Regular expressions library

Defined in header <regex>

The regular expressions library provides a class that represents regular expressions , which are a kind of mini-language used to perform pattern matching within strings. Almost all operations with regexes can be characterized by operating on several of the following objects:

- **Target sequence**. The character sequence that is searched for a pattern. This may be a range specified by two iterators, a null-terminated character string or `std::string`.
- **Pattern**. This is the regular expression itself. It determines what constitutes a match. It is an object of type `std::basic_regex`, constructed from a string with special syntax. See syntax_option_type for the description of supported syntax variations.
- **Matched array**. The information about matches may be retrieved as an object of type `std::match_results`.
- **Replacement string**. This is a string that determines how to replace the matches, see match_flag_type for the description of supported syntax variations.

Main classes

These classes encapsulate a regular expression and the results of matching a regular expression within a target sequence of characters.

basic_regex (C++11)	regular expression object (class template)
sub_match (C++11)	identifies the sequence of characters matched by a sub-expression (class template)
match_results (C++11)	identifies one regular expression match, including all sub-expression matches (class template)

Algorithms

These functions are used to apply the regular expression encapsulated in a regex to a target sequence of characters.

regex_match (C++11)	attempts to match a regular expression to an entire character sequence (function template)
regex_search (C++11)	attempts to match a regular expression to any part of a character sequence (function template)
regex_replace (C++11)	replaces occurrences of a regular expression with formatted replacement text (function template)

Iterators

The regex iterators are used to traverse the entire set of regular expression matches found within a sequence.

regex_iterator (C++11)	iterates through all regex matches within a character sequence (class template)
regex_token_iterator (C++11)	iterates through the specified sub-expressions within all regex matches in a given string or through unmatched substrings (class template)

Exceptions

This class defines the type of objects thrown as exceptions to report errors from the regular expressions library.

regex_error (C++11)	reports errors generated by the regular expressions library (class)

Traits

The regex traits class is used to encapsulate the localizable aspects of a regex.

regex_traits (C++11)	provides metainformation about a character type, required by the regex library (class template)

Constants

Defined in namespace `std::regex_constants`

`syntax_option_type` (C++11)	general options controlling regex behavior (typedef)
`match_flag_type` (C++11)	options specific to matching (typedef)
`error_type` (C++11)	describes different types of matching errors (typedef)

Example

Run this code

```
#include <iostream>
#include <iterator>
#include <string>
#include <regex>

int main()
{
    std::string s = "Some people, when confronted with a problem, think "
        "\"I know, I'll use regular expressions.\" "
        "Now they have two problems.";

    std::regex self_regex("REGULAR EXPRESSIONS",
            std::regex_constants::ECMAScript | std::regex_constants::icase);
    if (std::regex_search(s, self_regex)) {
        std::cout << "Text contains the phrase 'regular expressions'\n";
    }

    std::regex word_regex("(\\S+)");
    auto words_begin =
        std::sregex_iterator(s.begin(), s.end(), word_regex);
    auto words_end = std::sregex_iterator();

    std::cout << "Found "
              << std::distance(words_begin, words_end)
              << " words\n";

    const int N = 6;
    std::cout << "Words longer than " << N << " characters:\n";
    for (std::sregex_iterator i = words_begin; i != words_end; ++i) {
        std::smatch match = *i;
        std::string match_str = match.str();
        if (match_str.size() > N) {
            std::cout << "  " << match_str << '\n';
        }
    }

    std::regex long_word_regex("(\\w{7,})");
    std::string new_s = std::regex_replace(s, long_word_regex, "[$&]");
    std::cout << new_s << '\n';
}
```

Output:

```
Text contains the phrase 'regular expressions'
Found 19 words
Words longer than 6 characters:
  people,
  confronted
  problem,
  regular
  expressions."
  problems.
Some people, when [confronted] with a [problem], think
"I know, I'll use [regular] [expressions]." Now they have two [problems].
```

std::basic_regex

Defined in header <regex>

```
template <
    class CharT,
    class Traits = std::regex_traits<CharT>
> class basic_regex;
```
(since C++11)

The class template `basic_regex` provides a general framework for holding regular expressions.

Several specializations for common character types are provided:

Defined in header <regex>

Type	Definition
`regex`	`basic_regex<char>`
`wregex`	`basic_regex<wchar_t>`

Member types

Member type	Definition
`value_type`	`CharT`
`traits_type`	`Traits`
`string_type`	`Traits::string_type`
`locale_type`	`Traits::locale_type`
`flag_type`	`std::regex_constants::syntax_option_type`

Member functions

(constructor)	constructs the regex object (public member function)
(destructor)	destructs the regex object (public member function)
`operator=`	assigns the contents (public member function)
`assign`	assigns the contents (public member function)

Observers

`mark_count`	returns the number of marked sub-expressions within the regular expression (public member function)
`flags`	returns the syntax flags (public member function)

Locale

`getloc`	get locale information (public member function)
`imbue`	set locale information (public member function)

Modifiers

`swap`	swaps the contents (public member function)

Constants

Value	Effect(s)
`icase`	Character matching should be performed without regard to case.
`nosubs`	When performing matches, all marked sub-expressions **`(`*`expr`*`)`** are treated as non-marking sub-expressions **`(?:`*`expr`*`)`**. No matches are stored in the supplied `std::regex_match` structure and `mark_count()` is zero
`optimize`	Instructs the regular expression engine to make matching faster, with the potential cost of making construction slower. For example, this might mean converting a non-deterministic FSA

	to a deterministic FSA.
`collate`	Character ranges of the form *"[a-b]"* will be locale sensitive.
`ECMAScript`	Use the Modified ECMAScript regular expression grammar
`basic`	Use the basic POSIX regular expression grammar (grammar documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03)).
`extended`	Use the extended POSIX regular expression grammar (grammar documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_04)).
`awk`	Use the regular expression grammar used by the *awk* utility in POSIX (grammar documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html#tag_20_06_13_04))
`grep`	Use the regular expression grammar used by the *grep* utility in POSIX. This is effectively the same as the `basic` option with the addition of newline '\n' as an alternation separator.
`egrep`	Use the regular expression grammar used by the *grep* utility, with the *-E* option, in POSIX. This is effectively the same as the `extended` option with the addition of newline '\n' as an alternation separator in addtion to '|'.

At most one grammar option must be chosen out of `ECMAScript`, `basic`, `extended`, `awk`, `grep`, `egrep`. If no grammar is chosen, `ECMAScript` is assumed to be selected. The other options serve as modifiers, such that `std::regex("meow", std::regex::icase)` is equivalent to `std::regex("meow", std::regex::ECMAScript|std::regex::icase)`

The member constants in `basic_regex` are duplicates of the syntax_option_type constants defined in the namespace `std::regex_constants`.

Non-member functions

`std::swap``(std::basic_regex)` (C++11)	specializes the `std::swap` algorithm (function template)

std::sub_match

Defined in header <regex>

```
template<
    class BidirIt      (since C++11)
> class sub_match;
```

The class template `sub_match` is used by the regular expression engine to denote sequences of characters matched by marked sub-expressions. A match is a `[begin, end)` pair within the target range matched by the regular expression, but with additional observer functions to enhance code clarity.

Only the default constructor is publicly accessible. Instances of `sub_match` are normally constructed and populated as a part of a `std::match_results` container during the processing of one of the regex algorithms.

The member functions return defined default values unless the `matched` member is `true`.

`sub_match` inherits from `std::pair<BidirIt, BidirIt>`, although it cannot be treated as a `std::pair` object because member functions such as swap and assignment will not work as expected.

Type requirements

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

Specializations

Several specializations for common character sequence types are provided:

Defined in header <regex>

Type	Definition
`csub_match`	`sub_match<const char*>`
`wcsub_match`	`sub_match<const wchar_t*>`
`ssub_match`	`sub_match<std::string::const_iterator>`
`wssub_match`	`sub_match<std::wstring::const_iterator>`

Member types

Member type	Definition
`iterator`	`BidirIt`
`value_type`	`std::iterator_traits<BidirIt>::value_type`
`difference_type`	`std::iterator_traits<BidirIt>::difference_type`
`string_type`	`std::basic_string<value_type>`

Member objects

`bool` **`matched`**	Indicates if this match was successful (public member object)

Inherited from std::pair

`BidirIt` **`first`**	Start of the match sequence. (public member object)
`BidirIt` **`second`**	One-past-the-end of the match sequence. (public member object)

Member functions

(constructor)	constructs the match object (public member function)

Observers

length	returns the length of the match (if any) (public member function)
str **operator string_type**	converts to the underlying string type (public member function)
compare	compares matched subsequence (if any) (public member function)

Non-member functions

operator== **operator!=** **operator<** **operator<=** **operator>** **operator>=**	compares two `sub_match` objects (function)
operator<<	outputs the matched character subsequence (function template)

See also

regex_token_iterator (C++11)	iterates through regex submatches (class template)

std::match_results

Defined in header <regex>

```
template<
    class BidirIt,
    class Alloc = std::allocator<std::sub_match<BidirIt>>
> class match_results;
```
(since C++11)

The class template **std::match_results** holds a collection of character sequences that represent the result of a regular expression match.

