Boost.Regex 5.0.0

John Maddock

Table of Contents

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

Configuration

Compiler Setup

You shouldn't need to do anything special to configure Boost.Regex for use with your compiler - the Boost.Config subsystem should already take care of it, if you do have problems (or you are using a particularly obscure compiler or platform) then Boost.Config has a configure script that you can run.

Locale and traits class selection

The following macros (see user.hpp) control how Boost.Regex interacts with the user's locale:

macro	**description**
BOOST_REGEX_USE_C_LOCALE	Forces Boost.Regex to use the global C locale in its traits class support: this is now deprecated in favour of the C++ locale.
BOOST_REGEX_USE_CPP_LOCALE	Forces Boost.Regex to use std::locale in it's default traits class, regular expressions can then be imbued with an instance specific locale. This is the default behaviour on non-Windows platforms.
BOOST_REGEX_NO_W32	Tells Boost.Regex not to use any Win32 API's even when available (implies BOOST_REGEX_USE_CPP_LOCALE unless BOOST_REGEX_USE_C_LOCALE is set).

Linkage Options

macro	**description**
BOOST_REGEX_DYN_LINK	For Microsoft and Borland C++ builds, this tells Boost.Regex that it should link to the dll build of the Boost.Regex. By default boost.regex will link to its static library build, even if the dynamic C runtime library is in use.
BOOST_REGEX_NO_LIB	For Microsoft and Borland C++ builds, this tells Boost.Regex that it should not automatically select the library to link to.
BOOST_REGEX_NO_FASTCALL	For Microsoft builds, this tells Boost.Regex to use the `__cdecl` calling convention rather than `__fastcall`. Useful if you want to use the same library from both managed and unmanaged code.

Algorithm Selection

macro	description
BOOST_REGEX_RECURSIVE	Tells Boost.Regex to use a stack-recursive matching algorithm. This is generally the fastest option (although there is very little in it), but can cause stack overflow in extreme cases, on Win32 this can be handled safely, but this is not the case on other platforms.
BOOST_REGEX_NON_RECURSIVE	Tells Boost.Regex to use a non-stack recursive matching algorithm, this can be slightly slower than the alternative, but is always safe no matter how pathological the regular expression. This is the default on non-Win32 platforms.

Algorithm Tuning

The following option applies only if BOOST_REGEX_RECURSIVE is set.

macro	description
BOOST_REGEX_HAS_MS_STACK_GUARD	Tells Boost.Regex that Microsoft style __try - __except blocks are supported, and can be used to safely trap stack overflow.

The following options apply only if BOOST_REGEX_NON_RECURSIVE is set.

macro	description
BOOST_REGEX_BLOCKSIZE	In non-recursive mode, Boost.Regex uses largish blocks of memory to act as a stack for the state machine, the larger the block size then the fewer allocations that will take place. This defaults to 4096 bytes, which is large enough to match the vast majority of regular expressions without further allocations, however, you can choose smaller or larger values depending upon your platforms characteristics.
BOOST_REGEX_MAX_BLOCKS	Tells Boost.Regex how many blocks of size BOOST_REGEX_BLOCKSIZE it is permitted to use. If this value is exceeded then Boost.Regex will stop trying to find a match and throw a std::runtime_error. Defaults to 1024, don't forget to tweak this value if you alter BOOST_REGEX_BLOCKSIZE by much.
BOOST_REGEX_MAX_CACHE_BLOCKS	Tells Boost.Regex how many memory blocks to store in it's internal cache - memory blocks are taken from this cache rather than by calling ::operator new. Generally speaking this can be an order of magnitude faster than calling ::opertator new each time a memory block is required, but has the downside that Boost.Regex can end up caching a large chunk of memory (by default up to 16 blocks each of BOOST_REGEX_BLOCKSIZE size). If memory is tight then try defining this to 0 (disables all caching), or if that is too slow, then a value of 1 or 2, may be sufficient. On the other hand, on large multi-processor, multi-threaded systems, you may find that a higher value is in order.

Building and Installing the Library

When you extract the library from its zip file, you must preserve its internal directory structure (for example by using the -d option when extracting). If you didn't do that when extracting, then you'd better stop reading this, delete the files you just extracted, and try again!

This library should not need configuring before use; most popular compilers/standard libraries/platforms are already supported "as is". If you do experience configuration problems, or just want to test the configuration with your compiler, then the process is the same as for all of boost; see the configuration library documentation.

The library will encase all code inside namespace boost.

Unlike some other template libraries, this library consists of a mixture of template code (in the headers) and static code and data (in cpp files). Consequently it is necessary to build the library's support code into a library or archive file before you can use it, instructions for specific platforms are as follows:

Building with bjam

This is now the preferred method for building and installing this library, please refer to the getting started guide for more information.

Building With Unicode and ICU Support

Boost.Regex is now capable of performing a configuration check to test whether ICU is already installed in your compiler's search paths. When you build you should see a message like this:

```
Performing configuration checks

    - has_icu builds           : yes
```

Whick means that ICU has been found, and support for it will be enabled in the library build.

Tip

If you don't want the regex library to use ICU then build with the "--disable-icu" command line option.

If instead you see:

```
Performing configuration checks

    - has_icu builds           : no
```

Then ICU was not found and support for it will not be compiled into the library. If you think that it should have been found, then you will need to take a look at the contents of the file *boost-root/bin.v2/config.log* for the actual error messages obtained when the build carried out the configuration check. You will then need to fix these errors by ensuring your compiler gets invoked with the correct options, for example:

```
bjam include=some-include-path --toolset=toolset-name install
```

will add "some-include-path" to your compilers header include path, or if ICU has been built with non-standard names for it's binaries, then:

```
bjam -sICU_LINK="linker-options-for-icu" --toolset=toolset-name install
```

Will use *"linker-options-for-icu"* when linking the library rather than the default ICU binary names.

You might also need to use the options "cxxflags=-option" and "linkflags=-option" to set compiler and linker specific options.

Important

Configuration results are cached - if you try rebuilding with different compiler options then add an "-a" to the bjam command line to force all targets to be rebuilt.

If ICU is not already in your compiler's path, but instead headers, libraries and binaries are located at *path-to-icu/include*, *path-to-icu/lib* and *path-to-icu/bin* respectively then you need to set the environment variable `ICU_PATH` to point to the root directory of your ICU installation: this typically happens if you're building with MSVC. For example if ICU was installed to `c:\download\icu` you might use:

```
bjam -sICU_PATH=c:\download\icu --toolset=toolset-name install
```

Important

ICU is a C++ library just like Boost is, as such your copy of ICU must have been built with the same C++ compiler (and compiler version) that you are using to build Boost. Boost.Regex will not work correctly unless you ensure that this is the case: it is up to you to ensure that the version of ICU you are using is binary compatible with the toolset you use to build Boost.

And finally, if you want to build/test with multiple compiler versions, all with different ICU builds, then the only way to achieve that currently is to modify your user-config.jam so that each toolset has the necessary compiler/linker options set so that ICU is found automatically by the configuration step (providing the ICU binaries use the standard names, all you have to add is the appropriate header-include and linker-search paths).

Building from Source

The Regex library is "just a bunch of source files": nothing special is required to build them.

You can either build the files under boost-path/libs/regex/src/*.cpp as a library, or add them directly to your project. This is particularly useful if you need to use specific compiler options not supported by the default Boost build.

There are two #defines you should be aware of:

- BOOST_HAS_ICU should be defined if you want ICU support compiled in.
- BOOST_REGEX_DYN_LINK should be defined if you are building a DLL on Windows.

Important

The makefiles that were supplied with Boost.Regex are now deprecated and will be removed in the next release.

Introduction and Overview

Regular expressions are a form of pattern-matching that are often used in text processing; many users will be familiar with the Unix utilities grep, sed and awk, and the programming language Perl, each of which make extensive use of regular expressions. Traditionally C++ users have been limited to the POSIX C API's for manipulating regular expressions, and while Boost.Regex does provide these API's, they do not represent the best way to use the library. For example Boost.Regex can cope with wide character strings, or search and replace operations (in a manner analogous to either sed or Perl), something that traditional C libraries can not do.

The class `basic_regex` is the key class in this library; it represents a "machine readable" regular expression, and is very closely modeled on `std::basic_string`, think of it as a string plus the actual state-machine required by the regular expression algorithms. Like `std::basic_string` there are two typedefs that are almost always the means by which this class is referenced:

```
namespace boost{

template <class charT,
          class traits = regex_traits<charT> >
class basic_regex;

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

}
```

To see how this library can be used, imagine that we are writing a credit card processing application. Credit card numbers generally come as a string of 16-digits, separated into groups of 4-digits, and separated by either a space or a hyphen. Before storing a credit card number in a database (not necessarily something your customers will appreciate!), we may want to verify that the number is in the correct format. To match any digit we could use the regular expression [0-9], however ranges of characters like this are actually locale dependent. Instead we should use the POSIX standard form [[:digit:]], or the Boost.Regex and Perl shorthand for this \d (note that many older libraries tended to be hard-coded to the C-locale, consequently this was not an issue for them). That leaves us with the following regular expression to validate credit card number formats:

```
(\d{4}[- ]){3}\d{4}
```

Here the parenthesis act to group (and mark for future reference) sub-expressions, and the {4} means "repeat exactly 4 times". This is an example of the extended regular expression syntax used by Perl, awk and egrep. Boost.Regex also supports the older "basic" syntax used by sed and grep, but this is generally less useful, unless you already have some basic regular expressions that you need to reuse.

Now let's take that expression and place it in some C++ code to validate the format of a credit card number:

```
bool validate_card_format(const std::string& s)
{
   static const boost::regex e("(\\d{4}[- ]){3}\\d{4}");
   return regex_match(s, e);
}
```

Note how we had to add some extra escapes to the expression: remember that the escape is seen once by the C++ compiler, before it gets to be seen by the regular expression engine, consequently escapes in regular expressions have to be doubled up when embedding them in C/C++ code. Also note that all the examples assume that your compiler supports argument-dependent-lookup lookup, if yours doesn't (for example VC6), then you will have to add some `boost::` prefixes to some of the function calls in the examples.

Those of you who are familiar with credit card processing, will have realized that while the format used above is suitable for human readable card numbers, it does not represent the format required by online credit card systems; these require the number as a string of 16 (or possibly 15) digits, without any intervening spaces. What we need is a means to convert easily between the two formats, and this is where search and replace comes in. Those who are familiar with the utilities sed and Perl will already be ahead here; we need two strings - one a regular expression - the other a "format string" that provides a description of the text to replace the match

with. In Boost.Regex this search and replace operation is performed with the algorithm `regex_replace`, for our credit card example we can write two algorithms like this to provide the format conversions:

```
// match any format with the regular expression:
const boost::regex e("\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z");
const std::string machine_format("\\1\\2\\3\\4");
const std::string human_format("\\1-\\2-\\3-\\4");

std::string machine_readable_card_number(const std::string s)
{
   return regex_replace(s, e, machine_format, boost::match_default | boost::format_sed);
}

std::string human_readable_card_number(const std::string s)
{
   return regex_replace(s, e, human_format, boost::match_default | boost::format_sed);
}
```

Here we've used marked sub-expressions in the regular expression to split out the four parts of the card number as separate fields, the format string then uses the sed-like syntax to replace the matched text with the reformatted version.

In the examples above, we haven't directly manipulated the results of a regular expression match, however in general the result of a match contains a number of sub-expression matches in addition to the overall match. When the library needs to report a regular expression match it does so using an instance of the class `match_results`, as before there are typedefs of this class for the most common cases:

```
namespace boost{

typedef match_results<const char*>                cmatch;
typedef match_results<const wchar_t*>             wcmatch;
typedef match_results<std::string::const_iterator>  smatch;
typedef match_results<std::wstring::const_iterator> wsmatch;

}
```

The algorithms `regex_search` and `regex_match` make use of `match_results` to report what matched; the difference between these algorithms is that `regex_match` will only find matches that consume *all* of the input text, where as `regex_search` will search for a match anywhere within the text being matched.

Note that these algorithms are not restricted to searching regular C-strings, any bidirectional iterator type can be searched, allowing for the possibility of seamlessly searching almost any kind of data.

For search and replace operations, in addition to the algorithm `regex_replace` that we have already seen, the `match_results` class has a `format` member that takes the result of a match and a format string, and produces a new string by merging the two.

For iterating through all occurrences of an expression within a text, there are two iterator types: `regex_iterator` will enumerate over the `match_results` objects found, while `regex_token_iterator` will enumerate a series of strings (similar to perl style split operations).

For those that dislike templates, there is a high level wrapper class `RegEx` that is an encapsulation of the lower level template code - it provides a simplified interface for those that don't need the full power of the library, and supports only narrow characters, and the "extended" regular expression syntax. This class is now deprecated as it does not form part of the regular expressions C++ standard library proposal.

The POSIX API functions: `regcomp`, `regexec`, `regfree` and [regerr], are available in both narrow character and Unicode versions, and are provided for those who need compatibility with these API's.

Finally, note that the library now has run-time localization support, and recognizes the full POSIX regular expression syntax - including advanced features like multi-character collating elements and equivalence classes - as well as providing compatibility with other regular expression libraries including GNU and BSD4 regex packages, PCRE and Perl 5.

Unicode and Boost.Regex

There are two ways to use Boost.Regex with Unicode strings:

Rely on wchar_t

If your platform's `wchar_t` type can hold Unicode strings, and your platform's C/C++ runtime correctly handles wide character constants (when passed to `std::iswspace` `std::iswlower` etc), then you can use `boost::wregex` to process Unicode. However, there are several disadvantages to this approach:

- It's not portable: there's no guarantee on the width of `wchar_t`, or even whether the runtime treats wide characters as Unicode at all, most Windows compilers do so, but many Unix systems do not.
- There's no support for Unicode-specific character classes: `[[:Nd:]]`, `[[:Po:]]` etc.
- You can only search strings that are encoded as sequences of wide characters, it is not possible to search UTF-8, or even UTF-16 on many platforms.

Use a Unicode Aware Regular Expression Type.

If you have the ICU library, then Boost.Regex can be configured to make use of it, and provide a distinct regular expression type (boost::u32regex), that supports both Unicode specific character properties, and the searching of text that is encoded in either UTF-8, UTF-16, or UTF-32. See: ICU string class support.

Understanding Marked Sub-Expressions and Captures

Captures are the iterator ranges that are "captured" by marked sub-expressions as a regular expression gets matched. Each marked sub-expression can result in more than one capture, if it is matched more than once. This document explains how captures and marked sub-expressions in Boost.Regex are represented and accessed.

Marked sub-expressions

Every time a Perl regular expression contains a parenthesis group `()`, it spits out an extra field, known as a marked sub-expression, for example the expression:

```
(\w+)\W+(\w+)
```

Has two marked sub-expressions (known as $1 and $2 respectively), in addition the complete match is known as $&, everything before the first match as $`, and everything after the match as $'. So if the above expression is searched for within `"@abc def--"`, then we obtain:

Sub-expression	**Text found**
$`	"@"
$&	"abc def"
$1	"abc"
$2	"def"
$'	"--"

In Boost.Regex all these are accessible via the `match_results` class that gets filled in when calling one of the regular expression matching algorithms (`regex_search`, `regex_match`, or `regex_iterator`). So given:

```
boost::match_results<IteratorType> m;
```

The Perl and Boost.Regex equivalents are as follows:

Perl	**Boost.Regex**
$`	`m.prefix()`
$&	`m[0]`
$n	`m[n]`
$'	`m.suffix()`

In Boost.Regex each sub-expression match is represented by a `sub_match` object, this is basically just a pair of iterators denoting the start and end position of the sub-expression match, but there are some additional operators provided so that objects of type `sub_match` behave a lot like a `std::basic_string`: for example they are implicitly convertible to a `basic_string`, they can be compared to a string, added to a string, or streamed out to an output stream.

Unmatched Sub-Expressions

When a regular expression match is found there is no need for all of the marked sub-expressions to have participated in the match, for example the expression:

```
(abc)|(def)
```

can match either $1 or $2, but never both at the same time. In Boost.Regex you can determine which sub-expressions matched by accessing the `sub_match::matched` data member.

Repeated Captures

When a marked sub-expression is repeated, then the sub-expression gets "captured" multiple times, however normally only the final capture is available, for example if

```
(?:(\w+)\W+)+
```

is matched against

```
one fine day
```

Then $1 will contain the string "day", and all the previous captures will have been forgotten.

However, Boost.Regex has an experimental feature that allows all the capture information to be retained - this is accessed either via the `match_results::captures` member function or the `sub_match::captures` member function. These functions return a container that contains a sequence of all the captures obtained during the regular expression matching. The following example program shows how this information may be used:

```
#include <boost/regex.hpp>
#include <iostream>

void print_captures(const std::string& regx, const std::string& text)
{
   boost::regex e(regx);
   boost::smatch what;
   std::cout << "Expression:  \"" << regx << "\"\n";
   std::cout << "Text:        \"" << text << "\"\n";
   if(boost::regex_match(text, what, e, boost::match_extra))
   {
      unsigned i, j;
      std::cout << "** Match found **\n   Sub-Expressions:\n";
      for(i = 0; i < what.size(); ++i)
         std::cout << "      $" << i << " = \"" << what[i] << "\"\n";
      std::cout << "   Captures:\n";
      for(i = 0; i < what.size(); ++i)
      {
         std::cout << "      $" << i << " = {";
         for(j = 0; j < what.captures(i).size(); ++j)
         {
            if(j)
               std::cout << ", ";
            else
               std::cout << " ";
            std::cout << "\"" << what.captures(i)[j] << "\"";
         }
         std::cout << " }\n";
      }
   }
   else
   {
      std::cout << "** No Match found **\n";
   }
}

int main(int , char* [])
{
   print_captures("((([:lower:]]+)|([[:upper:]]+))+", "aBBcccDDDDDeeeeeeee");
   print_captures("(.*)bar|(.*)bah", "abcbar");
   print_captures("(.*)bar|(.*)bah", "abcbah");
   print_captures("^(?:(\\w+)|(?>\\W+))*$",
      "now is the time for all good men to come to the aid of the party");
   return 0;
}
```

Which produces the following output:

```
Expression:  "((([:lower:]]+)|([[:upper:]]+))+"
Text:        "aBBcccDDDDDeeeeeeee"
** Match found **
   Sub-Expressions:
      $0 = "aBBcccDDDDDeeeeeeee"
      $1 = "eeeeeeee"
      $2 = "eeeeeeee"
      $3 = "DDDDD"
   Captures:
      $0 = { "aBBcccDDDDDeeeeeeee" }
      $1 = { "a", "BB", "ccc", "DDDDD", "eeeeeeee" }
      $2 = { "a", "ccc", "eeeeeeee" }
      $3 = { "BB", "DDDDD" }
Expression:  "(.*)bar|(.*)bah"
Text:        "abcbar"
** Match found **
   Sub-Expressions:
      $0 = "abcbar"
      $1 = "abc"
      $2 = ""
   Captures:
      $0 = { "abcbar" }
      $1 = { "abc" }
      $2 = { }
Expression:  "(.*)bar|(.*)bah"
Text:        "abcbah"
** Match found **
   Sub-Expressions:
      $0 = "abcbah"
      $1 = ""
      $2 = "abc"
   Captures:
      $0 = { "abcbah" }
      $1 = { }
      $2 = { "abc" }
Expression:  "^(?:(\w+)|(?>\W+))*$"
Text:        "now is the time for all good men to come to the aid of the party"
** Match found **
   Sub-Expressions:
      $0 = "now is the time for all good men to come to the aid of the party"
      $1 = "party"
   Captures:
      $0 = { "now is the time for all good men to come to the aid of the party" }
      $1 = { "now", "is", "the", "time", "for", "all", "good", "men", "to",
         "come", "to", "the", "aid", "of", "the", "party" }
```

Unfortunately enabling this feature has an impact on performance (even if you don't use it), and a much bigger impact if you do use it, therefore to use this feature you need to:

- Define BOOST_REGEX_MATCH_EXTRA for all translation units including the library source (the best way to do this is to uncomment this define in boost/regex/user.hpp and then rebuild everything.
- Pass the match_extra flag to the particular algorithms where you actually need the captures information (regex_search, regex_match, or regex_iterator).

Partial Matches

The `match_flag_type` `match_partial` can be passed to the following algorithms: `regex_match`, `regex_search`, and `regex_grep`, and used with the iterator `regex_iterator`. When used it indicates that partial as well as full matches should be found. A partial match is one that matched one or more characters at the end of the text input, but did not match all of the regular expression (although it may have done so had more input been available). Partial matches are typically used when either validating data input (checking each character as it is entered on the keyboard), or when searching texts that are either too long to load into memory (or even into a memory mapped file), or are of indeterminate length (for example the source may be a socket or similar). Partial and full matches can be differentiated as shown in the following table (the variable M represents an instance of `match_results` as filled in by `regex_match`, `regex_search` or `regex_grep`):

	Result	**M[0].matched**	**M[0].first**	**M[0].second**
No match	False	Undefined	Undefined	Undefined
Partial match	True	False	Start of partial match.	End of partial match (end of text).
Full match	True	True	Start of full match.	End of full match.

Be aware that using partial matches can sometimes result in somewhat imperfect behavior:

- There are some expressions, such as ".*abc" that will always produce a partial match. This problem can be reduced by careful construction of the regular expressions used, or by setting flags like match_not_dot_newline so that expressions like .* can't match past line boundaries.
- Boost.Regex currently prefers leftmost matches to full matches, so for example matching "abc|b" against "ab" produces a partial match against the "ab" rather than a full match against "b". It's more efficient to work this way, but may not be the behavior you want in all situations.

The following example tests to see whether the text could be a valid credit card number, as the user presses a key, the character entered would be added to the string being built up, and passed to `is_possible_card_number`. If this returns true then the text could be a valid card number, so the user interface's OK button would be enabled. If it returns false, then this is not yet a valid card number, but could be with more input, so the user interface would disable the OK button. Finally, if the procedure throws an exception the input could never become a valid number, and the inputted character must be discarded, and a suitable error indication displayed to the user.

```
#include <string>
#include <iostream>
#include <boost/regex.hpp>

boost::regex e("(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})");

bool is_possible_card_number(const std::string& input)
{
   //
   // return false for partial match, true for full match, or throw for
   // impossible match based on what we have so far...
   boost::match_results<std::string::const_iterator> what;
   if(0 == boost::regex_match(input, what, e, boost::match_default | boost::match_partial))
   {
      // the input so far could not possibly be valid so reject it:
      throw std::runtime_error(
         "Invalid data entered - this could not possibly be a valid card number");
   }
   // OK so far so good, but have we finished?
   if(what[0].matched)
   {
      // excellent, we have a result:
      return true;
   }
   // what we have so far is only a partial match...
   return false;
}
```

In the following example, text input is taken from a stream containing an unknown amount of text; this example simply counts the number of html tags encountered in the stream. The text is loaded into a buffer and searched a part at a time, if a partial match was encountered, then the partial match gets searched a second time as the start of the next batch of text:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <boost/regex.hpp>

// match some kind of html tag:
boost::regex e("<[^>]*>");
// count how many:
unsigned int tags = 0;

void search(std::istream& is)
{
   // buffer we'll be searching in:
   char buf[4096];
   // saved position of end of partial match:
   const char* next_pos = buf + sizeof(buf);
   // flag to indicate whether there is more input to come:
   bool have_more = true;

   while(have_more)
   {
      // how much do we copy forward from last try:
      unsigned leftover = (buf + sizeof(buf)) - next_pos;
      // and how much is left to fill:
      unsigned size = next_pos - buf;
      // copy forward whatever we have left:
      std::memmove(buf, next_pos, leftover);
      // fill the rest from the stream:
      is.read(buf + leftover, size);
      unsigned read = is.gcount();
      // check to see if we've run out of text:
      have_more = read == size;
      // reset next_pos:
      next_pos = buf + sizeof(buf);
      // and then iterate:
      boost::cregex_iterator a(
         buf,
         buf + read + leftover,
         e,
         boost::match_default | boost::match_partial);
      boost::cregex_iterator b;

      while(a != b)
      {
         if((*a)[0].matched == false)
         {
            // Partial match, save position and break:
            next_pos = (*a)[0].first;
            break;
         }
         else
         {
            // full match:
            ++tags;
         }

         // move to next match:
         ++a;
      }
   }
}
```

Regular Expression Syntax

This section covers the regular expression syntax used by this library, this is a programmers guide, the actual syntax presented to your program's users will depend upon the flags used during expression compilation.

There are three main syntax options available, depending upon how you construct the regular expression object:

- Perl (this is the default behavior).
- POSIX extended (including the egrep and awk variations).
- POSIX Basic (including the grep and emacs variations).

You can also construct a regular expression that treats every character as a literal, but that's not really a "syntax"!

Perl Regular Expression Syntax

Synopsis

The Perl regular expression syntax is based on that used by the programming language Perl . Perl regular expressions are the default behavior in Boost.Regex or you can pass the flag `perl` to the `basic_regex` constructor, for example:

```
// e1 is a case sensitive Perl regular expression:
// since Perl is the default option there's no need to explicitly specify the syntax used here:
boost::regex e1(my_expression);
// e2 a case insensitive Perl regular expression:
boost::regex e2(my_expression, boost::regex::perl|boost::regex::icase);
```

Perl Regular Expression Syntax

In Perl regular expressions, all characters match themselves except for the following special characters:

```
.[{}()\*+?|^$
```

Wildcard

The single character '.' when used outside of a character set will match any single character except:

- The NULL character when the flag `match_not_dot_null` is passed to the matching algorithms.
- The newline character when the flag `match_not_dot_newline` is passed to the matching algorithms.

Anchors

A '^' character shall match the start of a line.

A '$' character shall match the end of a line.

Marked sub-expressions

A section beginning `(` and ending `)` acts as a marked sub-expression. Whatever matched the sub-expression is split out in a separate field by the matching algorithms. Marked sub-expressions can also repeated, or referred to by a back-reference.

Non-marking grouping

A marked sub-expression is useful to lexically group part of a regular expression, but has the side-effect of spitting out an extra field in the result. As an alternative you can lexically group part of a regular expression, without generating a marked sub-expression by using `(?:` and `)`, for example `(?:ab)+` will repeat `ab` without splitting out any separate sub-expressions.

Repeats

Any atom (a single character, a marked sub-expression, or a character class) can be repeated with the `*`, `+`, `?`, and `{}` operators.

The `*` operator will match the preceding atom zero or more times, for example the expression `a*b` will match any of the following:

```
b
ab
aaaaaaaab
```

The `+` operator will match the preceding atom one or more times, for example the expression `a+b` will match any of the following:

```
ab
aaaaaaaab
```

But will not match:

```
b
```

The `?` operator will match the preceding atom zero or one times, for example the expression ca?b will match any of the following:

```
cb
cab
```

But will not match:

```
caab
```

An atom can also be repeated with a bounded repeat:

`a{n}` Matches 'a' repeated exactly n times.

`a{n,}` Matches 'a' repeated n or more times.

`a{n, m}` Matches 'a' repeated between n and m times inclusive.

For example:

```
^a{2,3}$
```

Will match either of:

```
aa
aaa
```

But neither of:

```
a
aaaa
```

It is an error to use a repeat operator, if the preceding construct can not be repeated, for example:

```
a(*)
```

Will raise an error, as there is nothing for the * operator to be applied to.

Non greedy repeats

The normal repeat operators are "greedy", that is to say they will consume as much input as possible. There are non-greedy versions available that will consume as little input as possible while still producing a match.

*? Matches the previous atom zero or more times, while consuming as little input as possible.

+? Matches the previous atom one or more times, while consuming as little input as possible.

?? Matches the previous atom zero or one times, while consuming as little input as possible.

{n,}? Matches the previous atom n or more times, while consuming as little input as possible.

{n,m}? Matches the previous atom between n and m times, while consuming as little input as possible.

Possessive repeats

By default when a repeated pattern does not match then the engine will backtrack until a match is found. However, this behaviour can sometime be undesireble so there are also "possessive" repeats: these match as much as possible and do not then allow backtracking if the rest of the expression fails to match.

*+ Matches the previous atom zero or more times, while giving nothing back.

++ Matches the previous atom one or more times, while giving nothing back.

?+ Matches the previous atom zero or one times, while giving nothing back.

{n,}+ Matches the previous atom n or more times, while giving nothing back.

{n,m}+ Matches the previous atom between n and m times, while giving nothing back.

Back references

An escape character followed by a digit *n*, where *n* is in the range 1-9, matches the same string that was matched by sub-expression *n*. For example the expression:

```
^(a*).*\1$
```

Will match the string:

```
aaabbaaa
```

But not the string:

```
aaabba
```

You can also use the \g escape for the same function, for example:

Escape	Meaning
`\g1`	Match whatever matched sub-expression 1
`\g{1}`	Match whatever matched sub-expression 1: this form allows for safer parsing of the expression in cases like `\g{1}2` or for indexes higher than 9 as in `\g{1234}`
`\g-1`	Match whatever matched the last opened sub-expression
`\g{-2}`	Match whatever matched the last but one opened sub-expression
`\g{one}`	Match whatever matched the sub-expression named "one"

Finally the \k escape can be used to refer to named subexpressions, for example `\k<two>` will match whatever matched the subexpression named "two".

Alternation

The `|` operator will match either of its arguments, so for example: `abc|def` will match either "abc" or "def".

Parenthesis can be used to group alternations, for example: `ab(d|ef)` will match either of "abd" or "abef".

Empty alternatives are not allowed (these are almost always a mistake), but if you really want an empty alternative use `(?:)` as a placeholder, for example:

`|abc` is not a valid expression, but

`(?:)|abc` is and is equivalent, also the expression:

`(?:abc)??` has exactly the same effect.

Character sets

A character set is a bracket-expression starting with `[` and ending with], it defines a set of characters, and matches any single character that is a member of that set.

A bracket expression may contain any combination of the following:

Single characters

For example `[abc]`, will match any of the characters 'a', 'b', or 'c'.

Character ranges

For example `[a-c]` will match any single character in the range 'a' to 'c'. By default, for Perl regular expressions, a character x is within the range y to z, if the code point of the character lies within the codepoints of the endpoints of the range. Alternatively, if you set the `collate` flag when constructing the regular expression, then ranges are locale sensitive.

Negation

If the bracket-expression begins with the ^ character, then it matches the complement of the characters it contains, for example `[^a-c]` matches any character that is not in the range `a-c`.

Character classes

An expression of the form `[[:name:]]` matches the named character class "name", for example `[[:lower:]]` matches any lower case character. See character class names.

Collating Elements

An expression of the form `[[.col.]]` matches the collating element *col*. A collating element is any single character, or any sequence of characters that collates as a single unit. Collating elements may also be used as the end point of a range, for example: `[[.ae.]-c]` matches the character sequence "ae", plus any single character in the range "ae"-c, assuming that "ae" is treated as a single collating element in the current locale.

As an extension, a collating element may also be specified via it's symbolic name, for example:

```
[[.NUL.]]
```

matches a `\0` character.

Equivalence classes

An expression of the form `[[=col=]]`, matches any character or collating element whose primary sort key is the same as that for collating element *col*, as with collating elements the name *col* may be a symbolic name. A primary sort key is one that ignores case, accentation, or locale-specific tailorings; so for example `[[=a=]]` matches any of the characters: a, À, Á, Â, Ã, Ä, Å, A, à, á, â, ã, ä and å. Unfortunately implementation of this is reliant on the platform's collation and localisation support; this feature can not be relied upon to work portably across all platforms, or even all locales on one platform.

Escaped Characters

All the escape sequences that match a single character, or a single character class are permitted within a character class definition. For example `[\[\]]` would match either of `[` or `]` while `[\W\d]` would match any character that is either a "digit", *or* is *not* a "word" character.

Combinations

All of the above can be combined in one character set declaration, for example: `[[:digit:]a-c[.NUL.]]`.

Escapes

Any special character preceded by an escape shall match itself.

The following escape sequences are all synonyms for single characters:

Escape	Character
`\a`	`\a`
`\e`	`0x1B`
`\f`	`\f`
`\r`	`\r`
`\t`	`\t`
`\v`	`\v`
`\b`	`\b` (but only inside a character class declaration).
`\cX`	An ASCII escape sequence - the character whose code point is X % 32
`\xdd`	A hexadecimal escape sequence - matches the single character whose code point is 0xdd.
`\x{dddd}`	A hexadecimal escape sequence - matches the single character whose code point is 0xdddd.
`\0ddd`	An octal escape sequence - matches the single character whose code point is 0ddd.
`\N{name}`	Matches the single character which has the symbolic name *name*. For example `\N{newline}` matches the single character \n.

"Single character" character classes:

Any escaped character *x*, if *x* is the name of a character class shall match any character that is a member of that class, and any escaped character *X*, if *x* is the name of a character class, shall match any character not in that class.

The following are supported by default:

Escape sequence	Equivalent to
`\d`	`[[:digit:]]`
`\l`	`[[:lower:]]`
`\s`	`[[:space:]]`
`\u`	`[[:upper:]]`
`\w`	`[[:word:]]`
`\h`	Horizontal whitespace
`\v`	Vertical whitespace
`\D`	`[^[:digit:]]`
`\L`	`[^[:lower:]]`
`\S`	`[^[:space:]]`
`\U`	`[^[:upper:]]`
`\W`	`[^[:word:]]`
`\H`	Not Horizontal whitespace
`\V`	Not Vertical whitespace

Character Properties

The character property names in the following table are all equivalent to the names used in character classes.

Form	Description	Equivalent character set form
`\pX`	Matches any character that has the property X.	`[[:X:]]`
`\p{Name}`	Matches any character that has the property Name.	`[[:Name:]]`
`\PX`	Matches any character that does not have the property X.	`[^[:X:]]`
`\P{Name}`	Matches any character that does not have the property Name.	`[^[:Name:]]`

For example `\pd` matches any "digit" character, as does `\p{digit}`.

Word Boundaries

The following escape sequences match the boundaries of words:

< Matches the start of a word.

> Matches the end of a word.

\b Matches a word boundary (the start or end of a word).

\B Matches only when not at a word boundary.

Buffer boundaries

The following match only at buffer boundaries: a "buffer" in this context is the whole of the input text that is being matched against (note that ^ and $ may match embedded newlines within the text).

\` Matches at the start of a buffer only.

\' Matches at the end of a buffer only.

\A Matches at the start of a buffer only (the same as \`).

\z Matches at the end of a buffer only (the same as \').

\Z Matches a zero-width assertion consisting of an optional sequence of newlines at the end of a buffer: equivalent to the regular expression `(?=\v*\z)`. Note that this is subtly different from Perl which behaves as if matching `(?=\n?\z)`.

Continuation Escape

The sequence `\G` matches only at the end of the last match found, or at the start of the text being matched if no previous match was found. This escape useful if you're iterating over the matches contained within a text, and you want each subsequence match to start where the last one ended.

Quoting escape

The escape sequence `\Q` begins a "quoted sequence": all the subsequent characters are treated as literals, until either the end of the regular expression or \E is found. For example the expression: `\Q*+\Ea+` would match either of:

```
\*+a
\*+aaa
```

Unicode escapes

`\C` Matches a single code point: in Boost regex this has exactly the same effect as a "." operator. `\X` Matches a combining character sequence: that is any non-combining character followed by a sequence of zero or more combining characters.

Matching Line Endings

The escape sequence `\R` matches any line ending character sequence, specifically it is identical to the expression `(?>\x0D\x0A?|[\x0A-\x0C\x85\x{2028}\x{2029}])`.

Keeping back some text

`\K` Resets the start location of $0 to the current text position: in other words everything to the left of \K is "kept back" and does not form part of the regular expression match. $` is updated accordingly.

For example `foo\Kbar` matched against the text "foobar" would return the match "bar" for $0 and "foo" for $`. This can be used to simulate variable width lookbehind assertions.

Any other escape

Any other escape sequence matches the character that is escaped, for example \@ matches a literal '@'.

Perl Extended Patterns

Perl-specific extensions to the regular expression syntax all start with `(?`.

Named Subexpressions

You can create a named subexpression using:

```
(?<NAME>expression)
```

Which can be then be referred to by the name *NAME*. Alternatively you can delimit the name using 'NAME' as in:

```
(?'NAME'expression)
```

These named subexpressions can be referred to in a backreference using either `\g{NAME}` or `\k<NAME>` and can also be referred to by name in a Perl format string for search and replace operations, or in the `match_results` member functions.

Comments

`(?# ... )` is treated as a comment, it's contents are ignored.

Modifiers

`(?imsx-imsx ... )` alters which of the perl modifiers are in effect within the pattern, changes take effect from the point that the block is first seen and extend to any enclosing `)`. Letters before a '-' turn that perl modifier on, letters afterward, turn it off.

`(?imsx-imsx:pattern)` applies the specified modifiers to pattern only.

Non-marking groups

`(?:pattern)` lexically groups pattern, without generating an additional sub-expression.

Branch reset

`(?|pattern)` resets the subexpression count at the start of each "|" alternative within *pattern*.

The sub-expression count following this construct is that of whichever branch had the largest number of sub-expressions. This construct is useful when you want to capture one of a number of alternative matches in a single sub-expression index.

In the following example the index of each sub-expression is shown below the expression:

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

Lookahead

`(?=pattern)` consumes zero characters, only if pattern matches.

`(?!pattern)` consumes zero characters, only if pattern does not match.

Lookahead is typically used to create the logical AND of two regular expressions, for example if a password must contain a lower case letter, an upper case letter, a punctuation symbol, and be at least 6 characters long, then the expression:

```
(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}
```

could be used to validate the password.

Lookbehind

`(?<=pattern)` consumes zero characters, only if pattern could be matched against the characters preceding the current position (pattern must be of fixed length).

`(?<!pattern)` consumes zero characters, only if pattern could not be matched against the characters preceding the current position (pattern must be of fixed length).

Independent sub-expressions

`(?>pattern)` *pattern* is matched independently of the surrounding patterns, the expression will never backtrack into *pattern*. Independent sub-expressions are typically used to improve performance; only the best possible match for pattern will be considered, if this doesn't allow the expression as a whole to match then no match is found at all.

Recursive Expressions

`(?N) (?-N) (?+N) (?R) (?0) (?&NAME)`

`(?R)` and `(?0)` recurse to the start of the entire pattern.

`(?N)` executes sub-expression *N* recursively, for example `(?2)` will recurse to sub-expression 2.

`(?-N)` and `(?+N)` are relative recursions, so for example `(?-1)` recurses to the last sub-expression to be declared, and `(?+1)` recurses to the next sub-expression to be declared.

`(?&NAME)` recurses to named sub-expression *NAME*.

Conditional Expressions

`(?(condition)yes-pattern|no-pattern)` attempts to match *yes-pattern* if the *condition* is true, otherwise attempts to match *no-pattern*.

`(?(condition)yes-pattern)` attempts to match *yes-pattern* if the *condition* is true, otherwise matches the NULL string.

condition may be either: a forward lookahead assert, the index of a marked sub-expression (the condition becomes true if the sub-expression has been matched), or an index of a recursion (the condition become true if we are executing directly inside the specified recursion).

Here is a summary of the possible predicates:

- `(?(?=assert)yes-pattern|no-pattern)` Executes *yes-pattern* if the forward look-ahead assert matches, otherwise executes *no-pattern*.
- `(?(?!assert)yes-pattern|no-pattern)` Executes *yes-pattern* if the forward look-ahead assert does not match, otherwise executes *no-pattern*.
- `(?(N)yes-pattern|no-pattern)` Executes *yes-pattern* if subexpression *N* has been matched, otherwise executes *no-pattern*.
- `(?(<name>)yes-pattern|no-pattern)` Executes *yes-pattern* if named subexpression *name* has been matched, otherwise executes *no-pattern*.
- `(?('name')yes-pattern|no-pattern)` Executes *yes-pattern* if named subexpression *name* has been matched, otherwise executes *no-pattern*.
- `(?(R)yes-pattern|no-pattern)` Executes *yes-pattern* if we are executing inside a recursion, otherwise executes *no-pattern*.
- `(?(RN)yes-pattern|no-pattern)` Executes *yes-pattern* if we are executing inside a recursion to sub-expression *N*, otherwise executes *no-pattern*.
- `(?(R&name)yes-pattern|no-pattern)` Executes *yes-pattern* if we are executing inside a recursion to named sub-expression *name*, otherwise executes *no-pattern*.
- `(?(DEFINE)never-exectuted-pattern)` Defines a block of code that is never executed and matches no characters: this is usually used to define one or more named sub-expressions which are referred to from elsewhere in the pattern.

Operator precedence

The order of precedence for of operators is as follows:

1. Collation-related bracket symbols `[==] [::] [..]`
2. Escaped characters [^]
3. Character set (bracket expression) `[]`
4. Grouping `()`
5. Single-character-ERE duplication `* + ? {m,n}`
6. Concatenation
7. Anchoring ^$
8. Alternation |

What gets matched

If you view the regular expression as a directed (possibly cyclic) graph, then the best match found is the first match found by a depth-first-search performed on that graph, while matching the input text.

Alternatively:

The best match found is the leftmost match, with individual elements matched as follows;

Construct	What gets matched
`AtomA AtomB`	Locates the best match for *AtomA* that has a following match for *AtomB*.
`Expression1 | Expression2`	If *Expresion1* can be matched then returns that match, otherwise attempts to match *Expression2*.
`S{N}`	Matches *S* repeated exactly N times.
`S{N,M}`	Matches S repeated between N and M times, and as many times as possible.
`S{N,M}?`	Matches S repeated between N and M times, and as few times as possible.
`S?, S*, S+`	The same as `S{0,1}`, `S{0,UINT_MAX}`, `S{1,UINT_MAX}` respectively.
`S??, S*?, S+?`	The same as `S{0,1}?`, `S{0,UINT_MAX}?`, `S{1,UINT_MAX}?` respectively.
`(?>S)`	Matches the best match for *S*, and only that.
`(?=S), (?<=S)`	Matches only the best match for *S* (this is only visible if there are capturing parenthesis within *S*).
`(?!S), (?<!S)`	Considers only whether a match for S exists or not.
`(?(condition)yes-pattern | no-pattern)`	If condition is true, then only yes-pattern is considered, otherwise only no-pattern is considered.

Variations

The options `normal`, `ECMAScript`, `JavaScript` and `JScript` are all synonyms for `perl`.

Options

There are a variety of flags that may be combined with the `perl` option when constructing the regular expression, in particular note that the `newline_alt` option alters the syntax, while the `collate`, `nosubs` and `icase` options modify how the case and locale sensitivity are to be applied.

Pattern Modifiers

The perl `smix` modifiers can either be applied using a `(?smix-smix)` prefix to the regular expression, or with one of the regex-compile time flags `no_mod_m`, `mod_x`, `mod_s`, and `no_mod_s`.

References

Perl 5.8.

POSIX Extended Regular Expression Syntax

Synopsis

The POSIX-Extended regular expression syntax is supported by the POSIX C regular expression API's, and variations are used by the utilities `egrep` and `awk`. You can construct POSIX extended regular expressions in Boost.Regex by passing the flag `extended` to the regex constructor, for example:

```
// e1 is a case sensitive POSIX-Extended expression:
boost::regex e1(my_expression, boost::regex::extended);
// e2 a case insensitive POSIX-Extended expression:
boost::regex e2(my_expression, boost::regex::extended|boost::regex::icase);
```

POSIX Extended Syntax

In POSIX-Extended regular expressions, all characters match themselves except for the following special characters:

```
.[{}()\*+?|^$
```

Wildcard:

The single character '.' when used outside of a character set will match any single character except:

- The NULL character when the flag `match_no_dot_null` is passed to the matching algorithms.
- The newline character when the flag `match_not_dot_newline` is passed to the matching algorithms.

Anchors:

A '^' character shall match the start of a line when used as the first character of an expression, or the first character of a sub-expression.

A '$' character shall match the end of a line when used as the last character of an expression, or the last character of a sub-expression.

Marked sub-expressions:

A section beginning `(` and ending `)` acts as a marked sub-expression. Whatever matched the sub-expression is split out in a separate field by the matching algorithms. Marked sub-expressions can also repeated, or referred to by a back-reference.

Repeats:

Any atom (a single character, a marked sub-expression, or a character class) can be repeated with the `*`, `+`, `?`, and `{}` operators.

The `*` operator will match the preceding atom *zero or more times*, for example the expression `a*b` will match any of the following:

```
b
ab
aaaaaaaab
```

The + operator will match the preceding atom *one or more times*, for example the expression a+b will match any of the following:

```
ab
aaaaaaaab
```

But will not match:

```
b
```

The `?` operator will match the preceding atom *zero or one times*, for example the expression `ca?b` will match any of the following:

```
cb
cab
```

But will not match:

```
caab
```

An atom can also be repeated with a bounded repeat:

`a{n}` Matches 'a' repeated *exactly n times*.

`a{n,}` Matches 'a' repeated *n or more times*.

`a{n, m}` Matches 'a' repeated *between n and m times inclusive*.

For example:

```
^a{2,3}$
```

Will match either of:

```
aa
aaa
```

But neither of:

```
a
aaaa
```

It is an error to use a repeat operator, if the preceding construct can not be repeated, for example:

```
a(*)
```

Will raise an error, as there is nothing for the `*` operator to be applied to.

Back references:

An escape character followed by a digit *n*, where *n* is in the range 1-9, matches the same string that was matched by sub-expression *n*. For example the expression:

```
^(a*).*\1$
```

Will match the string:

```
aaabbaaa
```

But not the string:

```
aaabba
```

Caution

The POSIX standard does not support back-references for "extended" regular expressions, this is a compatible extension to that standard.

Alternation

The `|` operator will match either of its arguments, so for example: `abc|def` will match either "abc" or "def".

Parenthesis can be used to group alternations, for example: `ab(d|ef)` will match either of "abd" or "abef".

Character sets:

A character set is a bracket-expression starting with [and ending with], it defines a set of characters, and matches any single character that is a member of that set.

A bracket expression may contain any combination of the following:

Single characters:

For example `[abc]`, will match any of the characters 'a', 'b', or 'c'.

Character ranges:

For example `[a-c]` will match any single character in the range 'a' to 'c'. By default, for POSIX-Extended regular expressions, a character *x* is within the range *y* to *z*, if it collates within that range; this results in locale specific behavior . This behavior can be turned off by unsetting the `collate` option flag - in which case whether a character appears within a range is determined by comparing the code points of the characters only.

Negation:

If the bracket-expression begins with the ^ character, then it matches the complement of the characters it contains, for example `[^a-c]` matches any character that is not in the range `a-c`.

Character classes:

An expression of the form `[[:name:]]` matches the named character class "name", for example `[[:lower:]]` matches any lower case character. See character class names.

Collating Elements:

An expression of the form `[[.col.]]` matches the collating element *col*. A collating element is any single character, or any sequence of characters that collates as a single unit. Collating elements may also be used as the end point of a range, for example: `[[.ae.]-c]` matches the character sequence "ae", plus any single character in the range "ae"-c, assuming that "ae" is treated as a single collating element in the current locale.

Collating elements may be used in place of escapes (which are not normally allowed inside character sets), for example `[[.^.]abc]` would match either one of the characters 'abc^'.

As an extension, a collating element may also be specified via its symbolic name, for example:

```
[[.NUL.]]
```

matches a NUL character.

Equivalence classes:

An expression of the form `[[=col=]]`, matches any character or collating element whose primary sort key is the same as that for collating element *col*, as with collating elements the name *col* may be a symbolic name. A primary sort key is one that ignores case,

accentation, or locale-specific tailorings; so for example `[[=a=]]` matches any of the characters: a, À, Á, Â, Ã, Ä, Å, A, à, á, â, ã, ä and å. Unfortunately implementation of this is reliant on the platform's collation and localisation support; this feature can not be relied upon to work portably across all platforms, or even all locales on one platform.

Combinations:

All of the above can be combined in one character set declaration, for example: `[[:digit:]a-c[.NUL.]]`.

Escapes

The POSIX standard defines no escape sequences for POSIX-Extended regular expressions, except that:

- Any special character preceded by an escape shall match itself.
- The effect of any ordinary character being preceded by an escape is undefined.
- An escape inside a character class declaration shall match itself: in other words the escape character is not "special" inside a character class declaration; so `[\^]` will match either a literal '\' or a '^'.

However, that's rather restrictive, so the following standard-compatible extensions are also supported by Boost.Regex:

Escapes matching a specific character

The following escape sequences are all synonyms for single characters:

Escape	Character
\a	'\a'
\e	0x1B
\f	\f
\n	\n
\r	\r
\t	\t
\v	\v
\b	\b (but only inside a character class declaration).
\cX	An ASCII escape sequence - the character whose code point is X % 32
\xdd	A hexadecimal escape sequence - matches the single character whose code point is 0xdd.
\x{dddd}	A hexadecimal escape sequence - matches the single character whose code point is 0xdddd.
\0ddd	An octal escape sequence - matches the single character whose code point is 0ddd.
\N{Name}	Matches the single character which has the symbolic name *Name*. For example `\\N{newline}` matches the single character \n.

"Single character" character classes:

Any escaped character *x*, if *x* is the name of a character class shall match any character that is a member of that class, and any escaped character *X*, if *x* is the name of a character class, shall match any character not in that class.

The following are supported by default:

Escape sequence	Equivalent to
`\d`	`[[:digit:]]`
`\l`	`[[:lower:]]`
`\s`	`[[:space:]]`
`\u`	`[[:upper:]]`
`\w`	`[[:word:]]`
`\D`	`[^[:digit:]]`
`\L`	`[^[:lower:]]`
`\S`	`[^[:space:]]`
`\U`	`[^[:upper:]]`
`\W`	`[^[:word:]]`

Character Properties

The character property names in the following table are all equivalent to the names used in character classes.

Form	Description	Equivalent character set form
`\pX`	Matches any character that has the property X.	`[[:X:]]`
`\p{Name}`	Matches any character that has the property Name.	`[[:Name:]]`
`\PX`	Matches any character that does not have the property X.	`[^[:X:]]`
`\P{Name}`	Matches any character that does not have the property Name.	`[^[:Name:]]`

For example `\pd` matches any "digit" character, as does `\p{digit}`.

Word Boundaries

The following escape sequences match the boundaries of words:

Escape	Meaning
\<	Matches the start of a word.
\>	Matches the end of a word.
\b	Matches a word boundary (the start or end of a word).
\B	Matches only when not at a word boundary.

Buffer boundaries

The following match only at buffer boundaries: a "buffer" in this context is the whole of the input text that is being matched against (note that ^ and $ may match embedded newlines within the text).

Escape	Meaning
\`	Matches at the start of a buffer only.
\'	Matches at the end of a buffer only.
\A	Matches at the start of a buffer only (the same as \`).
\z	Matches at the end of a buffer only (the same as \').
\Z	Matches an optional sequence of newlines at the end of a buffer: equivalent to the regular expression \n*\z

Continuation Escape

The sequence \G matches only at the end of the last match found, or at the start of the text being matched if no previous match was found. This escape useful if you're iterating over the matches contained within a text, and you want each subsequence match to start where the last one ended.

Quoting escape

The escape sequence \Q begins a "quoted sequence": all the subsequent characters are treated as literals, until either the end of the regular expression or \E is found. For example the expression: \Q*+\Ea+ would match either of:

```
\*+a
\*+aaa
```

Unicode escapes

Escape	Meaning
\C	Matches a single code point: in Boost regex this has exactly the same effect as a "." operator.
\X	Matches a combining character sequence: that is any non-combining character followed by a sequence of zero or more combining characters.

Any other escape

Any other escape sequence matches the character that is escaped, for example \@ matches a literal '@'.

Operator precedence

The order of precedence for of operators is as follows:

1. Collation-related bracket symbols `[==] [::] [..]`
2. Escaped characters \
3. Character set (bracket expression) `[]`
4. Grouping `()`
5. Single-character-ERE duplication `* + ? {m,n}`
6. Concatenation
7. Anchoring ^$
8. Alternation `|`

What Gets Matched

When there is more that one way to match a regular expression, the "best" possible match is obtained using the leftmost-longest rule.

Variations

Egrep

When an expression is compiled with the flag `egrep` set, then the expression is treated as a newline separated list of POSIX-Extended expressions, a match is found if any of the expressions in the list match, for example:

```
boost::regex e("abc\ndef", boost::regex::egrep);
```

will match either of the POSIX-Basic expressions "abc" or "def".

As its name suggests, this behavior is consistent with the Unix utility `egrep`, and with grep when used with the -E option.

awk

In addition to the POSIX-Extended features the escape character is special inside a character class declaration.

In addition, some escape sequences that are not defined as part of POSIX-Extended specification are required to be supported - however Boost.Regex supports these by default anyway.

Options

There are a variety of flags that may be combined with the `extended` and `egrep` options when constructing the regular expression, in particular note that the `newline_alt` option alters the syntax, while the `collate`, `nosubs` and `icase` options modify how the case and locale sensitivity are to be applied.

References

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions.

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, egrep.

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, awk.

POSIX Basic Regular Expression Syntax

Synopsis

The POSIX-Basic regular expression syntax is used by the Unix utility `sed`, and variations are used by `grep` and `emacs`. You can construct POSIX basic regular expressions in Boost.Regex by passing the flag `basic` to the regex constructor (see `syntax_option_type`), for example:

```
// e1 is a case sensitive POSIX-Basic expression:
boost::regex e1(my_expression, boost::regex::basic);
// e2 a case insensitive POSIX-Basic expression:
boost::regex e2(my_expression, boost::regex::basic|boost::regex::icase);
```

POSIX Basic Syntax

In POSIX-Basic regular expressions, all characters are match themselves except for the following special characters:

```
.[\*^$
```

Wildcard:

The single character '.' when used outside of a character set will match any single character except:

- The NULL character when the flag `match_no_dot_null` is passed to the matching algorithms.
- The newline character when the flag `match_not_dot_newline` is passed to the matching algorithms.

Anchors:

A '^' character shall match the start of a line when used as the first character of an expression, or the first character of a sub-expression.

A '$' character shall match the end of a line when used as the last character of an expression, or the last character of a sub-expression.

Marked sub-expressions:

A section beginning `\(` and ending `\)` acts as a marked sub-expression. Whatever matched the sub-expression is split out in a separate field by the matching algorithms. Marked sub-expressions can also repeated, or referred-to by a back-reference.

Repeats:

Any atom (a single character, a marked sub-expression, or a character class) can be repeated with the * operator.

For example `a*` will match any number of letter a's repeated zero or more times (an atom repeated zero times matches an empty string), so the expression `a*b` will match any of the following:

```
b
ab
aaaaaaaab
```

An atom can also be repeated with a bounded repeat:

`a\{n\}` Matches 'a' repeated exactly n times.

`a\{n,\}` Matches 'a' repeated n or more times.

`a\{n, m\}` Matches 'a' repeated between n and m times inclusive.

For example:

```
^a{2,3}$
```

Will match either of:

```
aa
aaa
```

But neither of:

```
a
aaaa
```

It is an error to use a repeat operator, if the preceding construct can not be repeated, for example:

```
a(*)
```

Will raise an error, as there is nothing for the * operator to be applied to.

Back references:

An escape character followed by a digit *n*, where *n* is in the range 1-9, matches the same string that was matched by sub-expression *n*. For example the expression:

```
^\(a*\).*\1$
```

Will match the string:

```
aaabbaaa
```

But not the string:

```
aaabba
```

Character sets:

A character set is a bracket-expression starting with [and ending with], it defines a set of characters, and matches any single character that is a member of that set.

A bracket expression may contain any combination of the following:

Single characters:

For example `[abc]`, will match any of the characters 'a', 'b', or 'c'.

Character ranges:

For example `[a-c]` will match any single character in the range 'a' to 'c'. By default, for POSIX-Basic regular expressions, a character *x* is within the range *y* to *z*, if it collates within that range; this results in locale specific behavior. This behavior can be turned off by unsetting the `collate` option flag when constructing the regular expression - in which case whether a character appears within a range is determined by comparing the code points of the characters only.

Negation:

If the bracket-expression begins with the ^ character, then it matches the complement of the characters it contains, for example `[^a-c]` matches any character that is not in the range a-c.

Character classes:

An expression of the form `[[:name:]]` matches the named character class "name", for example `[[:lower:]]` matches any lower case character. See character class names.

Collating Elements:

An expression of the form `[[.col.]]` matches the collating element *col*. A collating element is any single character, or any sequence of characters that collates as a single unit. Collating elements may also be used as the end point of a range, for example: `[[.ae.]-c]` matches the character sequence "ae", plus any single character in the rangle "ae"-c, assuming that "ae" is treated as a single collating element in the current locale.

Collating elements may be used in place of escapes (which are not normally allowed inside character sets), for example `[[.^.]abc]` would match either one of the characters 'abc^'.

As an extension, a collating element may also be specified via its symbolic name, for example:

```
[[.NUL.]]
```

matches a 'NUL' character. See collating element names.

Equivalence classes:

An expression of the form `[[=col=]]`, matches any character or collating element whose primary sort key is the same as that for collating element *col*, as with collating elements the name *col* may be a collating symbolic name. A primary sort key is one that ignores case, accentation, or locale-specific tailorings; so for example `[[=a=]]` matches any of the characters: a, À, Á, Â, Ã, Ä, Å, A, à, á, â, ã, ä and å. Unfortunately implementation of this is reliant on the platform's collation and localisation support; this feature can not be relied upon to work portably across all platforms, or even all locales on one platform.

Combinations:

All of the above can be combined in one character set declaration, for example: `[[:digit:]a-c[.NUL.]]`.

Escapes

With the exception of the escape sequences \{, \}, \(, and \), which are documented above, an escape followed by any character matches that character. This can be used to make the special characters

```
.[\*^$
```

"ordinary". Note that the escape character loses its special meaning inside a character set, so `[\^]` will match either a literal '\' or a '^'.

What Gets Matched

When there is more that one way to match a regular expression, the "best" possible match is obtained using the leftmost-longest rule.

Variations

Grep

When an expression is compiled with the flag `grep` set, then the expression is treated as a newline separated list of POSIX-Basic expressions, a match is found if any of the expressions in the list match, for example:

```
boost::regex e("abc\ndef", boost::regex::grep);
```

will match either of the POSIX-Basic expressions "abc" or "def".

As its name suggests, this behavior is consistent with the Unix utility grep.

emacs

In addition to the POSIX-Basic features the following characters are also special:

Character	Description
+	repeats the preceding atom one or more times.
?	repeats the preceding atom zero or one times.
*?	A non-greedy version of *.
+?	A non-greedy version of +.
??	A non-greedy version of ?.

And the following escape sequences are also recognised:

Escape	Description
|	specifies an alternative.
\(?: ...)	is a non-marking grouping construct - allows you to lexically group something without spitting out an extra sub-expression.
\w	matches any word character.
\W	matches any non-word character.
\sx	matches any character in the syntax group x, the following emacs groupings are supported: 's', ' ', '_', 'w', '.', ')', '(', '"', '\', '>' and '<'. Refer to the emacs docs for details.
\Sx	matches any character not in the syntax grouping x.
\c and \C	These are not supported.
\`	matches zero characters only at the start of a buffer (or string being matched).
\'	matches zero characters only at the end of a buffer (or string being matched).
\b	matches zero characters at a word boundary.
\B	matches zero characters, not at a word boundary.
\<	matches zero characters only at the start of a word.
\>	matches zero characters only at the end of a word.

Finally, you should note that emacs style regular expressions are matched according to the Perl "depth first search" rules. Emacs expressions are matched this way because they contain Perl-like extensions, that do not interact well with the POSIX-style leftmost-longest rule.

Options

There are a variety of flags that may be combined with the `basic` and `grep` options when constructing the regular expression, in particular note that the `newline_alt`, `no_char_classes`, `no-intervals`, `bk_plus_qm` and `bk_plus_vbar` options all alter the syntax, while the `collate` and `icase` options modify how the case and locale sensitivity are to be applied.

References

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1).

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep (FWD.1).

Emacs Version 21.3.

Character Class Names

Character Classes that are Always Supported

The following character class names are always supported by Boost.Regex:

Name	POSIX-standard name	Description
alnum	Yes	Any alpha-numeric character.
alpha	Yes	Any alphabetic character.
blank	Yes	Any whitespace character that is not a line separator.
cntrl	Yes	Any control character.
d	No	Any decimal digit
digit	Yes	Any decimal digit.
graph	Yes	Any graphical character.
l	No	Any lower case character.
lower	Yes	Any lower case character.
print	Yes	Any printable character.
punct	Yes	Any punctuation character.
s	No	Any whitespace character.
space	Yes	Any whitespace character.
unicode	No	Any extended character whose code point is above 255 in value.
u	No	Any upper case character.
upper	Yes	Any upper case character.
w	No	Any word character (alphanumeric characters plus the underscore).
word	No	Any word character (alphanumeric characters plus the underscore).
xdigit	Yes	Any hexadecimal digit character.

Character classes that are supported by Unicode Regular Expressions

The following character classes are only supported by Unicode Regular Expressions: that is those that use the `u32regex` type. The names used are the same as those from Chapter 4 of the Unicode standard.

Short Name	Long Name
	ASCII
	Any
	Assigned
C*	Other
Cc	Control
Cf	Format
Cn	Not Assigned
Co	Private Use
Cs	Surrogate
L*	Letter
Ll	Lowercase Letter
Lm	Modifier Letter
Lo	Other Letter
Lt	Titlecase
Lu	Uppercase Letter
M*	Mark
Mc	Spacing Combining Mark
Me	Enclosing Mark
Mn	Non-Spacing Mark
N*	Number
Nd	Decimal Digit Number
Nl	Letter Number
No	Other Number
P*	Punctuation
Pc	Connector Punctuation
Pd	Dash Punctuation
Pe	Close Punctuation
Pf	Final Punctuation

Short Name	Long Name
Pi	Initial Punctuation
Po	Other Punctuation
Ps	Open Punctuation
S*	Symbol
Sc	Currency Symbol
Sk	Modifier Symbol
Sm	Math Symbol
So	Other Symbol
Z*	Separator
Zl	Line Separator
Zp	Paragraph Separator
Zs	Space Separator

Collating Names

Digraphs

The following are treated as valid digraphs when used as a collating name:

"ae", "Ae", "AE", "ch", "Ch", "CH", "ll", "Ll", "LL", "ss", "Ss", "SS", "nj", "Nj", "NJ", "dz", "Dz", "DZ", "lj", "Lj", "LJ".

So for example the expression:

```
[[.ae.]-c] ↵
```

will match any character that collates between the digraph "ae" and the character "c".

POSIX Symbolic Names

The following symbolic names are recognised as valid collating element names, in addition to any single character, this allows you to write for example:

```
[[.left-square-bracket.][.right-square-bracket.]]
```

if you wanted to match either "[" or "]".

Name	Character
NUL	\x00
SOH	\x01
STX	\x02
ETX	\x03
EOT	\x04
ENQ	\x05
ACK	\x06
alert	\x07
backspace	\x08
tab	\t
newline	\n
vertical-tab	\v
form-feed	\f
carriage-return	\r
SO	\xE
SI	\xF
DLE	\x10
DC1	\x11
DC2	\x12
DC3	\x13
DC4	\x14
NAK	\x15
SYN	\x16
ETB	\x17
CAN	\x18
EM	\x19
SUB	\x1A
ESC	\x1B

Name	Character
IS4	\x1C
IS3	\x1D
IS2	\x1E
IS1	\x1F
space	\x20
exclamation-mark	!
quotation-mark	"
number-sign	#
dollar-sign	$
percent-sign	%
ampersand	&
apostrophe	'
left-parenthesis	(
right-parenthesis	)
asterisk	*
plus-sign	+
comma	,
hyphen	-
period	.
slash	/
zero	0
one	1
two	2
three	3
four	4
five	5
six	6
seven	7

Name	Character
eight	8
nine	9
colon	:
semicolon	;
less-than-sign	<
equals-sign	=
greater-than-sign	>
question-mark	?
commercial-at	@
left-square-bracket	[
backslash	\
right-square-bracket	]
circumflex	~
underscore	_
grave-accent	`
left-curly-bracket	{
vertical-line	|
right-curly-bracket	}
tilde	~
DEL	\x7F

Named Unicode Characters

When using Unicode aware regular expressions (with the `u32regex` type), all the normal symbolic names for Unicode characters (those given in Unidata.txt) are recognised. So for example:

```
[[.CYRILLIC CAPITAL LETTER I.]] ↵
```

would match the Unicode character 0x0418.

The Leftmost Longest Rule

Often there is more than one way of matching a regular expression at a particular location, for POSIX basic and extended regular expressions, the "best" match is determined as follows:

1. Find the leftmost match, if there is only one match possible at this location then return it.
2. Find the longest of the possible matches, along with any ties. If there is only one such possible match then return it.
3. If there are no marked sub-expressions, then all the remaining alternatives are indistinguishable; return the first of these found.
4. Find the match which has matched the first sub-expression in the leftmost position, along with any ties. If there is only on such match possible then return it.
5. Find the match which has the longest match for the first sub-expression, along with any ties. If there is only one such match then return it.
6. Repeat steps 4 and 5 for each additional marked sub-expression.
7. If there is still more than one possible match remaining, then they are indistinguishable; return the first one found.

Search and Replace Format String Syntax

Format strings are used by the algorithm `regex_replace` and by `match_results<>::format`, and are used to transform one string into another.

There are three kind of format string: Sed, Perl and Boost-Extended.

Alternatively, when the flag `format_literal` is passed to one of these functions, then the format string is treated as a string literal, and is copied unchanged to the output.

Sed Format String Syntax

Sed-style format strings treat all characters as literals except:

character	description
&	The ampersand character is replaced in the output stream by the whole of what matched the regular expression. Use \& to output a literal '&' character.
\	Specifies an escape sequence.

An escape character followed by any character x, outputs that character unless x is one of the escape sequences shown below.

Escape	Meaning
\a	Outputs the bell character: '\a'.
\e	Outputs the ANSI escape character (code point 27).
\f	Outputs a form feed character: '\f'
\n	Outputs a newline character: '\n'.
\r	Outputs a carriage return character: '\r'.
\t	Outputs a tab character: '\t'.
\v	Outputs a vertical tab character: '\v'.
\xDD	Outputs the character whose hexadecimal code point is 0xDD
\x{DDDD}	Outputs the character whose hexadecimal code point is 0xDDDD
\cX	Outputs the ANSI escape sequence "escape-X".
\D	If D is a decimal digit in the range 1-9, then outputs the text that matched sub-expression D.

Perl Format String Syntax

Perl-style format strings treat all characters as literals except '$' and '\' which start placeholder and escape sequences respectively.

Placeholder sequences specify that some part of what matched the regular expression should be sent to output as follows:

Placeholder	Meaning
$&	Outputs what matched the whole expression.
$MATCH	As $&
${^MATCH}	As $&
$`	Outputs the text between the end of the last match found (or the start of the text if no previous match was found), and the start of the current match.
$PREMATCH	As $`
${^PREMATCH}	As $`
$'	Outputs all the text following the end of the current match.
$POSTMATCH	As $'
${^POSTMATCH}	As $'
$+	Outputs what matched the last marked sub-expression in the regular expression.
$LAST_PAREN_MATCH	As $+
$LAST_SUBMATCH_RESULT	Outputs what matched the last sub-expression to be actually matched.
$^N	As $LAST_SUBMATCH_RESULT
$$	Outputs a literal '$'
$n	Outputs what matched the n'th sub-expression.
${n}	Outputs what matched the n'th sub-expression.
$+{NAME}	Outputs whatever matched the sub-expression named "NAME".

Any $-placeholder sequence not listed above, results in '$' being treated as a literal.

An escape character followed by any character x, outputs that character unless x is one of the escape sequences shown below.

Escape	Meaning
\a	Outputs the bell character: '\a'.
\e	Outputs the ANSI escape character (code point 27).
\f	Outputs a form feed character: '\f'
\n	Outputs a newline character: '\n'.
\r	Outputs a carriage return character: '\r'.
\t	Outputs a tab character: '\t'.
\v	Outputs a vertical tab character: '\v'.
\xDD	Outputs the character whose hexadecimal code point is 0xDD
\x{DDDD}	Outputs the character whose hexadecimal code point is 0xDDDD
\cX	Outputs the ANSI escape sequence "escape-X".
\D	If D is a decimal digit in the range 1-9, then outputs the text that matched sub-expression D.
\l	Causes the next character to be outputted, to be output in lower case.
\u	Causes the next character to be outputted, to be output in upper case.
\L	Causes all subsequent characters to be output in lower case, until a \E is found.
\U	Causes all subsequent characters to be output in upper case, until a \E is found.
\E	Terminates a \L or \U sequence.

Boost-Extended Format String Syntax

Boost-Extended format strings treat all characters as literals except for '$', '\', '(', ')', '?', and ':'.

Grouping

The characters '(' and ')' perform lexical grouping, so use \(and \) if you want a to output literal parenthesis.

Conditionals

The character '?' begins a conditional expression, the general form is:

?Ntrue-expression:false-expression

where N is decimal digit.

If sub-expression N was matched, then true-expression is evaluated and sent to output, otherwise false-expression is evaluated and sent to output.

You will normally need to surround a conditional-expression with parenthesis in order to prevent ambiguities.

For example, the format string "(?1foo:bar)" will replace each match found with "foo" if the sub-expression $1 was matched, and with "bar" otherwise.

For sub-expressions with an index greater than 9, or for access to named sub-expressions use:

?{INDEX}true-expression:false-expression

or

?{NAME}true-expression:false-expression

Placeholder Sequences

Placeholder sequences specify that some part of what matched the regular expression should be sent to output as follows:

Placeholder	**Meaning**
$&	Outputs what matched the whole expression.
$MATCH	As $&
${^MATCH}	As $&
$`	Outputs the text between the end of the last match found (or the start of the text if no previous match was found), and the start of the current match.
$PREMATCH	As $`
${^PREMATCH}	As $`
$'	Outputs all the text following the end of the current match.
$POSTMATCH	As $'
${^POSTMATCH}	As $'
$+	Outputs what matched the last marked sub-expression in the regular expression.
$LAST_PAREN_MATCH	As $+
$LAST_SUBMATCH_RESULT	Outputs what matched the last sub-expression to be actually matched.
$^N	As $LAST_SUBMATCH_RESULT
$$	Outputs a literal '$'
$n	Outputs what matched the n'th sub-expression.
${n}	Outputs what matched the n'th sub-expression.
$+{NAME}	Outputs whatever matched the sub-expression named "NAME".

Any $-placeholder sequence not listed above, results in '$' being treated as a literal.

Escape Sequences

An escape character followed by any character x, outputs that character unless x is one of the escape sequences shown below.

Escape	Meaning
\a	Outputs the bell character: '\a'.
\e	Outputs the ANSI escape character (code point 27).
\f	Outputs a form feed character: '\f'
\n	Outputs a newline character: '\n'.
\r	Outputs a carriage return character: '\r'.
\t	Outputs a tab character: '\t'.
\v	Outputs a vertical tab character: '\v'.
\xDD	Outputs the character whose hexadecimal code point is 0xDD
\x{DDDD}	Outputs the character whose hexadecimal code point is 0xDDDD
\cX	Outputs the ANSI escape sequence "escape-X".
\D	If D is a decimal digit in the range 1-9, then outputs the text that matched sub-expression D.
\l	Causes the next character to be outputted, to be output in lower case.
\u	Causes the next character to be outputted, to be output in upper case.
\L	Causes all subsequent characters to be output in lower case, until a \E is found.
\U	Causes all subsequent characters to be output in upper case, until a \E is found.
\E	Terminates a \L or \U sequence.

Reference

basic_regex

Synopsis

```
#include <boost/regex.hpp>
```

The template class `basic_regex` encapsulates regular expression parsing and compilation. The class takes two template parameters:

- `charT`: determines the character type, i.e. either `char` or `wchar_t`; see charT concept.
- `traits`: determines the behavior of the character type, for example which character class names are recognized. A default traits class is provided: `regex_traits<charT>`. See also traits concept.

For ease of use there are two typedefs that define the two standard `basic_regex` instances, unless you want to use custom traits classes or non-standard character types (for example see unicode support), you won't need to use anything other than these:

```
namespace boost{

template <class charT, class traits = regex_traits<charT>  >
class basic_regex;

typedef basic_regex<char>      regex;
typedef basic_regex<wchar_t>   wregex;

}
```

The definition of `basic_regex` follows: it is based very closely on class `basic_string`, and fulfils the requirements for a constant-container of `charT`.

```
namespace boost{

template <class  charT, class traits = regex_traits<charT> >
class basic_regex {
   public:
   // types:
   typedef          charT                                value_type;
   typedef          implementation-specific              const_iterator;
   typedef          const_iterator                       iterator;
   typedef          charT&                               reference;
   typedef          const charT&                         const_reference;
   typedef          std::ptrdiff_t                       difference_type;
   typedef          std::size_t                          size_type;
   typedef          regex_constants:: syntax_option_type  flag_type;
   typedef typename traits::locale_type                  locale_type;

   // constants:
   // main option selection:
   static const regex_constants:: syntax_option_type normal
                                                = regex_constants::normal;
   static const regex_constants:: syntax_option_type ECMAScript
                                                = normal;
   static const regex_constants:: syntax_option_type JavaScript
                                                = normal;
   static const regex_constants:: syntax_option_type JScript
                                                = normal;
   static const regex_constants:: syntax_option_type basic
                                                = regex_constants::basic;
   static const regex_constants:: syntax_option_type extended
                                                = regex_constants::extended;
   static const regex_constants:: syntax_option_type awk
                                                = regex_constants::awk;
   static const regex_constants:: syntax_option_type grep
                                                = regex_constants::grep;
   static const regex_constants:: syntax_option_type egrep
                                                = regex_constants::egrep;
   static const regex_constants:: syntax_option_type sed
                                                = basic = regex_constants::sed;
   static const regex_constants:: syntax_option_type perl
                                                = regex_constants::perl;
   static const regex_constants:: syntax_option_type literal
                                                = regex_constants::literal;

   // modifiers specific to perl expressions:
   static const regex_constants:: syntax_option_type no_mod_m
                                                = regex_constants::no_mod_m;
   static const regex_constants:: syntax_option_type no_mod_s
                                                = regex_constants::no_mod_s;
   static const regex_constants:: syntax_option_type mod_s
                                                = regex_constants::mod_s;
   static const regex_constants:: syntax_option_type mod_x
                                                = regex_constants::mod_x;

   // modifiers specific to POSIX basic expressions:
   static const regex_constants:: syntax_option_type bk_plus_qm
                                                = regex_constants::bk_plus_qm;
   static const regex_constants:: syntax_option_type bk_vbar
                                                = regex_constants::bk_vbar
   static const regex_constants:: syntax_option_type no_char_classes
                                                = regex_constants::no_char_classes
   static const regex_constants:: syntax_option_type no_intervals
                                                = regex_constants::no_intervals
```

```
// common modifiers:
static const regex_constants:: syntax_option_type nosubs
                                                = regex_constants::nosubs;
static const regex_constants:: syntax_option_type optimize
                                                = regex_constants::optimize;
static const regex_constants:: syntax_option_type collate
                                                = regex_constants::collate;
static const regex_constants:: syntax_option_type newline_alt
                                                = regex_constants::newline_alt;
static const regex_constants:: syntax_option_type no_except
                                                = regex_constants::newline_alt;

// construct/copy/destroy:
explicit basic_regex ();
explicit basic_regex(const  charT* p, flag_type f = regex_constants::normal);
basic_regex(const charT* p1, const  charT* p2,
            flag_type f = regex_constants::normal);
basic_regex(const charT* p, size_type len, flag_type  f);
basic_regex(const basic_regex&);

template <class ST, class SA>
explicit basic_regex(const basic_string<charT, ST,  SA>& p,
                     flag_type f = regex_constants::normal);

template <class InputIterator>
basic_regex(InputIterator first,  InputIterator last,
            flag_type f = regex_constants::normal);

~basic_regex();
basic_regex& operator=(const basic_regex&);
basic_regex& operator= (const charT* ptr);

template <class ST, class SA>
basic_regex& operator= (const basic_string<charT, ST, SA>& p);
// iterators:
std::pair<const_iterator, const_iterator> subexpression(size_type n) const;
const_iterator begin() const;
const_iterator end() const;
// capacity:
size_type size() const;
size_type max_size() const;
bool empty() const;
size_type mark_count()const;
//
// modifiers:
basic_regex& assign(const basic_regex& that);
basic_regex& assign(const charT* ptr,
                    flag_type f = regex_constants::normal);
basic_regex& assign(const charT* ptr, unsigned int len, flag_type f);

template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                    flag_type f = regex_constants::normal);

template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
                    flag_type f = regex_constants::normal);

// const operations:
flag_type flags() const;
int status()const;
basic_string<charT> str() const;
int compare(basic_regex&) const;
```

```
   // locale:
   locale_type imbue(locale_type loc);
   locale_type getloc() const;
   // swap
   void swap(basic_regex&) throw();
};

template <class charT, class traits>
bool operator == (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator != (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator < (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator <= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator >= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator > (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);

template <class charT, class io_traits, class re_traits>
basic_ostream<charT, io_traits>&
   operator << (basic_ostream<charT, io_traits>& os,
                const basic_regex<charT, re_traits>& e);

template <class charT, class traits>
void swap(basic_regex<charT, traits>& e1,
          basic_regex<charT, traits>& e2);

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

} // namespace boost
```

Description

Class `basic_regex` has the following public members:

```
// main option selection:
static const regex_constants:: syntax_option_type normal
                                          = regex_constants::normal;
static const regex_constants:: syntax_option_type ECMAScript
                                          = normal;
static const regex_constants:: syntax_option_type JavaScript
                                          = normal;
static const regex_constants:: syntax_option_type JScript
                                          = normal;
static const regex_constants:: syntax_option_type basic
                                          = regex_constants::basic;
static const regex_constants:: syntax_option_type extended
                                          = regex_constants::extended;
static const regex_constants:: syntax_option_type awk
                                          = regex_constants::awk;
static const regex_constants:: syntax_option_type grep
                                          = regex_constants::grep;
static const regex_constants:: syntax_option_type egrep
                                          = regex_constants::egrep;
static const regex_constants:: syntax_option_type sed
                                          = regex_constants::sed;
static const regex_constants:: syntax_option_type perl
                                          = regex_constants::perl;
static const regex_constants:: syntax_option_type literal
                                          = regex_constants::literal;

// modifiers specific to perl expressions:
static const regex_constants:: syntax_option_type no_mod_m
                                          = regex_constants::no_mod_m;
static const regex_constants:: syntax_option_type no_mod_s
                                          = regex_constants::no_mod_s;
static const regex_constants:: syntax_option_type mod_s
                                          = regex_constants::mod_s;
static const regex_constants:: syntax_option_type mod_x
                                          = regex_constants::mod_x;

// modifiers specific to POSIX basic expressions:
static const regex_constants:: syntax_option_type bk_plus_qm
                                          = regex_constants::bk_plus_qm;
static const regex_constants:: syntax_option_type bk_vbar
                                          = regex_constants::bk_vbar
static const regex_constants:: syntax_option_type no_char_classes
                                          = regex_constants::no_char_classes
static const regex_constants:: syntax_option_type no_intervals
                                          = regex_constants::no_intervals

// common modifiers:
static const regex_constants:: syntax_option_type nosubs
                                          = regex_constants::nosubs;
static const regex_constants:: syntax_option_type optimize
                                          = regex_constants::optimize;
static const regex_constants:: syntax_option_type collate
                                          = regex_constants::collate;
static const regex_constants:: syntax_option_type newline_alt
                                          = regex_constants::newline_alt;
```

The meaning of these options is documented in the `syntax_option_type` section.

The static constant members are provided as synonyms for the constants declared in namespace `boost::regex_constants`; for each constant of type `syntax_option_type` declared in namespace `boost::regex_constants` then a constant with the same name, type and value is declared within the scope of basic_regex.

```
basic_regex();
```

Effects: Constructs an object of class `basic_regex`.

Table 1. basic_regex default construction postconditions

Element	Value
`empty()`	`true`
`size()`	`0`
`str()`	`basic_string<charT>()`

```
basic_regex(const charT* p, flag_type f = regex_constants::normal);
```

Requires: *p* shall not be a null pointer.

Throws: `bad_expression` if *p* is not a valid regular expression, unless the flag `no_except` is set in *f*.

Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the null-terminated string *p*, and interpreted according to the option flags specified in *f*.

Table 2. Postconditions for basic_regex construction

Element	Value
`empty()`	`false`
`size()`	`char_traits<charT>::length(p)`
`str()`	`basic_string<charT>(p)`
`flags()`	*f*
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex(const charT* p1, const charT* p2,
            flag_type f = regex_constants::normal);
```

Requires: *p1* and *p2* are not null pointers, `p1 < p2`.

Throws: bad_expression if [p1,p2) is not a valid regular expression, unless the flag `no_except` is set in *f*.

Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters [p1,p2), and interpreted according the option flags specified in *f*.

Table 3. Postconditions for basic_regex construction

Element	Value
`empty()`	`false`
`size()`	`std::distance(p1,p2)`
`str()`	`basic_string<charT>(p1,p2)`
`flags()`	*f*
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex(const charT* p, size_type len, flag_type f);
```

Requires: *p* shall not be a null pointer, `len < max_size()`.

Throws: `bad_expression` if *p* is not a valid regular expression, unless the flag `no_except` is set in *f*.

Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters [p, p+len), and interpreted according the option flags specified in *f*.

Table 4. Postconditions for basic_regex construction

Element	Value
`empty()`	`false`
`size()`	*len*
`str()`	`basic_string<charT>(p, len)`
`flags()`	*f*
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex(const basic_regex& e);
```

Effects: Constructs an object of class `basic_regex` as a copy of the object *e*.

```
template <class ST, class SA>
basic_regex(const basic_string<charT, ST, SA>& s,
            flag_type f = regex_constants::normal);
```

Throws: `bad_expression` if *s* is not a valid regular expression, unless the flag `no_except` is set in *f*.

Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the string *s*, and interpreted according to the option flags specified in *f*.

Table 5. Postconditions for basic_regex construction

Element	Value
`empty()`	`false`
`size()`	`s.size()`
`str()`	*s*
`flags()`	*f*
`mark_count()`	The number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
            flag_type f = regex_constants::normal);
```

Throws: `bad_expression` if the sequence [first, last) is not a valid regular expression, unless the flag `no_except` is set in *f*.

Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters [first, last), and interpreted according to the option flags specified in *f*.

Table 6. Postconditions for basic_regex construction

Element	Value
`empty()`	`false`
`size()`	`distance(first,last)`
`str()`	`basic_string<charT>(first,last)`
`flags()`	*f*
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex& operator=(const basic_regex& e);
```

Effects: Returns the result of `assign(e.str(), e.flags())`.

```
basic_regex& operator=(const charT* ptr);
```

Requires: *p* shall not be a null pointer.

Effects: Returns the result of `assign(ptr)`.

```
template <class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

Effects: Returns the result of `assign(p)`.

```
std::pair<const_iterator, const_iterator> subexpression(size_type n) const;
```

Effects: Returns a pair of iterators denoting the location of marked subexpression *n* within the original regular expression string. The returned iterators are relative to `begin()` and `end()`.

Requires: The expression must have been compiled with the `syntax_option_type` save_subexpression_location set. Argument *n* must be in within the range `0 <= n < mark_count()`.

```
const_iterator begin() const;
```

Effects: Returns a starting iterator to a sequence of characters representing the regular expression.

```
const_iterator end() const;
```

Effects: Returns termination iterator to a sequence of characters representing the regular expression.

```
size_type size() const;
```

Effects: Returns the length of the sequence of characters representing the regular expression.

```
size_type max_size() const;
```

Effects: Returns the maximum length of the sequence of characters representing the regular expression.

```
bool empty() const;
```

Effects: Returns true if the object does not contain a valid regular expression, otherwise false.

```
size_type mark_count() const;
```

Effects: Returns the number of marked sub-expressions within the regular expression.

```
basic_regex& assign(const basic_regex& that);
```

Effects: Returns `assign(that.str(), that.flags())`.

```
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::normal);
```

Effects: Returns `assign(string_type(ptr), f)`.

```
basic_regex& assign(const charT* ptr, unsigned int len, flag_type f);
```

Effects: Returns `assign(string_type(ptr, len), f)`.

```
template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                    flag_type f = regex_constants::normal);
```

Throws: `bad_expression` if *s* is not a valid regular expression, unless the flag `no_except` is set in *f*.

Returns: *this.

Effects: Assigns the regular expression contained in the string *s*, interpreted according the option flags specified in *f*.

Table 7. Postconditions for basic_regex::assign

Element	Value
`empty()`	`false`
`size()`	`s.size()`
`str()`	*s*
`flags()`	*f*
`mark_count()`	The number of marked sub-expressions within the expression.

```
template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
                    flag_type f = regex_constants::normal);
```

Requires: The type `InputIterator` corresponds to the Input Iterator requirements (24.1.1).

Effects: Returns `assign(string_type(first, last), f)`.

```
flag_type flags() const;
```

Effects: Returns a copy of the regular expression syntax flags that were passed to the object's constructor, or the last call to `assign`.

```
int status() const;
```

Effects: Returns zero if the expression contains a valid regular expression, otherwise an error code. This member function is retained for use in environments that cannot use exception handling.

```
basic_string<charT> str() const;
```

Effects: Returns a copy of the character sequence passed to the object's constructor, or the last call to assign.

```
int compare(basic_regex& e)const;
```

Effects: If `flags() == e.flags()` then returns `str().compare(e.str())`, otherwise returns `flags() - e.flags()`.

```
locale_type imbue(locale_type l);
```

Effects: Returns the result of `traits_inst.imbue(l)` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object. Calls to `imbue` invalidate any currently contained regular expression.

Postcondition: `empty() == true`.

```
locale_type getloc() const;
```

Effects: Returns the result of `traits_inst.getloc()` where `traits_inst` is a (default initialized) instance of the template parameter traits stored within the object.

```
void swap(basic_regex& e) throw();
```

Effects: Swaps the contents of the two regular expressions.

Postcondition: `*this` contains the regular expression that was in *e*, *e* contains the regular expression that was in `*this`.

Complexity: constant time.

Note

Comparisons between `basic_regex` objects are provided on an experimental basis: please note that these are not present in the Technical Report on C++ Library Extensions, so use with care if you are writing code that may need to be ported to other implementations of `basic_regex`.

```
template <class charT, class traits>
bool operator == (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

Effects: Returns `lhs.compare(rhs) == 0`.

```
template <class charT, class traits>
bool operator != (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

Effects: Returns `lhs.compare(rhs) != 0`.

```
template <class charT, class traits>
bool operator < (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);
```

Effects: Returns `lhs.compare(rhs) < 0`.

```
template <class charT, class traits>
bool operator <= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

Effects: Returns `lhs.compare(rhs) <= 0`.

```
template <class charT, class traits>
bool operator >= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

Effects: Returns `lhs.compare(rhs) >= 0`.

```
template <class charT, class traits>
bool operator > (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);
```

Effects: Returns `lhs.compare(rhs) > 0`.

Note

The basic_regex stream inserter is provided on an experimental basis, and outputs the textual representation of the expression to the stream.

```
template <class charT, class io_traits, class re_traits>
basic_ostream<charT, io_traits>&
   operator << (basic_ostream<charT, io_traits>& os
                const basic_regex<charT, re_traits>& e);
```

Effects: Returns `(os << e.str())`.

```
template <class charT, class traits>
void swap(basic_regex<charT, traits>& lhs,
         basic_regex<charT, traits>& rhs);
```

Effects: calls `lhs.swap(rhs)`.

match_results

Synopsis

```
#include <boost/regex.hpp>
```

Regular expressions are different from many simple pattern-matching algorithms in that as well as finding an overall match they can also produce sub-expression matches: each sub-expression being delimited in the pattern by a pair of parenthesis (...). There has to be some method for reporting sub-expression matches back to the user: this is achieved this by defining a class `match_results` that acts as an indexed collection of sub-expression matches, each sub-expression match being contained in an object of type `sub_match`.

Template class `match_results` denotes a collection of character sequences representing the result of a regular expression match. Objects of type `match_results` are passed to the algorithms `regex_match` and `regex_search`, and are returned by the iterator `regex_iterator`. Storage for the collection is allocated and freed as necessary by the member functions of class `match_results`.

The template class `match_results` conforms to the requirements of a Sequence, as specified in (lib.sequence.reqmts), except that only operations defined for const-qualified Sequences are supported.

Class template `match_results` is most commonly used as one of the typedefs `cmatch`, `wcmatch`, `smatch`, or `wsmatch`:

```
template <class BidirectionalIterator,
         class Allocator = std::allocator<sub_match<BidirectionalIterator> >
class match_results;

typedef match_results<const char*>              cmatch;
typedef match_results<const wchar_t*>           wcmatch;
typedef match_results<string::const_iterator>   smatch;
typedef match_results<wstring::const_iterator>  wsmatch;

template <class BidirectionalIterator,
         class Allocator = std::allocator<sub_match<BidirectionalIterator> >
class match_results
{
public:
   typedef          sub_match<BidirectionalIterator>                        value_type;
   typedef          const value_type&                                       const_reference;
   typedef          const_reference                                         reference;
   typedef          implementation defined                                  const_iterator;
   typedef          const_iterator                                          iterator;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
   typedef typename Allocator::size_type                                    size_type;
   typedef          Allocator                                               allocator_type;
   typedef typename iterator_traits<BidirectionalIterator>::value_type      char_type;
   typedef          basic_string<char_type>                                 string_type;

   // construct/copy/destroy:
   explicit match_results(const Allocator& a = Allocator());
   match_results(const match_results& m);
   match_results& operator=(const match_results& m);
   ~match_results();

   // size:
   size_type size() const;
   size_type max_size() const;
   bool empty() const;
   // element access:
   difference_type length(int sub = 0) const;
   difference_type length(const char_type* sub) const;
   template <class charT>
   difference_type length(const charT* sub) const;
   template <class charT, class Traits, class A>
   difference_type length(const std::basic_string<charT, Traits, A>& sub) const;
   difference_type position(unsigned int sub = 0) const;
   difference_type position(const char_type* sub) const;
   template <class charT>
   difference_type position(const charT* sub) const;
   template <class charT, class Traits, class A>
   difference_type position(const std::basic_string<charT, Traits, A>& sub) const;
   string_type str(int sub = 0) const;
   string_type str(const char_type* sub)const;
   template <class Traits, class A>
   string_type str(const std::basic_string<char_type, Traits, A>& sub)const;
   template <class charT>
   string_type str(const charT* sub)const;
   template <class charT, class Traits, class A>
   string_type str(const std::basic_string<charT, Traits, A>& sub)const;
   const_reference operator[](int n) const;
   const_reference operator[](const char_type* n) const;
   template <class Traits, class A>
   const_reference operator[](const std::basic_string<char_type, Traits, A>& n) const;
   template <class charT>
   const_reference operator[](const charT* n) const;
   template <class charT, class Traits, class A>
```

```
   const_reference operator[](const std::basic_string<charT, Traits, A>& n) const;

   const_reference prefix() const;

   const_reference suffix() const;
   const_iterator begin() const;
   const_iterator end() const;
   // format:
   template <class OutputIterator, class Formatter>
   OutputIterator format(OutputIterator out,
                         Formatter fmt,
                         match_flag_type flags = format_default) const;
   template <class Formatter>
   string_type format(Formatter fmt,
                      match_flag_type flags = format_default) const;

   allocator_type get_allocator() const;
   void swap(match_results& that);

#ifdef BOOST_REGEX_MATCH_EXTRA
   typedef typename value_type::capture_sequence_type capture_sequence_type;
   const capture_sequence_type& captures(std::size_t i)const;
#endif

};

template <class BidirectionalIterator, class Allocator>
bool operator == (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
bool operator != (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);

template <class charT, class traits, class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const match_results<BidirectionalIterator, Allocator>& m);

template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);
```

Description

In all `match_results` constructors, a copy of the Allocator argument is used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

```
match_results(const Allocator& a = Allocator());
```

Effects: Constructs an object of class `match_results`. The postconditions of this function are indicated in the table:

Element	Value
empty()	true
size()	0
str()	basic_string<charT>()

```
match_results(const match_results& m);
```

Effects: Constructs an object of class match_results, as a copy of m.

```
match_results& operator=(const match_results& m);
```

Effects: Assigns m to *this. The postconditions of this function are indicated in the table:

Element	Value
empty()	m.empty().
size()	m.size().
str(n)	m.str(n) for all integers n < m.size().
prefix()	m.prefix().
suffix()	m.suffix().
(*this)[n]	m[n] for all integers n < m.size().
length(n)	m.length(n) for all integers n < m.size().
position(n)	m.position(n) for all integers n < m.size().

```
size_type size()const;
```

Effects: Returns the number of `sub_match` elements stored in *this; that is the number of marked sub-expressions in the regular expression that was matched plus one.

```
size_type max_size()const;
```

Effects: Returns the maximum number of `sub_match` elements that can be stored in *this.

```
bool empty()const;
```

Effects: Returns size() == 0.

```
difference_type length(int sub = 0)const;
difference_type length(const char_type* sub)const;
template <class charT>
difference_type length(const charT* sub)const;
template <class charT, class Traits, class A>
difference_type length(const std::basic_string<charT, Traits, A>&)const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: Returns the length of sub-expression *sub*, that is to say: `(*this)[sub].length()`.

The overloads that accept a string refer to a named sub-expression *n*. In the event that there is no such named sub-expression then returns zero.

The template overloads of this function, allow the string and/or character type to be different from the character type of the underlying sequence and/or regular expression: in this case the characters will be widened to the underlying character type of the original regular expression. A compiler error will occur if the argument passes a wider character type than the underlying sequence. These overloads allow a normal narrow character C string literal to be used as an argument, even when the underlying character type of the expression being matched may be something more exotic such as a Unicode character type.

```
difference_type position(unsigned int sub = 0)const;
difference_type position(const char_type* sub)const;
template <class charT>
difference_type position(const charT* sub)const;
template <class charT, class Traits, class A>
difference_type position(const std::basic_string<charT, Traits, A>&)const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: Returns the starting location of sub-expression *sub*, or -1 if *sub* was not matched. Note that if this represents a partial match , then `position()` will return the location of the partial match even though `(*this)[0].matched` is false.

The overloads that accept a string refer to a named sub-expression *n*. In the event that there is no such named sub-expression then returns -1.

The template overloads of this function, allow the string and/or character type to be different from the character type of the underlying sequence and/or regular expression: in this case the characters will be widened to the underlying character type of the original regular expression. A compiler error will occur if the argument passes a wider character type than the underlying sequence. These overloads allow a normal narrow character C string literal to be used as an argument, even when the underlying character type of the expression being matched may be something more exotic such as a Unicode character type.

