Mandating the Standard Library:
Clause 30 - Regular Expressions library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

— Change 'participate in overload resolution' wording into "Constraints' elements
— Change 'Requires' elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.
— Drive-by fixes (hopefully very few)

This paper covers Clause 30 (Regular Expressions), and is based on N4810.

The entire clause is reproduced here, but the changes are confined to a few sections:

— re.badexp 30.6
— re.traits 30.7
— re.regex.construct 30.8.1
— re.regex.assign 30.8.2
— re.results.const 30.10.1
— re.results.acc 30.10.4
— re.results.form 30.10.5
— re.alg.match 30.11.2
— re.alg.search 30.11.3

Drive-by fixes:

— "De-shalled" re.traits/1 (30.7).
— reworded a bunch of constructor details in re.badexp (30.6), re.regex.construct (30.8.1), and re.results.const (30.10.1).

Thanks to Daniel Krügler for his advice and reviews.

Help for the editors: The changes here can be viewed as latex sources with the following commands

```bash
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter30 regex.tex
```
30 Regular expressions library [re]

30.1 General [re.general]

1 This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

2 The following subclauses describe a basic regular expression class template and its traits that can handle char-like (?) template arguments, two specializations of this class template that handle sequences of char and wchar_t, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as summarized in Table 122.

Table 122 — Regular expressions library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2 Definitions</td>
<td>[re.def]</td>
</tr>
<tr>
<td>30.3 Requirements</td>
<td></td>
</tr>
<tr>
<td>30.5 Constants</td>
<td>&lt;regex&gt;</td>
</tr>
<tr>
<td>30.6 Exception type</td>
<td></td>
</tr>
<tr>
<td>30.7 Traits</td>
<td></td>
</tr>
<tr>
<td>30.8 Regular expression template</td>
<td></td>
</tr>
<tr>
<td>30.9 Submatches</td>
<td></td>
</tr>
<tr>
<td>30.10 Match results</td>
<td></td>
</tr>
<tr>
<td>30.11 Algorithms</td>
<td></td>
</tr>
<tr>
<td>30.12 Iterators</td>
<td></td>
</tr>
<tr>
<td>30.13 Grammar</td>
<td></td>
</tr>
</tbody>
</table>

30.2 Definitions [re.def]

1 The following definitions shall apply to this Clause:

30.2.1 collating element [defns.regex.collating.element]
a sequence of one or more characters within the current locale that collate as if they were a single character.

30.2.2 finite state machine [defns.regex.finite.state.machine]
an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

30.2.3 format specifier [defns.regex.format.specifier]
a sequence of one or more characters that is to be replaced with some part of a regular expression match.

30.2.4 matched [defns.regex.matched]
a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

30.2.5 primary equivalence class [defns.regex.primary.equivalence.class]
a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accents, case, or locale specific tailorings.
30.2.6 [defns.regex.regular.expression]  
regular expression  
a pattern that selects specific strings from a set of character strings.

30.2.7 [defns.regex.subexpression]  
sub-expression  
a subset of a regular expression that has been marked by parenthesis.

30.3 Requirements [re.req]  
1 This subclause defines requirements on classes representing regular expression traits.  
[Note: The class template regex_traits, defined in 30.7, satisfies these requirements. — end note]

2 The class template basic_regex, defined in 30.8, needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member typedef-names and functions in the template parameter traits used by the basic_regex class template. This subclause defines the semantics of these members.

3 To specialize class template basic_regex for a character container CharT and its related regular expression traits class Traits, use basic_regex<CharT, Traits>.

4 In Table 123 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 (??); F1 and F2 are forward 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; cl is an object of type X::char_class_type, and loc is an object of type X::locale_type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>The character container type used</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in the implementation of class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>template basic_regex.</td>
</tr>
<tr>
<td>X::string_type</td>
<td>basic_string&lt;charT&gt;</td>
<td>A copy constructible type</td>
</tr>
<tr>
<td>X::locale_type</td>
<td>A copy</td>
<td>A type that represents the locale</td>
</tr>
<tr>
<td></td>
<td>constructible type</td>
<td>used by the traits class.</td>
</tr>
<tr>
<td>X::char_class_type</td>
<td>A bitmask</td>
<td>A bitmask type representing a</td>
</tr>
<tr>
<td></td>
<td>type (??).</td>
<td>particular character classification.</td>
</tr>
<tr>
<td>X::length(p)</td>
<td>size_t</td>
<td>Yields the smallest i such that</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p[i] == 0. Complexity is linear in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i.</td>
</tr>
<tr>
<td>v.translate(c)</td>
<td>X::char_type</td>
<td>Returns a character such that for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>any character d that is to be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>considered equivalent to c then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v.translate(c) == v.translate(d).</td>
</tr>
<tr>
<td>v.translate_nocase(c)</td>
<td>X::char_type</td>
<td>For all characters C that are to be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>considered equivalent to c when</td>
</tr>
<tr>
<td></td>
<td></td>
<td>comparisons are to be performed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>without regard to case, then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v.translate_nocase(c) ==</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v.translate_nocase(C).</td>
</tr>
<tr>
<td>v.transform(F1, F2)</td>
<td>X::string_type</td>
<td>Returns a sort key for the character</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequence designated by the iterator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>range [F1, F2] such that if the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>character sequence [G1, G2] sorts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>before the character sequence [H1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2] then v.transform(G1, G2) &lt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v.transform(H1, H2).</td>
</tr>
</tbody>
</table>

Table 123 — Regular expression traits class requirements
Table 123 — Regular expression traits class requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.transform_primary(F1, F2)</td>
<td>X::string_type</td>
<td>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) &lt; v.transform_primary(H1, H2).</td>
</tr>
<tr>
<td>v.lookup_collatename(F1, F2)</td>
<td>X::string_type</td>
<td>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.</td>
</tr>
<tr>
<td>v.lookup_classname(F1, F2, b)</td>
<td>X::char_class_type</td>
<td>Converts the character sequence designated by the iterator range [F1, F2) into a value of a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be bitwise OR’ed together; the resulting value represents membership in either of the corresponding character classes. If b is true, the returned bitmask is suitable for matching characters without regard to their case. 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.</td>
</tr>
<tr>
<td>v.isctype(c, cl)</td>
<td>bool</td>
<td>Returns true if character c is a member of one of the character classes designated by cl, false otherwise.</td>
</tr>
<tr>
<td>v.value(c, I)</td>
<td>int</td>
<td>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]</td>
</tr>
<tr>
<td>u.imbue(loc)</td>
<td>X::locale_type</td>
<td>Imbues u with the locale loc and returns the previous locale used by u if any.</td>
</tr>
<tr>
<td>v.getloc()</td>
<td>X::locale_type</td>
<td>Returns the current locale used by v, if any.</td>
</tr>
</tbody>
</table>

5 [Note: Class template regex_traits satisfies the requirements for a regular expression traits class when it is specialized for char or wchar_t. This class template is described in the header <regex>, and is described in 30.7. —end note]

30.4 Header <regex> synopsis

```cpp
#include <initializer_list>

namespace std {
    // 30.5, regex constants
    namespace regex_constants {
        using syntax_option_type = T1;
        using match_flag_type = T2;
        using error_type = T3;
    }

    // 30.6, class regex_error
    class regex_error;
}
```
// 30.7, class template regex_traits
template<class charT> struct regex_traits;

// 30.8, class template basic_regex
template<class charT, class traits = regex_traits<charT>> class basic_regex;
using regex = basic_regex<char>;
using wregex = basic_regex<wchar_t>;

// 30.8.5, basic_regex swap
template<class charT, class traits>
void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

// 30.9, class template sub_match
template<class BidirectionalIterator>
class sub_match;
using csub_match = sub_match<const char*>;
using wcsub_match = sub_match<const wchar_t*>;
using assub_match = sub_match<string::const_iterator>;
using wssub_match = sub_match<wstring::const_iterator>;

// 30.9.2, sub_match non-member operators
template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template<class BiIter>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template<class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template<class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template<class BiIter>
bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template<class BiIter>
bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator>=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
template<class BiIter, class ST, class SA>
bool operator==(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template<class BiIter, class ST, class SA>
bool operator!=(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template<class BiIter, class ST, class SA>
bool operator<(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template<class BiIter, class ST, class SA>
bool operator>(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template<class BiIter, class ST, class SA>
bool operator<=(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template<class BiIter, class ST, class SA>
bool operator>=(const sub_match<BiIter>& lhs, const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

template<class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 30.10, class template match_results
template<class BidirectionalIterator, class Allocator = allocator<sub_match<BidirectionalIterator>>> class match_results;