This is a specialized allocator-aware container. It can only be default created, obtained from std::regex_iterator, or modified by std::regex_search or std::regex_match. Because std::match_results holds std::sub_matches, each of which is a pair of iterators into the original character sequence that was matched, it's undefined behavior to examine std::match_results if the original character sequence was destroyed or iterators to it were invalidated for other reasons.

The first sub_match (index 0) contained in a match_result always represents the full match within a target sequence made by a regex, and subsequent sub_matches represent sub-expression matches corresponding in sequence to the left parenthesis delimiting the sub-expression in the regex.

Type requirements

- BidirIt must meet the requirements of BidirectionalIterator.
- Alloc must meet the requirements of Allocator.

Specializations

Several specializations for common character sequence types are provided:

Defined in header <regex>

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

Member types

Member type	Definition
allocator_type	Allocator
value_type	std::sub_match<BidirIt>
const_reference	const value_type&
reference	value_type&
const_iterator	*implementation defined* (depends on the underlying container)
iterator	const_iterator
difference_type	std::iterator_traits<BidirIt>::difference_type
size_type	std::allocator_traits<Alloc>::size_type
char_type	std::iterator_traits<BidirIt>::value_type
string_type	std::basic_string<char_type>

Member functions

(constructor)	constructs the object (public member function)
(destructor)	destructs the object (public member function)
operator=	assigns the contents (public member function)
get_allocator	returns the associated allocator (public member function)

State

ready	checks if the results are available (public member function)

Size

empty	checks whether the match was successful (public member function)
size	returns the number of matches in a fully-established result state (public member function)
max_size	returns the maximum possible number of sub-matches (public member function)

Element access

length	returns the length of the particular sub-match (public member function)
position	returns the position of the first character of the particular sub-match (public member function)
str	returns the sequence of characters for the particular sub-match (public member function)
operator[]	returns specified sub-match (public member function)
prefix	returns sub-sequence between the beginning of the target sequence and the beginning of the full match. (public member function)
suffix	returns sub-sequence between the end of the full match and the end of the target sequence (public member function)

Iterators

begin **cbegin**	returns iterator to the beginning of the list of sub-matches (public member function)
end **cend**	returns iterator to the end of the list of sub-matches (public member function)

Format

format	formats match results for output (public member function)

Modifiers

swap	swaps the contents (public member function)

Non-member functions

operator== **operator!=**	lexicographically compares the values in the two match result (function template)
std::swap(std::match_results) (C++11)	specializes the std::swap() algorithm (function template)

std::regex_match

Defined in header <regex>

```
template< class BidirIt,
          class Alloc, class CharT, class Traits >
bool regex_match( BidirIt first, BidirIt last,
                  std::match_results<BidirIt,Alloc>& m,
                  const std::basic_regex<CharT,Traits>& e,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default );
```
(1) (since C++11)

```
template< class BidirIt,
          class CharT, class Traits >
bool regex_match( BidirIt first, BidirIt last,
                  const std::basic_regex<CharT,Traits>& e,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default );
```
(2) (since C++11)

```
template< class CharT, class Alloc, class Traits >
bool regex_match( const CharT* str,
                  std::match_results<const CharT*,Alloc>& m,
                  const std::basic_regex<CharT,Traits>& e,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default );
```
(3) (since C++11)

```
template< class STraits, class SAlloc,
          class Alloc, class CharT, class Traits >
bool regex_match( const std::basic_string<CharT,STraits,SAlloc>& s,
                  std::match_results<
                      typename std::basic_string<CharT,STraits,SAlloc>::const_iterator,
                      Alloc
                  >& m,
                  const std::basic_regex<CharT,Traits>& e,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default );
```
(4) (since C++11)

```
template< class CharT, class Traits >
bool regex_match( const CharT* str,
                  const std::basic_regex<CharT,Traits>& e,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default );
```
(5) (since C++11)

```
template< class STraits, class SAlloc,
          class CharT, class Traits >
bool regex_match( const std::basic_string<CharT, STraits, SAlloc>& s,
                  const std::basic_regex<CharT,Traits>& e,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default );
```
(6) (since C++11)

```
template< class STraits, class SAlloc,
          class Alloc, class CharT, class Traits >
bool regex_match( const std::basic_string<CharT,STraits,SAlloc>&&,
                  std::match_results<
                      typename std::basic_string<CharT,STraits,SAlloc>::const_iterator,
                      Alloc
                  >&,
                  const std::basic_regex<CharT,Traits>&,
                  std::regex_constants::match_flag_type flags =
                      std::regex_constants::match_default ) = delete;
```
(7) (since C++14)

Determines if the regular expression `e1` matches the given target.

1) Determines if there is a match between the regular expression `e` and the entire target character sequence `[first,last)`, taking into account the effect of `flags`. Match results are returned in `m`.
2) Behaves as (1) above, omitting the match results.
3) Returns `std::regex_match(str, str + std::char_traits<charT>::length(str), m, e, flags)`.
4) Returns `std::regex_match(s.begin(), s.end(), m, e, flags)`.
5) Returns `std::regex_match(str, str + std::char_traits<charT>::length(str), e, flags)`.
6) Returns `std::regex_match(s.begin(), s.end(), e, flags)`.
7) The overload 4 is prohibited from accepting temporary strings, otherwise this function populates match_results m with string iterators that become invalid immediately.

Note that `regex_match` will only successfully match a regular expression to an *entire* character sequence, whereas `std::regex_search` will successfully match subsequences.

Parameters

first, last	-	the target character range to apply the regex to, given as iterators
m	-	the match results
str	-	the target string, given as a null-terminated C-style string
s	-	the target string, given as a `std::basic_string`
e	-	the regular expression
flags	-	flags used to determine how the match will be performed

Type requirements

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

Return value

Returns `true` if a match exists, `false` otherwise. In either case, the object `m` is updated, as follows:

If the match does not exist:

`m.ready() == true`
`m.empty() == true`
`m.size() == 0`

If the match exists:

`m.ready()`	`true`
`m.empty()`	`false`
`m.size()`	number of marked subexpressions plus 1, that is, `1+e.mark_count()`
`m.prefix().first`	`first`
`m.prefix().second`	`first`
`m.prefix().matched`	`false` (the match prefix is empty)
`m.suffix().first`	`last`
`m.suffix().second`	`last`
`m.suffix().matched`	`false` (the match suffix is empty)
`m[0].first`	`first`
`m[0].second`	`last`
`m[0].matched`	`true` (the entire sequence is matched)
`m[n].first`	the start of the sequence that matched marked sub-expression n, or `last` if the subexpression did not participate in the match
`m[n].second`	the end of the sequence that matched marked sub-expression n, or `last` if the subexpression did not participate in the match
`m[n].matched`	`true` if sub-expression n participated in the match, `false` otherwise

Example

Run this code

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    // Simple regular expression matching
    std::string fnames[] = {"foo.txt", "bar.txt", "baz.dat", "zoidberg"};
    std::regex txt_regex("[a-z]+\\.txt");

    for (const auto &fname : fnames) {
        std::cout << fname << ": " << std::regex_match(fname, txt_regex) << '\n';
    }

    // Extraction of a sub-match
    std::regex base_regex("([a-z]+)\\.txt");
    std::smatch base_match;
```

```
    for (const auto &fname : fnames) {
        if (std::regex_match(fname, base_match, base_regex)) {
            // The first sub_match is the whole string; the next
            // sub_match is the first parenthesized expression.
            if (base_match.size() == 2) {
                std::ssub_match base_sub_match = base_match[1];
                std::string base = base_sub_match.str();
                std::cout << fname << " has a base of " << base << '\n';
            }
        }
    }

