String_view support for regex

Mark de Wever koraq@xs4all.nl

2019-05-04

1 Introduction

This proposals adds several `string_view` overloads to the classes and functions in the `<regex>` header. This makes using the functions in `<regex>` easier when a developer uses `string_view`. It also reduces the number of temporary `string` objects created.

This proposal fixes LWG issue 3126.

2 History

Changes since the first draft.

— Updated the motivation section with before and after samples.

— Added a standard library feature test macro.

— Changed the proposed wording in 29.9.2. It is now based on LWG issue 3126.

— Improved wording and formatting.

3 Motivation

C++11 added regex support to the standard library. Its `match_results` contains a set of `sub_match` objects. These `sub_match` objects contain a view of the original input of the `regex_match` and `regex_search` functions.

C++17 added the `string_view` to the standard library. If the regex engine had been added after `string_view` I expect its design would be different. For example the `sub_match` would probably be build around `string_view` instead of `pair`.

The functions in the `<regex>` header haven't been modified to add `string_view` support. Therefore using `string_view` with the functions feels clumbersome:

— Using `regex_match` or `regex_search` with `string_view` is only possible with the iterator interface, but `string` has its own overload.

— Using the `sub_match` has a simple interface to create a `string` of the result. It is possible to create a `string_view` using the iterators but it's not easy. It encourages to use its `str()` function, which creates a temporary string. This is more expensive than creating a `string_view`.

The proposal has been implemented in libc++ of the LLVM project. The proof of concept implementation is available at GitHub.

3.1 Before and after samples

The naïve approach to get the regex working with a `string_view` was to simply create a `string` with the input. Paying for the unneeded creation of a `string`.

```
void foo(std::string_view input)
{
        std::regex re{"foo"};
        std::smatch m;
        std::string i{input};
        if(std::regex_match(i, m, re)) {
                ...
        }
}
```

The better approach avoids the creation of a `string`, but the code feels rather verbose.

```
void foo(std::string_view input)
{
        std::regex re{"foo"};
        std::match_results<std::string_view::const_iterator> m;
        if(std::regex_match(input.begin(), input.end(), m, re)) {
                ...
        }
}
```

Users may not know you can specialise `match_results`, so they still may use the naïve approach.

With this proposal the user can write the following simple version.

```
void foo(std::string_view input)
{
        std::regex re{"foo"};
        std::svmatch m;
        if(std::regex_match(input, m, re)) {
                ...
        }
}
```

In order to extract the data to a `string_view` we again have several ways:

`std::string_view sv{m[0].str()};` seems the simple solution, but it causes overhead by creating a temporary `string`. Worse, the `string_view` has been bound to a temporary that no longer exists when `sv` will be used.

`std::string_view sv(&*m[0].first, m[0].length());` feels verbose and can't use uniform initialisation since `length()` returns a `difference_type` where the constructor expects a `size_type`.

`std::string_view sv{m[0].view()};` seems the simple and safe solution.

4 Impact On the Standard

This proposal is a library only proposal. It only affects the `<regex>` header:

- Adds several function overloads and typedefs to `<regex>`.
- Adds functions returning a `string_view` from `sub_match`.
- Changes some implementation details:
  - Replaces creating temporary `string` objects with temporary `string_view` objects, which should be faster. (This claim hasn't been profiled.)
  - Lets the comparison operator use hidden friend functions.

5 Design Decisions

This design adds additional overloads and functions instead of replacing existing functions. P0506R2 attempted to replace existing functions and has been rejected. This proposal attempts not to break the existing API.

The name of the `view` function is based on P0408R5.

I based the choices for adding `noexcept` and `constexpr` to the functions on the other functions in the header. If P1149 is accepted it would make sense to add `constexpr` to several functions.

Based on LWG issue 3126 the comparison operators are hidden friend functions.