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

// 30.10.7, match_results swap
template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1, match_results<BidirectionalIterator, Allocator>& m2);

§ 30.4
// 30.11.2, function template regex_match

```cpp
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,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

```cpp
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,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

```cpp
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,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

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

```cpp
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,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

// 30.11.3, function template regex_search

```cpp
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,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```

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

```cpp
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,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```

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

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

```cpp
template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>&&,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator,
                  Allocator>&,
                  const basic_regex<charT, traits>&,
                  regex_constants::match_flag_type = regex_constants::match_default) = delete;
```
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>&,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type
    = regex_constants::match_default) = delete;

// 30.11.4, function template regex_replace

// 30.12.1, class template regex_iterator

// 30.12.2, class template regex_token_iterator
class regex_token_iterator;

using cregex_token_iterator  = regex_token_iterator<const char*>;
using wcregex_token_iterator = regex_token_iterator<const wchar_t*>;
using sregex_token_iterator = regex_token_iterator<string::const_iterator>;
using wsregex_token_iterator = regex_token_iterator<wstring::const_iterator>;

namespace pmr {
  template<class BidirectionalIterator>
  using match_results = 
    std::match_results<BidirectionalIterator, 
    polymorphic_allocator<std::sub_match<BidirectionalIterator>>>;

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

30.5 Namespace std::regex_constants

The namespace std::regex_constants holds symbolic constants used by the regular expression library. This namespace provides three types, syntax_option_type, match_flag_type, and error_type, along with several constants of these types.

30.5.1 Bitmask type syntax_option_type

namespace std::regex_constants {
  using syntax_option_type = T1;
  inline constexpr syntax_option_type icase = unspecified;
  inline constexpr syntax_option_type nosubs = unspecified;
  inline constexpr syntax_option_type optimize = unspecified;
  inline constexpr syntax_option_type collate = unspecified;
  inline constexpr syntax_option_type ECMAScript = unspecified;
  inline constexpr syntax_option_type basic = unspecified;
  inline constexpr syntax_option_type extended = unspecified;
  inline constexpr syntax_option_type awk = unspecified;
  inline constexpr syntax_option_type grep = unspecified;
  inline constexpr syntax_option_type egrep = unspecified;
  inline constexpr syntax_option_type multiline = unspecified;
}

The type syntax_option_type is an implementation-defined bitmask type (??). Setting its elements has the effects listed in Table 124. A valid value of type syntax_option_type shall have at most one of the grammar elements ECMAScript, basic, extended, awk, grep, egrep, set. If no grammar element is set, the default grammar is ECMAScript.

30.5.2 Bitmask type match_flag_type

namespace std::regex_constants {
  using match_flag_type = T2;
  inline constexpr match_flag_type match_default = {};
  inline constexpr match_flag_type match_not_bol = unspecified;
  inline constexpr match_flag_type match_not_eol = unspecified;
  inline constexpr match_flag_type match_not_bow = unspecified;
  inline constexpr match_flag_type match_not_eow = unspecified;
  inline constexpr match_flag_type match_any = unspecified;
  inline constexpr match_flag_type match_not_null = unspecified;
  inline constexpr match_flag_type match_continuous = unspecified;
  inline constexpr match_flag_type match_prev_avail = unspecified;
  inline constexpr match_flag_type format_default = {};
  inline constexpr match_flag_type format_sed = unspecified;
  inline constexpr match_flag_type format_no_copy = unspecified;
  inline constexpr match_flag_type format_first_only = unspecified;
}

§ 30.5.2
Table 124 — syntax_option_type effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>icase</td>
<td>Specifies that matching of regular expressions against a character container</td>
</tr>
<tr>
<td></td>
<td>sequence shall be performed without regard to case.</td>
</tr>
<tr>
<td>nosubs</td>
<td>Specifies that no sub-expressions shall be considered to be marked, so that</td>
</tr>
<tr>
<td></td>
<td>when a regular expression is matched against a character container sequence,</td>
</tr>
<tr>
<td></td>
<td>no sub-expression matches shall be stored in the supplied match_results</td>
</tr>
<tr>
<td></td>
<td>object.</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies that the regular expression engine should pay more attention to the</td>
</tr>
<tr>
<td></td>
<td>speed with which regular expressions are matched, and less to the speed with</td>
</tr>
<tr>
<td></td>
<td>which regular expression objects are constructed. Otherwise it has no</td>
</tr>
<tr>
<td></td>
<td>detectable effect on the program output.</td>
</tr>
<tr>
<td>collate</td>
<td>Specifies that character ranges of the form &quot;[a-b]&quot; shall be locale sensitive.</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by ECMAScript in ECMA-262, as modified in 30.13.</td>
</tr>
<tr>
<td></td>
<td>See also: ECMA-262 15.10</td>
</tr>
<tr>
<td>basic</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by basic regular expressions in POSIX.</td>
</tr>
<tr>
<td></td>
<td>See also: POSIX, Base Definitions and Headers, Section 9.3</td>
</tr>
<tr>
<td>extended</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by extended regular expressions in POSIX.</td>
</tr>
<tr>
<td></td>
<td>See also: POSIX, Base Definitions and Headers, Section 9.4</td>
</tr>
<tr>
<td>awk</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by the utility awk in POSIX.</td>
</tr>
<tr>
<td>grep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by the utility grep in POSIX.</td>
</tr>
<tr>
<td>egrep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by the utility grep when given the -E option in POSIX.</td>
</tr>
<tr>
<td>multiline</td>
<td>Specifies that ^ shall match the beginning of a line and $ shall match the</td>
</tr>
<tr>
<td></td>
<td>end of a line, if the ECMAScript engine is selected.</td>
</tr>
</tbody>
</table>

1 The type match_flag_type is an implementation-defined bitmask type (??). The constants of that type, except for match_default and format_default, are bitmask elements. The match_default and format_default constants are empty bitmasks. Matching a regular expression against a sequence of characters [first, last) proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 125 for any bitmask elements set.

Table 125 — regex_constants::match_flag_type effects when obtaining a match against a character container sequence [first, last).

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_not_bol</td>
<td>The first character in the sequence [first, last) shall be treated as</td>
</tr>
<tr>
<td></td>
<td>though it is not at the beginning of a line, so the character ^ in the regular</td>
</tr>
<tr>
<td></td>
<td>expression shall not match [first, first).</td>
</tr>
<tr>
<td>match_not_eol</td>
<td>The last character in the sequence [first, last) shall be treated as though</td>
</tr>
<tr>
<td></td>
<td>it is not at the end of a line, so the character &quot;$&quot; in the regular expression</td>
</tr>
<tr>
<td></td>
<td>shall not match [last, last).</td>
</tr>
<tr>
<td>match_not_bow</td>
<td>The expression ^\b shall not match the sub-sequence [first, first).</td>
</tr>
<tr>
<td>match_not_eow</td>
<td>The expression \b shall not match the sub-sequence [last, last).</td>
</tr>
<tr>
<td>match_any</td>
<td>If more than one match is possible then any match is an acceptable result.</td>
</tr>
<tr>
<td>match_not_null</td>
<td>The expression shall not match an empty sequence.</td>
</tr>
<tr>
<td>match_continuous</td>
<td>The expression shall only match a sub-sequence that begins at first.</td>
</tr>
</tbody>
</table>
Table 125 — regex_constants::match_flag_type effects when obtaining a match against a character container sequence [first, last). (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_prev_avail</td>
<td>--first is a valid iterator position. When this flag is set the flags match_not_bol and match_not_bow shall be ignored by the regular expression algorithms (30.11) and iterators (30.12).</td>
</tr>
<tr>
<td>format_default</td>
<td>When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.</td>
</tr>
<tr>
<td>format_sed</td>
<td>When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.</td>
</tr>
<tr>
<td>format_no_copy</td>
<td>During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.</td>
</tr>
<tr>
<td>format_first_only</td>
<td>When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.</td>
</tr>
</tbody>
</table>

30.5.3 Implementation-defined error_type

namespace std::regex_constants {
using error_type = T3;
inline constexpr error_type error_collate = unspecified;
inline constexpr error_type error_cctype = unspecified;
inline constexpr error_type error_escape = unspecified;
inline constexpr error_type error_backref = unspecified;
inline constexpr error_type error_brack = unspecified;
inline constexpr error_type error_paren = unspecified;
inline constexpr error_type error_brace = unspecified;
inline constexpr error_type error_badbrace = unspecified;
inline constexpr error_type error_range = unspecified;
inline constexpr error_type error_space = unspecified;
inline constexpr error_type error_badrepeat = unspecified;
inline constexpr error_type error_complexity = unspecified;
inline constexpr error_type error_stack = unspecified;
}