```
string_type str(int sub = 0)const;
string_type str(const char_type* sub)const;
template <class Traits, class A>
string_type str(const std::basic_string<char_type, Traits, A>& sub)const;
template <class charT>
string_type str(const charT* sub)const;
template <class charT, class Traits, class A>
string_type str(const std::basic_string<charT, Traits, A>& sub)const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: Returns sub-expression *sub* as a string: `string_type((*this)[sub])`.

The overloads that accept a string, return the string that matched the named sub-expression *n*. In the event that there is no such named sub-expression then returns an empty string.

The template overloads of this function, allow the string and/or character type to be different from the character type of the underlying sequence and/or regular expression: in this case the characters will be widened to the underlying character type of the original regular expression. A compiler error will occur if the argument passes a wider character type than the underlying sequence. These overloads allow a normal narrow character C string literal to be used as an argument, even when the underlying character type of the expression being matched may be something more exotic such as a Unicode character type.

```
const_reference operator[](int n) const;
const_reference operator[](const char_type* n) const;
template <class Traits, class A>
const_reference operator[](const std::basic_string<char_type, Traits, A>& n) const;
template <class charT>
const_reference operator[](const charT* n) const;
template <class charT, class Traits, class A>
const_reference operator[](const std::basic_string<charT, Traits, A>& n) const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: Returns a reference to the `sub_match` object representing the character sequence that matched marked sub-expression *n*. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression. If *n* is out of range, or if *n* is an unmatched sub-expression, then returns a `sub_match` object whose matched member is false.

The overloads that accept a string, return a reference to the `sub_match` object representing the character sequence that matched the named sub-expression *n*. In the event that there is no such named sub-expression then returns a `sub_match` object whose matched member is false.

The template overloads of this function, allow the string and/or character type to be different from the character type of the underlying sequence and/or regular expression: in this case the characters will be widened to the underlying character type of the original regular expression. A compiler error will occur if the argument passes a wider character type than the underlying sequence. These overloads allow a normal narrow character C string literal to be used as an argument, even when the underlying character type of the expression being matched may be something more exotic such as a Unicode character type.

```
const_reference prefix()const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: Returns a reference to the `sub_match` object representing the character sequence from the start of the string being matched or searched, to the start of the match found.

```
const_reference suffix()const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: Returns a reference to the `sub_match` object representing the character sequence from the end of the match found to the end of the string being matched or searched.

```
const_iterator begin()const;
```

Effects: Returns a starting iterator that enumerates over all the marked sub-expression matches stored in *this.

```
const_iterator end()const;
```

Effects: Returns a terminating iterator that enumerates over all the marked sub-expression matches stored in *this.

```
template <class OutputIterator, class Formatter>
OutputIterator format(OutputIterator out,
                      Formatter fmt,
                      match_flag_type flags = format_default);
```

Requires: The type `OutputIterator` conforms to the Output Iterator requirements (C++ std 24.1.2).

The type `Formatter` must be either a pointer to a null-terminated string of type `char_type[]`, or be a container of `char_type`'s (for example `std::basic_string<char_type>`) or be a unary, binary or ternary functor that computes the replacement string from a function call: either `fmt(*this)` which must return a container of `char_type`'s to be used as the replacement text, or either `fmt(*this, out)` or `fmt(*this, out, flags)`, both of which write the replacement text to `*out`, and then return the new OutputIterator position. Note that if the formatter is a functor, then it is *passed by value*: users that want to pass function objects with internal state might want to use Boost.Ref to wrap the object so that it's passed by reference.

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: If `fmt` is either a null-terminated string, or a container of `char_type`'s, then copies the character sequence `[fmt.begin(), fmt.end())` to `OutputIterator` *out*. For each format specifier or escape sequence in *fmt*, replace that sequence with either the character(s) it represents, or the sequence of characters within `*this` to which it refers. The bitmasks specified in flags determines what format specifiers or escape sequences are recognized, by default this is the format used by ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace.

If `fmt` is a function object, then depending on the number of arguments the function object accepts, it will either:

- Call `fmt(*this)` and copy the string returned to `OutputIterator` *out*.
- Call `fmt(*this, out)`.
- Call `fmt(*this, out, flags)`.

In all cases the new position of the `OutputIterator` is returned.

See the format syntax guide for more information.

Returns: out.

```
template <class Formatter>
string_type format(Formatter fmt,
                   match_flag_type flags = format_default);
```

Requires The type `Formatter` must be either a pointer to a null-terminated string of type `char_type[]`, or be a container of `char_type`'s (for example `std::basic_string<char_type>`) or be a unary, binary or ternary functor that computes the replacement string from a function call: either `fmt(*this)` which must return a container of `char_type`'s to be used as the replacement text, or either `fmt(*this, out)` or `fmt(*this, out, flags)`, both of which write the replacement text to `*out`, and then return the new OutputIterator position.

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: If `fmt` is either a null-terminated string, or a container of `char_type`'s, then copies the string *fmt*: For each format specifier or escape sequence in *fmt*, replace that sequence with either the character(s) it represents, or the sequence of characters within `*this` to which it refers. The bitmasks specified in flags determines what format specifiers or escape sequences are recognized, by default this is the format used by ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace.

If `fmt` is a function object, then depending on the number of arguments the function object accepts, it will either:

- Call `fmt(*this)` and return the result.
- Call `fmt(*this, unspecified-output-iterator)`, where `unspecified-output-iterator` is an unspecified OutputIterator type used to copy the output to the string result.
- Call `fmt(*this, unspecified-output-iterator, flags)`, where `unspecified-output-iterator` is an unspecified OutputIterator type used to copy the output to the string result.

See the format syntax guide for more information.

```
allocator_type get_allocator()const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

```
void swap(match_results& that);
```

Effects: Swaps the contents of the two sequences.

Postcondition: *this contains the sequence of matched sub-expressions that were in that, that contains the sequence of matched sub-expressions that were in *this.

Complexity: constant time.

```
typedef typename value_type::capture_sequence_type capture_sequence_type;
```

Defines an implementation-specific type that satisfies the requirements of a standard library Sequence (21.1.1 including the optional Table 68 operations), whose value_type is a `sub_match<BidirectionalIterator>`. This type happens to be `std::vector<sub_match<BidirectionalIterator> >`, but you shouldn't actually rely on that.

```
const capture_sequence_type& captures(std::size_t i)const;
```

Requires: that the match_results object has been initialized as a result of a successful call to `regex_search` or `regex_match` or was returned from a `regex_iterator`, and that the underlying iterators have not been subsequently invalidated. Will raise a `std::logic_error` if the match_results object was not initialized.

Effects: returns a sequence containing all the captures obtained for sub-expression i.

Returns: `(*this)[i].captures();`

Preconditions: the library must be built and used with BOOST_REGEX_MATCH_EXTRA defined, and you must pass the flag match_extra to the regex matching functions (`regex_match`, `regex_search`, `regex_iterator` or `regex_token_iterator`) in order for this member function to be defined and return useful information.

Rationale: Enabling this feature has several consequences:

- sub_match occupies more memory resulting in complex expressions running out of memory or stack space more quickly during matching.
- The matching algorithms are less efficient at handling some features (independent sub-expressions for example), even when match_extra is not used.
- The matching algorithms are much less efficient (i.e. slower), when match_extra is used. Mostly this is down to the extra memory allocations that have to take place.

```
template <class BidirectionalIterator, class Allocator>
bool operator == (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
```

Effects: Compares the two sequences for equality.

```
template <class BidirectionalIterator, class Allocator>
bool operator != (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
```

Effects: Compares the two sequences for inequality.

```
template <class charT, class traits, class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const match_results<BidirectionalIterator, Allocator>& m);
```

Effects: Writes the contents of *m* to the stream *os* as if by calling `os << m.str()`; Returns *os*.

```
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);
```

Effects: Swaps the contents of the two sequences.

sub_match

```
#include <boost/regex.hpp>
```

Regular expressions are different from many simple pattern-matching algorithms in that as well as finding an overall match they can also produce sub-expression matches: each sub-expression being delimited in the pattern by a pair of parenthesis (...). There has to be some method for reporting sub-expression matches back to the user: this is achieved this by defining a class `match_results` that acts as an indexed collection of sub-expression matches, each sub-expression match being contained in an object of type `sub_match`.

Objects of type `sub_match` may only be obtained by subscripting an object of type `match_results`.

Objects of type `sub_match` may be compared to objects of type `std::basic_string`, or `const charT*` or `const charT`.

Objects of type `sub_match` may be added to objects of type `std::basic_string`, or `const charT*` or `const charT`, to produce a new `std::basic_string` object.

When the marked sub-expression denoted by an object of type `sub_match` participated in a regular expression match then member *matched* evaluates to *true*, and members *first* and *second* denote the range of characters [first,second) which formed that match. Otherwise *matched* is *false*, and members *first* and *second* contained undefined values.

When the marked sub-expression denoted by an object of type `sub_match` was repeated, then the `sub_match` object represents the match obtained by the *last* repeat. The complete set of all the captures obtained for all the repeats, may be accessed via the captures() member function (Note: this has serious performance implications, you have to explicitly enable this feature).

If an object of type `sub_match` represents sub-expression 0 - that is to say the whole match - then member *matched* is always *true*, unless a partial match was obtained as a result of the flag `match_partial` being passed to a regular expression algorithm, in which case member *matched* is *false*, and members *first* and *second* represent the character range that formed the partial match.

```
namespace boost{

template <class BidirectionalIterator>
class sub_match;

typedef sub_match<const char*>                    csub_match;
typedef sub_match<const wchar_t*>                 wcsub_match;
typedef sub_match<std::string::const_iterator>    ssub_match;
typedef sub_match<std::wstring::const_iterator>   wssub_match;

template <class BidirectionalIterator>
class sub_match : public std::pair<BidirectionalIterator, BidirectionalIterator>
{
public:
   typedef typename iterator_traits<BidirectionalIterator>::value_type       value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type  difference_type;
   typedef          BidirectionalIterator                                    iterator;

   bool matched;

   difference_type length()const;
   operator basic_string<value_type>()const;
   basic_string<value_type> str()const;

   int compare(const sub_match& s)const;
   int compare(const basic_string<value_type>& s)const;
   int compare(const value_type* s)const;
#ifdef BOOST_REGEX_MATCH_EXTRA
   typedef implementation-private capture_sequence_type;
   const capture_sequence_type& captures()const;
#endif
};
//
// comparisons to another sub_match:
//
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);


//
// comparisons to a basic_string:
//
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);

//
// comparisons to a pointer to a character array:
//
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);

//
// comparisons to a single character:
//
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
//
// addition operators:
//
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s,
               const sub_match<BidirectionalIterator>& m);
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const sub_match<BidirectionalIterator>& m,
               const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const* s,
               const sub_match<BidirectionalIterator>& m);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const * s);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const& s,
               const sub_match<BidirectionalIterator>& m);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const& s);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m1,
               const sub_match<BidirectionalIterator>& m2);

//
// stream inserter:
//
template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const sub_match<BidirectionalIterator>& m);

} // namespace boost
```

Description

Members

```
typedef typename std::iterator_traits<iterator>::value_type value_type;
```

The type pointed to by the iterators.

```
typedef typename std::iterator_traits<iterator>::difference_type difference_type;
```

A type that represents the difference between two iterators.

```
typedef BidirectionalIterator iterator;
```

The iterator type.

```
iterator first
```

An iterator denoting the position of the start of the match.

```
iterator second
```

An iterator denoting the position of the end of the match.

```
bool matched
```

A Boolean value denoting whether this sub-expression participated in the match.

```
static difference_type length();
```

Effects: returns the length of this matched sub-expression, or 0 if this sub-expression was not matched: `matched ? distance(first, second) : 0`.

```
operator basic_string<value_type>()const;
```

Effects: converts `*this` into a string: returns `(matched ? basic_string<value_type>(first, second) : basic_string<value_type>())`.

```
basic_string<value_type> str()const;
```

Effects: returns a string representation of `*this`: `(matched ? basic_string<value_type>(first, second) : basic_string<value_type>())`.

```
int compare(const sub_match& s)const;
```

Effects: performs a lexical comparison to *s*: returns `str().compare(s.str())`.

```
int compare(const basic_string<value_type>& s)const;
```

Effects: compares `*this` to the string *s*: returns `str().compare(s)`.

```
int compare(const value_type* s)const;
```

Effects: compares `*this` to the null-terminated string *s*: returns `str().compare(s)`.

```
typedef implementation-private capture_sequence_type;
```

Defines an implementation-specific type that satisfies the requirements of a standard library Sequence (21.1.1 including the optional Table 68 operations), whose value_type is a `sub_match<BidirectionalIterator>`. This type happens to be `std::vector<sub_match<BidirectionalIterator> >`, but you shouldn't actually rely on that.

```
const capture_sequence_type& captures()const;
```

Effects: returns a sequence containing all the captures obtained for this sub-expression.

Preconditions: the library must be built and used with BOOST_REGEX_MATCH_EXTRA defined, and you must pass the flag `match_extra` to the regex matching functions (`regex_match`, `regex_search`, `regex_iterator` or `regex_token_iterator`) in order for this member #function to be defined and return useful information.

Rationale: Enabling this feature has several consequences:

- sub_match occupies more memory resulting in complex expressions running out of memory or stack space more quickly during matching.
- The matching algorithms are less efficient at handling some features (independent sub-expressions for example), even when match_extra is not used.
- The matching algorithms are much less efficient (i.e. slower), when match_extra is used. Mostly this is down to the extra memory allocations that have to take place.

sub_match non-member operators

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs.compare(rhs) == 0`.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs.compare(rhs) != 0`.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs.compare(rhs) < 0`.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs.compare(rhs) <= 0`.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs.compare(rhs) >= 0`.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs.compare(rhs) > 0`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                                          traits,
                                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs == rhs.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                                          traits,
                                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs != rhs.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs < rhs.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                                         traits,
                                                         Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs > rhs.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs >= rhs.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs <= rhs.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

Effects: returns `lhs.str() == rhs`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

Effects: returns `lhs.str() != rhs`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
```

Effects: returns `lhs.str() < rhs`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
```

Effects: returns `lhs.str() > rhs`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

Effects: returns `lhs.str() >= rhs`.

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

Effects: returns `lhs.str() <= rhs`.

```
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs == rhs.str()`.

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs != rhs.str()`.

```
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs < rhs.str()`.

```
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs > rhs.str()`.

```
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs >= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs <= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

Effects: returns `lhs.str() == rhs`.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

Effects: returns `lhs.str() != rhs`.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

Effects: returns `lhs.str() < rhs`.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

Effects: returns `lhs.str() > rhs`.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

Effects: returns `lhs.str() >= rhs`.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

Effects: returns `lhs.str() <= rhs`.

```
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs == rhs.str()`.

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs != rhs.str()`.

```
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs < rhs.str()`.

```
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs > rhs.str()`.

```
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs >= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Effects: returns `lhs <= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

Effects: returns `lhs.str() == rhs`.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

Effects: returns `lhs.str() != rhs`.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

Effects: returns `lhs.str() < rhs`.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

Effects: returns `lhs.str() > rhs`.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

Effects: returns `lhs.str() >= rhs`.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

Effects: returns `lhs.str() <= rhs`.

The addition operators for `sub_match` allow you to add a `sub_match` to any type to which you can add a `std::string` and obtain a new string as the result.

```
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s,
               const sub_match<BidirectionalIterator>& m);
```

Effects: returns `s + m.str()`.

```
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const sub_match<BidirectionalIterator>& m,
               const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s);
```

Effects: returns `m.str() + s`.

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const* s,
               const sub_match<BidirectionalIterator>& m);
```

Effects: returns `s + m.str()`.

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const * s);
```

Effects: returns `m.str() + s`.

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const& s,
               const sub_match<BidirectionalIterator>& m);
```

Effects: returns `s + m.str()`.

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const& s);
```

Effects: returns `m.str() + s`.

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m1,
               const sub_match<BidirectionalIterator>& m2);
```

Effects: returns `m1.str() + m2.str()`.

Stream inserter

```
template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os
               const sub_match<BidirectionalIterator>& m);
```

Effects: returns `(os << m.str())`.

regex_match

```
#include <boost/regex.hpp>
```

The algorithm `regex_match` determines whether a given regular expression matches **all** of a given character sequence denoted by a pair of bidirectional-iterators, the algorithm is defined as follows, the main use of this function is data input validation.

Important

Note that the result is true only if the expression matches the **whole** of the input sequence. If you want to search for an expression somewhere within the sequence then use `regex_search`. If you want to match a prefix of the character string then use `regex_search` with the flag match_continuous set.

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
               match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

Description

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

Requires: Type BidirectionalIterator meets the requirements of a Bidirectional Iterator (24.1.4).

Effects: Determines whether there is an exact match between the regular expression *e*, and all of the character sequence [first, last), parameter *flags* (see `match_flag_type`) is used to control how the expression is matched against the character sequence. Returns true if such a match exists, false otherwise.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Postconditions: If the function returns false, then the effect on parameter *m* is undefined, otherwise the effects on parameter *m* are given in the table:

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	`false`
`m.prefix().first`	`first`
`m.prefix().last`	`first`
`m.prefix().matched`	`false`
`m.suffix().first`	`last`
`m.suffix().last`	`last`
`m.suffix().matched`	`false`
`m[0].first`	`first`
`m[0].second`	`last`
`m[0].matched`	true if a full match was found, and false if it was a partial match (found as a result of the match_partial flag being set).
`m[n].first`	For all integers `n < m.size()`, the start of the sequence that matched sub-expression *n*. Alternatively, if sub-expression *n* did not participate in the match, then `last`.
`m[n].second`	For all integers `n < m.size()`, the end of the sequence that matched sub-expression *n*. Alternatively, if sub-expression *n* did not participate in the match, then `last`.
`m[n].matched`	For all integers `n < m.size()`, true if sub-expression *n* participated in the match, false otherwise.

```
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

Effects: Behaves "as if" by constructing an instance of `match_results<BidirectionalIterator> what`, and then returning the result of `regex_match(first, last, what, e, flags)`.

```
template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_match(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator,
         class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
             match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocator>& m,
               const basic_regex <charT, traits>& e,
               match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_match(s.begin(), s.end(), m, e, flags)`.

```
template <class charT, class traits>
bool regex_match(const charT* str,
               const basic_regex <charT, traits>& e,
               match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_match(str, str + char_traits<charT>::length(str), e, flags)`.

```
template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
               const basic_regex <charT, traits>& e,
               match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_match(s.begin(), s.end(), e, flags)`.

Examples

The following example processes an ftp response:

```
#include <stdlib.h>
#include <boost/regex.hpp>
#include <string>
#include <iostream>

using namespace boost;

regex expression("([0-9]+)(\\-| |$)(.*)");

// process_ftp:
// on success returns the ftp response code, and fills
// msg with the ftp response message.
int process_ftp(const char* response, std::string* msg)
{
   cmatch what;
   if(regex_match(response, what, expression))
   {
      // what[0] contains the whole string
      // what[1] contains the response code
      // what[2] contains the separator character
      // what[3] contains the text message.
      if(msg)
         msg->assign(what[3].first, what[3].second);
      return std::atoi(what[1].first);
   }
   // failure did not match
   if(msg)
      msg->erase();
   return -1;
}
```

regex_search

```
#include <boost/regex.hpp>
```

The algorithm regex_search will search a range denoted by a pair of bidirectional-iterators for a given regular expression. The algorithm uses various heuristics to reduce the search time by only checking for a match if a match could conceivably start at that position. The algorithm is defined as follows:

```
template <class BidirectionalIterator,
         class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template <class ST, class SA,
         class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<
                     typename basic_string<charT, ST,SA>::const_iterator,
                     Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template<class charT, class Allocator, class traits>
bool regex_search(const charT* str,
                  match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template <class charT, class traits>
bool regex_search(const charT* str,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template<class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Description

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Requires: Type BidirectionalIterator meets the requirements of a Bidirectional Iterator (24.1.4).

Effects: Determines whether there is some sub-sequence within [first,last) that matches the regular expression *e*, parameter *flags* is used to control how the expression is matched against the character sequence. Returns true if such a sequence exists, false otherwise.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Postconditions: If the function returns false, then the effect on parameter *m* is undefined, otherwise the effects on parameter *m* are given in the table:

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	`false`
`m.prefix().first`	`first`
`m.prefix().last`	`m[0].first`
`m.prefix().matched`	`m.prefix().first != m.prefix().second`
`m.suffix().first`	`m[0].second`
`m.suffix().last`	`last`
`m.suffix().matched`	`m.suffix().first != m.suffix().second`
`m[0].first`	The start of the sequence of characters that matched the regular expression
`m[0].second`	The end of the sequence of characters that matched the regular expression
`m[0].matched`	true if a full match was found, and false if it was a partial match (found as a result of the match_partial flag being set).
`m[n].first`	For all integers `n < m.size()`, the start of the sequence that matched sub-expression *n*. Alternatively, if sub-expression *n* did not participate in the match, then last.
`m[n].second`	For all integers `n < m.size()`, the end of the sequence that matched sub-expression *n*. Alternatively, if sub-expression *n* did not participate in the match, then `last`.
`m[n].matched`	For all integers `n < m.size()`, true if sub-expression *n* participated in the match, false otherwise.

```
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_search(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator, class charT,
         class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocat↵
or>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_search(s.begin(), s.end(), m, e, flags)`.

```
template <class iterator, class charT, class traits>
bool regex_search(iterator first, iterator last,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Effects: Behaves "as if" by constructing an instance of `match_results<BidirectionalIterator> what`, and then returning the result of `regex_search(first, last, what, e, flags)`.

```
template <class charT, class traits>
bool regex_search(const charT* str
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_search(str, str + char_traits<charT>::length(str), e, flags)`.

```
template <class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

Effects: Returns the result of `regex_search(s.begin(), s.end(), e, flags)`.

Examples

The following example, takes the contents of a file in the form of a string, and searches for all the C++ class declarations in the file. The code will work regardless of the way that `std::string` is implemented, for example it could easily be modified to work with the SGI rope class, which uses a non-contiguous storage strategy.

```
#include <string>
#include <map>
#include <boost/regex.hpp>

// purpose:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's
typedef std::map<std::string, int, std::less<std::string> > map_type;

boost::regex expression(
   "^(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   "(class|struct)[[:space:]]*"
   "(\\<\\w+\\>([[:blank:]]*\\([^)]*\\))?"
   "[[:space:]]*)*(\\<\\w*\\>)[[:space:]]*"
   "(<[^;:{]+>[[:space:]]*)?(\\{|:[^;\\{()]*\\{)");

void IndexClasses(map_type& m, const std::string& file)
{
   std::string::const_iterator start, end;
   start = file.begin();
   end = file.end();
      boost::match_results<std::string::const_iterator> what;
   boost::match_flag_type flags = boost::match_default;
   while(regex_search(start, end, what, expression, flags))
   {
      // what[0] contains the whole string
      // what[5] contains the class name.
      // what[6] contains the template specialisation if any.
      // add class name and position to map:
      m[std::string(what[5].first, what[5].second)
            + std::string(what[6].first, what[6].second)]
         = what[5].first - file.begin();
      // update search position:
      start = what[0].second;
      // update flags:
      flags |= boost::match_prev_avail;
      flags |= boost::match_not_bob;
   }
}
```

regex_replace

```
#include <boost/regex.hpp>
```

The algorithm `regex_replace` searches through a string finding all the matches to the regular expression: for each match it then calls `match_results<>::format` to format the string and sends the result to the output iterator. Sections of text that do not match are copied to the output unchanged only if the *flags* parameter does not have the flag `format_no_copy` set. If the flag `format_first_only` is set then only the first occurrence is replaced rather than all occurrences.

```
template <class OutputIterator, class BidirectionalIterator, class traits, class Formatter>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
                             Formatter fmt,
                             match_flag_type flags = match_default);

template <class traits, class Formatter>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                                  const basic_regex<charT, traits>& e,
                                  Formatter fmt,
                                  match_flag_type flags = match_default);
```

Description

```
template <class OutputIterator, class BidirectionalIterator, class traits, class Formatter>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
                             Formatter fmt,
                             match_flag_type flags = match_default);
```

Enumerates all the occurrences of expression *e* in the sequence [first, last), replacing each occurrence with the string that results by merging the match found with the format string *fmt*, and copies the resulting string to *out*. In the case that *fmt* is a unary, binary or ternary function object, then the character sequence generated by that object is copied unchanged to the output when performing a substitution.

If the flag `format_no_copy` is set in *flags* then unmatched sections of text are not copied to output.

If the flag `format_first_only` is set in flags then only the first occurrence of *e* is replaced.

The manner in which the format string *fmt* is interpreted, along with the rules used for finding matches, are determined by the flags set in *flags*: see `match_flag_type`.

Requires The type `Formatter` must be either a pointer to a null-terminated string of type `char_type[]`, or be a container of `char_type`'s (for example `std::basic_string<char_type>`) or be a unary, binary or ternary functor that computes the replacement string from a function call: either `fmt(what)` which must return a container of `char_type`'s to be used as the replacement text, or either `fmt(what, out)` or `fmt(what, out, flags)`, both of which write the replacement text to `*out`, and then return the new OutputIterator position. In each case `what` is the `match_results` object that represents the match found. Note that if the formatter is a functor, then it is *passed by value*: users that want to pass function objects with internal state might want to use Boost.Ref to wrap the object so that it's passed by reference.

Effects: Constructs an `regex_iterator` object:

```
regex_iterator<BidirectionalIterator, charT, traits, Allocator>
                                        i(first, last, e, flags),
```

and uses *i* to enumerate through all of the matches *m* of type `match_results <BidirectionalIterator>` that occur within the sequence [first, last).

If no such matches are found and

```
!(flags & format_no_copy)
```

then calls

```
std::copy(first, last, out).
```

Otherwise, for each match found, if

```
!(flags & format_no_copy)
```

calls

```
std::copy(m.prefix().first, m.prefix().last, out),
```

and then calls

```
m.format(out, fmt, flags).
```

Finally if

```
!(flags & format_no_copy)
```

calls

```
std::copy(last_m.suffix().first, last_m,suffix().last, out)
```

where *last_m* is a copy of the last match found.

If `flags & format_first_only` is non-zero then only the first match found is replaced.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Returns: out.

```
template <class traits, class Formatter>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                                  const basic_regex<charT, traits>& e,
                                  Formatter fmt,
                                  match_flag_type flags = match_default);
```

Requires The type `Formatter` must be either a pointer to a null-terminated string of type `char_type[]`, or be a container of `char_type`'s (for example `std::basic_string<char_type>`) or be a unary, binary or ternary functor that computes the replacement string from a function call: either `fmt(what)` which must return a container of `char_type`'s to be used as the replacement text, or either `fmt(what, out)` or `fmt(what, out, flags)`, both of which write the replacement text to `*out`, and then return the new OutputIterator position. In each case `what` is the `match_results` object that represents the match found.

Effects: Constructs an object `basic_string<charT> result`, calls `regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)`, and then returns `result`.

Examples

The following example takes C/C++ source code as input, and outputs syntax highlighted HTML code.

```
#include <fstream>
#include <sstream>
#include <string>
#include <iterator>
#include <boost/regex.hpp>
#include <fstream>
#include <iostream>

// purpose:
// takes the contents of a file and transform to
// syntax highlighted code in html format

boost::regex e1, e2;
extern const char* expression_text;
extern const char* format_string;
extern const char* pre_expression;
extern const char* pre_format;
extern const char* header_text;
extern const char* footer_text;

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, const char** argv)
{
   try{
   e1.assign(expression_text);
   e2.assign(pre_expression);
   for(int i = 1; i < argc; ++i)
   {
      std::cout << "Processing file " << argv[i] << std::endl;
      std::ifstream fs(argv[i]);
      std::string in;
      load_file(in, fs);
      std::string out_name(std::string(argv[i]) + std::string(".htm"));
      std::ofstream os(out_name.c_str());
      os << header_text;
      // strip '<' and '>' first by outputting to a
      // temporary string stream
      std::ostringstream t(std::ios::out | std::ios::binary);
      std::ostream_iterator<char, char> oi(t);
      boost::regex_replace(oi, in.begin(), in.end(),
      e2, pre_format, boost::match_default | boost::format_all);
      // then output to final output stream
      // adding syntax highlighting:
      std::string s(t.str());
      std::ostream_iterator<char, char> out(os);
      boost::regex_replace(out, s.begin(), s.end(),
      e1, format_string, boost::match_default | boost::format_all);
      os << footer_text;
   }
   }
   catch(...)
```

```
   { return -1; }
   return 0;
}

extern const char* pre_expression = "(<)|(>)|(&)|\\r";
extern const char* pre_format = "(?1&lt;)(?2&gt;)(?3&amp;)";


const char* expression_text =
   // preprocessor directives: index 1
   "(^[[:blank:]]*#(?:[^\\\\\\n]|\\\\[^\\n[:punct:][:word:]]*[\\n[:punct:][:word:]])*)|"
   // comment: index 2
   "(//[^\\n]*|/\\*.*?\\*/)|"
   // literals: index 3
   "\\<([+-]?(?:(?:0x[[:xdigit:]]+)|(?:(?:[[:digit:]]*\\.)?[[:digit:]]+"
   "(?:[eE][+-]?[[:digit:]]+)?))u?(?:(?:int(?:8|16|32|64))|L)?)\\>|"
   // string literals: index 4
   "('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
   // keywords: index 5
   "\\<(__asm|__cdecl|__declspec|__export|__far16|__fastcall|__fortran|__import"
   "|__pascal|__rtti|__stdcall|_asm|_cdecl|__except|_export|_far16|_fastcall"
   "|__finally|_fortran|_import|_pascal|_stdcall|__thread|__try|asm|auto|bool"
   "|break|case|catch|cdecl|char|class|const|const_cast|continue|default|delete"
   "|do|double|dynamic_cast|else|enum|explicit|extern|false|float|for|friend|goto"
   "|if|inline|int|long|mutable|namespace|new|operator|pascal|private|protected"
   "|public|register|reinterpret_cast|return|short|signed|sizeof|static|static_cast"
   "|struct|switch|template|this|throw|true|try|typedef|typeid|typename|union|unsigned"
   "|using|virtual|void|volatile|wchar_t|while)\\>"
   ;

const char* format_string = "(?1<font color=\"#008040\">$&</font>)"
                            "(?2<I><font color=\"#000080\">$&</font></I>)"
                            "(?3<font color=\"#0000A0\">$&</font>)"
                            "(?4<font color=\"#0000FF\">$&</font>)"
                            "(?5<B>$&</B>)";

const char* header_text =
   "<HTML>\n<HEAD>\n"
   "<TITLE>Auto-generated html formatted source</TITLE>\n"
   "<META HTTP-EQUIV=\"Content-Type\" CONTENT=\"text/html; charset=windows-1252\">\n"
   "</HEAD>\n"
   "<BODY LINK=\"#0000ff\" VLINK=\"#800080\" BGCOLOR=\"#ffffff\">\n"
   "<P> </P>\n<PRE>";

const char* footer_text = "</PRE>\n</BODY>\n\n";
```

regex_iterator

The iterator type `regex_iterator` will enumerate all of the regular expression matches found in some sequence: dereferencing a `regex_iterator` yields a reference to a `match_results` object.

```
template <class BidirectionalIterator,
         class charT = iterator_traits<BidirectionalIterator>::value_type,
         class traits = regex_traits<charT> >
class regex_iterator
{
public:
   typedef          basic_regex<charT, traits>                              regex_type;
   typedef          match_results<BidirectionalIterator>                    value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
   typedef          const value_type*                                       pointer;
   typedef          const value_type&                                       reference;
   typedef          std::forward_iterator_tag                               iterator_category;

   regex_iterator();
   regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                  const regex_type& re,
                  match_flag_type m = match_default);
   regex_iterator(const regex_iterator&);
   regex_iterator& operator=(const regex_iterator&);
   bool operator==(const regex_iterator&)const;
   bool operator!=(const regex_iterator&)const;
   const value_type& operator*()const;
   const value_type* operator->()const;
   regex_iterator& operator++();
   regex_iterator operator++(int);
};

typedef regex_iterator<const char*>                 cregex_iterator;
typedef regex_iterator<std::string::const_iterator>  sregex_iterator;

#ifndef  BOOST_NO_WREGEX
typedef regex_iterator<const wchar_t*>              wcregex_iterator;
typedef regex_iterator<std::wstring::const_iterator> wsregex_iterator;
#endif

template <class charT, class traits> regex_iterator<const charT*, charT, traits>
   make_regex_iterator(const charT* p, const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
   regex_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
      make_regex_iterator(const std::basic_string<charT, ST, SA>& p,
                          const basic_regex<charT, traits>& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);
```

Description

A `regex_iterator` is constructed from a pair of iterators, and enumerates all occurrences of a regular expression within that iterator range.

```
regex_iterator();
```

Effects: constructs an end of sequence `regex_iterator`.