    // Extraction of several sub-matches
    std::regex pieces_regex("([a-z]+)\\.([a-z]+)");
    std::smatch pieces_match;

    for (const auto &fname : fnames) {
        if (std::regex_match(fname, pieces_match, pieces_regex)) {
            std::cout << fname << '\n';
            for (size_t i = 0; i < pieces_match.size(); ++i) {
                std::ssub_match sub_match = pieces_match[i];
                std::string piece = sub_match.str();
                std::cout << "  submatch " << i << ": " << piece << '\n';
            }
        }
    }
}
```

Output:

```
foo.txt: 1
bar.txt: 1
baz.dat: 0
zoidberg: 0
foo.txt has a base of foo
bar.txt has a base of bar
foo.txt
  submatch 0: foo.txt
  submatch 1: foo
  submatch 2: txt
bar.txt
  submatch 0: bar.txt
  submatch 1: bar
  submatch 2: txt
baz.dat
  submatch 0: baz.dat
  submatch 1: baz
  submatch 2: dat
```

See also

basic_regex (C++11)	regular expression object (class template)
match_results (C++11)	identifies one regular expression match, including all sub-expression matches (class template)
regex_search (C++11)	attempts to match a regular expression to any part of a character sequence (function template)

std::regex_search

Defined in header <regex>

```
template< class BidirIt,
          class Alloc, class CharT, class Traits >
bool regex_search( BidirIt first, BidirIt last,
                   std::match_results<BidirIt,Alloc>& m,
                   const std::basic_regex<CharT,Traits>& e,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(1) (since C++11)

```
template< class Alloc, class CharT, class Traits >
bool regex_search( const CharT* str,
                   std::match_results<BidirIt,Alloc>& m,
                   const std::basic_regex<CharT,Traits>& e,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(2) (since C++11)

```
template< class STraits, class SAlloc,
          class Alloc, class CharT, class Traits >
bool regex_search( const std::basic_string<CharT,STraits,SAlloc>& s,
                   std::match_results<
                       typename std::basic_string<CharT,STraits,SAlloc>::const_iterator,
                       Alloc
                   >& m,
                   const std::basic_regex<CharT, Traits>& e,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(3) (since C++11)

```
template< class BidirIt,
          class CharT, class Traits >
bool regex_search( BidirIt first, BidirIt last,
                   const std::basic_regex<CharT,Traits>& e,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(4) (since C++11)

```
template< class CharT, class Traits >
bool regex_search( const CharT* str,
                   const std::basic_regex<CharT,Traits>& e,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(5) (since C++11)

```
template< class STraits, class SAlloc,
          class CharT, class Traits >
bool regex_search( const std::basic_string<CharT,STraits,SAlloc>& s,
                   const std::basic_regex<CharT,Traits>& e,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(6) (since C++11)

```
template< class STraits, class SAlloc,
          class Alloc, class CharT, class Traits >
bool regex_search( const std::basic_string<CharT,STraits,SAlloc>&&,
                   std::match_results<
                       typename std::basic_string<CharT,STraits,SAlloc>::const_iterator,
                       Alloc
                   >&,
                   const std::basic_regex<CharT, Traits>&,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default ) = delete;
```
(7) (since C++14)

Determines if there is a match between the regular expression `e` and some subsequence in the target character sequence.

1) Analyzes generic range `[first,last)`. Match results are returned in `m`.
2) Analyzes a null-terminated string pointed to by `str`. Match results are returned in `m`.
3) Analyzes a string `s`. Match results are returned in `m`.
4-6) Equivalent to (1-3), just omits the match results.
7) The overload 3 is prohibited from accepting temporary strings, otherwise this function populates match_results m with string iterators that become invalid immediately.

`regex_search` will successfully match any subsequence of the given sequence, whereas `std::regex_match` will only return `true` if the regular expression matches the *entire* sequence.

Parameters

first, last - a range identifying the target character sequence
str - a pointer to a null-terminated target character sequence

s - a string identifying target character sequence
e - the `std::regex` that should be applied to the target character sequence
m - the match results
flags - `std::regex_constants::match_flag_type` governing search behavior

Type requirements

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

Return value

Returns `true` if a match exists, `false` otherwise. In either case, the object `m` is updated, as follows:

If the match does not exist:

`m.ready() == true`
`m.empty() == true`
`m.size() == 0`

If the match exists:

`m.ready()`	`true`
`m.empty()`	`false`
`m.size()`	number of marked subexpressions plus 1, that is, `1+e.mark_count()`
`m.prefix().first`	`first`
`m.prefix().second`	`m[0].first`
`m.prefix().matched`	`m.prefix().first != m.prefix().second`
`m.suffix().first`	`m[0].second`
`m.suffix().second`	`last`
`m.suffix().matched`	`m.suffix().first != m.suffix().second`
`m[0].first`	the start of the matching sequence
`m[0].second`	the end of the matching sequence
`m[0].matched`	`true`
`m[n].first`	the start of the sequence that matched marked sub-expression n, or `last` if the subexpression did not participate in the match
`m[n].second`	the end of the sequence that matched marked sub-expression n, or `last` if the subexpression did not participate in the match
`m[n].matched`	`true` if sub-expression n participated in the match, `false` otherwise

Example

Run this code

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::string lines[] = {"Roses are #ff0000",
                           "violets are #0000ff",
                           "all of my base are belong to you"};

    std::regex color_regex("#([a-f0-9]{2})"
                           "([a-f0-9]{2})"
                           "([a-f0-9]{2})");

    for (const auto &line : lines) {
        std::cout << line << ": "
                  << std::regex_search(line, color_regex) << '\n';
    }

    std::smatch color_match;
    for (const auto &line : lines) {
        std::regex_search(line, color_match, color_regex);
        std::cout << "matches for '" << line << "'\n";
```

```
        for (size_t i = 0; i < color_match.size(); ++i)
            std::cout << i << ": " << color_match[i] << '\n';
    }
}
```

Output:

```
Roses are #ff0000: 1
violets are #0000ff: 1
all of my base are belong to you: 0
matches for 'Roses are #ff0000'
0: #ff0000
1: ff
2: 00
3: 00
matches for 'violets are #0000ff'
0: #0000ff
1: 00
2: 00
3: ff
matches for 'all of my base are belong to you'
```

See also

basic_regex (C++11)	regular expression object (class template)
match_results (C++11)	identifies one regular expression match, including all sub-expression matches (class template)
regex_match (C++11)	attempts to match a regular expression to an entire character sequence (function template)

std::regex_replace

Defined in header <regex>

```
template< class OutputIt, class BidirIt,
          class Traits, class CharT,
          class STraits, class SAlloc >
OutputIt regex_replace( OutputIt out, BidirIt first, BidirIt last,
                        const std::basic_regex<CharT,Traits>& re,
                        const std::basic_string<CharT,STraits,SAlloc>& fmt,
                        std::regex_constants::match_flag_type flags =
                            std::regex_constants::match_default );
```
(1) (since C++11)

```
template< class OutputIt, class BidirIt,
          class Traits, class CharT >
OutputIt regex_replace( OutputIt out, BidirIt first, BidirIt last,
                        const std::basic_regex<CharT,Traits>& re,
                        const CharT* fmt,
                        std::regex_constants::match_flag_type flags =
                            std::regex_constants::match_default );
```
(2) (since C++11)

```
template< class Traits, class CharT,
          class STraits, class SAlloc,
          class FTraits, class FAlloc >
std::basic_string<CharT,STraits,SAlloc>
    regex_replace( const std::basic_string<CharT,STraits,SAlloc>& s,
                   const std::basic_regex<CharT,Traits>& re,
                   const std::basic_string<CharT,FTraits,FAlloc>& fmt,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(3) (since C++11)

```
template< class Traits, class CharT,
          class STraits, class SAlloc >
std::basic_string<CharT,STraits,SAlloc>
    regex_replace( const std::basic_string<CharT,STraits,SAlloc>& s,
                   const std::basic_regex<CharT,Traits>& re,
                   const CharT* fmt,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(4) (since C++11)

```
template< class Traits, class CharT,
          class STraits, class SAlloc >
std::basic_string<CharT>
    regex_replace( const CharT* s,
                   const std::basic_regex<CharT,Traits>& re,
                   const std::basic_string<CharT,STraits,SAlloc>& fmt,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(5) (since C++11)