The type error_type is an implementation-defined enumerated type (??). Values of type error_type represent the error conditions described in Table 126:

Table 126 — error_type values in the C locale

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_collate</td>
<td>The expression contained an invalid collating element name.</td>
</tr>
<tr>
<td>error_cctype</td>
<td>The expression contained an invalid character class name.</td>
</tr>
<tr>
<td>error_escape</td>
<td>The expression contained an invalid escaped character, or a trailing escape.</td>
</tr>
<tr>
<td>error_backref</td>
<td>The expression contained an invalid back reference.</td>
</tr>
<tr>
<td>error_brack</td>
<td>The expression contained mismatched [ and ].</td>
</tr>
<tr>
<td>error_paren</td>
<td>The expression contained mismatched ( and ).</td>
</tr>
<tr>
<td>error_brace</td>
<td>The expression contained mismatched { and }.</td>
</tr>
<tr>
<td>error_badbrace</td>
<td>The expression contained an invalid range in a { } expression.</td>
</tr>
<tr>
<td>error_range</td>
<td>The expression contained an invalid character range, such as [b-a] in most encodings.</td>
</tr>
<tr>
<td>error_space</td>
<td>There was insufficient memory to convert the expression into a finite state machine.</td>
</tr>
</tbody>
</table>
Table 126 — error_type values in the C locale (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_badrepeat</td>
<td>One of *?+{ was not preceded by a valid regular expression.</td>
</tr>
<tr>
<td>error_complexity</td>
<td>The complexity of an attempted match against a regular expression exceeded a pre-set level.</td>
</tr>
<tr>
<td>error_stack</td>
<td>There was insufficient memory to determine whether the regular expression could match the specified character sequence.</td>
</tr>
</tbody>
</table>

30.6 Class regex_error

```cpp
regex_error(regex_constants::error_type ecode);  
Effects: Constructs an object of class regex_error.
Ensures: ecode == code().
```

regex_constants::error_type code() const;
Returns: The error code that was passed to the constructor.

30.7 Class template regex_traits

```cpp
namespace std {
    template<class charT>
    struct regex_traits {
        using char_type = charT;
        using string_type = basic_string<char_type>;
        using locale_type = locale;
        using char_class_type = bitmask_type;

        regex_traits();
        static size_t length(const char_type* p);
        charT translate(charT c) const;
        charT translate_nocase(charT c) const;
        template<class ForwardIterator>
        string_type transform(ForwardIterator first, ForwardIterator last) const;
        template<class ForwardIterator>
        string_type transform_primary( 
            ForwardIterator first, ForwardIterator last) const;
        template<class ForwardIterator>
        string_type lookup_collatename( 
            ForwardIterator first, ForwardIterator last) const;
        template<class ForwardIterator>
        char_class_type lookup_classname( 
            ForwardIterator first, ForwardIterator last, bool icase = false) const;
        bool isctype(charT c, char_class_type f) const;
        int value(charT ch, int radix) const;
        locale_type imbue(locale_type l);
        locale_type getloc() const;
    };
}
```

1 The specializations regex_traits<char> and regex_traits<wchar_t> shall be valid and shall satisfy the requirements for a regular expression traits class (30.3).
using char_class_type = bitmask_type;

The type char_class_type is used to represent a character classification and is capable of holding an implementation specific set returned by lookup_classname.

static size_t length(const char_type* p);

Returns: char_traits<charT>::length(p).

charT translate(charT c) const;

Returns: c.

charT translate_nocase(charT c) const;

Returns: use_facet<ctype<charT>>().tolower(c).

template<class ForwardIterator>
string_type transform(ForwardIterator first, ForwardIterator last) const;

Effects: As if by:

string_type str(first, last);
return use_facet<collate<charT>>(getloc()).transform(str.data(), str.data() + str.length());

template<class ForwardIterator>
string_type transform_primary(ForwardIterator first, ForwardIterator last) const;

Effects: If

typeid(use_facet<collate<charT>>()) == typeid(collate_byname<charT>)

and the form of the sort key returned by collate_byname<charT>::transform(first, last) is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

template<class ForwardIterator>
string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;

Returns: A sequence of one or more characters that represents the collating element consisting of the character sequence designated by the iterator range [first, last). Returns an empty string if the character sequence is not a valid collating element.

template<class ForwardIterator>
char_class_type lookup_classname(
    ForwardIterator first, ForwardIterator last, bool icase = false) const;

Returns: An unspecified value that represents the character classification named by the character sequence designated by the iterator range [first, last). If the parameter icase is true then the returned mask identifies the character classification without regard to the case of the characters being matched, otherwise it does honor the case of the characters being matched. The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns char_class_type().

Remarks: For regex_traits<char>, at least the narrow character names in Table 127 shall be recognized. For regex_traits<wchar_t>, at least the wide character names in Table 127 shall be recognized.

bool isctype(charT c, char_class_type f) const;

Effects: Determines if the character c is a member of the character classification represented by f.

Returns: Given the following function declaration:

// for exposition only
template<class C>
ctype_base::mask convert(typename regex_traits<C>::char_class_type f);

that returns a value in which each ctype_base::mask value corresponding to a value in f named in Table 127 is set, then the result is determined as if by:

329) For example, if the parameter icase is true then \[[:lower:]] is the same as \[[:alpha:]].
ctype_base::mask m = convert<charT>(f);
const ctype<charT>& ct = use_facet<ctype<charT>>(getloc());
if (ct.is(m, c)) {
    return true;
} else if (c == ct.widen('_')) {
    charT w[1] = { ct.widen('w') };  
    char_class_type x = lookup_classname(w, w+1);
    return (f&x) == x;
} else {
    return false;
}

[Example:
  regex_traits<char> t;
  string d("d");
  string u("upper");
  regex_traits<char>::char_class_type f;
  f = t.lookup_classname(d.begin(), d.end());
  f |= t.lookup_classname(u.begin(), u.end());
  ctype_base::mask m = convert<char>(f); // m == ctype_base::digit|ctype_base::upper
  —end example]

int value(charT ch, int radix) const;

13 Requires—Excepts: The value of radix shall be 8, 10, or 16.
14 Returns: The value represented by the digit ch in base radix if the character ch is a valid digit in base radix; otherwise returns -1.

locale_type imbue(locale_type loc);

15 Effects: Imbues this with a copy of the locale loc. [Note: Calling imbue with a different locale than the one currently in use invalidates all cached data held by *this. —end note]
16 Returns: If no locale has been previously imbued then a copy of the global locale in effect at the time of construction of *this, otherwise a copy of the last argument passed to imbue.
17 Ensures: getloc() == loc.

locale_type getloc() const;

18 Returns: If no locale has been imbued then a copy of the global locale in effect at the time of construction of *this, otherwise a copy of the last argument passed to imbue.

30.8 Class template basic_regex

1 For a char-like type charT, specializations of class template basic_regex represent regular expressions constructed from character sequences of charT characters. In the rest of 30.8, charT denotes a given char-like type. Storage for a regular expression is allocated and freed as necessary by the member functions of class basic_regex.

2 Objects of type specialization of basic_regex are responsible for converting the sequence of charT objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions. [Note: Implementations will typically declare some function templates as friends of basic_regex to achieve this —end note]