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               match_flag_type m = match_default);
```

Effects: constructs a `regex_iterator` that will enumerate all occurrences of the expression *re*, within the sequence [a,b), and found using `match_flag_type` *m*. The object *re* must exist for the lifetime of the `regex_iterator`.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

```
regex_iterator(const regex_iterator& that);
```

Effects: constructs a copy of `that`.

Postconditions: `*this == that`.

```
regex_iterator& operator=(const regex_iterator&);
```

Effects: sets `*this` equal to those in `that`.

Postconditions: *this == that.

```
bool operator==(const regex_iterator& that)const;
```

Effects: returns true if *this is equal to that.

```
bool operator!=(const regex_iterator&)const;
```

Effects: returns `!(*this == that)`.

```
const value_type& operator*()const;
```

Effects: dereferencing a `regex_iterator` object it yields a const reference to a `match_results` object, whose members are set as follows:

Element	Value
`(*it).size()`	`1 + re.mark_count()`
`(*it).empty()`	`false`
`(*it).prefix().first`	The end of the last match found, or the start of the underlying sequence if this is the first match enumerated
`(*it).prefix().last`	The same as the start of the match found: `(*it)[0].first`
`(*it).prefix().matched`	True if the prefix did not match an empty string: `(*it).prefix().first != (*it).prefix().second`
`(*it).suffix().first`	The same as the end of the match found: `(*it)[0].second`
`(*it).suffix().last`	The end of the underlying sequence.
`(*it).suffix().matched`	True if the suffix did not match an empty string: `(*it).suffix().first != (*it).suffix().second`
`(*it)[0].first`	The start of the sequence of characters that matched the regular expression
`(*it)[0].second`	The end of the sequence of characters that matched the regular expression
`(*it)[0].matched`	true if a full match was found, and false if it was a partial match (found as a result of the match_partial flag being set).
`(*it)[n].first`	For all integers `n < (*it).size()`, the start of the sequence that matched sub-expression *n*. Alternatively, if sub-expression *n* did not participate in the match, then last.
`(*it)[n].second`	For all integers `n < (*it).size()`, the end of the sequence that matched sub-expression *n*. Alternatively, if sub-expression *n* did not participate in the match, then last.
`(*it)[n].matched`	For all integers `n < (*it).size()`, true if sub-expression *n* participated in the match, false otherwise.
`(*it).position(n)`	For all integers `n < (*it).size()`, then the distance from the start of the underlying sequence to the start of sub-expression match *n*.

```
const value_type* operator->()const;
```

Effects: returns `&(*this)`.

```
regex_iterator& operator++();
```

Effects: moves the iterator to the next match in the underlying sequence, or the end of sequence iterator if none if found. When the last match found matched a zero length string, then the `regex_iterator` will find the next match as follows: if there exists a non-zero length match that starts at the same location as the last one, then returns it, otherwise starts looking for the next (possibly zero length) match from one position to the right of the last match.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Returns: *this.

```
regex_iterator operator++(int);
```

Effects: constructs a copy result of `*this`, then calls `++(*this)`.

Returns: result.

```
template <class charT, class traits>
regex_iterator<const charT*, charT, traits>
   make_regex_iterator(const charT* p, const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_iterator(const std::basic_string<charT, ST, SA>& p,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);
```

Effects: returns an iterator that enumerates all occurrences of expression *e* in text *p* using `match_flag_type` *m*.

Examples

The following example takes a C++ source file and builds up an index of class names, and the location of that class in the file.

```
#include <string>
#include <map>
#include <fstream>
#include <iostream>
#include <boost/regex.hpp>

using namespace std;

// purpose:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's

typedef std::map<std::string, std::string::difference_type, std::less<std::string> > map_type;

const char* re =
   // possibly leading whitespace:
   "^[[:space:]]*"
   // possible template declaration:
   "(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   // class or struct:
   "(class|struct)[[:space:]]*"
   // leading declspec macros etc:
   "("
      "\\<\\w+\\>"
      "("
         "[[:blank:]]*\\([^)]*\\)"
      ")?"
      "[[:space:]]*"
   ")*"
   // the class name
   "(\\<\\w*\\>)[[:space:]]*"
   // template specialisation parameters
   "(<[^;:{]+>)?[[:space:]]*"
   // terminate in { or :
   "(\\{|:[^;\\{()]*\\{)";


boost::regex expression(re);
map_type class_index;

bool regex_callback(const boost::match_results<std::string::const_iterator>& what)
{
   // what[0] contains the whole string
   // what[5] contains the class name.
   // what[6] contains the template specialisation if any.
   // add class name and position to map:
   class_index[what[5].str() + what[6].str()] = what.position(5);
   return true;
}

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}
```

```
int main(int argc, const char** argv)
{
   std::string text;
   for(int i = 1; i < argc; ++i)
   {
      cout << "Processing file " << argv[i] << endl;
      std::ifstream fs(argv[i]);
      load_file(text, fs);
      // construct our iterators:
      boost::sregex_iterator m1(text.begin(), text.end(), expression);
      boost::sregex_iterator m2;
      std::for_each(m1, m2, &regex_callback);
      // copy results:
      cout << class_index.size() << " matches found" << endl;
      map_type::iterator c, d;
      c = class_index.begin();
      d = class_index.end();
      while(c != d)
      {
         cout << "class \"" << (*c).first << "\" found at index: " << (*c).second << endl;
         ++c;
      }
      class_index.erase(class_index.begin(), class_index.end());
   }
   return 0;
}
```

regex_token_iterator

The template class `regex_token_iterator` is an iterator adapter; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more character sequence for each match found. Each position enumerated by the iterator is a `sub_match` object that represents what matched a particular sub-expression within the regular expression. When class `regex_token_iterator` is used to enumerate a single sub-expression with index -1, then the iterator performs field splitting: that is to say it enumerates one character sequence for each section of the character container sequence that does not match the regular expression specified.

```
template <class BidirectionalIterator,
         class charT = iterator_traits<BidirectionalIterator>::value_type,
         class traits = regex_traits<charT> >
class regex_token_iterator
{
public:
   typedef          basic_regex<charT, traits>                              regex_type;
   typedef          sub_match<BidirectionalIterator>                        value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
   typedef          const value_type*                                       pointer;
   typedef          const value_type&                                       reference;
   typedef          std::forward_iterator_tag                               iterator_category;

   regex_token_iterator();
   regex_token_iterator(BidirectionalIterator a,
                        BidirectionalIterator b,
                        const regex_type& re,
                        int submatch = 0,
                        match_flag_type m = match_default);
   regex_token_iterator(BidirectionalIterator a,
                        BidirectionalIterator b,
                        const regex_type& re,
                        const std::vector<int>& submatches,
                        match_flag_type m = match_default);
   template <std::size_t N>
   regex_token_iterator(BidirectionalIterator a,
                        BidirectionalIterator b,
                        const regex_type& re,
                        const int (&submatches)[N],
                        match_flag_type m = match_default);
   regex_token_iterator(const regex_token_iterator&);
   regex_token_iterator& operator=(const regex_token_iterator&);
   bool operator==(const regex_token_iterator&)const;
   bool operator!=(const regex_token_iterator&)const;
   const value_type& operator*()const;
   const value_type* operator->()const;
   regex_token_iterator& operator++();
   regex_token_iterator operator++(int);
};

typedef regex_token_iterator<const char*>                    cregex_token_iterator;
typedef regex_token_iterator<std::string::const_iterator>    sregex_token_iterator;
#ifndef BOOST_NO_WREGEX
typedef regex_token_iterator<const wchar_t*>                 wcregex_token_iterator;
typedef regex_token_iterator<<std::wstring::const_iterator> wsregex_token_iterator;
#endif

template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

```
template <class charT, class traits, std::size_t N>
regex_token_iterator<const charT*, charT, traits>
make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA, std::size_t N>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<
      typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

Description

```
regex_token_iterator();
```

Effects: constructs an end of sequence iterator.

```
regex_token_iterator(BidirectionalIterator a,
                     BidirectionalIterator b,
                     const regex_type& re,
                     int submatch = 0,
                     match_flag_type m = match_default);
```

Preconditions: `!re.empty()`. Object *re* shall exist for the lifetime of the iterator constructed from it.

Effects: constructs a `regex_token_iterator` that will enumerate one string for each regular expression match of the expression *re* found within the sequence [a,b), using match flags *m* (see `match_flag_type`). The string enumerated is the sub-expression *submatch* for each match found; if *submatch* is -1, then enumerates all the text sequences that did not match the expression *re* (that is to performs field splitting).

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

```
regex_token_iterator(BidirectionalIterator a,
                     BidirectionalIterator b,
                     const regex_type& re,
                     const std::vector<int>& submatches,
                     match_flag_type m = match_default);
```

Preconditions: `submatches.size() && !re.empty()`. Object *re* shall exist for the lifetime of the iterator constructed from it.

Effects: constructs a `regex_token_iterator` that will enumerate `submatches.size()` strings for each regular expression match of the expression *re* found within the sequence [a,b), using match flags *m* (see `match_flag_type`). For each match found one string will be enumerated for each sub-expression index contained within submatches vector; if `submatches[0]` is -1, then the first string enumerated for each match will be all of the text from end of the last match to the start of the current match, in addition there will be one extra string enumerated when no more matches can be found: from the end of the last match found, to the end of the underlying sequence.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

```
template <std::size_t N>
regex_token_iterator(BidirectionalIterator a,
                     BidirectionalIterator b,
                     const regex_type& re,
                     const int (&submatches)[R],
                     match_flag_type m = match_default);
```

Preconditions: `!re.empty()`. Object *re* shall exist for the lifetime of the iterator constructed from it.

Effects: constructs a `regex_token_iterator` that will enumerate *R* strings for each regular expression match of the expression *re* found within the sequence [a,b), using match flags *m* (see `match_flag_type`). For each match found one string will be enumerated for each sub-expression index contained within the *submatches* array; if `submatches[0]` is -1, then the first string enumerated for each match will be all of the text from end of the last match to the start of the current match, in addition there will be one extra string enumerated when no more matches can be found: from the end of the last match found, to the end of the underlying sequence.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

```
regex_token_iterator(const regex_token_iterator& that);
```

Effects: constructs a copy of `that`.

Postconditions: `*this == that`.

```
regex_token_iterator& operator=(const regex_token_iterator& that);
```

Effects: sets `*this` to be equal to `that`.

Postconditions: `*this == that`.

```
bool operator==(const regex_token_iterator&)const;
```

Effects: returns true if `*this` is the same position as `that`.

```
bool operator!=(const regex_token_iterator&)const;
```

Effects: returns `!(*this == that)`.

```
const value_type& operator*()const;
```

Effects: returns the current character sequence being enumerated.

```
const value_type* operator->()const;
```

Effects: returns `&(*this)`.

```
regex_token_iterator& operator++();
```

Effects: Moves on to the next character sequence to be enumerated.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Returns: `*this`.

```
regex_token_iterator& operator++(int);
```

Effects: constructs a copy result of `*this`, then calls `++(*this)`.

Returns: result.

```
template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, std::size_t N>
regex_token_iterator<const charT*, charT, traits>
make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA, std::size_t N>
regex_token_iterator<
      typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<
      typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

Effects: returns a `regex_token_iterator` that enumerates one `sub_match` for each value in *submatch* for each occurrence of regular expression *e* in string *p*, matched using `match_flag_type` *m*.

Examples

The following example takes a string and splits it into a series of tokens:

```
#include <iostream>
#include <boost/regex.hpp>

using namespace std;

int main(int argc)
{
   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";

      boost::regex re("\\s+");
      boost::sregex_token_iterator i(s.begin(), s.end(), re, -1);
      boost::sregex_token_iterator j;

      unsigned count = 0;
      while(i != j)
      {
         cout << *i++ << endl;
         count++;
      }
      cout << "There were " << count << " tokens found." << endl;

   }while(argc == 1);
   return 0;
}
```

The following example takes a html file and outputs a list of all the linked files:

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <boost/regex.hpp>

boost::regex e("<\\s*A\\s+[^>]*href\\s*=\\s*\"([^\"]*)\"",
               boost::regex::normal | boost::regbase::icase);

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   //
   // attempt to grow string buffer to match file size,
   // this doesn't always work...
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
      // use logarithmic growth strategy, in case
      // in_avail (above) returned zero:
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, char** argv)
{
   std::string s;
   int i;
   for(i = 1; i < argc; ++i)
   {
      std::cout << "Findings URL's in " << argv[i] << ":" << std::endl;
      s.erase();
      std::ifstream is(argv[i]);
      load_file(s, is);
      boost::sregex_token_iterator i(s.begin(), s.end(), e, 1);
      boost::sregex_token_iterator j;
      while(i != j)
      {
         std::cout << *i++ << std::endl;
      }
   }
   //
   // alternative method:
   // test the array-literal constructor, and split out the whole
   // match as well as $1....
   //
   for(i = 1; i < argc; ++i)
   {
      std::cout << "Findings URL's in " << argv[i] << ":" << std::endl;
      s.erase();
      std::ifstream is(argv[i]);
      load_file(s, is);
      const int subs[] = {1, 0,};
      boost::sregex_token_iterator i(s.begin(), s.end(), e, subs);
      boost::sregex_token_iterator j;
      while(i != j)
      {
```

```
         std::cout << *i++ << std::endl;
      }
   }

   return 0;
}
```

bad_expression

Synopsis

```
#include <boost/pattern_except.hpp>
```

The class `regex_error` defines the type of objects thrown as exceptions to report errors during the conversion from a string representing a regular expression to a finite state machine.

```
namespace boost{

class regex_error : public std::runtime_error
{
public:
   explicit regex_error(const std::string& s, regex_constants::error_type err, std::ptrdiff_t pos);
    explicit regex_error(boost::regex_constants::error_type err);
    boost::regex_constants::error_type code()const;
    std::ptrdiff_t position()const;
};

typedef regex_error bad_pattern; // for backwards compatibility
typedef regex_error bad_expression; // for backwards compatibility

} // namespace boost
```

Description

```
regex_error(const std::string& s, regex_constants::error_type err, std::ptrdiff_t pos);
regex_error(boost::regex_constants::error_type err);
```

Effects: Constructs an object of class regex_error.

```
boost::regex_constants::error_type code()const;
```

Effects: returns the error code that represents parsing error that occurred.

```
std::ptrdiff_t position()const;
```

Effects: returns the location in the expression where parsing stopped.

Footnotes: the choice of `std::runtime_error` as the base class for `regex_error` is moot; depending upon how the library is used exceptions may be either logic errors (programmer supplied expressions) or run time errors (user supplied expressions). The library previously used `bad_pattern` and `bad_expression` for errors, these have been replaced by the single class `regex_error` to keep the library in synchronization with the Technical Report on C++ Library Extensions.

syntax_option_type

syntax_option_type Synopsis

Type `syntax_option_type` is an implementation specific bitmask type that controls how a regular expression string is to be interpreted. For convenience note that all the constants listed here, are also duplicated within the scope of class template `basic_regex`.

```
namespace std{ namespace regex_constants{

typedef implementation-specific-bitmask-type syntax_option_type;

// these flags are standardized:
static const syntax_option_type normal;
static const syntax_option_type ECMAScript = normal;
static const syntax_option_type JavaScript = normal;
static const syntax_option_type JScript = normal;
static const syntax_option_type perl = normal;
static const syntax_option_type basic;
static const syntax_option_type sed = basic;
static const syntax_option_type extended;
static const syntax_option_type awk;
static const syntax_option_type grep;
static const syntax_option_type egrep;
static const syntax_option_type icase;
static const syntax_option_type nosubs;
static const syntax_option_type optimize;
static const syntax_option_type collate;

//
// The remaining options are specific to Boost.Regex:
//

// Options common to both Perl and POSIX regular expressions:
static const syntax_option_type newline_alt;
static const syntax_option_type no_except;
static const syntax_option_type  save_subexpression_location;

// Perl specific options:
static const syntax_option_type no_mod_m;
static const syntax_option_type no_mod_s;
static const syntax_option_type mod_s;
static const syntax_option_type mod_x;
static const syntax_option_type no_empty_expressions;

// POSIX extended specific options:
static const syntax_option_type no_escape_in_lists;
static const syntax_option_type no_bk_refs;

// POSIX basic specific options:
static const syntax_option_type no_escape_in_lists;
static const syntax_option_type no_char_classes;
static const syntax_option_type no_intervals;
static const syntax_option_type bk_plus_qm;
static const syntax_option_type bk_vbar;

} // namespace regex_constants
} // namespace std
```

Overview of syntax_option_type

The type `syntax_option_type` is an implementation specific bitmask type (see C++ standard 17.3.2.1.2). Setting its elements has the effects listed in the table below, a valid value of type `syntax_option_type` will always have exactly one of the elements `normal`, `basic`, `extended`, `awk`, `grep`, `egrep`, `sed`, `literal` or `perl` set.

Note that for convenience all the constants listed here are duplicated within the scope of class template `basic_regex`, so you can use any of:

```
boost::regex_constants::constant_name
```

or

```
boost::regex::constant_name
```

or

```
boost::wregex::constant_name
```

in an interchangeable manner.

Options for Perl Regular Expressions

One of the following must always be set for perl regular expressions:

Element	Standardized	Effect when set
ECMAScript	Yes	Specifies that the grammar recognized by the regular expression engine uses its normal semantics: that is the same as that given in the ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1). This is functionally identical to the Perl regular expression syntax. Boost.Regex also recognizes all of the perl-compatible `(?...)` extensions in this mode.
perl	No	As above.
normal	No	As above.
JavaScript	No	As above.
JScript	No	As above.

The following options may also be set when using perl-style regular expressions:

Element	Standardized	Effect when set
icase	Yes	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
nosubs	Yes	Specifies that when a regular expression is matched against a character container sequence, then no sub-expression matches are to be stored in the supplied `match_results` structure.
optimize	Yes	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output. This currently has no effect for Boost.Regex.
collate	Yes	Specifies that character ranges of the form `[a-b]` should be locale sensitive.
newline_alt	No	Specifies that the \n character has the same effect as the alternation operator |. Allows newline separated lists to be used as a list of alternatives.
no_except	No	Prevents `basic_regex` from throwing an exception when an invalid expression is encountered.
no_mod_m	No	Normally Boost.Regex behaves as if the Perl m-modifier is on: so the assertions ^ and $ match after and before embedded newlines respectively, setting this flags is equivalent to prefixing the expression with (?-m).
no_mod_s	No	Normally whether Boost.Regex will match "." against a newline character is determined by the match flag `match_dot_not_newline`. Specifying this flag is equivalent to prefixing the expression with `(?-s)` and therefore causes "." not to match a newline character regardless of whether `match_not_dot_newline` is set in the match flags.

Element	Standardized	Effect when set
mod_s	No	Normally whether Boost.Regex will match "." against a newline character is determined by the match flag `match_dot_not_newline`. Specifying this flag is equivalent to prefixing the expression with `(?s)` and therefore causes "." to match a newline character regardless of whether `match_not_dot_newline` is set in the match flags.
mod_x	No	Turns on the perl x-modifier: causes unescaped whitespace in the expression to be ignored.
no_empty_expressions	No	When set then empty expressions/alternatives are prohibited.
save_subexpression_location	No	When set then the locations of individual sub-expressions within the *original regular expression string* can be accessed via the `subexpression()` member function of `basic_regex`.

Options for POSIX Extended Regular Expressions

Exactly one of the following must always be set for POSIX extended regular expressions:

Element	Standardized	Effect when set
extended	Yes	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX extended regular expressions in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1). Refer to the POSIX extended regular expression guide for more information. In addition some perl-style escape sequences are supported (The POSIX standard specifies that only "special" characters may be escaped, all other escape sequences result in undefined behavior).
egrep	Yes	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility grep when given the -E option in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep (FWD.1). That is to say, the same as POSIX extended syntax, but with the newline character acting as an alternation character in addition to "|".
awk	Yes	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility awk in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, awk (FWD.1). That is to say: the same as POSIX extended syntax, but with escape sequences in character classes permitted. In addition some perl-style escape sequences are supported (actually the awk syntax only requires \a \b \t \v \f \n and \r to be recognised, all other Perl-style escape sequences invoke undefined behavior according to the POSIX standard, but are in fact recognised by Boost.Regex).

The following options may also be set when using POSIX extended regular expressions:

Element	Standardized	Effect when set
icase	Yes	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
nosubs	Yes	Specifies that when a regular expression is matched against a character container sequence, then no sub-expression matches are to be stored in the supplied `match_results` structure.
optimize	Yes	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output. This currently has no effect for Boost.Regex.
collate	Yes	Specifies that character ranges of the form `[a-b]` should be locale sensitive. This bit is on by default for POSIX-Extended regular expressions, but can be unset to force ranges to be compared by code point only.
newline_alt	No	Specifies that the \n character has the same effect as the alternation operator |. Allows newline separated lists to be used as a list of alternatives.
no_escape_in_lists	No	When set this makes the escape character ordinary inside lists, so that `[\b]` would match either '\' or 'b'. This bit is on by default for POSIX-Extended regular expressions, but can be unset to force escapes to be recognised inside lists.
no_bk_refs	No	When set then backreferences are disabled. This bit is on by default for POSIX-Extended regular expressions, but can be unset to support for backreferences on.
no_except	No	Prevents `basic_regex` from throwing an exception when an invalid expression is encountered.
save_subexpression_location	No	When set then the locations of individual sub-expressions within the *original regular expression string* can be accessed via the `subexpression()` member function of `basic_regex`.

Options for POSIX Basic Regular Expressions

Exactly one of the following must always be set for POSIX basic regular expressions:

Element	Standardized	Effect When Set
basic	Yes	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX basic regular expressions in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1).
sed	No	As Above.
grep	Yes	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility `grep` in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilit\ies, grep (FWD.1). That is to say, the same as POSIX basic syntax, but with the newline character acting as an alternation character; the expression is treated as a newline separated list of alternatives.
emacs	No	Specifies that the grammar recognised is the superset of the POSIX-Basic syntax used by the emacs program.

The following options may also be set when using POSIX basic regular expressions:

Element	Standardized	Effect when set
icase	Yes	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
nosubs	Yes	Specifies that when a regular expression is matched against a character container sequence, then no sub-expression matches are to be stored in the supplied `match_results` structure.
optimize	Yes	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output. This currently has no effect for Boost.Regex.
collate	Yes	Specifies that character ranges of the form `[a-b]` should be locale sensitive. This bit is on by default for POSIX-Basic regular expressions, but can be unset to force ranges to be compared by code point only.
newline_alt	No	Specifies that the \n character has the same effect as the alternation operator |. Allows newline separated lists to be used as a list of alternatives. This bit is already set, if you use the `grep` option.
no_char_classes	No	When set then character classes such as `[[:alnum:]]` are not allowed.
no_escape_in_lists	No	When set this makes the escape character ordinary inside lists, so that `[\b]` would match either '\' or 'b'. This bit is on by default for POSIX-basic regular expressions, but can be unset to force escapes to be recognised inside lists.
no_intervals	No	When set then bounded repeats such as a{2,3} are not permitted.
bk_plus_qm	No	When set then `\?` acts as a zero-or-one repeat operator, and `\+` acts as a one-or-more repeat operator.
bk_vbar	No	When set then `|` acts as the alternation operator.
no_except	No	Prevents `basic_regex` from throwing an exception when an invalid expression is encountered.

Element	Standardized	Effect when set
save_subexpression_location	No	When set then the locations of individual sub-expressions within the *original regular expression string* can be accessed via the `subexpression()` member function of `basic_regex`.

Options for Literal Strings

The following must always be set to interpret the expression as a string literal:

Element	Standardized	Effect when set
literal	Yes	Treat the string as a literal (no special characters).

The following options may also be combined with the literal flag:

Element	Standardized	Effect when set
icase	Yes	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
optimize	Yes	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output. This currently has no effect for Boost.Regex.

match_flag_type

The type `match_flag_type` is an implementation specific bitmask type (see C++ std 17.3.2.1.2) that controls how a regular expression is matched against a character sequence. The behavior of the format flags is described in more detail in the format syntax guide.

```
namespace boost{ namespace regex_constants{

typedef implemenation-specific-bitmask-type match_flag_type;

static const match_flag_type match_default = 0;
static const match_flag_type match_not_bob;
static const match_flag_type match_not_eob;
static const match_flag_type match_not_bol;
static const match_flag_type match_not_eol;
static const match_flag_type match_not_bow;
static const match_flag_type match_not_eow;
static const match_flag_type match_any;
static const match_flag_type match_not_null;
static const match_flag_type match_continuous;
static const match_flag_type match_partial;
static const match_flag_type match_single_line;
static const match_flag_type match_prev_avail;
static const match_flag_type match_not_dot_newline;
static const match_flag_type match_not_dot_null;
static const match_flag_type match_posix;
static const match_flag_type match_perl;
static const match_flag_type match_nosubs;
static const match_flag_type match_extra;

static const match_flag_type format_default = 0;
static const match_flag_type format_sed;
static const match_flag_type format_perl;
static const match_flag_type format_literal;

static const match_flag_type format_no_copy;
static const match_flag_type format_first_only;
static const match_flag_type format_all;

} // namespace regex_constants
} // namespace boost
```

Description

The type `match_flag_type` is an implementation specific bitmask type (see C++ std 17.3.2.1.2). When matching a regular expression against a sequence of characters [first, last) then setting its elements has the effects listed in the table below:

Element	Effect if set
match_default	Specifies that matching of regular expressions proceeds without any modification of the normal rules used in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1)
match_not_bob	Specifies that the expressions "\A" and "\`" should not match against the sub-sequence [first,first).
match_not_eob	Specifies that the expressions "\'", "\z" and "\Z" should not match against the sub-sequence [last,last).
match_not_bol	Specifies that the expression "^" should not be matched against the sub-sequence [first,first).
match_not_eol	Specifies that the expression "$" should not be matched against the sub-sequence [last,last).
match_not_bow	Specifies that the expressions "\<" and "\b" should not be matched against the sub-sequence [first,first).
match_not_eow	Specifies that the expressions "\>" and "\b" should not be matched against the sub-sequence [last,last).
match_any	Specifies that if more than one match is possible then any match is an acceptable result: this will still find the leftmost match, but may not find the "best" match at that position. Use this flag if you care about the speed of matching, but don't care what was matched (only whether there is one or not).
match_not_null	Specifies that the expression can not be matched against an empty sequence.
match_continuous	Specifies that the expression must match a sub-sequence that begins at first.
match_partial	Specifies that if no match can be found, then it is acceptable to return a match [from, last) such that from!= last, if there could exist some longer sequence of characters [from,to) of which [from,last) is a prefix, and which would result in a full match. This flag is used when matching incomplete or very long texts, see the partial matches documentation for more information.
match_extra	Instructs the matching engine to retain all available capture information; if a capturing group is repeated then information about every repeat is available via match_results::captures() or sub_match_captures().
match_single_line	Equivalent to the inverse of Perl's m/ modifier; prevents ^ from matching after an embedded newline character (so that it only matches at the start of the text being matched), and $ from matching before an embedded newline (so that it only matches at the end of the text being matched).

Element	Effect if set
match_prev_avail	Specifies that --first is a valid iterator position, when this flag is set then the flags match_not_bol and match_not_bow are ignored by the regular expression algorithms (RE.7) and iterators (RE.8).
match_not_dot_newline	Specifies that the expression "." does not match a newline character. This is the inverse of Perl's s/ modifier.
match_not_dot_null	Specifies that the expression "." does not match a character null '\0'.
match_posix	Specifies that the expression should be matched according to the POSIX leftmost-longest rule, regardless of what kind of expression was compiled. Be warned that these rules do not work well with many Perl-specific features such as non-greedy repeats.
match_perl	Specifies that the expression should be matched according to the Perl matching rules, irrespective of what kind of expression was compiled.
match_nosubs	Makes the expression behave as if it had no marked subexpressions, no matter how many capturing groups are actually present. The `match_results` class will only contain information about the overall match, and not any sub-expressions.
format_default	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the rules used by the ECMAScript replace function in ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace. (FWD.1). This is functionally identical to the Perl format string rules. In addition during search and replace operations then all non-overlapping occurrences of the regular expression are located and replaced, and sections of the input that did not match the expression, are copied unchanged to the output string.
format_sed	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the rules used by the Unix sed utility in IEEE Std 1003.1-2001, Portable Operating SystemInterface (POSIX), Shells and Utilities. See also the Sed Format string reference.
format_perl	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the same rules as Perl 5.
format_literal	Specifies that when a regular expression match is to be replaced by a new string, that the new string is a literal copy of the replacement text.

Element	Effect if set
format_all	Specifies that all syntax extensions are enabled, including conditional (?ddexpression1:expression2) replacements: see the format string guide for more details.
format_no_copy	When specified during a search and replace operation, then sections of the character container sequence being searched that do match the regular expression, are not copied to the output string.
format_first_only	When specified during a search and replace operation, then only the first occurrence of the regular expression is replaced.

error_type

Synopsis

Type error type represents the different types of errors that can be raised by the library when parsing a regular expression.