```
template< class Traits, class CharT >
std::basic_string<CharT>
    regex_replace( const CharT* s,
                   const std::basic_regex<CharT,Traits>& re,
                   const CharT* fmt,
                   std::regex_constants::match_flag_type flags =
                       std::regex_constants::match_default );
```
(6) (since C++11)

`regex_replace` uses a regular expression to perform substitution on a sequence of characters:

1) Copies characters in the range `[first,last)` to `out`, replacing any sequences that match `re` with characters formatted by `fmt`. In other words:

- Constructs a `std::regex_iterator` object `i` as if by `std::regex_iterator<BidirIt, CharT, traits> i(first, last, re, flags)`, and uses it to step through every match of `re` within the sequence `[first,last)`.
- For each such match `m`, copies the non-matched subsequence (`m.prefix()`) into `out` as if by `out = std::copy(m.prefix().first, m.prefix().second, out)` and then replaces the matched subsequence with the formatted replacement string as if by calling `out = m.format(out, fmt, flags)`.
- When no more matches are found, copies the remaining non-matched characters to `out` as if by `out = std::copy(last_m.suffix().first, last_m.suffix().second, out)` where `last_m` is a copy of the last match found.
- If there are no matches, copies the entire sequence into `out` as-is, by `out = std::copy(first, last, out)`
- If `flags` contains `std::regex_constants::format_no_copy`, the non-matched subsequences are not copied into `out`.
- If `flags` contains `std::regex_constants::format_first_only`, only the first match is replaced.

2) same as 1), but the formatted replacement is performed as if by calling `out = m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)`

3-4) Constructs an empty string `result` of type `std::basic_string<CharT, ST, SA>` and calls `std::regex_replace(std::back_inserter(result), s.begin(), s.end(), re, fmt, flags)`.

5-6) Constructs an empty string `result` of type `std::basic_string<CharT>` and calls `std::regex_replace(std::back_inserter(result), s, s + std::char_traits<CharT>::length(s), re, fmt, flags)`.

Parameters

first, last	-	the input character sequence, represented as a pair of iterators
s	-	the input character sequence, represented as `std::basic_string` or character array
re	-	the `std::basic_regex` that will be matched against the input sequence
flags	-	the match flags of type `std::regex_constants::match_flag_type`
fmt	-	the regex replacement format string, exact syntax depends on the value of `flags`
out	-	output iterator to store the result of the replacement

Type requirements

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

Return value

1-2) Returns a copy of the output iterator `out` after all the insertions.

3-6) Returns the string `result` which contains the output.

Exceptions

May throw `std::regex_error` to indicate an error condition.

Example

Run this code

```
#include <iostream>
#include <iterator>
#include <regex>
#include <string>

int main()
{
   std::string text = "Quick brown fox";
   std::regex vowel_re("a|e|i|o|u");

   // write the results to an output iterator
   std::regex_replace(std::ostreambuf_iterator<char>(std::cout),
                      text.begin(), text.end(), vowel_re, "*");

   // construct a string holding the results
   std::cout << '\n' << std::regex_replace(text, vowel_re, "[$&]") << '\n';
}
```

Output:

```
Q**ck br*wn f*x
Q[u][i]ck br[o]wn f[o]x
```

See also

regex_search (C++11)	attempts to match a regular expression to any part of a character sequence (function template)
match_flag_type (C++11)	options specific to matching (typedef)

std::regex_iterator

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

`std::regex_iterator` is a read-only `ForwardIterator` that accesses the individual matches of a regular expression within the underlying character sequence.

On construction, and on every increment, it calls `std::regex_search` and remembers the result (that is, saves a copy of the value `std::match_results<BidirIt>`). The first object may be read when the iterator is constructed or when the first dereferencing is done. Otherwise, dereferencing only returns a copy of the most recently obtained regex match.

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

A typical implementation of `std::regex_iterator` holds the begin and the end iterators for the underlying sequence (two instances of BidirIt), a pointer to the regular expression (`const regex_type*`) and the match flags (`std::regex_constants::match_flag_type`), and the current match (`std::match_results<BidirIt>`).

Type requirements

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

Specializations

Several specializations for common character sequence types are defined:

Defined in header `<regex>`

Type	Definition
`cregex_iterator`	`regex_iterator<const char*>`
`wcregex_iterator`	`regex_iterator<const wchar_t*>`
`sregex_iterator`	`regex_iterator<std::string::const_iterator>`
`wsregex_iterator`	`regex_iterator<std::wstring::const_iterator>`

Member types

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

Member functions

(constructor)	constructs a new `regex_iterator` (public member function)
(destructor) (implicitly declared)	destructs a `regex_iterator`, including the cached value (public member function)
`operator=`	assigns contents (public member function)
`operator==` **`operator!=`**	compares two `regex_iterators` (public member function)
`operator*` **`operator->`**	accesses the current match (public member function)
	advances the iterator to the next match

operator++ **operator++**(int)	(public member function)

Notes

It is the programmer's responsibility to ensure that the `std::basic_regex` object passed to the iterator's constructor outlives the iterator. Because the iterator stores a pointer to the regex, incrementing the iterator after the regex was destroyed accesses a dangling pointer.

If the part of the regular expression that matched is just an assertion (`^`, `$`, `\b`, `\B`), the match stored in the iterator is a zero-length match, that is, `match[0].first == match[0].second`.

Example

Run this code

```
#include <regex>
#include <iterator>
#include <iostream>
#include <string>

int main()
{
    const std::string s = "Quick brown fox.";

    std::regex words_regex("[^\\s]+");
    auto words_begin =
        std::sregex_iterator(s.begin(), s.end(), words_regex);
    auto words_end = std::sregex_iterator();

    std::cout << "Found "
              << std::distance(words_begin, words_end)
              << " words:\n";

    for (std::sregex_iterator i = words_begin; i != words_end; ++i) {
        std::smatch match = *i;
        std::string match_str = match.str();
        std::cout << match_str << '\n';
    }
}
```

Output:

```
Found 3 words:
Quick
brown
fox.
```

See also

match_results (C++11)	identifies one regular expression match, including all sub-expression matches (class template)
regex_search (C++11)	check if a regular expression occurs anywhere within a string (function template)

std::regex_token_iterator

Defined in header <regex>

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

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

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

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

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

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

Type requirements

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

Specializations

Several specializations for common character sequence types are defined:

Defined in header <regex>

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

Member types

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

Member functions

(constructor)	constructs a new `regex_token_iterator` (public member function)
(destructor) (implicitly declared)	destructs a `regex_token_iterator`, including the cached value (public member function)
`operator=`	assigns contents

	(public member function)
operator== **operator!=**	compares two `regex_token_iterator`s (public member function)
operator* **operator->**	accesses current submatch (public member function)
operator++ **operator++**`(int)`	advances the iterator to the next submatch (public member function)

Notes

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

Example

Run this code

```
#include <fstream>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <regex>

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

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

Output:

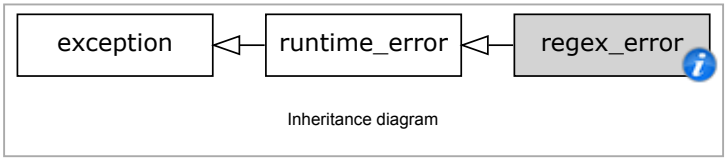
```
Quick
brown
fox.
http://google.com
http://cppreference.com
```

std::regex_error

Defined in header <regex>

```
class regex_error;    (since C++11)
```

Defines the type of exception object thrown to report errors in the regular expressions library.



Inheritance diagram

Member functions

(constructor)	constructs a `regex_error` object (public member function)
code	gets the `std::regex_constants::error_type` for a `regex_error` (public member function)

Inherited from std::exception

Member functions

(destructor) [virtual]	destructs the exception object (virtual public member function of `std::exception`)
what [virtual]	returns an explanatory string (virtual public member function of `std::exception`)

Example

Run this code

```
#include <regex>
#include <iostream>

int main()
{
    try {
        std::regex re("[a-b][a");
    }

    catch (const std::regex_error& e) {
        std::cout << "regex_error caught: " << e.what() << '\n';
        if (e.code() == std::regex_constants::error_brack) {
            std::cout << "The code was error_brack\n";
        }
    }
}
```

Output:

```
regex_error caught: The expression contained mismatched [ and ].
The code was error_brack
```

std::regex_traits

Defined in header <regex>

```
template< class CharT >
class regex_traits;
```
(since C++11)

The type trait template `regex_traits` supplies `std::basic_regex` with the set of types and functions necessary to operate on the type `CharT`.