3 The functions described in this Clause report errors by throwing exceptions of type regex_error.

§ 30.8 14
Table 127 — Character class names and corresponding `ctype` masks

<table>
<thead>
<tr>
<th>Narrow character name</th>
<th>Wide character name</th>
<th>Corresponding <code>ctype_base::mask</code> value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;alnum&quot;</td>
<td>L&quot;alnum&quot;</td>
<td><code>ctype_base::alnum</code></td>
</tr>
<tr>
<td>&quot;alpha&quot;</td>
<td>L&quot;alpha&quot;</td>
<td><code>ctype_base::alpha</code></td>
</tr>
<tr>
<td>&quot;blank&quot;</td>
<td>L&quot;blank&quot;</td>
<td><code>ctype_base::blank</code></td>
</tr>
<tr>
<td>&quot;cntrl&quot;</td>
<td>L&quot;cntrl&quot;</td>
<td><code>ctype_base::cntrl</code></td>
</tr>
<tr>
<td>&quot;digit&quot;</td>
<td>L&quot;digit&quot;</td>
<td><code>ctype_base::digit</code></td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>L&quot;d&quot;</td>
<td><code>ctype_base::digit</code></td>
</tr>
<tr>
<td>&quot;graph&quot;</td>
<td>L&quot;graph&quot;</td>
<td><code>ctype_base::graph</code></td>
</tr>
<tr>
<td>&quot;lower&quot;</td>
<td>L&quot;lower&quot;</td>
<td><code>ctype_base::lower</code></td>
</tr>
<tr>
<td>&quot;print&quot;</td>
<td>L&quot;print&quot;</td>
<td><code>ctype_base::print</code></td>
</tr>
<tr>
<td>&quot;punct&quot;</td>
<td>L&quot;punct&quot;</td>
<td><code>ctype_base::punct</code></td>
</tr>
<tr>
<td>&quot;space&quot;</td>
<td>L&quot;space&quot;</td>
<td><code>ctype_base::space</code></td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>L&quot;s&quot;</td>
<td><code>ctype_base::space</code></td>
</tr>
<tr>
<td>&quot;upper&quot;</td>
<td>L&quot;upper&quot;</td>
<td><code>ctype_base::upper</code></td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>L&quot;w&quot;</td>
<td><code>ctype_base::alnum</code></td>
</tr>
<tr>
<td>&quot;xdigit&quot;</td>
<td>L&quot;xdigit&quot;</td>
<td><code>ctype_base::xdigit</code></td>
</tr>
</tbody>
</table>

```cpp
namespace std {
  template<class charT, class traits = regex_traits<charT>>
  class basic_regex {
    public:
      // types
      using value_type = charT;
      using traits_type = traits;
      using string_type = typename traits::string_type;
      using flag_type = regex_constants::syntax_option_type;
      using locale_type = typename traits::locale_type;
      
      // 30.5.1, constants
      static constexpr flag_type icase = regex_constants::icase;
      static constexpr flag_type nosubs = regex_constants::nosubs;
      static constexpr flag_type optimize = regex_constants::optimize;
      static constexpr flag_type collate = regex_constants::collate;
      static constexpr flag_type ECMAScript = regex_constants::ECMAScript;
      static constexpr flag_type basic = regex_constants::basic;
      static constexpr flag_type extended = regex_constants::extended;
      static constexpr flag_type awk = regex_constants::awk;
      static constexpr flag_type grep = regex_constants::grep;
      static constexpr flag_type egrep = regex_constants::egrep;
      static constexpr flag_type multiline = regex_constants::multiline;
      
      // 30.8.1, construct/copy/destroy
      basic_regex();
      explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
      basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
      basic_regex(const basic_regex&); 
      basic_regex(basic_regex&&) noexcept;
      template<class ST, class SA>
      explicit basic_regex(const basic_string<charT, ST, SA>& p,
                      flag_type f = regex_constants::ECMAScript);
      template<class ForwardIterator>
      basic_regex(ForwardIterator first, ForwardIterator last,
                  flag_type f = regex_constants::ECMAScript);
      basic_regex(initializer_list<charT>, flag_type = regex_constants::ECMAScript);
  ~basic_regex();
}
```

§ 30.8
basic_regex& operator=(const basic_regex&); 
basic_regex& operator=(basic_regex&&) noexcept; 
basic_regex& operator=(const charT* ptr); 
basic_regex& operator=(initializer_list<charT> il); 
template<class ST, class SA>
  basic_regex& operator=(const charT* ptr); 
operator=(const basic_string<charT, ST, SA>& p); 
// 30.8.2, assign
basic_regex& assign(const basic_regex& that); 
basic_regex& assign(basic_regex&& that) noexcept; 
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript); 
template<class string_traits, class A>
  basic_regex& assign(const string_traits<charT, A>& s, 
  flag_type f = regex_constants::ECMAScript); 
operator=(const basic_string<charT, string_traits, A>& s, 
  flag_type f = regex_constants::ECMAScript); 
operator=(initializer_list<charT>, 
  flag_type = regex_constants::ECMAScript); 
// 30.8.3, const operations
unsigned mark_count() const; 
flag_type flags() const; 
// 30.8.4, locale
locale_type imbue(locale_type loc); 
locale_type getloc() const; 
// 30.8.5, swap
void swap(basic_regex&); 
};

template<class ForwardIterator>
  basic_regex(ForwardIterator, ForwardIterator, 
  regex_constants::syntax_option_type = regex_constants::ECMAScript)
  -> basic_regex<typename iterator_traits<ForwardIterator>::value_type>;

30.8.1 Constructors [re.regex.construct]

basic_regex();
  Effects: Constructs an object of class basic_regex that does not match any character sequence. 
Ensures: *this does not match any character sequence.

explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
  Requires: p shall not be a null pointer.  Expects: [p, p+char_traits<charT>::length(p)) is a valid range.
  Throws: regex_error if [p, p+char_traits<charT>::length(p)) is not a valid regular expression.
  Effects: Constructs an object of class basic_regex: the object’s internal finite state machine is constructed from the regular expression contained in the array of charT of length char_traits<charT>::length(p), whose first element is designated by sequence of characters [p, p+char_traits<charT>::length(p)], and interpreted according to the flags f.
  Ensures: flags() returns f. mark_count() returns the number of marked sub-expressions within the expression.

basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
  Requires: p shall not be a null pointer.  Expects: [p, p+len) is a valid range.
  Throws: regex_error if [p, p+len) is not a valid regular expression.

§ 30.8.1 16
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 \([p, p+\text{len})\), and interpreted according the flags specified in \(f\).

Ensures: `flags()` returns \(f\). `mark_count()` returns the number of marked sub-expressions within the expression.

```cpp
basic_regex(const basic_regex& e);
Effects: Constructs an object of class `basic_regex` as a copy of the object \(e\).
Ensures: `flags()` and `mark_count()` return \(e\).`flags()` and \(e\).`mark_count()`, respectively.
```

```cpp
basic_regex(basic_regex&& e) noexcept;
Effects: Move constructs an object of class `basic_regex` from \(e\).
Ensures: `flags()` and `mark_count()` return the values that \(e\).`flags()` and \(e\).`mark_count()`, respectively, had before construction. \(e\) is in a valid state with unspecified value.
```

```cpp
template<class ST, class SA>
explicit basic_regex(const basic_string<charT, ST, SA>& s,
                     flag_type f = regex_constants::ECMAScript);
Throws: `regex_error` if \(s\) is not a valid regular expression.
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 flags specified in \(f\).
Ensures: `flags()` returns \(f\). `mark_count()` returns the number of marked sub-expressions within the expression.
```

```cpp
template<class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
             flag_type f = regex_constants::ECMAScript);
Throws: `regex_error` if the sequence \([\text{first}, \text{last})\) is not a valid regular expression.
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 \([\text{first}, \text{last})\), and interpreted according to the flags specified in \(f\).
Ensures: `flags()` returns \(f\). `mark_count()` returns the number of marked sub-expressions within the expression.
```

```cpp
basic_regex(initializer_list<charT> il, flag_type f = regex_constants::ECMAScript);
Effects: Same as `basic_regex(il.begin(), il.end(), f)`.
```

### 30.8.2 Assignment

```cpp
basic_regex& operator=(const basic_regex& e);
Effects: Copies \(e\) into \(*\text{this}\) and returns \(*\text{this}\).
Ensures: `flags()` and `mark_count()` return \(e\).`flags()` and \(e\).`mark_count()`, respectively.
```

```cpp
basic_regex& operator=(basic_regex&& e) noexcept;
Effects: Move assigns from \(e\) into \(*\text{this}\) and returns \(*\text{this}\).
Ensures: `flags()` and `mark_count()` return the values that \(e\).`flags()` and \(e\).`mark_count()` had before assignment. \(e\) is in a valid state with unspecified value.
```

```cpp
basic_regex& operator=(const charT* ptr);
Requirements: \(\text{ptr}\) shall not be a null pointer.
Effects: Returns `assign(ptr)`.
```

```cpp
basic_regex& operator=(initializer_list<charT> il);
Effects: Returns `assign(il.begin(), il.end())`.
```
template<class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>& p);

Effects: Returns assign(p).

basic_regex& assign(const basic_regex& that);

Effects: Equivalent to: return *this = that;

basic_regex& assign(basic_regex&& that) noexcept;

Effects: Equivalent to: return *this = std::move(that);

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

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

basic_regex& assign(const charT* ptr, size_t len, flag_type f = regex_constants::ECMAScript);

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::ECMAScript);

Throws: regex_error if s is not a valid regular expression.

Returns: *this.

Effects: Assigns the regular expression contained in the string s, interpreted according the flags specified in f. If an exception is thrown, *this is unchanged.

Ensures: If no exception is thrown, flags() returns f and mark_count() returns the number of marked sub-expressions within the expression.