```
namespace boost{ namespace regex_constants{

typedef implementation-specific-type error_type;

static const error_type error_collate;
static const error_type error_ctype;
static const error_type error_escape;
static const error_type error_backref;
static const error_type error_brack;
static const error_type error_paren;
static const error_type error_brace;
static const error_type error_badbrace;
static const error_type error_range;
static const error_type error_space;
static const error_type error_badrepeat;
static const error_type error_complexity;
static const error_type error_stack;
static const error_type error_bad_pattern;

} // namespace regex_constants
} // namespace boost
```

Description

The type `error_type` is an implementation-specific enumeration type that may take one of the following values:

Constant	Meaning
error_collate	An invalid collating element was specified in a [[.name.]] block.
error_ctype	An invalid character class name was specified in a [[:name:]] block.
error_escape	An invalid or trailing escape was encountered.
error_backref	A back-reference to a non-existant marked sub-expression was encountered.
error_brack	An invalid character set [...] was encountered.
error_paren	Mismatched '(' and ')'.
error_brace	Mismatched '{' and '}'.
error_badbrace	Invalid contents of a {...} block.
error_range	A character range was invalid, for example [d-a].
error_space	Out of memory.
error_badrepeat	An attempt to repeat something that can not be repeated - for example a*+
error_complexity	The expression became too complex to handle.
error_stack	Out of program stack space.
error_bad_pattern	Other unspecified errors.

regex_traits

```
namespace boost{

template <class charT, class implementationT = sensible_default_choice>
struct regex_traits : public implementationT
{
   regex_traits() : implementationT() {}
};

template <class charT>
struct c_regex_traits;

template <class charT>
class cpp_regex_traits;

template <class charT>
class w32_regex_traits;

} // namespace boost
```

Description

The class `regex_traits` is just a thin wrapper around an actual implementation class, which may be one of:

- `c_regex_traits`: this class is deprecated, it wraps the C locale, and is used as the default implementation when the platform is not Win32, and the C++ locale is not available.
- `cpp_regex_traits`: the default traits class for non-Win32 platforms, allows the regex class to be imbued with a std::locale instance.
- `w32_regex_traits`: the default traits class implementation on Win32 platforms, allows the regex class to be imbued with an LCID.

The default behavior can be altered by defining one of the following configuration macros in boost/regex/user.hpp

- BOOST_REGEX_USE_C_LOCALE: makes `c_regex_traits` the default.
- BOOST_REGEX_USE_CPP_LOCALE: makes `cpp_regex_traits` the default.

All these traits classes fulfil the traits class requirements.

Interfacing With Non-Standard String Types

The Boost.Regex algorithms and iterators are all iterator-based, with convenience overloads of the algorithms provided that convert standard library string types to iterator pairs internally. If you want to search a non-standard string type then the trick is to convert that string into an iterator pair: so far I haven't come across any string types that can't be handled this way, even if they're not officially iterator based. Certainly any string type that provides access to it's internal buffer, along with it's length, can be converted into a pair of pointers (which can be used as iterators).

Some non-standard string types are sufficiently common that wappers have been provided for them already: currently this includes the ICU and MFC string class types.

Working With Unicode and ICU String Types

Introduction to using Regex with ICU

The header:

```
<boost/regex/icu.hpp>
```

contains the data types and algorithms necessary for working with regular expressions in a Unicode aware environment.

In order to use this header you will need the ICU library, and you will need to have built the Boost.Regex library with ICU support enabled.

The header will enable you to:

- Create regular expressions that treat Unicode strings as sequences of UTF-32 code points.
- Create regular expressions that support various Unicode data properties, including character classification.
- Transparently search Unicode strings that are encoded as either UTF-8, UTF-16 or UTF-32.

Unicode regular expression types

Header `<boost/regex/icu.hpp>` provides a regular expression traits class that handles UTF-32 characters:

```
class icu_regex_traits;
```

and a regular expression type based upon that:

```
typedef basic_regex<UChar32,icu_regex_traits> u32regex;
```

The type `u32regex` is regular expression type to use for all Unicode regular expressions; internally it uses UTF-32 code points, but can be created from, and used to search, either UTF-8, or UTF-16 encoded strings as well as UTF-32 ones.

The constructors, and assign member functions of `u32regex`, require UTF-32 encoded strings, but there are a series of overloaded algorithms called `make_u32regex` which allow regular expressions to be created from UTF-8, UTF-16, or UTF-32 encoded strings:

```
template <class InputIterator>
u32regex make_u32regex(InputIterator i,
                       InputIterator j,
                       boost::regex_constants::syntax_option_type opt);
```

Effects: Creates a regular expression object from the iterator sequence [i,j). The character encoding of the sequence is determined based upon sizeof(*i): 1 implies UTF-8, 2 implies UTF-16, and 4 implies UTF-32.

```
u32regex make_u32regex(const char* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

Effects: Creates a regular expression object from the Null-terminated UTF-8 character sequence *p*.

```
u32regex make_u32regex(const unsigned char* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

Effects: Creates a regular expression object from the Null-terminated UTF-8 character sequence p.

```
u32regex make_u32regex(const wchar_t* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

Effects: Creates a regular expression object from the Null-terminated character sequence p. The character encoding of the sequence is determined based upon sizeof(wchar_t): 1 implies UTF-8, 2 implies UTF-16, and 4 implies UTF-32.

```
u32regex make_u32regex(const UChar* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

Effects: Creates a regular expression object from the Null-terminated UTF-16 character sequence p.

```
template<class C, class T, class A>
u32regex make_u32regex(const std::basic_string<C, T, A>& s,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

Effects: Creates a regular expression object from the string s. The character encoding of the string is determined based upon sizeof(C): 1 implies UTF-8, 2 implies UTF-16, and 4 implies UTF-32.

```
u32regex make_u32regex(const UnicodeString& s,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

Effects: Creates a regular expression object from the UTF-16 encoding string s.

Unicode Regular Expression Algorithms

The regular expression algorithms `regex_match`, `regex_search` and `regex_replace` all expect that the character sequence upon which they operate, is encoded in the same character encoding as the regular expression object with which they are used. For Unicode regular expressions that behavior is undesirable: while we may want to process the data in UTF-32 "chunks", the actual data is much more likely to encoded as either UTF-8 or UTF-16. Therefore the header <boost/regex/icu.hpp> provides a series of thin wrappers around these algorithms, called `u32regex_match`, `u32regex_search`, and `u32regex_replace`. These wrappers use iterator-adapters internally to make external UTF-8 or UTF-16 data look as though it's really a UTF-32 sequence, that can then be passed on to the "real" algorithm.

u32regex_match

For each `regex_match` algorithm defined by `<boost/regex.hpp>`, then `<boost/regex/icu.hpp>` defines an overloaded algorithm that takes the same arguments, but which is called `u32regex_match`, and which will accept UTF-8, UTF-16 or UTF-32 encoded data, as well as an ICU UnicodeString as input.

Example: match a password, encoded in a UTF-16 UnicodeString:

```
//
// Find out if *password* meets our password requirements,
// as defined by the regular expression *requirements*.
//
bool is_valid_password(const UnicodeString& password, const UnicodeString& requirements)
{
   return boost::u32regex_match(password, boost::make_u32regex(requirements));
}
```

Example: match a UTF-8 encoded filename:

```
//
// Extract filename part of a path from a UTF-8 encoded std::string and return the result
// as another std::string:
//
std::string get_filename(const std::string& path)
{
   boost::u32regex r = boost::make_u32regex("(?:\\A|.*\\\\)([^\\\\]+)");
   boost::smatch what;
   if(boost::u32regex_match(path, what, r))
   {
      // extract $1 as a std::string:
      return what.str(1);
   }
   else
   {
      throw std::runtime_error("Invalid pathname");
   }
}
```

u32regex_search

For each `regex_search` algorithm defined by `<boost/regex.hpp>`, then `<boost/regex/icu.hpp>` defines an overloaded algorithm that takes the same arguments, but which is called `u32regex_search`, and which will accept UTF-8, UTF-16 or UTF-32 encoded data, as well as an ICU UnicodeString as input.

Example: search for a character sequence in a specific language block:

```
UnicodeString extract_greek(const UnicodeString& text)
{
   // searches through some UTF-16 encoded text for a block encoded in Greek,
   // this expression is imperfect, but the best we can do for now - searching
   // for specific scripts is actually pretty hard to do right.
   //
   // Here we search for a character sequence that begins with a Greek letter,
   // and continues with characters that are either not-letters ( [^[:L*:]] )
   // or are characters in the Greek character block ( [\\x{370}-\\x{3FF}] ).
   //
   boost::u32regex r = boost::make_u32regex(
         L"[\\x{370}-\\x{3FF}](?:[^[:L*:]]|[\\x{370}-\\x{3FF}])*");
   boost::u16match what;
   if(boost::u32regex_search(text, what, r))
   {
      // extract $0 as a UnicodeString:
      return UnicodeString(what[0].first, what.length(0));
   }
   else
   {
      throw std::runtime_error("No Greek found!");
   }
}
```

u32regex_replace

For each `regex_replace` algorithm defined by `<boost/regex.hpp>`, then `<boost/regex/icu.hpp>` defines an overloaded algorithm that takes the same arguments, but which is called `u32regex_replace`, and which will accept UTF-8, UTF-16 or UTF-32 encoded data, as well as an ICU UnicodeString as input. The input sequence and the format string specifier passed to the algorithm, can be encoded independently (for example one can be UTF-8, the other in UTF-16), but the result string / output iterator argument must use the same character encoding as the text being searched.

Example: Credit card number reformatting:

```
//
// Take a credit card number as a string of digits,
// and reformat it as a human readable string with "-"
// separating each group of four digit;,
// note that we're mixing a UTF-32 regex, with a UTF-16
// string and a UTF-8 format specifier, and it still all
// just works:
//
const boost::u32regex e = boost::make_u32regex(
      "\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z");
const char* human_format = "$1-$2-$3-$4";

UnicodeString human_readable_card_number(const UnicodeString& s)
{
   return boost::u32regex_replace(s, e, human_format);
}
```

Unicode Aware Regex Iterators

u32regex_iterator

Type `u32regex_iterator` is in all respects the same as `regex_iterator` except that since the regular expression type is always `u32regex` it only takes one template parameter (the iterator type). It also calls `u32regex_search` internally, allowing it to interface correctly with UTF-8, UTF-16, and UTF-32 data:

```
template <class BidirectionalIterator>
class u32regex_iterator
{
   // for members see regex_iterator
};

typedef u32regex_iterator<const char*>     utf8regex_iterator;
typedef u32regex_iterator<const UChar*>    utf16regex_iterator;
typedef u32regex_iterator<const UChar32*>  utf32regex_iterator;
```

In order to simplify the construction of a `u32regex_iterator` from a string, there are a series of non-member helper functions called make_u32regex_iterator:

```
u32regex_iterator<const char*>
   make_u32regex_iterator(const char* s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_iterator<const wchar_t*>
   make_u32regex_iterator(const wchar_t* s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_iterator<const UChar*>
   make_u32regex_iterator(const UChar* s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc>
u32regex_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_iterator(const std::basic_string<charT, Traits, Alloc>& s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_iterator<const UChar*>
   make_u32regex_iterator(const UnicodeString& s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);
```

Each of these overloads returns an iterator that enumerates all occurrences of expression *e*, in text *s*, using match_flags *m*.

Example: search for international currency symbols, along with their associated numeric value:

```
void enumerate_currencies(const std::string& text)
{
   // enumerate and print all the currency symbols, along
   // with any associated numeric values:
   const char* re =
      "([[:Sc:]][[:Cf:][:Cc:][:Z*:]]*)?"
      "([[:Nd:]]+(?:[[:Po:]][[:Nd:]]+)?)?"
      "(?(1)"
         "|(?(2)"
            "[[:Cf:][:Cc:][:Z*:]]*"
         ")"
         "[[:Sc:]]"
      ")";
   boost::u32regex r = boost::make_u32regex(re);
   boost::u32regex_iterator<std::string::const_iterator>
         i(boost::make_u32regex_iterator(text, r)), j;
   while(i != j)
   {
      std::cout << (*i)[0] << std::endl;
      ++i;
   }
}
```

Calling

```
enumerate_currencies(" $100.23 or £198.12 ");
```

Yields the output:

```
$100.23
£198.12
```

Provided of course that the input is encoded as UTF-8.

u32regex_token_iterator

Type `u32regex_token_iterator` is in all respects the same as `regex_token_iterator` except that since the regular expression type is always `u32regex` it only takes one template parameter (the iterator type). It also calls `u32regex_search` internally, allowing it to interface correctly with UTF-8, UTF-16, and UTF-32 data:

```
template <class BidirectionalIterator>
class u32regex_token_iterator
{
   // for members see regex_token_iterator
};

typedef u32regex_token_iterator<const char*>     utf8regex_token_iterator;
typedef u32regex_token_iterator<const UChar*>    utf16regex_token_iterator;
typedef u32regex_token_iterator<const UChar32*>  utf32regex_token_iterator;
```

In order to simplify the construction of a `u32regex_token_iterator` from a string, there are a series of non-member helper functions called `make_u32regex_token_iterator`:

```
u32regex_token_iterator<const char*>
   make_u32regex_token_iterator(
         const char* s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const wchar_t*>
   make_u32regex_token_iterator(
         const wchar_t* s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const UChar*>
   make_u32regex_token_iterator(
         const UChar* s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc>
u32regex_token_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_token_iterator(
         const std::basic_string<charT, Traits, Alloc>& s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const UChar*>
   make_u32regex_token_iterator(
         const UnicodeString& s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

Each of these overloads returns an iterator that enumerates all occurrences of marked sub-expression sub in regular expression *e*, found in text *s*, using match_flags *m*.

```
template <std::size_t N>
u32regex_token_iterator<const char*>
   make_u32regex_token_iterator(
         const char* p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <std::size_t N>
u32regex_token_iterator<const wchar_t*>
   make_u32regex_token_iterator(
         const wchar_t* p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <std::size_t N>
u32regex_token_iterator<const UChar*>
   make_u32regex_token_iterator(
         const UChar* p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc, std::size_t N>
u32regex_token_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_token_iterator(
         const std::basic_string<charT, Traits, Alloc>& p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <std::size_t N>
u32regex_token_iterator<const UChar*>
   make_u32regex_token_iterator(
         const UnicodeString& s,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);
```

Each of these overloads returns an iterator that enumerates one sub-expression for each submatch in regular expression *e*, found in text *s*, using match_flags *m*.

```
u32regex_token_iterator<const char*>
   make_u32regex_token_iterator(
         const char* p,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const wchar_t*>
   make_u32regex_token_iterator(
         const wchar_t* p,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const UChar*>
   make_u32regex_token_iterator(
         const UChar* p,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc>
u32regex_token_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_token_iterator(
         const std::basic_string<charT, Traits, Alloc>& p,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const UChar*>
   make_u32regex_token_iterator(
         const UnicodeString& s,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

Each of these overloads returns an iterator that enumerates one sub-expression for each submatch in regular expression *e*, found in text *s*, using match_flags *m*.

Example: search for international currency symbols, along with their associated numeric value:

```
void enumerate_currencies2(const std::string& text)
{
   // enumerate and print all the currency symbols, along
   // with any associated numeric values:
   const char* re =
      "([[:Sc:]][[:Cf:][:Cc:][:Z*:]]*)?"
      "([[:Nd:]]+(?:[[:Po:]][[:Nd:]]+)?)?"
      "(?(1)"
         "|(?(2)"
            "[[:Cf:][:Cc:][:Z*:]]*"
         ")"
         "[[:Sc:]]"
      ")";
   boost::u32regex r = boost::make_u32regex(re);
   boost::u32regex_token_iterator<std::string::const_iterator>
      i(boost::make_u32regex_token_iterator(text, r, 1)), j;
   while(i != j)
   {
      std::cout << *i << std::endl;
      ++i;
   }
}
```

Using Boost Regex With MFC Strings

Introduction to Boost.Regex and MFC Strings

The header `<boost/regex/mfc.hpp>` provides Boost.Regex support for MFC string types: note that this support requires Visual Studio .NET (Visual C++ 7) or later, where all of the MFC and ATL string types are based around the CSimpleStringT class template.

In the following documentation, whenever you see CSimpleStringT<charT>, then you can substitute any of the following MFC/ATL types (all of which inherit from CSimpleStringT):

```
CString
CStringA
CStringW
CAtlString
CAtlStringA
CAtlStringW
CStringT<charT,traits>
CFixedStringT<charT,N>
CSimpleStringT<charT>
```

Regex Types Used With MFC Strings

The following typedefs are provided for the convenience of those working with TCHAR's:

```
typedef basic_regex<TCHAR>                  tregex;
typedef match_results<TCHAR const*>         tmatch;
typedef regex_iterator<TCHAR const*>        tregex_iterator;
typedef regex_token_iterator<TCHAR const*>  tregex_token_iterator;
```

If you are working with explicitly narrow or wide characters rather than TCHAR, then use the regular Boost.Regex types `regex` and `wregex` instead.

Regular Expression Creation From an MFC String

The following helper function is available to assist in the creation of a regular expression from an MFC/ATL string type:

```
template <class charT>
basic_regex<charT>
   make_regex(const ATL::CSimpleStringT<charT>& s,
            ::boost::regex_constants::syntax_option_type f = boost::regex_constants::normal);
```

Effects: returns `basic_regex<charT>(s.GetString(), s.GetString() + s.GetLength(), f);`

Overloaded Algorithms For MFC String Types

For each regular expression algorithm that's overloaded for a `std::basic_string` argument, there is also one overloaded for the MFC/ATL string types. These algorithm signatures all look a lot more complex than they actually are, but for completeness here they are anyway:

regex_match

There are two overloads, the first reports what matched in a match_results structure, the second does not.

All the usual caveats for `regex_match` apply, in particular the algorithm will only report a successful match if all of the input text matches the expression, if this isn't what you want then use `regex_search` instead.

```
template <class charT, class T, class A>
bool regex_match(
   const ATL::CSimpleStringT<charT>& s,
   match_results<const B*, A>& what,
   const basic_regex<charT, T>& e,
   boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

Effects: returns `::boost::regex_match(s.GetString(), s.GetString() + s.GetLength(), what, e, f);`

Example:

```
//
// Extract filename part of a path from a CString and return the result
// as another CString:
//
CString get_filename(const CString& path)
{
   boost::tregex r(__T("(?:\\A|.*\\\\)([^\\\\]+)"));
   boost::tmatch what;
   if(boost::regex_match(path, what, r))
   {
      // extract $1 as a CString:
      return CString(what[1].first, what.length(1));
   }
   else
   {
      throw std::runtime_error("Invalid pathname");
   }
}
```

regex_match (second overload)

```
template <class charT, class T>
bool regex_match(
   const ATL::CSimpleStringT<charT>& s,
   const basic_regex<B, T>& e,
   boost::regex_constants::match_flag_type f = boost::regex_constants::match_default)
```

Effects: returns `::boost::regex_match(s.GetString(), s.GetString() + s.GetLength(), e, f);`

Example:

```
//
// Find out if *password* meets our password requirements,
// as defined by the regular expression *requirements*.
//
bool is_valid_password(const CString& password, const CString& requirements)
{
   return boost::regex_match(password, boost::make_regex(requirements));
}
```

regex_search

There are two additional overloads for `regex_search`, the first reports what matched the second does not:

```
template <class charT, class A, class T>
bool regex_search(const ATL::CSimpleStringT<charT>& s,
                  match_results<const charT*, A>& what,
                  const basic_regex<charT, T>& e,
                  boost::regex_constants::match_flag_type f = boost::regex_constants::match_de↵
fault)
```

Effects: returns ::boost::regex_search(s.GetString(), s.GetString() + s.GetLength(), what, e, f);

Example: Postcode extraction from an address string.

```
CString extract_postcode(const CString& address)
{
   // searches throw address for a UK postcode and returns the result,
   // the expression used is by Phil A. on www.regxlib.com:
   boost::tregex r(__T("^(([A-Z]{1,2}[0-9]{1,2})|([A-Z]{1,2}[0-9][A-Z]))\\s?([0-9][A-Z]{2})$"));
   boost::tmatch what;
   if(boost::regex_search(address, what, r))
   {
      // extract $0 as a CString:
      return CString(what[0].first, what.length());
   }
   else
   {
      throw std::runtime_error("No postcode found");
   }
}
```

regex_search (second overload)

```
template <class charT, class T>
inline bool regex_search(const ATL::CSimpleStringT<charT>& s,
                 const basic_regex<charT, T>& e,
                boost::regex_constants::match_flag_type f = boost::regex_constants::match_default)
```

Effects: returns `::boost::regex_search(s.GetString(), s.GetString() + s.GetLength(), e, f);`

regex_replace

There are two additional overloads for `regex_replace`, the first sends output to an output iterator, while the second creates a new string

```
template <class OutputIterator, class BidirectionalIterator, class traits, class
         charT>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
                             const ATL::CSimpleStringT<charT>& fmt,
                             match_flag_type flags = match_default)
```

Effects: returns `::boost::regex_replace(out, first, last, e, fmt.GetString(), flags);`

```
template <class traits, charT>
ATL::CSimpleStringT<charT> regex_replace(const ATL::CSimpleStringT<charT>& s,
                             const basic_regex<charT, traits>& e,
                             const ATL::CSimpleStringT<charT>& fmt,
                             match_flag_type flags = match_default)
```

Effects: returns a new string created using `regex_replace`, and the same memory manager as string *s*.

Example:

```
//
// Take a credit card number as a string of digits,
// and reformat it as a human readable string with "-"
// separating each group of four digits:
//
const boost::tregex e(__T("\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z"));
const CString human_format = __T("$1-$2-$3-$4");

CString human_readable_card_number(const CString& s)
{
   return boost::regex_replace(s, e, human_format);
}
```

Iterating Over the Matches Within An MFC String

The following helper functions are provided to ease the conversion from an MFC/ATL string to a `regex_iterator` or `regex_token_iterator`:

regex_iterator creation helper

```
template <class charT>
regex_iterator<charT const*>
   make_regex_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

Effects: returns `regex_iterator(s.GetString(), s.GetString() + s.GetLength(), e, f);`

Example:

```
void enumerate_links(const CString& html)
{
   // enumerate and print all the  links in some HTML text,
   // the expression used is by Andew Lee on www.regxlib.com:
   boost::tregex r(
      __T("href=[\"\']((http:\\/\\/|\\.\\/|\\/)?\\w+"
          "(\\.\\w+)*(\\/\\w+(\\.\\w+)?)*"
          "(\\/|\\?\\w*=\\w*(&\\w*=\\w*)*)?)[\"\']"));
   boost::tregex_iterator i(boost::make_regex_iterator(html, r)), j;
   while(i != j)
   {
      std::cout << (*i)[1] << std::endl;
      ++i;
   }
}
```

regex_token_iterator creation helpers

```
template <class charT>
regex_token_iterator<charT const*>
   make_regex_token_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      int sub = 0,
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

Effects: returns `regex_token_iterator(s.GetString(), s.GetString() + s.GetLength(), e, sub, f);`

```
template <class charT>
regex_token_iterator<charT const*>
   make_regex_token_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      const std::vector<int>& subs,
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

Effects: returns `regex_token_iterator(s.GetString(), s.GetString() + s.GetLength(), e, subs, f);`

```
template <class charT, std::size_t N>
regex_token_iterator<charT const*>
   make_regex_token_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      const int (& subs)[N],
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

Effects: returns `regex_token_iterator(s.GetString(), s.GetString() + s.GetLength(), e, subs, f);`

Example:

```
void enumerate_links2(const CString& html)
{
   // enumerate and print all the  links in some HTML text,
   // the expression used is by Andew Lee on www.regxlib.com:
   boost::tregex r(
         __T("href=[\"\']((http:\\/\\/|\\.\\/|\\/)?\\w+"
             "(\\.\\w+)*(\\/\\w+(\\.\\w+)?)*"
             "(\\/|\\?\\w*=\\w*(&\\w*=\\w*)*)?)[\"\']"));
   boost::tregex_token_iterator i(boost::make_regex_token_iterator(html, r, 1)), j;
   while(i != j)
   {
      std::cout << *i << std::endl;
      ++i;
   }
}
```

POSIX Compatible C API's

Note

this is an abridged reference to the POSIX API functions, these are provided for compatibility with other libraries, rather than as an API to be used in new code (unless you need access from a language other than C++). This version of these functions should also happily coexist with other versions, as the names used are macros that expand to the actual function names.

```
#include <boost/cregex.hpp>
```

or:

```
#include <boost/regex.h>
```

The following functions are available for users who need a POSIX compatible C library, they are available in both Unicode and narrow character versions, the standard POSIX API names are macros that expand to one version or the other depending upon whether UNICODE is defined or not.

Important

Note that all the symbols defined here are enclosed inside namespace `boost` when used in C++ programs, unless you use `#include <boost/regex.h>` instead - in which case the symbols are still defined in namespace boost, but are made available in the global namespace as well.

The functions are defined as:

```
extern "C" {

struct regex_tA;
struct regex_tW;

int regcompA(regex_tA*, const char*, int);
unsigned int regerrorA(int, const regex_tA*, char*, unsigned int);
int regexecA(const regex_tA*, const char*, unsigned int, regmatch_t*, int);
void regfreeA(regex_tA*);

int regcompW(regex_tW*, const wchar_t*, int);
unsigned int regerrorW(int, const regex_tW*, wchar_t*, unsigned int);
int regexecW(const regex_tW*, const wchar_t*, unsigned int, regmatch_t*, int);
void regfreeW(regex_tW*);

#ifdef UNICODE
#define regcomp regcompW
#define regerror regerrorW
#define regexec regexecW
#define regfree regfreeW
#define regex_t regex_tW
#else
#define regcomp regcompA
#define regerror regerrorA
#define regexec regexecA
#define regfree regfreeA
#define regex_t regex_tA
#endif
}
```

All the functions operate on structure regex_t, which exposes two public members:

Member	Meaning
`unsigned int re_nsub`	This is filled in by `regcomp` and indicates the number of sub-expressions contained in the regular expression.
`const TCHAR* re_endp`	Points to the end of the expression to compile when the flag REG_PEND is set.

Note

`regex_t` is actually a `#define` - it is either `regex_tA` or `regex_tW` depending upon whether `UNICODE` is defined or not, `TCHAR` is either `char` or `wchar_t` again depending upon the macro `UNICODE`.

regcomp

`regcomp` takes a pointer to a `regex_t`, a pointer to the expression to compile and a flags parameter which can be a combination of:

Flag	Meaning
REG_EXTENDED	Compiles modern regular expressions. Equivalent to `regbase::char_classes | regbase::intervals | regbase::bk_refs`.
REG_BASIC	Compiles basic (obsolete) regular expression syntax. Equivalent to `regbase::char_classes | regbase::intervals | regbase::limited_ops | regbase::bk_braces | regbase::bk_parens | regbase::bk_refs`.
REG_NOSPEC	All characters are ordinary, the expression is a literal string.
REG_ICASE	Compiles for matching that ignores character case.
REG_NOSUB	Has no effect in this library.
REG_NEWLINE	When this flag is set a dot does not match the newline character.
REG_PEND	When this flag is set the re_endp parameter of the regex_t structure must point to the end of the regular expression to compile.
REG_NOCOLLATE	When this flag is set then locale dependent collation for character ranges is turned off.
REG_ESCAPE_IN_LISTS	When this flag is set, then escape sequences are permitted in bracket expressions (character sets).
REG_NEWLINE_ALT	When this flag is set then the newline character is equivalent to the alternation operator |.
REG_PERL	Compiles Perl like regular expressions.
REG_AWK	A shortcut for awk-like behavior: `REG_EXTENDED | REG_ESCAPE_IN_LISTS`
REG_GREP	A shortcut for grep like behavior: `REG_BASIC | REG_NEWLINE_ALT`
REG_EGREP	A shortcut for egrep like behavior: `REG_EXTENDED | REG_NEWLINE_ALT`

regerror

regerror takes the following parameters, it maps an error code to a human readable string:

Parameter	Meaning
int code	The error code.
const regex_t* e	The regular expression (can be null).
char* buf	The buffer to fill in with the error message.
unsigned int buf_size	The length of buf.

If the error code is OR'ed with REG_ITOA then the message that results is the printable name of the code rather than a message, for example "REG_BADPAT". If the code is REG_ATIO then e must not be null and e->re_pend must point to the printable name of an error code, the return value is then the value of the error code. For any other value of code, the return value is the number of characters in the error message, if the return value is greater than or equal to buf_size then regerror will have to be called again with a larger buffer.

regexec

regexec finds the first occurrence of expression e within string buf. If len is non-zero then **m* is filled in with what matched the regular expression, m[0] contains what matched the whole string, m[1] the first sub-expression etc, see regmatch_t in the header file declaration for more details. The eflags parameter can be a combination of:

Flag	Meaning
REG_NOTBOL	Parameter buf does not represent the start of a line.
REG_NOTEOL	Parameter buf does not terminate at the end of a line.
REG_STARTEND	The string searched starts at buf + pmatch[0].rm_so and ends at buf + pmatch[0].rm_eo.

regfree

`regfree` frees all the memory that was allocated by regcomp.

Concepts

charT Requirements

Type `charT` used a template argument to class template `basic_regex`, must have a trivial default constructor, copy constructor, assignment operator, and destructor. In addition the following requirements must be met for objects; *c* of type `charT`, *c1* and *c2* of type `charT const`, and *i* of type `int`:

Expression	Return type	Assertion / Note / Pre- / Post-condition
charT c	charT	Default constructor (must be trivial).
charT c(c1)	charT	Copy constructor (must be trivial).
c1 = c2	charT	Assignment operator (must be trivial).
c1 == c2	bool	true if c1 has the same value as c2.
c1 != c2	bool	true if c1 and c2 are not equal.
c1 < c2	bool	true if the value of c1 is less than c2.
c1 > c2	bool	true if the value of c1 is greater than c2.
c1 <= c2	bool	true if c1 is less than or equal to c2.
c1 >= c2	bool	true if c1 is greater than or equal to c2.
intmax_t i = c1	int	charT must be convertible to an integral type. Note: type charT is not required to support this operation, if the traits class used supports the full Boost-specific interface, rather than the minimal standardised-interface (see traits class requirements below).
charT c(i);	charT	charT must be constructable from an integral type.

Traits Class Requirements

There are two sets of requirements for the `traits` template argument to `basic_regex`: a minimal interface (which is part of the regex standardization proposal), and an optional Boost-specific enhanced interface.

Minimal requirements.

In the following table `X` denotes a traits class defining types and functions for the character container type `charT`; *u* is an object of type `X`; *v* is an object of type `const X`; *p* is a value of type `const charT*`; *I1* and *I2* are Input Iterators; *c* is a value of type `const charT`; *s* is an object of type `X::string_type`; *cs* is an object of type `const X::string_type`; *b* is a value of type `bool`; *I* is a value of type `int`; *F1* and *F2* are values of type `const charT*`; and *loc* is an object of type `X::locale_type`.

Expression	Return type	Assertion / Note Pre / Post condition
X::char_type	charT	The character container type used in the implementation of class template basic_regex.
X::size_type		An unsigned integer type, capable of holding the length of a null-terminated string of charT's.
X::string_type	std::basic_string<charT> or std::vector<charT>	
X::locale_type	Implementation defined	A copy constructible type that represents the locale used by the traits class.
X::char_class_type	Implementation defined	A bitmask type representing a particular character classification. Multiple values of this type can be bitwise-or'ed together to obtain a new valid value.
X::length(p)	X::size_type	Yields the smallest i such that p[i] == 0. Complexity is linear in i.
v.translate(c)	X::char_type	Returns a character such that for any character d that is to be considered equivalent to c then v.translate(c) == v.translate(d).
v.translate_nocase(c)	X::char_type	For all characters C that are to be considered equivalent to c when comparisons are to be performed without regard to case, then v.translate_nocase(c) == v.translate_nocase(C).
v.transform(F1, F2)	X::string_type	Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) then v.transform(G1, G2) < v.transform(H1, H2).
v.transform_primary(F1, F2)	X::string_type	Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) when character case is not considered then v.transform_primary(G1, G2) < v.transform_primary(H1, H2).