Since many of regex operations are locale-sensitive (when `std::regex_constants::collate` flag is set), the regex_traits class typically holds an instance of a `std::locale` as a private member.

Standard specializations

Two specializations of `std::regex_traits` are defined by the standard library:

`std::regex_traits<char>`
`std::regex_traits<wchar_t>`

These specializations make it possible to use `std::basic_regex<char>` (aka `std::regex`) and `std::basic_regex<wchar_t>` (aka `std::wregex`), but in order to use, for example, `std::basic_regex<char32_t>`, user-provided specializtion `std::regex_traits<char32_t>` needs to be defined.

Member types

Type	Definition
`char_type`	`CharT`
`string_type`	`std::basic_string<CharT>`
`locale_type`	The locale used for localized behavior in the regular expression. Must be `CopyConstructible`
`char_class_type`	Represents a character classification and is capable of holding an implementation specific set returned by `lookup_classname`. Must be a `BitmaskType`.

Member functions

(constructor)	constructs the regex_traits object (public member function)
`length` [static]	calculates the length of a null-terminated character string (public static member function)
`translate`	determines the equivalence key for a character (public member function)
`translate_nocase`	determines the case-insensitive equivalence key for a character (public member function)
`transform`	determines the sort key for the given string, used to provide collation order (public member function)
`transform_primary`	determines the primary sort key for the character sequence, used to determine equivalence class (public member function)
`lookup_collatename`	gets a collation element by name (public member function)
`lookup_classname`	gets a character class by name (public member function)
`isctype`	indicates membership in a localized character class (public member function)
`value`	translates the character representing a numeric digit into an integral value (public member function)
`imbue`	sets the locale (public member function)
`getloc`	gets the locale (public member function)

std::regex_constants::syntax_option_type

Defined in header <regex>

```
typedef /*unspecified*/ syntax_option_type;
constexpr syntax_option_type icase = /*unspecified*/;
constexpr syntax_option_type nosubs = /*unspecified*/;
constexpr syntax_option_type optimize = /*unspecified*/;
constexpr syntax_option_type collate = /*unspecified*/;
constexpr syntax_option_type ECMAScript = /*unspecified*/;
constexpr syntax_option_type basic = /*unspecified*/;
constexpr syntax_option_type extended = /*unspecified*/;
constexpr syntax_option_type awk = /*unspecified*/;
constexpr syntax_option_type grep = /*unspecified*/;
constexpr syntax_option_type egrep = /*unspecified*/;
```

The `syntax_option_type` is a `BitmaskType` that contains options that govern how regular expressions behave.

The possible values for this type (`icase`, `optimize`, etc.) are duplicated inside std::basic_regex.

Constants

Value	Effect(s)
`icase`	Character matching should be performed without regard to case.
`nosubs`	When performing matches, all marked sub-expressions `(`***`expr`***`)` are treated as non-marking sub-expressions `(?:`***`expr`***`)`. No matches are stored in the supplied `std::regex_match` structure and `mark_count()` is zero
`optimize`	Instructs the regular expression engine to make matching faster, with the potential cost of making construction slower. For example, this might mean converting a non-deterministic FSA to a deterministic FSA.
`collate`	Character ranges of the form *"[a-b]"* will be locale sensitive.
`ECMAScript`	Use the Modified ECMAScript regular expression grammar
`basic`	Use the basic POSIX regular expression grammar (grammar documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_03)).
`extended`	Use the extended POSIX regular expression grammar (grammar documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_04)).
`awk`	Use the regular expression grammar used by the *awk* utility in POSIX (grammar documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html#tag_20_06_13_04))
`grep`	Use the regular expression grammar used by the *grep* utility in POSIX. This is effectively the same as the `basic` option with the addition of newline '\n' as an alternation separator.
`egrep`	Use the regular expression grammar used by the *grep* utility, with the *-E* option, in POSIX. This is effectively the same as the `extended` option with the addition of newline '\n' as an alternation separator in addtion to '|'.

At most one grammar option must be chosen out of `ECMAScript`, `basic`, `extended`, `awk`, `grep`, `egrep`. If no grammar is chosen, `ECMAScript` is assumed to be selected. The other options serve as modifiers, such that `std::regex("meow", std::regex::icase)` is equivalent to `std::regex("meow", std::regex::ECMAScript|std::regex::icase)`

Notes

Because POSIX uses "leftmost longest" matching rule (the longest matching subsequence is matched, and if there are several such subsequences, the first one is matched), it is not suitable, for example, for parsing markup languages: a POSIX regex such as `"<tag[^>]*>.*</tag>"` would match everything from the first `"<tag"` to the last `"</tag>"`, including every `"</tag>"` and `"<tag>"` inbetween. On the other hand, ECMAScript supports non-greedy matches, and the ECMAScript regex `"<tag[^>]*>.*?</tag>"` would match only until the first closing tag.

In C++11, these constants were specified with redundant keyword `static`, which was removed by C++14 via LWG issue 2053 (http%3A//www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#2053)

Example

Illustrates the difference in the matching algorithm between ECMAScript and POSIX regular expressions

Run this code

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::string str = "zzxayyzz";
    std::regex re1(".*(a|xayy)"); // ECMA
    std::regex re2(".*(a|xayy)", std::regex::extended); // POSIX

    std::cout << "Searching for .*(a|xayy) in zzxayyzz:\n";
    std::smatch m;
    std::regex_search(str, m, re1);
    std::cout << " ECMA (depth first search) match: " << m[0] << '\n';
    std::regex_search(str, m, re2);
    std::cout << " POSIX (leftmost longest)  match: " << m[0] << '\n';
}
```

Output:

```
Searching for .*(a|xayy) in zzxayyzz:
 ECMA (depth first search) match: zzxa
 POSIX (leftmost longest)  match: zzxayy
```

See also

basic_regex (C++11)	regular expression object (class template)

std::regex_constants::match_flag_type

Defined in header <regex>

```
typedef /*unspecified*/ match_flag_type;
constexpr match_flag_type match_default = 0;
constexpr match_flag_type match_not_bol = /*unspecified*/;
constexpr match_flag_type match_not_eol = /*unspecified*/;
constexpr match_flag_type match_not_bow = /*unspecified*/;
constexpr match_flag_type match_not_eow = /*unspecified*/;
constexpr match_flag_type match_any = /*unspecified*/;
constexpr match_flag_type match_not_null = /*unspecified*/;
constexpr match_flag_type match_continuous = /*unspecified*/;
constexpr match_flag_type match_prev_avail = /*unspecified*/;
constexpr match_flag_type format_default = 0;
constexpr match_flag_type format_sed = /*unspecified*/;
constexpr match_flag_type format_no_copy = /*unspecified*/;
constexpr match_flag_type format_first_only = /*unspecified*/;
```

`match_flag_type` is a `BitmaskType` that specifies additional regular expression matching options.

Constants

Note: `[first, last)` refers to the character sequence being matched.

Constant	Explanation
`match_not_bol`	The first character in *[first,last)* will be treated as if it is **not** at the beginning of a line (i.e. ^ will not match *[first,first)*
`match_not_eol`	The last character in *[first,last)* will be treated as if it is **not** at the end of a line (i.e. *$* will not match *[last,last)*
`match_not_bow`	*"\b"* will not match *[first,first)*
`match_not_eow`	*"\b"* will not match *[last,last)*
`match_any`	If more than one match is possible, then any match is an acceptable result
`match_not_null`	Do not match empty sequences
`match_continuous`	Only match a sub-sequence that begins at *first*
`match_prev_avail`	*--first* is a valid iterator position. When set, causes *match_not_bol* and *match_not_bow* to be ignored
`format_default`	Use ECMAScript rules to construct strings in `std::regex_replace` (syntax documentation (http%3A//ecma-international.org/ecma-262/5.1/#sec-15.5.4.11))
`format_sed`	Use POSIX *sed* utility rules in `std::regex_replace`. (syntax documentation (http%3A//pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html#tag_20_116_13_03))
`format_no_copy`	Do not copy un-matched strings to the output in `std::regex_replace`
`format_first_only`	Only replace the first match in `std::regex_replace`

All constants, except for `match_default` and `format_default`, are bitmask elements. The `match_default` and `format_default` constants are empty bitmasks

Notes

In C++11, these constants were specified with redundant keyword `static`, which was removed by C++14 via LWG issue 2053 (http%3A//www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#2053)

See also

`regex_match` (C++11)	attempts to match a regular expression to an entire character sequence (function template)
`syntax_option_type` (C++11)	general options controlling regex behavior (typedef)
`error_type` (C++11)	describes different types of matching errors (typedef)

std::regex_constants::error_type

Defined in header <regex>