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

Requires: InputIterator shall satisfy meets the Cpp17InputIterator requirements (??).

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

basic_regex& assign(initializer_list<charT> il, 
                    flag_type f = regex_constants::ECMAScript);

Effects: Same as assign(il.begin(), il.end(), f).

Returns: *this.

30.8.3 Constant operations [re.regex.operations]

unsigned mark_count() const;

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

flag_type flags() const;

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

30.8.4 Locale [re.regex.locale]

locale_type imbue(locale_type loc);

Effects: Returns the result of traits_inst.imbue(loc) where traits_inst is a (default-initialized) instance of the template type argument traits stored within the object. After a call to imbue the basic_regex object does not match any character sequence.

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.
30.8.5 Swap

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

- **Effects:** Swaps the contents of the two regular expressions.
- **Ensures:** *this contains the regular expression that was in e, e contains the regular expression that was in *this.
- **Complexity:** Constant time.

30.8.6 Non-member functions

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

- **Effects:** Calls lhs.swap(rhs).

30.9 Class template sub_match

- **Class template** `sub_match` denotes the sequence of characters matched by a particular marked sub-expression.

```cpp
namespace std {
    template<class BidirectionalIterator>
    class sub_match : public pair<BidirectionalIterator, BidirectionalIterator> {
        public:
            using value_type =
                typename iterator_traits<BidirectionalIterator>::value_type;
            using difference_type =
                typename iterator_traits<BidirectionalIterator>::difference_type;
            using iterator = BidirectionalIterator;
            using string_type = basic_string<value_type>;

            bool matched;
            constexpr sub_match();
            difference_type length() const;
            operator string_type() const;
            string_type str() const;
            int compare(const sub_match& s) const;
            int compare(const string_type& s) const;
            int compare(const value_type* s) const;
    };
}
```

30.9.1 Members

```cpp
constexpr sub_match();
```

- **Effects:** Value-initializes the pair base class subobject and the member matched.

```cpp
difference_type length() const;
```

- **Returns:** matched ? distance(first, second) : 0.

```cpp
operator string_type() const;
```

- **Returns:** matched ? string_type(first, second) : string_type().

```cpp
string_type str() const;
```

- **Returns:** matched ? string_type(first, second) : string_type().

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

- **Returns:** str().compare(s.str()).

```cpp
int compare(const string_type& s) const;
```

- **Returns:** str().compare(s).
int compare(const value_type* s) const;
    Returns: str().compare(s).

30.9.2 Non-member operators

template<class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    Returns: lhs.compare(rhs) == 0.

template<class BiIter>
    bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    Returns: lhs.compare(rhs) != 0.

template<class BiIter>
    bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    Returns: lhs.compare(rhs) < 0.

template<class BiIter>
    bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    Returns: lhs.compare(rhs) > 0.

template<class BiIter>
    bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    Returns: lhs.compare(rhs) <= 0.

template<class BiIter>
    bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
    Returns: lhs.compare(rhs) >= 0.

template<class BiIter, class ST, class SA>
    bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
    Returns: rhs.compare(typename sub_match<BiIter>::string_type(lhs.data(), lhs.size())) == 0.

template<class BiIter, class ST, class SA>
    bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
    Returns: !(lhs == rhs).

template<class BiIter, class ST, class SA>
    bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
    Returns: rhs.compare(typename sub_match<BiIter>::string_type(lhs.data(), lhs.size())) > 0.

template<class BiIter, class ST, class SA>
    bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
    Returns: rhs < lhs.

template<class BiIter, class ST, class SA>
    bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
    Returns: !(rhs < lhs).

template<class BiIter, class ST, class SA>
    bool operator>=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);
    Returns: !(rhs < lhs).
template<class BiIter, class ST, class SA>
bool operator>=(
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);

Returns: !(lhs < rhs).

12

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

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

13

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

Returns: !(lhs == rhs).

14

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

Returns: rhs.compare(lhs) > 0.

15

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

Returns: !(lhs == rhs).

16

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

Returns: !(lhs > rhs).

17

template<class BiIter, class ST, class SA>
bool operator<="(const typename iterator_traits<BiIter>::value_type* lhs,
                const sub_match<BiIter>& rhs);

Returns: !(lhs > rhs).

18

template<class BiIter, class ST, class SA>
bool operator>="(const typename iterator_traits<BiIter>::value_type* lhs,
                const sub_match<BiIter>& rhs);

Returns: !(lhs < rhs).

19

§ 30.9.2 21
template<class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type* lhs, 
const sub_match<BiIter>& rhs);

Returns: rhs < lhs.

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

Returns: !(rhs < lhs).

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

Returns: !(lhs < rhs).

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

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

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

Returns: !(lhs == rhs).

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

Returns: lhs.compare(rhs) < 0.

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

Returns: rhs < lhs.

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

Returns: !(rhs < lhs).

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

Returns: !(lhs < rhs).

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

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) == 0.

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

Returns: !(lhs == rhs).

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

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) > 0.
template<
class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type& lhs,
            const sub_match<BiIter>& rhs);

Returns: rhs < lhs.

34

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

Returns: !(rhs < lhs).

35

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

Returns: !(lhs < rhs).

36

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

Returns: lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) == 0.

37

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

Returns: !(lhs == rhs).

38

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

Returns: rhs < lhs.

39

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

Returns: rhs < lhs.

40

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

Returns: !(rhs < lhs).

41

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

Returns: !(lhs < rhs).

42

template<class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

Returns: os << m.str().

43

30.10 Class template match_results

Class template match_results denotes a collection of character sequences representing the result of a regular
expression match. Storage for the collection is allocated and freed as necessary by the member functions of
class template match_results.

The class template match_results satisfies the requirements of an allocator-aware container and of a
sequence container (??, ??) except that only copy assignment, move assignment, and operations defined
for const-qualified sequence containers are supported and that the semantics of comparison functions are
different from those required for a container.
A default-constructed `match_results` object has no fully established result state. A match result is `ready` when, as a consequence of a completed regular expression match modifying such an object, its result state becomes fully established. The effects of calling most member functions from a `match_results` object that is not ready are undefined.

The `sub_match` object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the `sub_match` member `matched` is always `true`. The `sub_match` object stored at index `n` denotes what matched the marked sub-expression `n` within the matched expression. If the sub-expression `n` participated in a regular expression match then the `sub_match` 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` point to the end of the sequence that was searched.  

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

        // 30.10.1, construct/copy/destroy
        match_results() : match_results(Allocator()) {}
        explicit match_results(const Allocator&);
        match_results(const match_results& m);
        match_results(match_results&& m) noexcept;
        match_results& operator=(const match_results& m);
        match_results& operator=(match_results&& m);
        ~match_results();

        // 30.10.2, state
        bool ready() const;

        // 30.10.3, size
        size_type size() const;
        size_type max_size() const;
        [[nodiscard]] bool empty() const;

        // 30.10.4, element access
        difference_type length(size_type sub = 0) const;
        difference_type position(size_type sub = 0) const;
        string_type str(size_type sub = 0) const;
        const_reference operator[](size_type n) const;

        const_reference prefix() const;
        const_reference suffix() const;
        const_iterator begin() const;
        const_iterator end() const;
        const_iterator cbegin() const;
        const_iterator cend() const;
    }
```
### 30.10.1 Constructors

**explicit** match_results(const Allocator& a);

1. **Effects:** Constructs an object of class `match_results`.
2. **Ensures:** `ready()` returns `false`, `size()` returns `0`.

**match_results(const match_results& m);**

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

**match_results(match_results&& m) noexcept;**

4. **Effects:** Move constructs an object of class `match_results` from `m` satisfying the same postconditions as Table 128. Additionally, the stored `Allocator` value is move constructed from `m.get_allocator()`.
5. **Throws:** Nothing.

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

6. **Effects:** Assigns `m` to `*this`. The postconditions of this function are indicated in Table 128.

**match_results& operator=(match_results&& m);**

7. **Effects:** Move-assigns `m` to `*this`. The postconditions of this function are indicated in Table 128.

<table>
<thead>
<tr>
<th>Table 128 — match_results assignment operator effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>ready()</td>
</tr>
<tr>
<td>size()</td>
</tr>
<tr>
<td>str(n)</td>
</tr>
<tr>
<td>prefix()</td>
</tr>
<tr>
<td>suffix()</td>
</tr>
<tr>
<td>(*this)[n]</td>
</tr>
<tr>
<td>length(n)</td>
</tr>
<tr>
<td>position(n)</td>
</tr>
</tbody>
</table>

§ 30.10.1
30.10.2 State

```cpp
bool ready() const;
```

*Returns:* `true` if `*this` has a fully established result state, otherwise `false`.

30.10.3 Size

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

*Returns:* One plus the number of marked sub-expressions in the regular expression that was matched if `*this` represents the result of a successful match. Otherwise returns 0. *[Note: The state of a `match_results` object can be modified only by passing that object to `regex_match` or `regex_search`. Subclauses 30.11.2 and 30.11.3 specify the effects of those algorithms on their `match_results` arguments. —end note]*

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

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

```cpp
[[nodiscard]] bool empty() const;
```

*Returns:* `size() == 0`.

30.10.4 Element access

```cpp
difference_type length(size_type sub = 0) const;
```

*Requires—Expects:* `ready() == true`.

*Returns:* `(*this)[sub].length()`.