Expression	Return type	Assertion / Note Pre / Post condition
v.lookup_classname(F1, F2)	X::char_class_type	Converts the character sequence designated by the iterator range [F1,F2) into a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be safely bitwise or'ed together. Returns 0 if the character sequence is not the name of a character class recognized by X. The value returned shall be independent of the case of the characters in the sequence.
v.lookup_collatename(F1, F2)	X::string_type	Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1, F2). Returns an empty string if the character sequence is not a valid collating element.
v.isctype(c, v.lookup_classname (F1, F2))	bool	Returns true if character c is a member of the character class designated by the iterator range [F1, F2), false otherwise.
v.value(c, I)	int	Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns -1. [Note: the value of I will only be 8, 10, or 16. -end note]
u.imbue(loc)	X::locale_type	Imbues u with the locale loc, returns the previous locale used by u if any.
v.getloc()	X::locale_type	Returns the current locale used by v if any.

Additional Optional Requirements

The following additional requirements are strictly optional, however in order for `basic_regex` to take advantage of these additional interfaces, all of the following requirements must be met; `basic_regex` will detect the presence or absence of the member `boost_extensions_tag` and configure itself appropriately.

Expression	Result	Assertion / Note Pre / Post condition
X::boost_extensions_tag	An unspecified type.	When present, all of the extensions listed in this table must be present.
v.syntax_type(c)	regex_constants::syntax_type	Returns a symbolic value of type regex_constants::syntax_type that signifies the meaning of character c within the regular expression grammar.
v.escape_syntax_type(c)	regex_constants::escape_syntax_type	Returns a symbolic value of type regex_constants::escape_syntax_type, that signifies the meaning of character c within the regular expression grammar, when c has been preceded by an escape character. Precondition: if b is the character preceding c in the expression being parsed then: `v.syntax_type(b) == syntax_escape`
v.translate(c, b)	X::char_type	Returns a character d such that: for any character d that is to be considered equivalent to c then `v.translate(c,false)==v.translate(d,false)`. Likewise for all characters C that are to be considered equivalent to c when comparisons are to be performed without regard to case, then `v.translate(c,true)==v.translate(C,true)`.
v.toi(I1, I2, i)	An integer type capable of holding either a charT or an int.	Behaves as follows: if `p == q` or if `*p` is not a digit character then returns -1. Otherwise performs formatted numeric input on the sequence [p,q) and returns the result as an int. Postcondition: either p == q or *p is a non-digit character.
v.error_string(I)	std::string	Returns a human readable error string for the error condition i, where i is one of the values enumerated by type regex_constants::error_type. If the value *I* is not recognized then returns the string "Unknown error" or a localized equivalent.
v.tolower(c)	X::char_type	Converts c to lower case, used for Perl-style \l and \L formatting operations.
v.toupper(c)	X::char_type	Converts c to upper case, used for Perl-style \u and \U formatting operations.

Iterator Requirements

The regular expression algorithms (and iterators) take all require a Bidirectional-Iterator.

Deprecated Interfaces

regex_format (Deprecated)

The algorithm `regex_format` is deprecated; new code should use `match_results<>::format` instead. Existing code will continue to compile, the following documentation is taken from the previous version of Boost.Regex and will not be further updated:

Algorithm regex_format

```
#include <boost/regex.hpp>
```

The algorithm `regex_format` takes the results of a match and creates a new string based upon a format string, `regex_format` can be used for search and replace operations:

```
template <class OutputIterator, class iterator, class Allocator, class Formatter>
OutputIterator regex_format(OutputIterator out,
                            const match_results<iterator, Allocator>& m,
                            Formatter fmt,
                            match_flag_type flags = 0);
```

The library also defines the following convenience variation of `regex_format`, which returns the result directly as a string, rather than outputting to an iterator.

Note

This version may not be available, or may be available in a more limited form, depending upon your compilers capabilities

```
template <class iterator, class Allocator, class Formatter>
std::basic_string<charT> regex_format
                                (const match_results<iterator, Allocator>& m,
                                Formatter fmt,
                                match_flag_type flags = 0);
```

Parameters to the main version of the function are passed as follows:

Parameter	Description
`OutputIterator out`	An output iterator type, the output string is sent to this iterator. Typically this would be a std::ostream_iterator.
`const match_results<iterator, Allocator>& m`	An instance of `match_results` obtained from one of the matching algorithms above, and denoting what matched.
`Formatter fmt`	Either a format string that determines how the match is transformed into the new string, or a functor that computes the new string from *m* - see `match_results<>::format`.
`unsigned flags`	Optional flags which describe how the format string is to be interpreted.

Format flags are described under `match_flag_type`.

The format string syntax (and available options) is described more fully under format strings.

regex_grep (Deprecated)

The algorithm `regex_grep` is deprecated in favor of `regex_iterator` which provides a more convenient and standard library friendly interface.

The following documentation is taken unchanged from the previous boost release, and will not be updated in future.

```
#include <boost/regex.hpp>
```

`regex_grep` allows you to search through a bidirectional-iterator range and locate all the (non-overlapping) matches with a given regular expression. The function is declared as:

```
template <class Predicate, class iterator, class charT, class traits>
unsigned int regex_grep(Predicate foo,
                        iterator first,
                        iterator last,
                        const basic_regex<charT, traits>& e,
                        boost::match_flag_type flags = match_default)
```

The library also defines the following convenience versions, which take either a `const charT*`, or a `const std::basic_string<>&` in place of a pair of iterators.

```
template <class Predicate, class charT, class traits>
unsigned int regex_grep(Predicate foo,
            const charT* str,
            const basic_regex<charT, traits>& e,
            boost::match_flag_type flags = match_default);

template <class Predicate, class ST, class SA, class charT, class traits>
unsigned int regex_grep(Predicate foo,
            const std::basic_string<charT, ST, SA>& s,
            const basic_regex<charT, traits>& e,
            boost::match_flag_type flags = match_default);
```

The parameters for the primary version of `regex_grep` have the following meanings:

foo: A predicate function object or function pointer, see below for more information.

first: The start of the range to search.

last: The end of the range to search.

e: The regular expression to search for.

flags: The flags that determine how matching is carried out, one of the match_flags enumerators.

The algorithm finds all of the non-overlapping matches of the expression *e*, for each match it fills a `match_results<iterator>` structure, which contains information on what matched, and calls the predicate *foo*, passing the `match_results<iterator>` as a single argument. If the predicate returns *true*, then the grep operation continues, otherwise it terminates without searching for further matches. The function returns the number of matches found.

The general form of the predicate is:

```
struct grep_predicate
{
   bool operator()(const match_results<iterator_type>& m);
};
```

For example the regular expression "a*b" would find one match in the string "aaaaab" and two in the string "aaabb".

Remember this algorithm can be used for a lot more than implementing a version of grep, the predicate can be and do anything that you want, grep utilities would output the results to the screen, another program could index a file based on a regular expression and store a set of bookmarks in a list, or a text file conversion utility would output to file. The results of one `regex_grep` can even be chained into another `regex_grep` to create recursive parsers.

The algorithm may throw `std::runtime_error` if the complexity of matching the expression against an *N* character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts it's permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Example: convert the example from `regex_search` to use `regex_grep` instead:

```
#include <string>
#include <map>
#include <boost/regex.hpp>

// IndexClasses:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's
typedef std::map<std::string, int, std::less<std::string> > map_type;

const char* re =
   // possibly leading whitespace:
   "^[[:space:]]*"
   // possible template declaration:
   "(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   // class or struct:
   "(class|struct)[[:space:]]*"
   // leading declspec macros etc:
   "("
      "\\<\\w+\\>"
      "("
         "[[:blank:]]*\\([^)]*\\)"
      ")?"
      "[[:space:]]*"
   ")*"
   // the class name
   "(\\<\\w*\\>)[[:space:]]*"
   // template specialisation parameters
   "(<[^;:{]+>)?[[:space:]]*"
   // terminate in { or :
   "(\\{|:[^;\\{()]*\\{)";

boost::regex expression(re);
class IndexClassesPred
{
   map_type& m;
   std::string::const_iterator base;
public:
   IndexClassesPred(map_type& a, std::string::const_iterator b) : m(a), base(b) {}
   bool operator()(const  smatch& what)
   {
      // what[0] contains the whole string
      // what[5] contains the class name.
      // what[6] contains the template specialisation if any.
      // add class name and position to map:
     m[std::string(what[5].first, what[5].second) + std::string(what[6].first, what[6].second)] =
               what[5].first - base;
      return true;
   }
```

```
};
void IndexClasses(map_type& m, const std::string& file)
{
   std::string::const_iterator start, end;
   start = file.begin();
   end = file.end();
   regex_grep(IndexClassesPred(m, start), start, end, expression);
}
```

Example: Use `regex_grep` to call a global callback function:

```
#include <string>
#include <map>
#include <boost/regex.hpp>

// purpose:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's
typedef std::map<std::string, int, std::less<std::string> > map_type;

const char* re =
   // possibly leading whitespace:
   "^[[:space:]]*"
   // possible template declaration:
   "(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   // class or struct:
   "(class|struct)[[:space:]]*"
   // leading declspec macros etc:
   "("
      "\\<\\w+\\>"
      "("
         "[[:blank:]]*\\([^)]*\\)"
      ")?"
      "[[:space:]]*"
   ")*"
   // the class name
   "(\\<\\w*\\>)[[:space:]]*"
   // template specialisation parameters
   "(<[^;:{]+>)?[[:space:]]*"
   // terminate in { or :
   "(\\{|:[^;\\{()]*\\{)";

boost::regex expression(re);
map_type class_index;
std::string::const_iterator base;

bool grep_callback(const  boost::smatch& what)
{
   // what[0] contains the whole string
   // what[5] contains the class name.
   // what[6] contains the template specialisation if any.
   // add class name and position to map:
   class_in↵
dex[std::string(what[5].first, what[5].second) + std::string(what[6].first, what[6].second)] =
               what[5].first - base;
   return true;
}
void IndexClasses(const std::string& file)
{
```

```
   std::string::const_iterator start, end;
   start = file.begin();
   end = file.end();
   base = start;
   regex_grep(grep_callback, start, end, expression, match_default);
}
```

Example: use `regex_grep` to call a class member function, use the standard library adapters `std::mem_fun` and `std::bind1st` to convert the member function into a predicate:

```
#include <string>
#include <map>
#include <boost/regex.hpp>
#include <functional>
// purpose:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's

typedef std::map<std::string, int, std::less<std::string> > map_type;
class class_index
{
   boost::regex expression;
   map_type index;
   std::string::const_iterator base;
   bool  grep_callback(boost::smatch what);
public:
   void IndexClasses(const std::string& file);
   class_index()
      : index(),
      expression("^(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
                 "(class|struct)[[:space:]]*(\\<\\w+\\>([[:blank:]]*\\([^)]*\\))?"
                 "[[:space:]]*)*(\\<\\w*\\>)[[:space:]]*(<[^;:{]+>[[:space:]]*)?"
                 "(\\{|:[^;\\{()]*\\{)"
                 ){}
};
bool  class_index::grep_callback(boost::smatch what)
{
   // what[0] contains the whole string
   // what[5] contains the class name.
   // what[6] contains the template specialisation if any.
   // add class name and position to map:
  index[std::string(what[5].first, what[5].second) + std::string(what[6].first, what[6].second)] =
              what[5].first - base;
   return true;
}

void class_index::IndexClasses(const std::string& file)
{
   std::string::const_iterator start, end;
   start = file.begin();
   end = file.end();
   base = start;
   regex_grep(std::bind1st(std::mem_fun(&class_index::grep_callback), this),
            start,
            end,
            expression);
}
```

Finally, C++ Builder users can use C++ Builder's closure type as a callback argument:

```
#include <string>
#include <map>
#include <boost/regex.hpp>
#include <functional>
// purpose:
// takes the contents of a file in the form of a string
// and searches for all the C++ class definitions, storing
// their locations in a map of strings/int's

typedef std::map<std::string, int, std::less<std::string> > map_type;
class class_index
{
   boost::regex expression;
   map_type index;
   std::string::const_iterator base;
   typedef  boost::smatch arg_type;
   bool grep_callback(const arg_type& what);
public:
   typedef bool (__closure* grep_callback_type)(const arg_type&);
   void IndexClasses(const std::string& file);
   class_index()
      : index(),
      expression("^(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
                 "(class|struct)[[:space:]]*(\\<\\w+\\>([[:blank:]]*\\([^)]*\\))?"
                 "[[:space:]]*)*(\\<\\w*\\>)[[:space:]]*(<[^;:{]+>[[:space:]]*)?"
                 "(\\{|:[^;\\{()]*\\{)"
                 ){}
};

bool class_index::grep_callback(const arg_type& what)
{
   // what[0] contains the whole string
// what[5] contains the class name.
// what[6] contains the template specialisation if any.
// add class name and position to map:
index[std::string(what[5].first, what[5].second) + std::string(what[6].first, what[6].second)] =
               what[5].first - base;
   return true;
}

void class_index::IndexClasses(const std::string& file)
{
   std::string::const_iterator start, end;
   start = file.begin();
   end = file.end();
   base = start;
   class_index::grep_callback_type cl = &(this->grep_callback);
   regex_grep(cl,
            start,
            end,
            expression);
}
```

regex_split (deprecated)

The algorithm `regex_split` has been deprecated in favor of the iterator `regex_token_iterator` which has a more flexible and powerful interface, as well as following the more usual standard library "pull" rather than "push" semantics.

Code which uses `regex_split` will continue to compile, the following documentation is taken from a previous Boost.Regex version:

```
#include <boost/regex.hpp>
```

Algorithm `regex_split` performs a similar operation to the perl split operation, and comes in three overloaded forms:

```
template <class OutputIterator, class charT, class Traits1, class Alloc1, class Traits2>
std::size_t regex_split(OutputIterator out,
                        std::basic_string<charT, Traits1, Alloc1>& s,
                        const basic_regex<charT, Traits2>& e,
                        boost::match_flag_type flags,
                        std::size_t max_split);

template <class OutputIterator, class charT, class Traits1, class Alloc1, class Traits2>
std::size_t regex_split(OutputIterator out,
                        std::basic_string<charT, Traits1, Alloc1>& s,
                        const basic_regex<charT, Traits2>& e,
                        boost::match_flag_type flags = match_default);

template <class OutputIterator, class charT, class Traits1, class Alloc1>
std::size_t regex_split(OutputIterator out,
                        std::basic_string<charT, Traits1, Alloc1>& s);
```

Effects: Each version of the algorithm takes an output-iterator for output, and a string for input. If the expression contains no marked sub-expressions, then the algorithm writes one string onto the output-iterator for each section of input that does not match the expression. If the expression does contain marked sub-expressions, then each time a match is found, one string for each marked sub-expression will be written to the output-iterator. No more than max_split strings will be written to the output-iterator. Before returning, all the input processed will be deleted from the string *s* (if *max_split* is not reached then all of *s* will be deleted). Returns the number of strings written to the output-iterator. If the parameter *max_split* is not specified then it defaults to `UINT_MAX`. If no expression is specified, then it defaults to "\s+", and splitting occurs on whitespace.

Throws: `std::runtime_error` if the complexity of matching the expression against an N character string begins to exceed $O(N^2)$, or if the program runs out of stack space while matching the expression (if Boost.Regex is configured in recursive mode), or if the matcher exhausts its permitted memory allocation (if Boost.Regex is configured in non-recursive mode).

Example: the following function will split the input string into a series of tokens, and remove each token from the string *s*:

```
unsigned tokenise(std::list<std::string>& l, std::string& s)
{
   return boost::regex_split(std::back_inserter(l), s);
}
```

Example: the following short program will extract all of the URL's from a html file, and print them out to cout:

```
#include <list>
#include <fstream>
#include <iostream>
#include <boost/regex.hpp>

boost::regex e("<\\s*A\\s+[^>]*href\\s*=\\s*\"([^\"]*)\"",
               boost::regbase::normal | boost::regbase::icase);

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   //
   // attempt to grow string buffer to match file size,
   // this doesn't always work...
   s.reserve(is.rdbuf()->gtin_avail());
   char c;
   while(is.get(c))
   {
      // use logarithmic growth stategy, in case
      // in_avail (above) returned zero:
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}


int main(int argc, char** argv)
{
   std::string s;
   std::list<std::string> l;

   for(int i = 1; i < argc; ++i)
   {
      std::cout << "Findings URL's in " << argv[i] << ":" << std::endl;
      s.erase();
      std::ifstream is(argv[i]);
      load_file(s, is);
      boost::regex_split(std::back_inserter(l), s, e);
      while(l.size())
      {
         s = *(l.begin());
         l.pop_front();
         std::cout << s << std::endl;
      }
   }
   return 0;
}
```

High Level Class RegEx (Deprecated)

The high level wrapper class RegEx is now deprecated and does not form part of the regular expression standardization proposal. This type still exists, and existing code will continue to compile, however the following documentation is unlikely to be further updated.

```
#include <boost/cregex.hpp>
```

The class RegEx provides a high level simplified interface to the regular expression library, this class only handles narrow character strings, and regular expressions always follow the "normal" syntax - that is the same as the perl / ECMAScript syntax.

```
typedef bool (*GrepCallback)(const RegEx& expression);
typedef bool (*GrepFileCallback)(const char* file, const RegEx& expression);
typedef bool (*FindFilesCallback)(const char* file);

class  RegEx
{
public:
   RegEx();
   RegEx(const RegEx& o);
   ~RegEx();
   RegEx(const char* c, bool icase = false);
   explicit RegEx(const std::string& s, bool icase = false);
   RegEx& operator=(const RegEx& o);
   RegEx& operator=(const char* p);
   RegEx& operator=(const std::string& s);
   unsigned int SetExpression(const char* p, bool icase = false);
   unsigned int SetExpression(const std::string& s, bool icase = false);
   std::string Expression()const;
   //
   // now matching operators:
   //
   bool Match(const char* p, boost::match_flag_type flags = match_default);
   bool Match(const std::string& s, boost::match_flag_type flags = match_default);
   bool Search(const char* p, boost::match_flag_type flags = match_default);
   bool Search(const std::string& s, boost::match_flag_type flags = match_default);
   unsigned int Grep(GrepCallback cb, const char* p,
                     boost::match_flag_type flags = match_default);
   unsigned int Grep(GrepCallback cb, const std::string& s,
                     boost::match_flag_type flags = match_default);
   unsigned int Grep(std::vector<std::string>& v, const char* p,
                     boost::match_flag_type flags = match_default);
   unsigned int Grep(std::vector<std::string>& v, const std::string& s,
                     boost::match_flag_type flags = match_default);
   unsigned int Grep(std::vector<unsigned int>& v, const char* p,
                     boost::match_flag_type flags = match_default);
   unsigned int Grep(std::vector<unsigned int>& v, const std::string& s,
                     boost::match_flag_type flags = match_default);
   unsigned int GrepFiles(GrepFileCallback cb, const char* files, bool recurse = false,
                          boost::match_flag_type flags = match_default);
   unsigned int GrepFiles(GrepFileCallback cb, const std::string& files,
                          bool recurse = false,
                          boost::match_flag_type flags = match_default);
   unsigned int FindFiles(FindFilesCallback cb, const char* files,
                          bool recurse = false,
                          boost::match_flag_type flags = match_default);
   unsigned int FindFiles(FindFilesCallback cb, const std::string& files,
                          bool recurse = false,
                          boost::match_flag_type flags = match_default);
   std::string Merge(const std::string& in, const std::string& fmt,
                     bool copy = true, boost::match_flag_type flags = match_default);
   std::string Merge(const char* in, const char* fmt, bool copy = true,
                     boost::match_flag_type flags = match_default);
   unsigned Split(std::vector<std::string>& v, std::string& s,
                  boost::match_flag_type flags = match_default,
                  unsigned max_count = ~0);
   //
   // now operators for returning what matched in more detail:
   //
   unsigned int Position(int i = 0)const;
   unsigned int Length(int i = 0)const;
   bool Matched(int i = 0)const;
   unsigned int Line()const;
```

```
   unsigned int Marks() const;
   std::string What(int i)const;
   std::string operator[](int i)const ;

   static const unsigned int npos;
};
```

Member functions for class RegEx are defined as follows:

Member	Description
`RegEx();`	Default constructor, constructs an instance of RegEx without any valid expression.
`RegEx(const RegEx& o);`	Copy constructor, all the properties of parameter *o* are copied.
`RegEx(const char* c, bool icase = false);`	Constructs an instance of RegEx, setting the expression to *c*, if *icase* is true then matching is insensitive to case, otherwise it is sensitive to case. Throws `bad_expression` on failure.
`RegEx(const std::string& s, bool icase = false);`	Constructs an instance of RegEx, setting the expression to *s*, if *icase* is true then matching is insensitive to case, otherwise it is sensitive to case. Throws `bad_expression` on failure.
`RegEx& operator=(const RegEx& o);`	Default assignment operator.
`RegEx& operator=(const char* p);`	Assignment operator, equivalent to calling `SetExpression(p, false)`. Throws `bad_expression` on failure.
`RegEx& operator=(const std::string& s);`	Assignment operator, equivalent to calling `SetExpression(s, false)`. Throws `bad_expression` on failure.
`unsigned int SetExpression(constchar* p, bool icase = false);`	Sets the current expression to *p*, if *icase* is true then matching is insensitive to case, otherwise it is sensitive to case. Throws `bad_expression` on failure.
`unsigned int SetExpression(const std::string& s, bool icase = false);`	Sets the current expression to *s*, if *icase* is true then matching is insensitive to case, otherwise it is sensitive to case. Throws `bad_expression` on failure.
`std::string Expression()const;`	Returns a copy of the current regular expression.
`bool Match(const char* p, boost::match_flag_type flags = match_default);`	Attempts to match the current expression against the text *p* using the match flags *flags* - see `match_flag_type`. Returns *true* if the expression matches the whole of the input string.
`bool Match(const std::string& s, boost::match_flag_type flags = match_default);`	Attempts to match the current expression against the text *s* using the `match_flag_type` *flags*. Returns *true* if the expression matches the whole of the input string.
`bool Search(const char* p, boost::match_flag_type flags = match_default);`	Attempts to find a match for the current expression somewhere in the text *p* using the `match_flag_type` *flags*. Returns *true* if the match succeeds.
`bool Search(const std::string& s, boost::match_flag_type flags = match_default);`	Attempts to find a match for the current expression somewhere in the text *s* using the `match_flag_type` flags. Returns *true* if the match succeeds.
`unsigned int Grep(GrepCallback cb, const char* p, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the text *p* using the `match_flag_type` *flags*. For each match found calls the call-back function cb as: `cb(*this);` If at any stage the call-back function returns *false* then the grep operation terminates, otherwise continues until no further matches are found. Returns the number of matches found.

Member	Description
`unsigned int Grep(GrepCallback cb, const std::string& s, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the text *s* using the `match_flag_type` flags. For each match found calls the call-back function cb as: `cb(*this);` If at any stage the call-back function returns false then the grep operation terminates, otherwise continues until no further matches are found. Returns the number of matches found.
`unsigned int Grep(std::vector<std::string>& v, const char* p, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the text *p* using the `match_flag_type` flags. For each match pushes a copy of what matched onto *v*. Returns the number of matches found.
`unsigned int Grep(std::vector<std::string>& v, const std::string& s, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the text *s* using the `match_flag_type` *flags*. For each match pushes a copy of what matched onto *v*. Returns the number of matches found.
`unsigned int Grep(std::vector<unsigned int>& v, const char* p, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the text *p* using the `match_flag_type` *flags*. For each match pushes the starting index of what matched onto *v*. Returns the number of matches found.
`unsigned int Grep(std::vector<unsigned int>& v, const std::string& s, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the text *s* using the `match_flag_type` *flags*. For each match pushes the starting index of what matched onto *v*. Returns the number of matches found.
`unsigned int GrepFiles(GrepFileCallback cb, const char* files, bool recurse = false, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the files files using the `match_flag_type` *flags*. For each match calls the call-back function cb. If the call-back returns false then the algorithm returns without considering further matches in the current file, or any further files. The parameter *files* can include wild card characters '*' and '?', if the parameter recurse is true then searches sub-directories for matching file names. Returns the total number of matches found. May throw an exception derived from `std::runtime_error` if file io fails.
`unsigned int GrepFiles(GrepFileCallback cb, const std::string& files, bool recurse = false, boost::match_flag_type flags = match_default);`	Finds all matches of the current expression in the files files using the `match_flag_type` *flags*. For each match calls the call-back function cb. If the call-back returns false then the algorithm returns without considering further matches in the current file, or any further files. The parameter *files* can include wild card characters '*' and '?', if the parameter recurse is true then searches sub-directories for matching file names. Returns the total number of matches found. May throw an exception derived from `std::runtime_error` if file io fails.

Member	Description
`unsigned int FindFiles(FindFilesCallback cb, const char* files, bool recurse = false, boost::match_flag_type flags = match_default);`	Searches files to find all those which contain at least one match of the current expression using the `match_flag_type` *flags*. For each matching file calls the call-back function cb. If the call-back returns false then the algorithm returns without considering any further files. The parameter *files* can include wild card characters '*' and '?', if the parameter *recurse* is true then searches sub-directories for matching file names. Returns the total number of files found. May throw an exception derived from `std::runtime_error` if file io fails.
`unsigned int FindFiles(FindFilesCallback cb, const std::string& files, bool recurse = false, boost::match_flag_type flags = match_default);`	Searches files to find all those which contain at least one match of the current expression using the `match_flag_type` *flags*. For each matching file calls the call-back function cb. If the call-back returns false then the algorithm returns without considering any further files. The parameter *files* can include wild card characters '*' and '?', if the parameter *recurse* is true then searches sub-directories for matching file names. Returns the total number of files found. May throw an exception derived from `std::runtime_error` if file io fails.
`std::string Merge(const std::string& in, const std::string& fmt, bool copy = true, boost::match_flag_type flags = match_default);`	Performs a search and replace operation: searches through the string *in* for all occurrences of the current expression, for each occurrence replaces the match with the format string *fmt*. Uses *flags* to determine what gets matched, and how the format string should be treated. If *copy* is true then all unmatched sections of input are copied unchanged to output, if the flag *format_first_only* is set then only the first occurance of the pattern found is replaced. Returns the new string. See also format string syntax, and `match_flag_type`.
`std::string Merge(const char* in, const char* fmt, bool copy = true, boost::match_flag_type flags = match_default);`	Performs a search and replace operation: searches through the string *in* for all occurrences of the current expression, for each occurrence replaces the match with the format string *fmt*. Uses *flags* to determine what gets matched, and how the format string should be treated. If *copy* is true then all unmatched sections of input are copied unchanged to output, if the flag *format_first_only* is set then only the first occurance of the pattern found is replaced. Returns the new string. See also format string syntax, and `match_flag_type`.

Member	Description
`unsigned Split(std::vector<std::string>& v, std::string& s, boost::match_flag_type flags = match_default, unsigned max_count = ~0);`	Splits the input string and pushes each one onto the vector. If the expression contains no marked sub-expressions, then one string is outputted for each section of the input that does not match the expression. If the expression does contain marked sub-expressions, then outputs one string for each marked sub-expression each time a match occurs. Outputs no more than *max_count* strings. Before returning, deletes from the input string *s* all of the input that has been processed (all of the string if *max_count* was not reached). Returns the number of strings pushed onto the vector.
`unsigned int Position(int i = 0)const;`	Returns the position of what matched sub-expression *i*. If `i = 0` then returns the position of the whole match. Returns `RegEx::npos` if the supplied index is invalid, or if the specified sub-expression did not participate in the match.
`unsigned int Length(int i = 0)const;`	Returns the length of what matched sub-expression i. If `i = 0` then returns the length of the whole match. Returns `RegEx::npos` if the supplied index is invalid, or if the specified sub-expression did not participate in the match.
`bool Matched(int i = 0)const;`	Returns true if sub-expression *i* was matched, false otherwise.
`unsigned int Line()const;`	Returns the line on which the match occurred, indexes start from 1 not zero, if no match occurred then returns `RegEx::npos`.
`unsigned int Marks() const;`	Returns the number of marked sub-expressions contained in the expression. Note that this includes the whole match (sub-expression zero), so the value returned is always >= 1.
`std::string What(int i)const;`	Returns a copy of what matched sub-expression *i*. If `i = 0` then returns a copy of the whole match. Returns a null string if the index is invalid or if the specified sub-expression did not participate in a match.
`std::string operator[](int i)const ;`	Returns `what(i);` Can be used to simplify access to sub-expression matches, and make usage more perl-like.

Internal Details

Unicode Iterators

Synopsis

```
#include <boost/regex/pending/unicode_iterator.hpp>
```

```
template <class BaseIterator, class U16Type = ::boost::uint16_t>
class u32_to_u16_iterator;

template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u16_to_u32_iterator;

template <class BaseIterator, class U8Type = ::boost::uint8_t>
class u32_to_u8_iterator;

template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u8_to_u32_iterator;

template <class BaseIterator>
class utf16_output_iterator;

template <class BaseIterator>
class utf8_output_iterator;
```

Description

This header contains a selection of iterator adaptors that make a sequence of characters in one encoding "look like" a read-only sequence of characters in another encoding.

```
template <class BaseIterator, class U16Type = ::boost::uint16_t>
class u32_to_u16_iterator
{
   u32_to_u16_iterator();
   u32_to_u16_iterator(BaseIterator start_position);

   // Other standard BidirectionalIterator members here...
};
```

A Bidirectional iterator adapter that makes an underlying sequence of UTF32 characters look like a (read-only) sequence of UTF16 characters. The UTF16 characters are encoded in the platforms native byte order.

```
template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u16_to_u32_iterator
{
   u16_to_u32_iterator();
   u16_to_u32_iterator(BaseIterator start_position);
   u16_to_u32_iterator(BaseIterator start_position, BaseIterator start_range, BaseIterat↵
or end_range);

   // Other standard BidirectionalIterator members here...
};
```

A Bidirectional iterator adapter that makes an underlying sequence of UTF16 characters (in the platforms native byte order) look like a (read-only) sequence of UTF32 characters.

The three-arg constructor of this class takes the start and end of the underlying sequence as well as the position to start iteration from. This constructor validates that the underlying sequence has validly encoded endpoints: this prevents accidentally incrementing/decrementing past the end of the underlying sequence as a result of invalid UTF16 code sequences at the endpoints of the underlying range.

```
template <class BaseIterator, class U8Type = ::boost::uint8_t>
class u32_to_u8_iterator
{
   u32_to_u8_iterator();
   u32_to_u8_iterator(BaseIterator start_position);

   // Other standard BidirectionalIterator members here...
};
```

A Bidirectional iterator adapter that makes an underlying sequence of UTF32 characters look like a (read-only) sequence of UTF8 characters.

```
template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u8_to_u32_iterator
{
   u8_to_u32_iterator();
   u8_to_u32_iterator(BaseIterator start_position);
   u8_to_u32_iterator(BaseIterator start_position, BaseIterator start_range, BaseIterat↵
or end_range);

   // Other standard BidirectionalIterator members here...
};
```

A Bidirectional iterator adapter that makes an underlying sequence of UTF8 characters look like a (read-only) sequence of UTF32 characters.

The three-arg constructor of this class takes the start and end of the underlying sequence as well as the position to start iteration from. This constructor validates that the underlying sequence has validly encoded endpoints: this prevents accidentally incrementing/decrementing past the end of the underlying sequence as a result of invalid UTF8 code sequences at the endpoints of the underlying range.

```
template <class BaseIterator>
class utf16_output_iterator
{
   utf16_output_iterator(const BaseIterator& b);
   utf16_output_iterator(const utf16_output_iterator& that);
   utf16_output_iterator& operator=(const utf16_output_iterator& that);

   // Other standard OutputIterator members here...
};
```

Simple OutputIterator adapter - accepts UTF32 values as input, and forwards them to *BaseIterator b* as UTF16. Both UTF32 and UTF16 values are in native byte order.

```
template <class BaseIterator>
class utf8_output_iterator
{
   utf8_output_iterator(const BaseIterator& b);
   utf8_output_iterator(const utf8_output_iterator& that);
   utf8_output_iterator& operator=(const utf8_output_iterator& that);

   // Other standard OutputIterator members here...
};
```

Simple OutputIterator adapter - accepts UTF32 values as input, and forwards them to *BaseIterator b* as UTF8. The UTF32 input values must be in native byte order.