```
typedef /*implementation defined*/ error_type;                    (since C++11)
```

```
constexpr error_type error_collate = /*unspecified*/;
constexpr error_type error_ctype = /*unspecified*/;
constexpr error_type error_escape = /*unspecified*/;
constexpr error_type error_backref = /*unspecified*/;
constexpr error_type error_brack = /*unspecified*/;
constexpr error_type error_paren = /*unspecified*/;
constexpr error_type error_brace = /*unspecified*/;                (since C++11)
constexpr error_type error_badbrace = /*unspecified*/;
constexpr error_type error_range = /*unspecified*/;
constexpr error_type error_space = /*unspecified*/;
constexpr error_type error_badrepeat = /*unspecified*/;
constexpr error_type error_complexity = /*unspecified*/;
constexpr error_type error_stack = /*unspecified*/;
```

The `error_type` is a type that describes errors that may occur during regular expression parsing.

Constants

Constant	Explanation
`error_collate`	the expression contains an invalid collating element name
`error_ctype`	the expression contains an invalid character class name
`error_escape`	the expression contains an invalid escaped character or a trailing escape
`error_backref`	the expression contains an invalid back reference
`error_brack`	the expression contains mismatched square brackets ('[' and ']')
`error_paren`	the expression contains mismatched parentheses ('(' and ')')
`error_brace`	the expression contains mismatched curly braces ('{' and '}')
`error_badbrace`	the expression contains an invalid range in a {} expression
`error_range`	the expression contains an invalid character range (e.g. [b-a])
`error_space`	there was not enough memory to convert the expression into a finite state machine
`error_badrepeat`	one of *?+{ was not preceded by a valid regular expression
`error_complexity`	the complexity of an attempted match exceeded a predefined level
`error_stack`	there was not enough memory to perform a match

Notes

In C++11, these constants were specified with redundant keyword `static`, which was removed by C++14 via LWG issue 2053 (http%3A//www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#2053)

See also

`regex_error` (C++11)	reports errors generated by the regular expressions library (class)

Modified ECMAScript regular expression grammar

This page describes the regular expression grammar that is used when `std::basic_regex` is constructed with syntax_option_type set to `ECMAScript` (the default). See syntax_option_type for the other supported regular expression grammars.

The `ECMAScript` regular expression grammar in C++ is ECMA-262 grammar (http%3A//ecma-international.org/ecma-262/5.1/#sec-15.10) with modifications marked with (C++ only) below.

Alternatives

A regular expression pattern is sequence of one or more *Alternative*s, separated by the disjunction operator `|` (in other words, the disjunction operator has the lowest precedence)

Pattern ::

Disjunction

Disjunction ::

Alternative
Alternative `|` *Disjunction*

The pattern first tries to skip the *Disjunction* and match the left *Alternative* followed by the rest of the regular expression (after the Disjunction).

If it fails, it tries to skip the left *Alternative* and match the right *Disjunction* (followed by the rest of the regular expression).

If the left *Alternative*, the right *Disjunction*, and the remainder of the regular expression all have choice points, all choices in the remainder of the expression are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*.

Any capturing parentheses inside a skipped *Alternative* produce empty submatches.

Run this code

```
#include <iostream>
#include <regex>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if(m.empty()) {
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
    } else {
        std::cout << "input=[" << in << "], regex=[" << re << "]: ";
        std::cout << "prefix=[" << m.prefix() << "] ";
        for(std::size_t n = 0; n < m.size(); ++n)
            std::cout << " m[" << n << "]=[" << m[n] << "] ";
        std::cout << "suffix=[" << m.suffix() << "]\n";
    }
}
int main()
{
    show_matches("abcdef", "abc|def");
    show_matches("abc", "ab|abc"); // left Alernative matched first

    // Match of the input against the left Alternative (a) followed
    // by the remained of the regex (c|bc) succeeds, which results
    // in m[1]="a" and m[4]="bc".
    // The skipped Alternatives (ab) and (c) leave their submatches
    // m[3] and m[5] empty.
    show_matches("abc", "((a)|(ab))((c)|(bc))");
}
```

Output:

```
input=[abcdef], regex=[abc|def]: prefix=[]  m[0]=[abc] suffix=[def]
input=[abc], regex=[ab|abc]: prefix=[]  m[0]=[ab] suffix=[c]
input=[abc], regex=[((a)|(ab))((c)|(bc))]: prefix=[]  m[0]=[abc]
m[1]=[a]  m[2]=[a]  m[3]=[] m[4]=[bc]  m[5]=[]  m[6]=[bc] suffix=[]
```

Terms

Each *Alternative* is either empty or is a sequence of *Term*s (with no separators between the *Term*s)

Alternative ::

[empty]
Alternative Term

Empty *Alternative* always matches and does not consume any input.

Consecutive *Term*s try to simultaneously match consecutive portions of the input.

If the left *Alternative*, the right *Term*, and the remainder of the regular expression all have choice points, all choices in the remained of the expression are tried before moving on to the next choice in the right *Term*, and all choices in the right *Term* are tried before moving on to the next choice in the left *Alternative*.

Run this code

```
#include <iostream>
#include <regex>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if(m.empty()) {
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
    } else {
        std::cout << "input=[" << in << "], regex=[" << re << "]: ";
        std::cout << "prefix=[" << m.prefix() << "] ";
        for(std::size_t n = 0; n < m.size(); ++n)
            std::cout << " m[" << n << "]=[" << m[n] << "] ";
        std::cout << "suffix=[" << m.suffix() << "]\n";
    }
}
int main()
{
    show_matches("abcdef", ""); // empty regex is a single empty Alternative
    show_matches("abc", "abc|"); // left Alernative matched first
    show_matches("abc", "|abc"); // left Alernative matched first, leaving abc unmatched
}
```

Output:

```
input=[abcdef], regex=[]: prefix=[]  m[0]=[] suffix=[abcdef]
input=[abc], regex=[abc|]: prefix=[]  m[0]=[abc] suffix=[]
input=[abc], regex=[|abc]: prefix=[]  m[0]=[] suffix=[abc]
```

Quantifiers

- Each *Term* is either an *Assertion* (see below), or an *Atom* (see below), or an *Atom* immediately followed by a *Quantifier*

Term ::

Assertion
Atom
Atom Quantifier

Each *Quantifier* is either a *greedy* quantifier (which consists of just one *QuantifierPrefix* or a *non-greedy* quantifier (which consists of one *QuantifierPrefix* followed by the question mark **?**.

Quantifier ::

QuantifierPrefix
QuantifierPrefix **?**

Each *QuantifierPrefix* determines two numbers: the minimum number of repetitions and the maximum number of repetitions, as follows:

QuantifierPrefix	Minimum	Maximum
*	zero	infinity
+	one	infinity
?	zero	one
{ *DecimalDigits* }	value of DecimalDigits	value of DecimalDigits
{ *DecimalDigits* , }	value of DecimalDigits	infinity
{ *DecimalDigits* , *DecimalDigits* }	value of DecimalDigits before the comma	value of DecimalDigits after the comma

The values of the individual *DecimalDigits* are obtained by calling `std::regex_traits::value`(C++ only) on each of the digits.

An *Atom* followed by a *Quantifier* is repeated the number of times specified by the *Quantifier*. A *Quantifier* can be *non-greedy*, in which case the *Atom* pattern is repeated as few times as possible while still matching the remainder of the regular expression, or it can be *greedy*, in which case the *Atom* pattern is repeated as many times as possible while still matching the remainder of the regular expression.

The *Atom* pattern is what is repeated, not the input that it matches, so different repetitions of the *Atom* can match different input substrings.

If the *Atom* and the remainder of the regular expression all have choice points, the *Atom* is first matched as many (or as few, if *non-greedy*) times as possible. All choices in the remainder of the regular expression are tried before moving on to the next choice in the last repetition of *Atom*. All choices in the last (nth) repetition of *Atom* are tried before moving on to the next choice in the next-to-last (n–1)st repetition of *Atom*; at which point it may turn out that more or fewer repetitions of *Atom* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the (n-1)st repetition of *Atom* and so on.

The *Atom*'s captures are cleared each time it is repeated (see the `"(z)((a+)?(b+)?(c))*"` example below)

Run this code