```cpp
difference_type position(size_type sub = 0) const;
```

*Requires—Expects:* `ready() == true`.

*Returns:* The distance from the start of the target sequence to `(*this)[sub].first`.

```cpp
string_type str(size_type sub = 0) const;
```

*Requires—Expects:* `ready() == true`.

*Returns:* `string_type(*this)[sub])`.

```cpp
const_reference operator[](size_type n) const;
```

*Requires—Expects:* `ready() == true`.

*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 >= size()` then returns a `sub_match` object representing an unmatched sub-expression.

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

*Requires—Expects:* `ready() == true`.

*Returns:* A reference to the `sub_match` object representing the character sequence from the start of the string being matched/searched to the start of the match found.

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

*Requires—Expects:* `ready() == true`.

*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/searched.

```cpp
const_iterator begin() const;
const_iterator cbegin() const;
```

*Returns:* A starting iterator that enumerates over all the sub-expressions stored in `*this`.

```cpp
const_iterator end() const;
```
const_iterator cend() const;

Returns: A terminating iterator that enumerates over all the sub-expressions stored in *this.

30.10.5 Formatting


template<class OutputIter>
OutputIter format(
    OutputIter out,
    const char_type* fmt_first, const char_type* fmt_last,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;

Requires: ready() == true and OutputIter shall satisfy meets
Effects: Copies the character sequence [fmt_first, fmt_last) to OutputIter out. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within *this to which it refers. The bitmasks specified in flags determine which format specifiers and escape sequences are recognized.

Returns: out.


typedef basic_string<char_type, ST, SA> string_type;

string_type format(
    const char_type* fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;

Requires: ready() == true.
Effects: Constructs an empty string result of type basic_string<char_type, ST, SA> and calls:
format(back_inserter(result), fmt, flags);

Returns: result.

30.10.6 Allocator

allocator_type get_allocator() const;

Returns: A copy of the Allocator that was passed to the object’s constructor or, if that allocator has been replaced, a copy of the most recent replacement.

30.10.7 Swap

void swap(match_results& that);

Effects: Swaps the contents of the two sequences.
Ensures: *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.


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

**Effects:** As if by `m1.swap(m2)`.

### 30.10.8 Non-member functions

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

**Returns:** `true` if neither match result is ready, `false` if one match result is ready and the other is not. If both match results are ready, returns `true` only if:

- (1.1) `m1.empty() && m2.empty()`, or
- (1.2) `!m1.empty() && !m2.empty()`, and the following conditions are satisfied:

  - (1.2.1) `m1.prefix() == m2.prefix()`,
  - (1.2.2) `m1.size() == m2.size() && equal(m1.begin(), m1.end(), m2.begin())`, and
  - (1.2.3) `m1.suffix() == m2.suffix()`.

**[Note: The algorithm equal is defined in ??...—end note]**

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

**Returns:** `!(m1 == m2)`.

### 30.11 Regular expression algorithms

#### 30.11.1 Exceptions

The algorithms described in this subclause may throw an exception of type `regex_error`. If such an exception `e` is thrown, `e.code()` shall return either `regex_constants::error_complexity` or `regex_constants::error_stack`.

#### 30.11.2 `regex_match`

```cpp
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,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

**Requires—Expects:** The type `BidirectionalIterator` shall satisfy the `Cpp17BidirectionalIterator` requirements (??).

**Effects:** Determines whether there is a match between the regular expression `e`, and all of the character sequence `[first, last)`. The parameter `flags` is used to control how the expression is matched against the character sequence. When determining if there is a match, only potential matches that match the entire character sequence are considered. Returns `true` if such a match exists, `false` otherwise. **[Example]**

```cpp
std::regex re("Get|GetValue");
std::cmatch m;
regex_search("GetValue", m, re);       // returns true, and m[0] contains "Get"
regex_match ("GetValue", m, re);       // returns true, and m[0] contains "GetValue"
regex_search("GetValues", m, re);     // returns true, and m[0] contains "Get"
regex_match ("GetValues", m, re);     // returns false
```

**Ensures:** `m.ready()` == `true` in all cases. If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in Table 129.
### Table 129 — Effects of regex_match algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m[0].first</td>
<td>first</td>
</tr>
<tr>
<td>m[0].second</td>
<td>last</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers 0 &lt; n &lt; 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.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers 0 &lt; n &lt; 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.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers 0 &lt; n &lt; m.size(), true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