Background Information

Headers

There are two main headers used by this library: `<boost/regex.hpp>` provides full access to the main template library, while `<boost/cregex.hpp>` provides access to the (deprecated) high level class RegEx, and the POSIX API functions.

There is also a header containing only forward declarations `<boost/regex_fwd.hpp>` for use when an interface is dependent upon `basic_regex`, but otherwise does not need the full definitions.

Localization

Boost.Regex provides extensive support for run-time localization, the localization model used can be split into two parts: front-end and back-end.

Front-end localization deals with everything which the user sees - error messages, and the regular expression syntax itself. For example a French application could change [[:word:]] to [[:mot:]] and \w to \m. Modifying the front end locale requires active support from the developer, by providing the library with a message catalogue to load, containing the localized strings. Front-end locale is affected by the LC_MESSAGES category only.

Back-end localization deals with everything that occurs after the expression has been parsed - in other words everything that the user does not see or interact with directly. It deals with case conversion, collation, and character class membership. The back-end locale does not require any intervention from the developer - the library will acquire all the information it requires for the current locale from the underlying operating system / run time library. This means that if the program user does not interact with regular expressions directly - for example if the expressions are embedded in your C++ code - then no explicit localization is required, as the library will take care of everything for you. For example embedding the expression [[:word:]]+ in your code will always match a whole word, if the program is run on a machine with, for example, a Greek locale, then it will still match a whole word, but in Greek characters rather than Latin ones. The back-end locale is affected by the LC_TYPE and LC_COLLATE categories.

There are three separate localization mechanisms supported by Boost.Regex:

Win32 localization model.

This is the default model when the library is compiled under Win32, and is encapsulated by the traits class `w32_regex_traits`. When this model is in effect each `basic_regex` object gets it's own LCID, by default this is the users default setting as returned by GetUserDefaultLCID, but you can call imbue on the `basic_regex` object to set it's locale to some other LCID if you wish. All the settings used by Boost.Regex are acquired directly from the operating system bypassing the C run time library. Front-end localization requires a resource dll, containing a string table with the user-defined strings. The traits class exports the function:

```
static std::string set_message_catalogue(const std::string& s);
```

which needs to be called with a string identifying the name of the resource dll, before your code compiles any regular expressions (but not necessarily before you construct any `basic_regex` instances):

```
boost::w32_regex_traits<char>::set_message_catalogue("mydll.dll");
```

The library provides full Unicode support under NT, under Windows 9x the library degrades gracefully - characters 0 to 255 are supported, the remainder are treated as "unknown" graphic characters.

C localization model.

This model has been deprecated in favor of the C++ locale for all non-Windows compilers that support it. This locale is encapsulated by the traits class `c_regex_traits`, Win32 users can force this model to take effect by defining the pre-processor symbol BOOST_REGEX_USE_C_LOCALE. When this model is in effect there is a single global locale, as set by `setlocale`. All settings are acquired from your run time library, consequently Unicode support is dependent upon your run time library implementation.

Front end localization is not supported.

Note that calling setlocale invalidates all compiled regular expressions, calling `setlocale(LC_ALL, "C")` will make this library behave equivalent to most traditional regular expression libraries including version 1 of this library.

C++ localization model.

This model is the default for non-Windows compilers.

When this model is in effect each instance of `basic_regex` has its own instance of `std::locale`, class `basic_regex` also has a member function `imbue` which allows the locale for the expression to be set on a per-instance basis. Front end localization requires a POSIX message catalogue, which will be loaded via the `std::messages` facet of the expression's locale, the traits class exports the symbol:

```
static std::string set_message_catalogue(const std::string& s);
```

which needs to be called with a string identifying the name of the message catalogue, before your code compiles any regular expressions (but not necessarily before you construct any basic_regex instances):

```
boost::cpp_regex_traits<char>::set_message_catalogue("mycatalogue");
```

Note that calling `basic_regex<>::imbue` will invalidate any expression currently compiled in that instance of `basic_regex`.

Finally note that if you build the library with a non-default localization model, then the appropriate pre-processor symbol (BOOST_REGEX_USE_C_LOCALE or BOOST_REGEX_USE_CPP_LOCALE) must be defined both when you build the support library, and when you include `<boost/regex.hpp>` or `<boost/cregex.hpp>` in your code. The best way to ensure this is to add the #define to `<boost/regex/user.hpp>`.

Providing a message catalogue

In order to localize the front end of the library, you need to provide the library with the appropriate message strings contained either in a resource dll's string table (Win32 model), or a POSIX message catalogue (C++ models). In the latter case the messages must appear in message set zero of the catalogue. The messages and their id's are as follows:

Message	id	Meaning	Default value
101	The character used to start a sub-expression.	"("	
102	The character used to end a sub-expression declaration.	")"	
103	The character used to denote an end of line assertion.	"$"	
104	The character used to denote the start of line assertion.	"^"	
105	The character used to denote the "match any character expression".	"."	
106	The match zero or more times repetition operator.	"*"	
107	The match one or more repetition operator.	"+"	
108	The match zero or one repetition operator.	"?"	
109	The character set opening character.	"["	
110	The character set closing character.	"]"	
111	The alternation operator.	"|"	
112	The escape character.	"\"	
113	The hash character (not currently used).	"#"	
114	The range operator.	"-"	
115	The repetition operator opening character.	"{"	
116	The repetition operator closing character.	"}"	
117	The digit characters.	"0123456789"	
118	The character which when preceded by an escape character represents the word boundary assertion.	"b"	
119	The character which when preceded by an escape character represents the non-word boundary assertion.	"B"	

Message	id	Meaning	Default value
120	The character which when preceded by an escape character represents the word-start boundary assertion.	"<"	
121	The character which when preceded by an escape character represents the word-end boundary assertion.	">"	
122	The character which when preceded by an escape character represents any word character.	"w"	
123	The character which when preceded by an escape character represents a non-word character.	"W"	
124	The character which when preceded by an escape character represents a start of buffer assertion.	"`A"	
125	The character which when preceded by an escape character represents an end of buffer assertion.	"'z"	
126	The newline character.	"\n"	
127	The comma separator.	","	
128	The character which when preceded by an escape character represents the bell character.	"a"	
129	The character which when preceded by an escape character represents the form feed character.	"f"	
130	The character which when preceded by an escape character represents the newline character.	"n"	
131	The character which when preceded by an escape character represents the carriage return character.	"r"	
132	The character which when preceded by an escape character represents the tab character.	"t"	
133	The character which when preceded by an escape character represents the vertical tab character.	"v"	

Message	id	Meaning	Default value
134	The character which when preceded by an escape character represents the start of a hexadecimal character constant.	"x"	
135	The character which when preceded by an escape character represents the start of an ASCII escape character.	"c"	
136	The colon character.	":"	
137	The equals character.	"="	
138	The character which when preceded by an escape character represents the ASCII escape character.	"e"	
139	The character which when preceded by an escape character represents any lower case character.	"l"	
140	The character which when preceded by an escape character represents any non-lower case character.	"L"	
141	The character which when preceded by an escape character represents any upper case character.	"u"	
142	The character which when preceded by an escape character represents any non-upper case character.	"U"	
143	The character which when preceded by an escape character represents any space character.	"s"	
144	The character which when preceded by an escape character represents any non-space character.	"S"	
145	The character which when preceded by an escape character represents any digit character.	"d"	
146	The character which when preceded by an escape character represents any non-digit character.	"D"	
147	The character which when preceded by an escape character represents the end quote operator.	"E"	

Message	id	Meaning	Default value
148	The character which when preceded by an escape character represents the start quote operator.	"Q"	
149	The character which when preceded by an escape character represents a Unicode combining character sequence.	"X"	
150	The character which when preceded by an escape character represents any single character.	"C"	
151	The character which when preceded by an escape character represents end of buffer operator.	"Z"	
152	The character which when preceded by an escape character represents the continuation assertion.	"G"	
153	The character which when preceeded by (? indicates a zero width negated forward lookahead assert.	!	

Custom error messages are loaded as follows:

Message ID	Error message ID	Default string
201	REG_NOMATCH	"No match"
202	REG_BADPAT	"Invalid regular expression"
203	REG_ECOLLATE	"Invalid collation character"
204	REG_ECTYPE	"Invalid character class name"
205	REG_EESCAPE	"Trailing backslash"
206	REG_ESUBREG	"Invalid back reference"
207	REG_EBRACK	"Unmatched [or "
209	REG_EBRACE	"Unmatched \{"
210	REG_BADBR	"Invalid content of \{\}"
211	REG_ERANGE	"Invalid range end"
212	REG_ESPACE	"Memory exhausted"
213	REG_BADRPT	"Invalid preceding regular expression"
214	REG_EEND	"Premature end of regular expression"
215	REG_ESIZE	"Regular expression too big"
216	REG_ERPAREN	"Unmatched) or \)"
217	REG_EMPTY	"Empty expression"
218	REG_E_UNKNOWN	"Unknown error"

[208	REG_EPAREN	"Unmatched (or \("

Custom character class names are loaded as followed:

Message ID	Description	Equivalent default class name
300	The character class name for alphanumeric characters.	"alnum"
301	The character class name for alphabetic characters.	"alpha"
302	The character class name for control characters.	"cntrl"
303	The character class name for digit characters.	"digit"
304	The character class name for graphics characters.	"graph"
305	The character class name for lower case characters.	"lower"
306	The character class name for printable characters.	"print"
307	The character class name for punctuation characters.	"punct"
308	The character class name for space characters.	"space"
309	The character class name for upper case characters.	"upper"
310	The character class name for hexadecimal characters.	"xdigit"
311	The character class name for blank characters.	"blank"
312	The character class name for word characters.	"word"
313	The character class name for Unicode characters.	"unicode"

Finally, custom collating element names are loaded starting from message id 400, and terminating when the first load thereafter fails. Each message looks something like: "tagname string" where tagname is the name used inside [[.tagname.]] and string is the actual text of the collating element. Note that the value of collating element [[.zero.]] is used for the conversion of strings to numbers - if you replace this with another value then that will be used for string parsing - for example use the Unicode character 0x0660 for [[.zero.]] if you want to use Unicode Arabic-Indic digits in your regular expressions in place of Latin digits.

Note that the POSIX defined names for character classes and collating elements are always available - even if custom names are defined, in contrast, custom error messages, and custom syntax messages replace the default ones.

Thread Safety

The Boost.Regex library is thread safe when Boost is: you can verify that Boost is in thread safe mode by checking to see if `BOOST_HAS_THREADS` is defined: this macro is set automatically by the config system when threading support is turned on in your compiler.

Class `basic_regex` and its typedefs regex and wregex are thread safe, in that compiled regular expressions can safely be shared between threads. The matching algorithms `regex_match`, `regex_search`, and `regex_replace` are all re-entrant and thread safe. Class `match_results` is now thread safe, in that the results of a match can be safely copied from one thread to another (for example one thread may find matches and push `match_results` instances onto a queue, while another thread pops them off the other end), otherwise use a separate instance of `match_results` per thread.

The POSIX API functions are all re-entrant and thread safe, regular expressions compiled with regcomp can also be shared between threads.

The class RegEx is only thread safe if each thread gets its own RegEx instance (apartment threading) - this is a consequence of RegEx handling both compiling and matching regular expressions.

Finally note that changing the global locale invalidates all compiled regular expressions, therefore calling `set_locale` from one thread while another uses regular expressions will produce unpredictable results.

There is also a requirement that there is only one thread executing prior to the start of main().

Test and Example Programs

Test Programs

regress:

A regression test application that gives the matching/searching algorithms a full workout. The presence of this program is your guarantee that the library will behave as claimed - at least as far as those items tested are concerned - if anyone spots anything that isn't being tested I'd be glad to hear about it.

Files:

- main.cpp
- basic_tests.cpp
- test_alt.cpp
- test_anchors.cpp
- test_asserts.cpp
- test_backrefs.cpp
- test_deprecated.cpp
- test_emacs.cpp
- test_escapes.cpp
- test_grep.cpp
- test_icu.cpp
- test_locale.cpp
- test_mfc.cpp

- test_non_greedy_repeats.cpp
- test_operators.cpp
- test_overloads.cpp
- test_perl_ex.cpp
- test_replace.cpp
- test_sets.cpp
- test_simple_repeats.cpp
- test_tricky_cases.cpp
- test_unicode.cpp

bad_expression_test:

Verifies that "bad" regular expressions don't cause the matcher to go into infinite loops, but to throw an exception instead.

Files: bad_expression_test.cpp.

recursion_test:

Verifies that the matcher can't overrun the stack (no matter what the expression).

Files: recursion_test.cpp.

concepts:

Verifies that the library meets all documented concepts (a compile only test).

Files: concept_check.cpp.

captures_test:

Test code for captures.

Files: captures_test.cpp.

Example programs

grep

A simple grep implementation, run with the -h command line option to find out its usage.

Files: grep.cpp

timer.exe

A simple interactive expression matching application, the results of all matches are timed, allowing the programmer to optimize their regular expressions where performance is critical.

Files: regex_timer.cpp.

Code snippets

The snippets examples contain the code examples used in the documentation:

captures_example.cpp: Demonstrates the use of captures.

credit_card_example.cpp: Credit card number formatting code.

partial_regex_grep.cpp: Search example using partial matches.

partial_regex_match.cpp: regex_match example using partial matches.

regex_iterator_example.cpp: Iterating through a series of matches.

regex_match_example.cpp: ftp based regex_match example.

regex_merge_example.cpp: regex_merge example: converts a C++ file to syntax highlighted HTML.

regex_replace_example.cpp: regex_replace example: converts a C++ file to syntax highlighted HTML

regex_search_example.cpp: regex_search example: searches a cpp file for class definitions.

regex_token_iterator_eg_1.cpp: split a string into a series of tokens.

regex_token_iterator_eg_2.cpp: enumerate the linked URL's in a HTML file.

The following are deprecated:

regex_grep_example_1.cpp: regex_grep example 1: searches a cpp file for class definitions.

regex_grep_example_2.cpp: regex_grep example 2: searches a cpp file for class definitions, using a global callback function.

regex_grep_example_3.cpp: regex_grep example 2: searches a cpp file for class definitions, using a bound member function callback.

regex_grep_example_4.cpp: regex_grep example 2: searches a cpp file for class definitions, using a C++ Builder closure as a callback.

regex_split_example_1.cpp: regex_split example: split a string into tokens.

regex_split_example_2.cpp : regex_split example: spit out linked URL's.

References and Further Information

Short tutorials on regular expressions can be found here and here.

The main book on regular expressions is Mastering Regular Expressions, published by O'Reilly.

Boost.Regex forms the basis for the regular expression chapter of the Technical Report on C++ Library Extensions.

The Open Unix Specification contains a wealth of useful material, including the POSIX regular expression syntax.

The Pattern Matching Pointers site is a "must visit" resource for anyone interested in pattern matching.

Glimpse and Agrep, use a simplified regular expression syntax to achieve faster search times.

Udi Manber and Ricardo Baeza-Yates both have a selection of useful pattern matching papers available from their respective web sites.

FAQ

Q. I can't get regex++ to work with escape characters, what's going on?

A. If you embed regular expressions in C++ code, then remember that escape characters are processed twice: once by the C++ compiler, and once by the Boost.Regex expression compiler, so to pass the regular expression \d+ to Boost.Regex, you need to embed "\d+" in your code. Likewise to match a literal backslash you will need to embed "\\" in your code.

Q. No matter what I do regex_match always returns false, what's going on?

A. The algorithm regex_match only succeeds if the expression matches **all** of the text, if you want to **find** a sub-string within the text that matches the expression then use regex_search instead.

Q. Why does using parenthesis in a POSIX regular expression change the result of a match?

A. For POSIX (extended and basic) regular expressions, but not for perl regexes, parentheses don't only mark; they determine what the best match is as well. When the expression is compiled as a POSIX basic or extended regex then Boost.Regex follows the POSIX standard leftmost longest rule for determining what matched. So if there is more than one possible match after considering the whole expression, it looks next at the first sub-expression and then the second sub-expression and so on. So...

"(0*)([0-9]*)" against "00123" would produce $1 = "00" $2 = "123"

where as

"0*([0-9])*" against "00123" would produce $1 = "00123"

If you think about it, had $1 only matched the "123", this would be "less good" than the match "00123" which is both further to the left and longer. If you want $1 to match only the "123" part, then you need to use something like:

"0*([1-9][0-9]*)"

as the expression.

Q. Why don't character ranges work properly (POSIX mode only)?

A. The POSIX standard specifies that character range expressions are locale sensitive - so for example the expression [A-Z] will match any collating element that collates between 'A' and 'Z'. That means that for most locales other than "C" or "POSIX", [A-Z] would match the single character 't' for example, which is not what most people expect - or at least not what most people have come to expect from regular expression engines. For this reason, the default behaviour of Boost.Regex (perl mode) is to turn locale sensitive collation off by not setting the `regex_constants::collate` compile time flag. However if you set a non-default compile time flag - for example `regex_constants::extended` or `regex_constants::basic`, then locale dependent collation will be enabled, this also applies to the POSIX API functions which use either `regex_constants::extended` or `regex_constants::basic` internally. [Note - when `regex_constants::nocollate` in effect, the library behaves "as if" the LC_COLLATE locale category were always "C", regardless of what its actually set to - end note].

Q. Why are there no throw specifications on any of the functions? What exceptions can the library throw?

A. Not all compilers support (or honor) throw specifications, others support them but with reduced efficiency. Throw specifications may be added at a later date as compilers begin to handle this better. The library should throw only three types of exception: [boost::regex_error] can be thrown by `basic_regex` when compiling a regular expression, `std::runtime_error` can be thrown when a call to `basic_regex::imbue` tries to open a message catalogue that doesn't exist, or when a call to `regex_search` or `regex_match` results in an "everlasting" search, or when a call to `RegEx::GrepFiles` or `RegEx::FindFiles` tries to open a file that cannot be opened, finally `std::bad_alloc` can be thrown by just about any of the functions in this library.

Q. Why can't I use the "convenience" versions of regex_match / regex_search / regex_grep / regex_format / regex_merge?

A. These versions may or may not be available depending upon the capabilities of your compiler, the rules determining the format of these functions are quite complex - and only the versions visible to a standard compliant compiler are given in the help. To find out what your compiler supports, run <boost/regex.hpp> through your C++ pre-processor, and search the output file for the function that you are interested in. Note however, that very few current compilers still have problems with these overloaded functions.

Performance

The performance of Boost.Regex in both recursive and non-recursive modes should be broadly comparable to other regular expression libraries: recursive mode is slightly faster (especially where memory allocation requires thread synchronisation), but not by much. The following pages compare Boost.Regex with various other regular expression libraries for the following compilers:

- Visual Studio.Net 2003 (recursive Boost.Regex implementation).
- Gcc 3.2 (cygwin) (non-recursive Boost.Regex implementation).

Standards Conformance

C++

Boost.Regex is intended to conform to the Technical Report on C++ Library Extensions.

ECMAScript / JavaScript

All of the ECMAScript regular expression syntax features are supported, except that:

The escape sequence \u matches any upper case character (the same as [[:upper:]]) rather than a Unicode escape sequence; use \x{DDDD} for Unicode escape sequences.

Perl

Almost all Perl features are supported, except for:

(?{code}) Not implementable in a compiled strongly typed language.

(??{code}) Not implementable in a compiled strongly typed language.

(*VERB) The backtracking control verbs are not recognised or implemented at this time.

In addition the following features behave slightly differently from Perl:

^ $ \Z These recognise any line termination sequence, and not just \n: see the Unicode requirements below.

POSIX

All the POSIX basic and extended regular expression features are supported, except that:

No character collating names are recognized except those specified in the POSIX standard for the C locale, unless they are explicitly registered with the traits class.

Character equivalence classes ([[=a=]] etc) are probably buggy except on Win32. Implementing this feature requires knowledge of the format of the string sort keys produced by the system; if you need this, and the default implementation doesn't work on your platform, then you will need to supply a custom traits class.

Unicode

The following comments refer to Unicode Technical Standard #18: Unicode Regular Expressions version 11.

Item	Feature	Support
1.1	Hex Notation	Yes: use \x{DDDD} to refer to code point UDDDD.
1.2	Character Properties	All the names listed under the General Category Property are supported. Script names and Other Names are not currently supported.

1.3	Subtraction and Intersection	Indirectly support by forward-lookahead:
		`(?=[[:X:]])[[:Y:]]`
		Gives the intersection of character properties X and Y.
		`(?![[:X:]])[[:Y:]]`
		Gives everything in Y that is not in X (subtraction).

1.4	Simple Word Boundaries	Conforming: non-spacing marks are included in the set of word characters.
1.5	Caseless Matching	Supported, note that at this level, case transformations are 1:1, many to many case folding operations are not supported (for example "ß" to "SS").
1.6	Line Boundaries	Supported, except that "." matches only one character of "\r\n". Other than that word boundaries match correctly; including not matching in the middle of a "\r\n" sequence.
1.7	Code Points	Supported: provided you use the u32* algorithms, then UTF-8, UTF-16 and UTF-32 are all treated as sequences of 32-bit code points.
2.1	Canonical Equivalence	Not supported: it is up to the user of the library to convert all text into the same canonical form as the regular expression.
2.2	Default Grapheme Clusters	Not supported.
2.3Default Word Boundaries	Not supported.	
2.4	Default Loose Matches	Not Supported.
2.5	Named Properties	Supported: the expression "[[:name:]]" or \N{name} matches the named character "name".
2.6	Wildcard properties	Not Supported.
3.1	Tailored Punctuation.	Not Supported.

Item	Feature	Support
3.2	Tailored Grapheme Clusters	Not Supported.
3.3	Tailored Word Boundaries.	Not Supported.
3.4	Tailored Loose Matches	Partial support: [[=c=]] matches characters with the same primary equivalence class as "c".
3.5	Tailored Ranges	Supported: [a-b] matches any character that collates in the range a to b, when the expression is constructed with the collate flag set.
3.6	Context Matches	Not Supported.
3.7	Incremental Matches	Supported: pass the flag `match_partial` to the regex algorithms.
3.8	Unicode Set Sharing	Not Supported.
3.9	Possible Match Sets	Not supported, however this information is used internally to optimise the matching of regular expressions, and return quickly if no match is possible.
3.10	Folded Matching	Partial Support: It is possible to achieve a similar effect by using a custom regular expression traits class.
3.11	Custom Submatch Evaluation	Not Supported.

Redistributables

If you are using Microsoft or Borland C++ and link to a dll version of the run time library, then you can choose to also link to a dll version of Boost.Regex by defining the symbol BOOST_REGEX_DYN_LINK when you compile your code. While these dll's are redistributable, there are no "standard" versions, so when installing on the users PC, you should place these in a directory private to your application, and not in the PC's directory path. Note that if you link to a static version of your run time library, then you will also link to a static version of Boost.Regex and no dll's will need to be distributed. The possible Boost.Regex dll and library names are computed according to the formula given in the getting started guide.

Note: you can disable automatic library selection by defining the symbol BOOST_REGEX_NO_LIB when compiling, this is useful if you want to build Boost.Regex yourself in your IDE, or if you need to debug Boost.Regex.

Acknowledgements

The author can be contacted at john - at - johnmaddock.co.uk; the home page for this library is at www.boost.org.

I am indebted to Robert Sedgewick's "Algorithms in C++" for forcing me to think about algorithms and their performance, and to the folks at boost for forcing me to think, period.

Eric Niebler, author of Boost.Expressive and the GRETA regular expression component, has shared several important ideas, in a series of long discussions.

Pete Becker, of Roundhouse Consulting, Ltd., has helped enormously with the standardisation proposal language.

The following people have all contributed useful comments or fixes: Dave Abrahams, Mike Allison, Edan Ayal, Jayashree Balasubramanian, Jan Bölsche, Beman Dawes, Paul Baxter, David Bergman, David Dennerline, Edward Diener, Peter Dimov, Robert Dunn, Fabio Forno, Tobias Gabrielsson, Rob Gillen, Marc Gregoire, Chris Hecker, Nick Hodapp, Jesse Jones, Martin Jost, Boris Krasnovskiy, Jan Hermelink, Max Leung, Wei-hao Lin, Jens Maurer, Richard Peters, Heiko Schmidt, Jason Shirk, Gerald Slacik, Scobie Smith, Mike Smyth, Alexander Sokolovsky, Hervé Poirier, Michael Raykh, Marc Recht, Scott VanCamp, Bruno Voigt, Alexey Voinov, Jerry Waldorf, Rob Ward, Lealon Watts, John Wismar, Thomas Witt and Yuval Yosef.

If I've missed your name off (I'm sure there are a few, just not who they are...) then please do get in touch.

I am also grateful to the manuals supplied with the Henry Spencer, PCRE, Perl and GNU regular expression libraries - wherever possible I have tried to maintain compatibility with these libraries and with the POSIX standard - the code however is entirely my own, including any bugs! I can absolutely guarantee that I will not fix any bugs I don't know about, so if you have any comments or spot any bugs, please get in touch.

History

New issues should be submitted at svn.boost.org - don't forget to include your email address in the ticket!

Currently open issues can be viewed here.

All issues including closed ones can be viewed here.

Boost.Regex-5.0.0 (Boost-1.56.0)

- Moved to library-specific version number post the move to Git. And since we have one (minor) breaking change this gets bumped up from v4 to v5.
- **Breaking change:** corrected behavior of `basic_regex<>::mark_count()` to match existing documentation, `basic_regex<>::subexpression(n)` changed to match, see #9227
- Fixed issue #8903.
- Fixed documentation typos from #9283.
- Fixed bug in collation code that failed if the locale generated collation strings with embedded nul's, see #9451.
- Apply patch for unusual thread usage (no statically initiallized mutexes), see #9461.
- Added better checks for invalid UTF-8 sequences, see #9473.

Boost-1.54

Fixed issue #8569.

Boost-1.53

Fixed Issues: #7744, #7644.

Boost-1.51

Fixed issues: #589, #7084, #7032, #6346.

Boost-1.50

Fixed issue with `(?!)` not being a valid expression, and updated docs on what constitutes a valid conditional expression.

Boost-1.48

Fixed issues: #698, #5835, #5958, #5736.

Boost 1.47

Fixed issues: #5223, #5353, #5363, #5462, #5472, #5504.

Boost 1.44

Fixed issues: #4309, #4215, #4212, #4191, #4132, #4123, #4114, #4036, #4020, #3941, #3902, #3890

Boost 1.42

- Added support for Functors rather than strings as format expressions.
- Improved error reporting when throwing exceptions to include better more relevant information.
- Improved performance and reduced stack usage of recursive expressions.
- Fixed tickets #2802, #3425, #3507, #3546, #3631, #3632, #3715, #3718, #3763, #3764

Boost 1.40

- Added support for many Perl 5.10 syntax elements including named sub-expressions, branch resets and recursive regular expressions.

Boost 1.38

- **Breaking change**: empty expressions, and empty alternatives are now allowed when using the Perl regular expression syntax. This change has been added for Perl compatibility, when the new `syntax_option_type` *no_empty_expressions* is set then the old behaviour is preserved and empty expressions are prohibited. This is issue #1081.
- Added support for Perl style ${n} expressions in format strings (issue #2556).
- Added support for accessing the location of sub-expressions within the regular expression string (issue #2269).
- Fixed compiler compatibility issues #2244, #2514, and #2458.

Boost 1.34

- Fix for non-greedy repeats and partial matches not working correctly in some cases.
- Fix for non-greedy repeats on VC++ not working in some cases (bug report 1515830).
- Changed match_results::position() to return a valid result when *this represents a partial match.
- Fixed the grep and egrep options so that the newline character gets treated the same as |.

Boost 1.33.1

- Fixed broken makefiles.
- Fixed configuration setup to allow building with VC7.1 - STLport-4.6.2 when using /Zc:wchar_t.
- Moved declarations class-inline in static_mutex.hpp so that SGI Irix compiler can cope.
- Added needed standard library #includes to fileiter.hpp, regex_workaround.hpp and cpp_regex_traits.hpp.
- Fixed a bug where non-greedy repeats could in certain strange circumstances repeat more times than their maximum value.
- Fixed the value returned by basic_regex<>::empty() from a default constructed object.
- Changed the definition of regex_error to make it backwards compatible with Boost-1.32.0.
- Disabled external templates for Intel C++ 8.0 and earlier - otherwise unresolved references can occur.

- Rewritten extern template code for gcc so that only specific member functions are exported: otherwise strange unresolved references can occur when linking and mixing debug and non-debug code.
- Initialise all the data members of the unicode_iterators: this keeps gcc from issuing needless warnings.
- Ported the ICU integration code to VC6 and VC7.
- Ensured code is STLport debug mode clean.
- Fixed lookbehind assertions so that fixed length repeats are permitted, and so that regex iteration allows lookbehind to look back before the current search range (into the last match).
- Fixed strange bug with non-greedy repeats inside forward lookahead assertions.
- Enabled negated character classes inside character sets.
- Fixed regression so that [a-z-] is a valid expression again.
- Fixed bug that allowed some invalid expressions to be accepted.

Boost 1.33.0

- Completely rewritten expression parsing code, and traits class support; now conforms to the standardization proposal.
- Breaking Change: The syntax options that can be passed to basic_regex constructors have been rationalized. The default option (perl) now has a value of zero, and it is now clearly documented which options apply to which regular expression syntax styles (perl, POSIX-extended, POSIX-basic etc). Some of the more esoteric options have now been removed, so there is the possibility that existing code may fail to compile: however equivalent functionality should still be available.
- Breaking Change: POSIX-extended and POSIX-basic regular expressions now enforce the letter of the POSIX standard much more closely than before.
- Added support for (?imsx-imsx) constructs.
- Added support for lookbehind expressions (?<=positive-lookbehind) and (?<!negative-lookbehind).
- Added support for conditional expressions (?(assertion)true-expresion|false-expression).
- Added MFC/ATL string wrappers.
- Added Unicode support; based on ICU.
- Changed newline support to recognise \f as a line separator (all character types), and \x85 as a line separator for wide characters / Unicode only.
- Added a new format flag format_literal that treats the replace string as a literal, rather than a Perl or Sed style format string.
- Errors are now reported by throwing exceptions of type regex_error. The types used previously - bad_expression and bad_pattern - are now just typedefs for regex_error. Type regex_error has a couple of new members: code() to report an error code rather than a string, and position() to report where in the expression the error occurred.

Boost 1.32.1

- Fixed bug in partial matches of bounded repeats of '.'.

Boost 1.31.0

- Completely rewritten pattern matching code - it is now up to 10 times faster than before.
- Reorganized documentation.

- Deprecated all interfaces that are not part of the regular expression standardization proposal.
- Added regex_iterator and regex_token_iterator .
- Added support for Perl style independent sub-expressions.
- Added non-member operators to the sub_match class, so that you can compare sub_match's with strings, or add them to a string to produce a new string.
- Added experimental support for extended capture information.
- Changed the match flags so that they are a distinct type (not an integer), if you try to pass the match flags as an integer rather than match_flag_type to the regex algorithms then you will now get a compiler error.