6 Questions

6.1 Implicit conversion in sub_match

The `sub_match` has an `operator string_view() const` member function. This allows an implicit conversion to a `string_view`. Since the class also has an `operator string() const` member it may make previous correct code ambiguous with this change. The question is what do we do about it:

- Nothing, we expect the case to be rare and fixing it is trivial. The creation of a `string_view` is cheaper than a `string` so the manual review is a good thing. If this option is chosen an entry needs to be added to the standard's Annex C Compatibility.
- Make the new overload explicit so it won't be implicitely selected. This changes the signature to `explicit operator string_view() const`.
- Make the new overload templated so the overload resolution prefers the non-templated conversion operator. This changes the signature to `template <class T> operator enable_if_t<is_same_v<T, string_view>, T>() const`.

6.2 Future test macro

What date should be assigned to the `__cpp_lib_string_view_regex` feature test macro?

7 Acknowledgements

I would like to thank the following persons for their input and suggestion: Arthur O'Dwyer, Jonathan Wakely, Peter Sommerlad, Thomas Köppe.

8 Proposed Wording

The modifications of standard are based on N4791

Note: The naming of function and template arguments needs a bit more polishing.
Note: The proposal will be rebased against the latest version of the standard draft before being submitted as a real proposal.

The proposed wording in 29.9.2 is based on LWG issue 3126.

16 Language support library [language.support]

16.3 Implementation properties [support.limits]

16.3.1 General [support.limits.general]

Table 36 — Standard library feature-test macros

Macro name	Value	Header(s)
`__cpp_lib_addressof_constexpr`	201603L	`<memory>`
`__cpp_lib_allocator_traits_is_always_equal`	201411L	`<memory> <scoped_allocator> <string> <deque> <forward_list> <list> <vector> <map> <set> <unordered_map> <unordered_set>`
`__cpp_lib_any`	201606L	`<any>`
`__cpp_lib_apply`	201603L	`<tuple>`
`__cpp_lib_array_constexpr`	201603L	`<iterator> <array>`
`__cpp_lib_as_const`	201510L	`<utility>`
`__cpp_lib_atomic_is_always_lock_free`	201603L	`<atomic>`

Table 36 — Standard library feature-test macros (continued)

Macro name	Value	Header(s)
__cpp_lib_atomic_ref	201806L	<atomic>
__cpp_lib_bit_cast	201806L	<bit>
__cpp_lib_bind_front	201811L	<functional>
__cpp_lib_bool_constant	201505L	<type_traits>
__cpp_lib_boyer_moore_searcher	201603L	<functional>
__cpp_lib_byte	201603L	<cstddef>
__cpp_lib_char8_t	201811L	<atomic> <filesystem> <istream> <limits> <locale> <ostream> <string> <string_view>
__cpp_lib_chrono	201611L	<chrono>
__cpp_lib_chrono_udls	201304L	<chrono>
__cpp_lib_clamp	201603L	<algorithm>
__cpp_lib_complex_udls	201309L	<complex>
__cpp_lib_concepts	201806L	<concepts>
__cpp_lib_constexpr_misc	201811L	<array> <functional> <iterator> <string_view> <tuple> <utility>
__cpp_lib_constexpr_swap_algorithms	201806L	<algorithm>
__cpp_lib_destroying_delete	201806L	<new>
__cpp_lib_enable_shared_from_this	201603L	<memory>
__cpp_lib_erase_if	201811L	<string> <deque> <forward_list> <list> <vector> <map> <set> <unordered_map> <unordered_set>
__cpp_lib_exchange_function	201304L	<utility>
__cpp_lib_execution	201603L	<execution>
__cpp_lib_filesystem	201703L	<filesystem>
__cpp_lib_gcd_lcm	201606L	<numeric>
__cpp_lib_generic_associative_lookup	201304L	<map> <set>
__cpp_lib_generic_unordered_lookup	201811L	<unordered_map> <unordered_set>
__cpp_lib_hardware_interference_size	201703L	<new>
__cpp_lib_has_unique_object_representations	201606L	<type_traits>
__cpp_lib_hypot	201603L	<cmath>
__cpp_lib_incomplete_container_elements	201505L	<forward_list> <list> <vector>
__cpp_lib_integer_sequence	201304L	<utility>
__cpp_lib_integral_constant_callable	201304L	<type_traits>
__cpp_lib_invoke	201411L	<functional>
__cpp_lib_is_aggregate	201703L	<type_traits>
__cpp_lib_is_constant_evaluated	201811L	<type_traits>
__cpp_lib_is_final	201402L	<type_traits>
__cpp_lib_is_invocable	201703L	<type_traits>
__cpp_lib_is_null_pointer	201309L	<type_traits>
__cpp_lib_is_swappable	201603L	<type_traits>
__cpp_lib_launder	201606L	<new>
__cpp_lib_list_remove_return_type	201806L	<forward_list> <list>
__cpp_lib_logical_traits	201510L	<type_traits>
__cpp_lib_make_from_tuple	201606L	<tuple>
__cpp_lib_make_reverse_iterator	201402L	<iterator>
__cpp_lib_make_unique	201304L	<memory>
__cpp_lib_map_try_emplace	201411L	<map>