```
#include <iostream>
#include <regex>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if(m.empty()) {
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
    } else {
        std::cout << "input=[" << in << "], regex=[" << re << "]: ";
        std::cout << "prefix=[" << m.prefix() << "] ";
        for(std::size_t n = 0; n < m.size(); ++n)
            std::cout << " m[" << n << "]=[" << m[n] << "] ";
        std::cout << "suffix=[" << m.suffix() << "]\n";
    }
}
int main()
{
    // greedy match, repeats [a-z] 4 times
    show_matches("abcdefghi", "a[a-z]{2,4}");
    // non-greedy match, repeats [a-z] 2 times
    show_matches("abcdefghi", "a[a-z]{2,4}?");

    // Choice point ordering for quantifiers results in a match
    // with two repetitions, first matching the substring "aa",
    // second matching the substring "ba", leaving "ac" not matched
    // ("ba" appears in the capture clause m[1])
    show_matches("aabaac", "(aa|aabaac|ba|b|c)*");

    // Choice point ordering for quantifiers makes this regex
    // calculate the greatest common divisor between 10 and 15
    // (the answer is 5, and it populates m[1] with "aaaaa")
    show_matches("aaaaaaaaaa,aaaaaaaaaaaaaaa", "^(a+)\\1*,\\1+$");

    // the substring "bbb" does not appear in the capture clause m[4]
    // because it is cleared when the second repetition of the atom
    // (a+)?(b+)?(c) is matching the substring "ac"
    show_matches("zaacbbbcac", "(z)((a+)?(b+)?(c))*");
}
```

Output:

```
input=[abcdefghi], regex=[a[a-z]{2,4}]: prefix=[]  m[0]=[abcde] suffix=[fghi]
input=[abcdefghi], regex=[a[a-z]{2,4}?]: prefix=[]  m[0]=[abc] suffix=[defghi]
input=[aabaac], regex=[(aa|aabaac|ba|b|c)*]:
 prefix=[]  m[0]=[aaba]  m[1]=[ba] suffix=[ac]
input=[aaaaaaaaaa,aaaaaaaaaaaaaa], regex=[^(a+)\1*,\1+$]:
 prefix=[]  m[0]=[aaaaaaaaaa,aaaaaaaaaaaaaa]  m[1]=[aaaaa] suffix=[]
input=[zaacbbbcac], regex=[(z)((a+)?(b+)?(c))*]:
prefix=[]  m[0]=[zaacbbbcac]  m[1]=[z]  m[2]=[ac]  m[3]=[a]  m[4]=[]  m[5]=[c] suffix=[]
```

Assertions

*Assertion*s match conditions, rather than substrings of the input string. They never consume any characters from the input. Each *Assertion* is one of the following

Assertion ::

- **^**
- **$**
- **\ b**
- **\ B**
- **(? =** *Disjunction* **)**
- **(? !** *Disjunction* **)**

The assertion ^ (beginning of line) matches

1) The position that immediately follows a *LineTerminator* character. (if supported, see LWG issue 2343 (http%3A//cplusplus.github.io/LWG/lwg-active.html#2343))
2) The beginning of the input (unless `std::regex_constants::match_not_bol`(C++ only) is enabled)

The assertion **$** (end of line) matches

1) The position of a *LineTerminator* character (if supported, see LWG issue 2343 (http%3A//cplusplus.github.io/LWG/lwg-active.html#2343))
2) The end of the input (unless `std::regex_constants::match_not_eol`(C++ only) is enabled)

In the two assertions above and in the Atom **.** below, *LineTerminator* is one of the following four charactes: `U+000A` (`\n` or line feed), `U+000D` (`\r` or carriage return), `U+2028` (line separator), or `U+2029` (paragraph separator)

The assertion **\b** (word boundary) matches

1) The beginning of a word (current character is a letter, digit, or underscore, and the previous character is not)
2) The end of a word (current character is not a letter, digit, or underscore, and the previous character is one of those)
3) The beginning of input if the first character is a letter, digit, or underscore (unless `std::regex_constants::match_not_bow`(C++ only) is enabled))
4) The end of input if the last character is a letter, digit, or underscore (unless `std::regex_constants::match_not_eow`(C++ only) is enabled))

The assertion **\B** (negative word boundary) matches everything EXCEPT the following

1) The beginning of a word (current character is a letter, digit, or underscore, and the previous character is not one of those or does not exist)
2) The end of a word (current character is not a letter, digit, or underscore (or the matcher is at the end of input), and the previous character is one of those)

The assertion **(? =** *Disjunction* **)** (zero-width positive lookahead) matches if *Disjunction* would match the input at the current position

The assertion **(? !** *Disjunction* **)** (zero-width negative lookahead) matches if *Disjunction* would NOT match the input at the current position.

For both Lookahead assertions, when matching the *Disjunction*, the position is not advanced before matching the remainder of the regular expression. Also, if *Disjunction* can match at the current position in several ways, only the first one is tried.

ECMAScript forbids backtracking into the lookahead Disjunctions, which affects the behavior of backreferences into a positive lookahead from the remainder of the regular expression (see example below). Backreferences into the negative lookahead from the rest of the regular expression are always undefined (since the lookahead Disjunction must fail to proceed).

Note: Lookahead assertions may be used to create logical AND between multiple regular expressions (see example below)

Run this code

```
#include <iostream>
#include <regex>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if(m.empty()) {
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
    } else {
        std::cout << "input=[" << in << "], regex=[" << re << "]: ";
        std::cout << "prefix=[" << m.prefix() << "] ";
        for(std::size_t n = 0; n < m.size(); ++n)
            std::cout << " m[" << n << "]=[" << m[n] << "] ";
        std::cout << "suffix=[" << m.suffix() << "]\n";
    }
}
int main()
{
    // matches the a at the end of input
    show_matches("aaa", "a$");

    // matches the o at the end of the first word
    show_matches("moo goo gai pan", "o\\b");

    // the lookahead matches the empty string immediately after the first b
    // this populates m[1] with "aaa" although m[0] is empty
    show_matches("baaabac", "(?=(a+))");

    // because backtracking into lookaheads is prohibited,
    // this matches aba rather than aaaba
    show_matches("baaabac", "(?=(a+))a*b\\1");

    // logical AND via lookahead: this password matches IF it contains
    // at least one lowercase letter
    // AND at least one uppercase letter
    // AND at least one punctuation character
    // AND be at least 6 characters long
    show_matches("abcdef", "(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}");
    show_matches("aB,def", "(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}");
}
```

Output:

```
input=[aaa], regex=[a$]: prefix=[aa]  m[0]=[a] suffix=[]
input=[moo goo gai pan], regex=[o\b]: prefix=[mo]  m[0]=[o] suffix=[ goo gai pan]
input=[baaabac], regex=[(?=(a+))]: prefix=[b]  m[0]=[]  m[1]=[aaa] suffix=[aaabac]
input=[baaabac], regex=[(?=(a+))a*b\1]: prefix=[baa]  m[0]=[aba]  m[1]=[a] suffix=[c]
input=[abcdef], regex=[(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}]:
 NO MATCH
input=[aB,def], regex=[(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}]:
 prefix=[]  m[0]=[aB,def] suffix=[]
```

Atoms

An *Atom* can be one of the following:

Atom ::

PatternCharacter
.
**** *AtomEscape*
CharacterClass
(*Disjunction* **)**
(**?** **:** *Disjunction* **)**

where *AtomEscape* ::

DecimalEscape
CharacterEscape
CharacterClassEscape

Different kinds of atoms evaluate differently.

Sub-expressions

The *Atom* `(` *Disjunction* `)` is marked subexpression: it executes the *Disjunction* and stores the copy of the input substring that was consumed by *Disjunctin* in the submatch array at the index that corresponds to the number of times the left open parenthesis `(` of marked subexpressions has been encountered in the entire regular expression at this point.

Besides being returned in the `std::match_results`, the captured submatches are accessible as back-references (`\1`, `\2`, ...) and can be referenced in `std::regex_replace`)

The *Atom* `(?:` *Disjunction* `)` (non-marking subexpression) simply evaluates the *Disjunction* and does not store its results in the submatch. This is a purely lexical grouping.

This section is incomplete
Reason: no example

Backreferences

DecimalEscape ::

DecimalIntegerLiteral [*lookahead* ∉ *DecimalDigit*]

If `\` is followed by a decimal number `N` whose first digit is not `0`, then the escape sequence is considered to be a *backreference*. The values `N` is obtained by calling `std::regex_traits::value`(C++ only) on each of the digits and combining their results using base-10 arithmetic. It is an error if `N` is greater than the total number of left capturing parentheses in the entire regular expression.

When a backreference `\N` appears as an Atom, it matches the same substring as what is currently stored in the N'th element of the submatch array.

The decimal escape `\0` is NOT a backreference: it is a character escape that represents the nul character. It cannot be followed by a decimal digit.

This section is incomplete
Reason: no example

Single character matches

The *Atom* `.` matches and consumes any one character from the input string except for *LineTerminator* (`U+000D`, `U+000A`, `U+2029`, or `U+2028`)

The *Atom PatternCharacter*, where *PatternCharacter* is any *SourceCharacter* EXCEPT the chracters `^ $ \ . * + ? ( ) [ ] { } |`, matches and consumes one character from the input if it is equal to this *PatternCharacter*.

The equality for this and all other single character matches is defined as follows:

1) If `std::regex_constants::icase` is set, the characters are equal if the return values of `std::regex_traits::translate_nocase` are equal.(C++ only)
2) otherwise, if `std::regex_constants::collate` is set, the characters are equal if the return values of `std::regex_traits::translate` are equal.(C++ only)
3) otherwise, the characters are equal if operator== returns true.

Each *Atom* that consists of the escape character `\` followed by *CharacterEscape* as well as the special DecimalEscape `\0`, matches and consumes one character from the input if it is equal to the character represented by the *CharacterEscape*. The following character escape sequences are recognized:

CharacterEscape ::

ControlEscape
`c` *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

Here, *ControlEscape* is one of the following five characters: `f n r t v`

ControlEscape	Code Unit	Name
`f`	U+000C	form feed
`n`	U+000A	new line
`r`	U+000D	carriage return
`t`	U+0009	horizontal tab
`v`	U+000B	vertical tab

ControlLetter is any lowercase or uppercase ASCII letters and this character escape matches the character whose code unit equals the remainder of dividing the value of the code unit of *ControlLetter* by `32`, for example, **`\cD`** and **`\cd`** both match code unit `U+0004` (EOT) because 'D' is `U+0044`, and `0x44 % 32 == 4` and 'd' is `U+0064` and `0x64 % 32 == 4`.

HexEscapeSequence is the letter **`x`** followed by exactly two *HexDigit*s (where *HexDigit* is one of **`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`**). This character escape matches the character whose code unit equals the numeric value of the two-digit hexadecimal number

UnicodeEscapeSequence is the letter **`u`** followed by exactly four *HexDigit*s. This character escape matches the character whose code unit equals the numeric value of this four-digit hexadecimal number. If the value does not fit in this `std::basic_regex`'s CharT, `std::regex_error` is thrown(C++ only).

IdentityEscape can be any non-alphanumeric character: for example, another backslash. It matches the character as-is.

This section is incomplete
Reason: no example

Character classes

An Atom can represent a character class, that is, it will match and consume one character if it belongs to one of the predefined groups of characters.

A character class can be introduced through a character class escape:

Atom ::

**** *CharacterClassEscape*

or directly

Atom ::

CharacterClass

The character class escapes are shorthands for some of the common characters classes, as follows:

CharacterClassEscape	ClassName expression(C++ only)	Meaning
`d`	**`[[:digit:]]`**	digits
`D`	**`[^[:digit:]]`**	non-digits
`s`	**`[[:space:]]`**	whitespace characters
`S`	**`[^[:space:]]`**	non-whitespace characters
`w`	**`[_[:alnum:]]`**	alphanumeric characters and the character _
`W`	**`[^_[:alnum:]]`**	characters other than alphanumeric or _

The exact meaning of each of these character class escapes in C++ is defined in terms of the locale-dependent named character classes, and not by explicitly listing the acceptable characters as in ECMAScript.

A *CharacterClass* is a bracket-enclosed sequence of *ClassRanges*, optionally beginning with the negation operator ^. If it begins with ^, this *Atom* matches any character that is NOT in the set of characters represented by the union of all *ClassRanges*. Otherwise, this *Atom* matches any character that IS in the set of the characters represented by the union of all *ClassRanges*.

CharacterClass ::

[**[** *lookahead* ∉ *{^}] ClassRanges* **]**
[**^** *ClassRanges* **]**

ClassRanges ::

[empty]
NonemptyClassRanges

NonemptyClassRanges ::

ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom **-** *ClassAtom ClassRanges*

If non-empty class range has the form ***`ClassAtom - ClassAtom`***, it matches any character from a range defined as follows: (C++ only)

The first *ClassAtom* must match a single collating element `c1` and the second *ClassAtom* must match a single collating element `c2`. To test if the input character `c` is matched by this range, the following steps are taken:

1) If `std::regex_constants::collate` is not on, the character is matched by direct comparison of

code points: `c` is matched if `c1 <= c && c <= c2`

1) Otherwise (if `std::regex_constants::collate` is enabled):
   1) If `std::regex_constants::icase` is enabled, all three characters (`c`, `c1`, and `c2` are passed `std::regex_traits::translate_nocase`
   2) Otherwise (if `std::regex_constants::icase` is not set), all three characters (`c`, `c1`, and `c2` are passed `std::regex_traits::translate`
2) The resulting strings are compared using `std::regex_traits::transform` and the character `c` is matched if `transformed c1 <= transformed c && transformed c <= transformed c2`

The character **-** is treated literally if it is

- the first or last character of *ClassRanges*
- the beginning or end ClassAtom of a dash-separated range specification
- immediately follows a dash-separated range specification.
- escaped with a backslash as a *CharacterEscape*

NonemptyClassRangesNoDash ::

ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash **-** *ClassAtom ClassRanges*

ClassAtom ::

-
ClassAtomNoDash
ClassAtomExClass(C++ only)
ClassAtomCollatingElement(C++ only)
ClassAtomEquivalence(C++ only)

ClassAtomNoDash ::

SourceCharacter but not one of **\ or] or -**
**** *ClassEscape*

Each *ClassAtomNoDash* represents a single character -- either *SourceCharacter* as-is or escaped as follows:

ClassEscape ::

DecimalEscape
b
CharacterEscape
CharacterClassEscape

The special *ClassEscape* **\b** produces a character set that matches the code unit U+0008 (backspace). Outside of *CharacterClass*, it is the word-boundary *Assertion*.

The use of **\B** and the use of any backreference (*DecimalEscape* other than zero) inside a *CharacterClass* is an error.

The characters **-** and **]** may need to be escaped in some situations in order to be treated as atoms. Other characters that have special meaning outside of *CharacterClass*, such as ***** or **?**, do not need to be escaped.

This section is incomplete
Reason: no example

POSIX-based character classes

These character classes are an extension to the ECMAScript grammar, and are equivalent to character classes found in the the POSIX regular expressions.

ClassAtomExClass(C++ only) ::

[: *ClassName* **:]**

Represents all characters that are members of the named character class *ClassName*. The name is valid only if `std::regex_traits::lookup_classname` returns non-zero for this name. As described in `std::regex_traits::lookup_classname`, the following names are guaranteed to be recognized: **alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit, d, s, w**. Additional names may be provided by system-supplied locales (such as **jdigit** or **jkanji** in Japanese) or implemented as a user-defined extension.

ClassAtomCollatingElement(C++ only) ::

[. *ClassName* **.]**

Represents the named collating element, which may represent a single character or a sequence of characters that collates as a single unit under the imbued locale, such as `[.tilde.]` or `[.ch.]` in Czech. The name is valid only if `std::regex_traits::lookup_collatename` is not an empty string.

When using `std::regex_constants::collate`, collating elements can always be used as ends points of a range (e.g. `[[.dz.]-g]` in Hungarian).

ClassAtomEquivalence(C++ only) ::

[= *ClassName* **=]**

Represents all characters that are members of the same equivalence class as the named collating element, that is, all characters whose whose primary collation key is the same as that for collating element *ClassName*. The name is valid only if `std::regex_traits::lookup_collatename` for that name is not an empty string and if the value returned by `std::regex_traits::transform_primary` for the result of the call to `std::regex_traits::lookup_collatename` is not an empty string.

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.`

ClassName(C++ only) ::

ClassNameCharacter
ClassNameCharacter ClassName

ClassNameCharacter(C++ only) ::

SourceCharacter but not one of **. = :**

This section is incomplete
Reason: no example