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

**Returns:** `regex_match(str, str + char_traits<charT>::length(str), m, e, flags)`.

**template<class ST, class SA, class charT, class traits>**
```cpp
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,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

**Returns:** `regex_match(s.begin(), s.end(), m, e, flags)`.

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

**Returns:** `regex_match(str, str + char_traits<charT>::length(str), e, flags)`.

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

**Returns:** `regex_match(s.begin(), s.end(), e, flags)`.
30.11.3 \texttt{regex\_search} \quad \textbf{[re.alg.search]} \\

\texttt{template<class BiDirectionalIterator, class Allocator, class charT, class traits>}

\texttt{bool regex\_search(BiDirectionalIterator \texttt{first}, BiDirectionalIterator \texttt{last},}

\texttt{match\_results<BiDirectionalIterator, Allocator>& \texttt{m},}

\texttt{const basic\_regex<charT, traits>& \texttt{e},}

\texttt{regex\_constants::match\_flag\_type flags = regex\_constants::match\_default);} \\

\textbf{1} \quad \textit{Requires–\textbf{Expects:}} Type \texttt{BiDirectionalIterator} shall satisfy the \texttt{Cpp17BiDirectionalIterator} requirements.\textbf{[??]}.

\textbf{2} \quad \textit{Effects:} Determines whether there is some sub-sequence within \texttt{[first, last)} that matches the regular expression \texttt{e}. The parameter \texttt{flags} is used to control how the expression is matched against the character sequence. Returns \texttt{true} if such a sequence exists, \texttt{false} otherwise.

\textbf{3} \quad \textit{Ensures:} \texttt{m.ready()} == \texttt{true} in all cases. If the function returns \texttt{false}, then the effect on parameter \texttt{m} is unspecified except that \texttt{m.size()} returns 0 and \texttt{m.empty()} returns \texttt{true}. Otherwise the effects on parameter \texttt{m} are given in Table 130.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Element & Value \\
\hline
\texttt{m.size()} & \texttt{1 + e.mark\_count()} \\
\texttt{m.empty()} & \texttt{false} \\
\texttt{m.prefix().first} & \texttt{first} \\
\texttt{m.prefix().second} & \texttt{m[0].first} \\
\texttt{m.prefix().matched} & \texttt{m.prefix().first \neq m.prefix().second} \\
\texttt{m.suffix().first} & \texttt{m[0].second} \\
\texttt{m.suffix().second} & \texttt{last} \\
\texttt{m.suffix().matched} & \texttt{m.suffix().first \neq m.suffix().second} \\
\texttt{m[0].first} & The start of the sequence of characters that matched the regular expression \\
\texttt{m[0].second} & The end of the sequence of characters that matched the regular expression \\
\texttt{m[0].matched} & \texttt{true} \\
\texttt{m[n].first} & For all integers \texttt{0 < n < m.size()}, the start of the sequence that matched sub-expression \texttt{n}. Alternatively, if sub-expression \texttt{n} did not participate in the match, then \texttt{last}. \\
\texttt{m[n].second} & For all integers \texttt{0 < n < m.size()}, the end of the sequence that matched sub-expression \texttt{n}. Alternatively, if sub-expression \texttt{n} did not participate in the match, then \texttt{last}. \\
\texttt{m[n].matched} & For all integers \texttt{0 < n < m.size()}, \texttt{true} if sub-expression \texttt{n} participated in the match, \texttt{false} otherwise. \\
\hline
\end{tabular}
\caption{Effects of \texttt{regex\_search} algorithm}
\end{table}

\texttt{template<class charT, class Allocator, class traits>}

\texttt{bool regex\_search(const charT* \texttt{str}, match\_results<const charT*, Allocator>& \texttt{m},}

\texttt{const basic\_regex<charT, traits>& \texttt{e},}

\texttt{regex\_constants::match\_flag\_type flags = regex\_constants::match\_default);} \\

\textbf{4} \quad \textit{Returns:} \texttt{regex\_search(str, str + char\_traits<charT>::length(str), m, e, flags)}.

\texttt{template<class ST, class SA, class Allocator, class charT, class traits>}

\texttt{bool regex\_search(const basic\_string<charT, ST, SA>& \texttt{s},}

\texttt{match\_results<typename basic\_string<charT, ST, SA>::const\_iterator,}

\texttt{Allocator>& \texttt{m},}

\texttt{const basic\_regex<charT, traits>& \texttt{e},}

\texttt{regex\_constants::match\_flag\_type flags = regex\_constants::match\_default);} \\

\textbf{5} \quad \textit{Returns:} \texttt{regex\_search(s.begin()), s.end(), m, e, flags)}.

\section*{§ 30.11.3}
template<class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

Effects: Behaves “as if” by constructing an object \texttt{what} of type \texttt{match_results<BidirectionalIterator>} and returning \texttt{regex_search(first, last, what, e, flags)}.

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

Returns: \texttt{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,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: \texttt{regex_search(s.begin(), s.end(), e, flags)}.

30.11.4 \texttt{regex_replace} \hfill \texttt{[re.alg.replace]}

\begin{verbatim}
template<class OutputIterator, class BidirectionalIterator, 
         class traits, class charT, class ST, class SA>
OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags = regex_constants::match_default);

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 charT* fmt,
                regex_constants::match_flag_type flags = regex_constants::match_default);
\end{verbatim}

Effects: Constructs a \texttt{regex_iterator} object \texttt{i} as if by

\begin{verbatim}
regex_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags)
\end{verbatim}

and uses \texttt{i} to enumerate through all of the matches \texttt{m} of type \texttt{match_results<BidirectionalIterator>} that occur within the sequence \texttt{[first, last)}. If no such matches are found and !(flags & regex_constants::format_no_copy), then calls

\begin{verbatim}
out = copy(first, last, out)
\end{verbatim}

If any matches are found then, for each such match:

\begin{enumerate}
\item If !(flags & regex_constants::format_no_copy), calls
\begin{verbatim}
out = copy(m.prefix().first, m.prefix().second, out)
\end{verbatim}
\item Then calls
\begin{verbatim}
out = m.format(out, fmt, flags)
\end{verbatim}
for the first form of the function and
\begin{verbatim}
out = m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)
\end{verbatim}
for the second.
\end{enumerate}

Finally, if such a match is found and !(flags & regex_constants::format_first_only) is nonzero, then only the first match found is replaced.

Returns: \texttt{out}.
template<class traits, class charT, class ST, class SA, class FST, class FSA>
    basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, FST, FSA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT, class ST, class SA>
    basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

    Effects: Constructs an empty string result of type basic_string<charT, ST, SA> and calls:
             regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags);

    Returns: result.

template<class traits, class charT, class ST, class SA>
    basic_string<charT>
    regex_replace(const charT* s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, ST, SA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

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

    Effects: Constructs an empty string result of type basic_string<charT> and calls:
             regex_replace(back_inserter(result), s, s + char_traits<charT>::length(s), e, fmt, flags);

    Returns: result.

30.12 Regular expression iterators [re.iter]
30.12.1 Class template regex_iterator [re.regiter]

The class template regex_iterator is an iterator adaptor. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A regex_iterator uses regex_search to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time operator++ is used, the iterator finds and stores a value of match_results<BidirectionalIterator>. If the end of the sequence is reached (regex_search returns false), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>* is returned. It is impossible to store things into regex_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

namespace std {
    template<class BidirectionalIterator,
             class charT = typename iterator_traits<BidirectionalIterator>::value_type,
             class traits = regex_traits<charT>>
    class regex_iterator {
public:
    using regex_type = basic_regex<charT, traits>;
    using iterator_category = forward_iterator_tag;
    using value_type = match_results<BidirectionalIterator>;
    using difference_type = ptrdiff_t;
    using pointer = const value_type*;
    using reference = const value_type&;

§ 30.12.1 32
regex_iterator();
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
  const regex_type& re,
  regex_constants::match_flag_type m = regex_constants::match_default);
regex_iterator(BidirectionalIterator, BidirectionalIterator,
  const regex_type&,
  regex_constants::match_flag_type = regex_constants::match_default) = delete;
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);

private:
  BidirectionalIterator begin; // exposition only
  BidirectionalIterator end; // exposition only
  const regex_type* pregex; // exposition only
  regex_constants::match_flag_type flags; // exposition only
  match_results<BidirectionalIterator> match; // exposition only
};

An object of type regex_iterator that is not an end-of-sequence iterator holds a zero-length match if
match[0].matched == true and match[0].first == match[0].second. [Note: For example, this can
occur when the part of the regular expression that matched consists only of an assertion (such as ^, $, '', \B). — end note]

30.12.1.1 Constructors

regex_iterator();
  Effects: Constructs an end-of-sequence iterator.

regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
  const regex_type& re,
  regex_constants::match_flag_type m = regex_constants::match_default);
  Effects: Initializes begin and end to a and b, respectively, sets pregex to addressof(re), sets flags
to m, then calls regex_search(begin, end, match, *pregex, flags). If this call returns false the
constructor sets *this to the end-of-sequence iterator.

30.12.1.2 Comparisons

bool operator==(const regex_iterator& right) const;
  Returns: true if *this and right are both end-of-sequence iterators or if the following conditions all
hold:
  (1.1)  begin == right.begin,
  (1.2)  end == right.end,
  (1.3)  pregex == right.pregex,
  (1.4)  flags == right.flags, and
  (1.5)  match[0] == right.match[0];
  otherwise false.

bool operator!=(const regex_iterator& right) const;
  Returns: !(this == right).
30.12.1.3 Indirection  \[\text{re.register.deref}\]

\[
\text{const value_type}\& \text{ operator*() const;}
\]

\text{Returns: match.}

\[
\text{const value_type}\* \text{ operator->() const;}
\]

\text{Returns: addressof(match).}

30.12.1.4 Increment  \[\text{re.register.incr}\]

regex_iterator\& \text{ operator++();}

\text{Effects: Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.}

\text{If the iterator holds a zero-length match and start == end the operator sets *this to the end-of-sequence iterator and returns *this.}

\text{Otherwise, if the iterator holds a zero-length match, the operator calls:}

\[
\text{regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous)}
\]

\text{If the call returns true the operator returns *this. Otherwise the operator increments start and continues as if the most recent match was not a zero-length match.}

\text{If the most recent match was not a zero-length match, the operator sets flags to flags | regex_constants::match_prev_avail and calls regex_search(start, end, match, *pregex, flags). If the call returns false the iterator sets *this to the end-of-sequence iterator. The iterator then returns *this.}

\text{In all cases in which the call to regex_search returns true, match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0, match.size()) for which match[i].matched is true, match.position(i) shall return distance(begin, match[i].first).}

\text{[Note: This means that match.position(i) gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to regex_search. — end note]}

\text{It is unspecified how the implementation makes these adjustments.}

\text{[Note: This means that a compiler may call an implementation-specific search function, in which case a program-defined specialization of regex_search will not be called. — end note]}

regex_iterator\& \text{ operator++(int);}

\text{Effects: As if by:}

\[
\text{regex_iterator tmp = *this; ++(*this); return tmp;}
\]

30.12.2 Class template regex_token_iterator  \[\text{re.tokiter}\]

The class template regex_token_iterator is an iterator adaptor; 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 sub-expressions for each match found. Each position enumerated by the iterator is a sub_match class template instance 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 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

After it is constructed, the iterator finds and stores a value regex_iterator\<BidirectionalIterator\> position and sets the internal count \(N\) to zero. It also maintains a sequence subs which contains a list of the sub-expressions which will be enumerated. Every time operator++ is used the count \(N\) is incremented; if \(N\) exceeds or equals subs.size(), then the iterator increments member position and sets count \(N\) to zero.
If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const sub_match<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const sub_match<BidirectionalIterator>* is returned.

It is impossible to store things into regex_token_iterator. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

namespace std {
  template<class BidirectionalIterator,
    class charT = typename iterator_traits<BidirectionalIterator>::value_type,
    class traits = regex_traits<charT>>
  class regex_token_iterator {
    public:
      using regex_type = basic_regex<charT, traits>;
      using iterator_category = forward_iterator_tag;
      using value_type = sub_match<BidirectionalIterator>;
      using difference_type = ptrdiff_t;
      using pointer = const value_type*;
      using reference = const value_type&;

      regex_token_iterator();
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re, int submatch = 0,
        regex_constants::match_flag_type m =
        regex_constants::match_default);
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        const vector<int>& submatches,
        regex_constants::match_flag_type m =
        regex_constants::match_default);
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        initializer_list<int> submatches,
        regex_constants::match_flag_type m =
        regex_constants::match_default);

      template<size_t N>
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        const int (&submatches)[N],
        regex_constants::match_flag_type m =
        regex_constants::match_default);

      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        int submatch = 0,
        regex_constants::match_flag_type m =
        regex_constants::match_default) = delete;
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        const vector<int>& submatches,
        regex_constants::match_flag_type m =
        regex_constants::match_default) = delete;
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
        const regex_type& re,
        initializer_list<int> submatches,
        regex_constants::match_flag_type m =
        regex_constants::match_default) = delete;

  };
}

§ 30.12.2
regex_token_iterator();

Effects: Constructs the end-of-sequence iterator.

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
int submatch = 0,
regex_constants::match_flag_type m = regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
const vector<int>& submatches,
regex_constants::match_flag_type m = regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
initializer_list<int> submatches,
regex_constants::match_flag_type m = regex_constants::match_default);

template<size_t N>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
const int (&submatches)[N],
regex_constants::match_flag_type m = regex_constants::match_default) = delete;

A suffix iterator is a regex_token_iterator object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member \texttt{result} holds a pointer to the data member \texttt{suffix}, the value of the member \texttt{suffix.match} is true, \texttt{suffix.first} points to the beginning of the final sequence, and \texttt{suffix.second} points to the end of the final sequence.

[Note: For a suffix iterator, data member \texttt{suffix.first} is the same as the end of the last match found, and \texttt{suffix.second} is the same as the end of the target sequence — end note]

The current match is (\texttt{*position}).prefix() if \texttt{sub[N]} == -1, or (\texttt{*position})[\texttt{sub[N]}] for any other value of \texttt{sub[N]}.

30.12.2.1 Constructors [re.tokiter.cnstr]

\texttt{regex_token_iterator};

\texttt{Effects: Constructs the end-of-sequence iterator.}

\texttt{template<size_t N>}
\texttt{regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
const int (&submatches)[N],
regex_constants::match_flag_type m = regex_constants::match_default) = delete;
Effects: The first constructor initializes the member `subs` to hold the single value `submatch`. The second, third, and fourth constructors initialize the member `subs` to hold a copy of the sequence of integer values pointed to by the iterator range `[begin(submatches), end(submatches))].

Each constructor then sets `N` to 0, and `position` to `position_iterator(a, b, re, m)`. If `position` is not an end-of-sequence iterator the constructor sets `result` to the address of the current match. Otherwise if any of the values stored in `subs` is equal to -1 the constructor sets `*this` to a suffix iterator that points to the range `[a, b)`, otherwise the constructor sets `*this` to an end-of-sequence iterator.

### 30.12.2.2 Comparisons