Table 36 — Standard library feature-test macros (continued)

Macro name	Value	Header(s)
`__cpp_lib_math_special_functions`	201603L	`<cmath>`
`__cpp_lib_memory_resource`	201603L	`<memory_resource>`
`__cpp_lib_node_extract`	201606L	`<map> <set> <unordered_map> <unordered_set>`
`__cpp_lib_nonmember_container_access`	201411L	`<iterator> <array> <deque> <forward_list> <list> <map> <regex> <set> <string> <unordered_map> <unordered_set> <vector>`
`__cpp_lib_not_fn`	201603L	`<functional>`
`__cpp_lib_null_iterators`	201304L	`<iterator>`
`__cpp_lib_optional`	201606L	`<optional>`
`__cpp_lib_parallel_algorithm`	201603L	`<algorithm> <numeric>`
`__cpp_lib_quoted_string_io`	201304L	`<iomanip>`
`__cpp_lib_ranges`	201811L	`<algorithm> <functional> <iterator> <memory> <ranges>`
`__cpp_lib_raw_memory_algorithms`	201606L	`<memory>`
`__cpp_lib_result_of_sfinae`	201210L	`<functional> <type_traits>`
`__cpp_lib_robust_nonmodifying_seq_ops`	201304L	`<algorithm>`
`__cpp_lib_sample`	201603L	`<algorithm>`
`__cpp_lib_scoped_lock`	201703L	`<mutex>`
`__cpp_lib_shared_mutex`	201505L	`<shared_mutex>`
`__cpp_lib_shared_ptr_arrays`	201611L	`<memory>`
`__cpp_lib_shared_ptr_weak_type`	201606L	`<memory>`
`__cpp_lib_shared_timed_mutex`	201402L	`<shared_mutex>`
`__cpp_lib_string_udls`	201304L	`<string>`
`__cpp_lib_string_view`	201606L	`<string> <string_view>`
`__cpp_lib_string_view_regex`	201901L	`<regex>`
`__cpp_lib_three_way_comparison`	201711L	`<compare>`
`__cpp_lib_to_chars`	201611L	`<charconv>`
`__cpp_lib_transformation_trait_aliases`	201304L	`<type_traits>`
`__cpp_lib_transparent_operators`	201510L	`<memory> <functional>`
`__cpp_lib_tuple_element_t`	201402L	`<tuple>`
`__cpp_lib_tuples_by_type`	201304L	`<utility> <tuple>`
`__cpp_lib_type_trait_variable_templates`	201510L	`<type_traits>`
`__cpp_lib_uncaught_exceptions`	201411L	`<exception>`
`__cpp_lib_unordered_map_try_emplace`	201411L	`<unordered_map>`
`__cpp_lib_variant`	201606L	`<variant>`
`__cpp_lib_void_t`	201411L	`<type_traits>`

29 Regular expressions library [re]

29.3 Requirements [re.req]

Table 123 — Regular expression traits class requirements

Expression	Return type	Assertion/note pre-/post-condition
`X::char_type`	`charT`	The character container type used in the implementation of class template `basic_regex`.

