ( int k = 0 ; k < what.size() ; k++ ) {
        strWhat = what[k] ;
        cout << "    what[" << k << "] = " << what[k] << " (" << strWhat.size()  << " chars)" << endl ;
      }

      // handle an ADD command
      if ( iequals( strWhat, "add" ) ) {
        cout << "  Command is ADD" << endl ;

        // test for a match on the second word in the command
        if ( regex_match( strCmd[n].c_str(), what, reAddCmd ) ) {
          for ( int k = 0 ; k < what.size() ; k++ ) {
            strWhat = what[k] ;
            cout << "    what[" << k << "] = " << what[k] << " (" << strWhat.size()  << " chars)" << endl ;
          }
          strWhat = what[1] ;

          // handle an ADD ELEMENT command
          if ( iequals( strWhat, "element" ) ) {
            cout << "  Command is ADD ELEMENT" << endl ;
            cout << "  Continue with adding an element here." << endl ;
          }
          // handle an ADD ATTRIBUTE command
          else if ( iequals( strWhat, "attribute" ) ) {
            cout << "  Command is ADD ATTRIBUTE" << endl ;
            cout << "  Continue with adding an attribute here." << endl ;
          }
          // parsing error: ADD is followed by an invalid keyword
          else {
            cout << "    Invalid ADD command: 2nd word must be 'element' or 'attribute'." << endl ;
          }
        }
        // parsing error: ADD command syntax does not match the regular expression
        else {
          cout << "    Invalid ADD command syntax." << endl ;
        }
      }

      // handle a PRINT command
      else if ( iequals( strWhat, "print" ) ) {
        cout << "  Command is PRINT" << endl ;
        cout << "  Call your print function here." << endl ;
      }

      // handle a QUIT command
      else if ( iequals( strWhat, "quit" ) ) {
        cout << "  Command is QUIT" << endl ;
        cout << "  Goodbye." << endl ;
        return ;
      }

      // parsing error: the first keyword is not ADD, PRINT, or QUIT
      else {
        cout << "  Invalid command: 1st word must be 'add', 'print', or 'quit'." << endl ;
      }
    }
  }
}




// this function is modeled after code found in regex_match_example.cpp
// note that this code demonstrates just one way to address the issue of parsing
//    a command line using regular expressions, other approaches are not only
//    possible, but perhaps even better
void test2_BasicCommandParsing_v2() {

  string strCmd[10] ;
  strCmd[0] = " add element root first one" ;
  strCmd[1] = "  add element root second" ;
  strCmd[2] = "  add  attribute  first   attr1 attr1value" ;
  strCmd[3] = "  add  attribute  second  attr2" ;
  strCmd[4] = "print" ;
  strCmd[5] = "a" ;
  strCmd[6] = "ad" ;
  strCmd[7] = "add" ;
  strCmd[8] = "quit" ;
  strCmd[9] = "another command" ;
  int nCmds = 10 ;

  cmatch what;
  // what[0] contains the entire matched string
  // what[1] contains the first matched group
  // what[2] contains the second matched group
  // what[3] etc.

  regex reAddCmd( "^\\s*a(d|dd)?.*", boost::regex::icase ) ;
  regex rePrintCmd( "^\\s*p(r|ri|rin|rint)?.*", boost::regex::icase ) ;
  regex reQuitCmd( "^\\s*q(u|ui|uit)?.*", boost::regex::icase ) ;

  // loop through all hard-coded command strings for testing purposes
  for ( int n = 0 ; n < nCmds ; n++ ) {

    // user entry point
    cout << "\nYour command: " ;
    // cin >> strCmd ;
    cout << strCmd[n] << endl ;

    // string version of a matched group
    // for building a bridge between the cmatch type and an STL sting so that we can
    //    process matches with STL string functions
    string strWhat ;

    // test for a match of an ADD command
    if ( regex_match( strCmd[n].c_str(), what, reAddCmd ) ) {
      cout << "  Command is ADD" << endl ;
      cout << "  Call a function to do your add command processing here." << endl ;
    }

    // test for a match of a PRINT command
    else if ( regex_match( strCmd[n].c_str(), what, rePrintCmd ) ) {
      cout << "  Command is PRINT" << endl ;
      cout << "  Call your print function here." << endl ;
    }

    // handle a QUIT command
    else if ( regex_match( strCmd[n].c_str(), what, reQuitCmd ) ) {
      cout << "  Command is QUIT" << endl ;
      cout << "  Goodbye." << endl ;
      return ;
    }

    // parsing error: the first keyword is not ADD, PRINT, or QUIT
    else {
      cout << "  Invalid command: 1st word must be 'add', 'print', or 'quit'." << endl ;
    }
  }
}


// standard C++ main function
int main( int argc, char* argv[] ) {
  // test1_BasicAndCaseInsensitiveMatches() ;
  // test2_BasicCommandParsing_v1() ;
  test2_BasicCommandParsing_v2() ;

  return 0 ;
}


======================================================================
```

regularexpressions

Anchors

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

Character Classes

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

POSIX Character Classes

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

Assertions

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

Note Items marked + should work in most regular expression implementations.

Sample Patterns

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

*	0 or more +
*?	0 or more, ungreedy +
+	1 or more +
+?	1 or more, ungreedy +
?	0 or 1 +
??	0 or 1, ungreedy +
{3}	Exactly 3 +
{3,}	3 or more +
{3,5}	3, 4 or 5 +
{3,5}?	3, 4 or 5, ungreedy +

Special Characters

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

String Replacement (Backreferences)

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

Ranges

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

Note *Ranges are inclusive.*

Pattern Modifiers

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

Metacharacters (must be escaped)

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

Available free from AddedBytes.com

"...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

Table of Contents

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

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

boost C++ LIBRARIES

"...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

Table of Contents

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 an 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