```cpp
bool operator==(const regex_token_iterator& right) const;
```

Returns: `true` if `*this` and `right` are both end-of-sequence iterators, or if `*this` and `right` are both suffix iterators and `suffix == right.suffix`; otherwise returns `false` if `*this` or `right` is an end-of-sequence iterator or a suffix iterator. Otherwise returns `true` if `position == right.position`, `N == right.N`, and `subs == right.subs`. Otherwise returns `false`.

```cpp
bool operator!=(const regex_token_iterator& right) const;
```

Returns: `!(*this == right)`.

### 30.12.2.3 Indirection

```cpp
const value_type& operator*() const;
```

Returns: `*result`.

```cpp
const value_type* operator->() const;
```

Returns: `result`.

### 30.12.2.4 Increment

```cpp
regex_token_iterator& operator++();
```

Effects: Constructs a local variable `prev` of type `position_iterator`, initialized with the value of `position`.

- If `*this` is a suffix iterator, sets `*this` to an end-of-sequence iterator.
- Otherwise, if `N + 1 < subs.size()`, increments `N` and sets `result` to the address of the current match.
- Otherwise, sets `N` to 0 and increments `position`. If `position` is not an end-of-sequence iterator the operator sets `result` to the address of the current match.
- Otherwise, if any of the values stored in `subs` is equal to -1 and `prev->suffix().length()` is not 0 the operator sets `*this` to a suffix iterator that points to the range `[prev->suffix().first, prev->suffix().second)`. Otherwise, sets `*this` to an end-of-sequence iterator.
- Returns: `*this`.

```cpp
regex_token_iterator& operator++(int);
```

Effects: Constructs a copy `tmp` of `*this`, then calls `++(*this)`.
- Returns: `tmp`.

### 30.13 Modified ECMAScript regular expression grammar

The regular expression grammar recognized by `basic_regex` objects constructed with the ECMAScript flag is that specified by ECMA-262, except as specified below.

Objects of type specialization of `basic_regex` store within themselves a default-constructed instance of their `traits` template parameter, henceforth referred to as `traits_inst`. This `traits_inst` object is used to support localization of the regular expression; `basic_regex` member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.

The following productions within the ECMAScript grammar are modified as follows:
The following new productions are then added:

```
ClassAtomExClass :
    [: ClassName :]
```

```
ClassAtomCollatingElement :
    [. ClassName .]
```

```
ClassAtomEquivalence :
    [= ClassName =]
```

```
ClassName :
    ClassNameCharacter
    ClassNameCharacter ClassName
```

```
ClassNameCharacter :
    SourceCharacter but not one of "." ";" ";-
```

The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in regular expressions in POSIX.

The regular expression grammar may be modified by any `regex_constants::syntax_option_type` flags specified when constructing an object of type specialization of `basic_regex` according to the rules in Table 124.

A `ClassName` production, when used in `ClassAtomExClass`, is not valid if `traits_inst.lookup_classname` returns zero for that name. The names recognized as valid `ClassName`s are determined by the type of the traits class, but at least the following names shall be recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, `w`. In addition the following expressions shall be equivalent:

- `\d` and `[:digit:]`
- `\D` and `[![:digit:]]`
- `\s` and `[:space:]`
- `\S` and `[![:space:]]`
- `\w` and `[[:alnum:]]`
- `\W` and `[![:alnum:]]`

A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string.

The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR'ed together and subsequently passed to `traits_inst.isctype`.

A `ClassName` production when used in a `ClassAtomEquivalence` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string or if the value returned by `traits_inst.transform_primary` for the result of the call to `traits_inst.lookup_collatename` is an empty string.

When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type `regex_error`. 
If the CV of a UnicodeEscapeSequence is greater than the largest value that can be held in an object of type charT the translator shall throw an exception object of type regex_error. [Note: This means that values of the form "uxxxx" that do not fit in a character are invalid. — end note]

Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling traits_inst.value.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags (30.5.2) specified when using the regular expression object in one of the regular expression algorithms (30.11). The behavior is also localized by interaction with the traits class template parameter as follows:

- During matching of a regular expression finite state machine against a sequence of characters, two characters c and d are compared using the following rules:
  - if (flags() & regex_constants::icase) the two characters are equal if traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d);
  - otherwise, if flags() & regex_constants::collate the two characters are equal if traits_inst.translate(c) == traits_inst.translate(d);
  - otherwise, the two characters are equal if c == d.

- During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range c1-c2 against a character c is conducted as follows: if flags() & regex_constants::collate is false then the character c is matched if c1 <= c && c <= c2, otherwise c is matched in accordance with the following algorithm:
  ```cpp
  string_type str1 = string_type(1,
    flags() & icase ?
      traits_inst.translate_nocase(c1) : traits_inst.translate(c1));
  string_type str2 = string_type(1,
    flags() & icase ?
      traits_inst.translate_nocase(c2) : traits_inst.translate(c2));
  string_type str = string_type(1,
    flags() & icase ?
      traits_inst.translate_nocase(c) : traits_inst.translate(c));
  return traits_inst.transform(str1.begin(), str1.end())
    <= traits_inst.transform(str.begin(), str.end())
    && traits_inst.transform(str.begin(), str.end())
    <= traits_inst.transform(str2.begin(), str2.end());
  ```

- During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using traits::transform_primary, and then comparing the sort keys for equality.

- During matching of a regular expression finite state machine against a sequence of characters, a character c is a member of a character class designated by an iterator range [first, last) if traits_inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase)) is true.

See also: ECMA-262 15.10