Table 123 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`X::string_type`	`basic_string<charT>`	
`X::string_view_type`	`basic_string_view<charT>`	
`X::locale_type`	A copy constructible type	A type that represents the locale used by the traits class.
`X::char_class_type`	A bitmask type (15.4.2.1.4).	A bitmask type representing a particular character classification.
`X::length(p)`	`size_t`	Yields the smallest `i` such that `p[i] == 0`. Complexity is linear in `i` .
`v.translate(c)`	`X::char_type`	Returns a character such that for any character `d` that is to be considered equivalent to `c` then `v.translate(c) == v.translate(d)`.
`v.translate_nocase(c)`	`X::char_type`	For all characters `C` that are to be considered equivalent to `c` when comparisons are to be performed without regard to case, then `v.translate_nocase(c) == v.translate_nocase(C)`.
`v.transform(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1, F2)` such that if the character sequence `[G1, G2)` sorts before the character sequence `[H1, H2)` then `v.transform(G1, G2) < v.transform(H1, H2)`.
`v.transform_primary(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1, F2)` such that if the character sequence `[G1, G2)` sorts before the character sequence `[H1, H2)` when character case is not considered then `v.transform_primary(G1, G2) < v.transform_primary(H1, H2)`.
`v.lookup_collatename(F1, F2)`	`X::string_type`	Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range `[F1, F2)`. Returns an empty string if the character sequence is not a valid collating element.
`v.lookup_classname(F1, F2, b)`	`X::char_class_type`	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.
`v.isctype(c, cl)`	`bool`	Returns `true` if character `c` is a member of one of the character classes designated by `cl`, `false` otherwise.

Table 123 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`v.value(c, I)`	`int`	Returns the value represented by the digit *c* in base *I* if the character *c* is a valid digit in base *I*; otherwise returns `-1`. [*Note*: The value of *I* will only be 8, 10, or 16. — *end note*]
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.

29.4 Header <regex> synopsis [re.syn]

```
// 29.9, 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 ssub_match  = sub_match<string::const_iterator>;
using wssub_match = sub_match<wstring::const_iterator>;

using svsub_match  = sub_match<string_view::const_iterator>;
using wsvsub_match = sub_match<wstring_view::const_iterator>;

// 29.9.2, 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 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);

// 29.10, 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>;

using svmatch  = match_results<string_view::const_iterator>;
using wvsmatch = match_results<wstring_view::const_iterator>;

// 29.11.2, function template regex_match
...
template<class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(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 Allocator, class charT, class traits>
  bool regex_match(basic_string_view<charT, ST> s,
                   match_results<typename basic_string_view<charT, ST>::const_iterator,
                                 Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
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);
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);

template<class ST, class charT, class traits>
  bool regex_match(basic_string_view<charT, ST> sv,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);

// 29.11.3, function template regex_search
...
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 charT, class traits>
  bool regex_search(basic_string_view<charT, ST> sv,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template<class ST, class Allocator, class charT, class traits>
  bool regex_search(basic_string_view<charT, ST> sv,
                    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);

// 29.11.4, function template regex_replace
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, class ST>
  OutputIterator
    regex_replace(OutputIterator out,
                  BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  basic_string_view<charT, ST> fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```

…

```
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, class FST>
  basic_string<charT, ST, SA>
```

```
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  basic_string_view<charT, FST> 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);
template<class traits, class charT, class ST, class FST, FSA>
  basic_string<charT, ST>
    regex_replace(basic_string_view<charT, ST> 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 FST>
  basic_string<charT, ST>
    regex_replace(basic_string_view<charT, ST> s,
                  const basic_regex<charT, traits>& e,
                  basic_string_view<charT, FST> fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template<class traits, class charT, class ST>
  basic_string<charT, ST>
    regex_replace(basic_string_view<charT, ST> s,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
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, class FST>
  basic_string<charT>
    regex_replace(const charT* s,
                  const basic_regex<charT, traits>& e,
                  basic_string_view<charT, FST> fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
// 29.12.1, class template regex_iterator
template<class BidirectionalIterator,
          class charT = typename iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT>>
  class regex_iterator;

using cregex_iterator  = regex_iterator<const char*>;
using wcregex_iterator = regex_iterator<const wchar_t*>;
using sregex_iterator  = regex_iterator<string::const_iterator>;
using wsregex_iterator = regex_iterator<wstring::const_iterator>;
using svregex_iterator  = regex_iterator<string_view::const_iterator>;
using wsvregex_iterator = regex_iterator<wstring_view::const_iterator>;
// 29.12.2, class template regex_token_iterator
template<class BidirectionalIterator,
          class charT = typename iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT>>
  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>;

  using svregex_token_iterator  = regex_token_iterator<string_view::const_iterator>;
  using wsvregex_token_iterator = regex_token_iterator<wstring_view::const_iterator>;

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

    using svmatch  = match_results<string_view::const_iterator>;
    using wsvmatch = match_results<wstring_view::const_iterator>;
  }
```

29.7 Class template regex_traits [re.traits]

```
namespace std {
  template<class charT>
    struct regex_traits {
      using char_type       = charT;
      using string_type     = basic_string<char_type>;

      using string_view_type = basic_string_view<char_type>;
```

29.8 Class template basic_regex [re.regex]

```
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 string_view_type = typename traits::string_view_type;

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

      template<class ST>
        explicit basic_regex(basic_string_view<charT, ST> p,
                             flag_type f = regex_constants::ECMAScript);

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

      template<class ST>
        basic_regex& operator=(basic_string_view<charT, ST> p);

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

      template<class ST>
        basic_regex& assign(basic_string_view<charT, ST> sv,
                            flag_type f = regex_constants::ECMAScript);
```

29.8.1 Constructors [re.regex.construct]

...

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

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

15 *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`.

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

```
template<class ST>
  explicit basic_regex(basic_string_view<charT, ST> sv,
                       flag_type f = regex_constants::ECMAScript);
```

17 *Throws:* `regex_error` if `sv` is not a valid regular expression.

18 *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_view sv`, and interpreted according to the flags specified in `f`.

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

29.8.2 Assignment [re.regex.assign]

...

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

8 *Effects:* Returns `assign(p)`.

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

9 *Effects:* Returns `assign(p)`.

...

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

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

14 *Returns:* `*this`.

15 *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.

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

```
template<class ST>
  basic_regex& assign(basic_string_view<charT, ST> sv,
                      flag_type f = regex_constants::ECMAScript);
```

17 *Throws:* `regex_error` if `sv` is not a valid regular expression.

18 *Returns:* `*this`.

19 *Effects:* Assigns the regular expression contained in the `string_view sv`, interpreted according the flags specified in `f`. If an exception is thrown, `*this` is unchanged.

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

29.9 Class template sub_match [re.submatch]

...

```
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>;
      using string_view_type = basic_string_view<value_type>;

      bool matched;

      constexpr sub_match();

      difference_type length() const;
      operator string_type() const;
      string_type str() const;
      operator string_view_type() const;
      string_view_type view() const;

      int compare(const sub_match& s) const;
      int compare(const string_type& s) const;
      int compare(string_view_type sv) const;
      int compare(const value_type* s) const;
    };
}
```

29.9.1 Members [re.submatch.members]

...

`operator string_type() const;`

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

`string_type str() const;`

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

`operator string_view_type() const;`

5 *Returns:* `matched ? string_view_type(addressof(*first), distance(first, second)) : string_view_type()`.

`string_view_type view() const;`

6 *Returns:* `matched ? string_view_type(addressof(*first), distance(first, second)) : string_view_type()`.

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

7 *Returns:* ~~str~~view().compare(s.~~str~~view()).

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

8 *Returns:* ~~str~~view().compare(s).

`int compare(string_view_type sv) const;`

9 *Returns:* `view().compare(sv)`.

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

10 *Returns:* ~~str~~view().compare(s).

29.9.2 Non-member operators [re.submatch.op]

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

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

...

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

42 *Returns:* !(lhs < rhs).

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

43 *Returns:* os « m.str().

44 Class template sub_match provides overloaded relational operators (7.6.9 [expr.rel]) and equality operators (7.6.10 [expr.eq]) for comparisons with another sub_match, with a string, or with a single character. The expressions shown in Table 128 are valid when one of the operands is a type S, that is a specialization of sub_match, and the other expression is one of:

(44.1) — a value x of a type S, in which case STR(x) is x.str();

(44.2) — a value x of type basic_string<S::value_type, T, A> for any types T and A, in which case STR(x) is basic_string_view<S::value_type>(x.data(), x.length());

(44.3) — a value x of type basic_string_view<S::value_type, T> for any type T, in which case STR(x) is basic_string_view<S::value_type>(x.data(), x.length());

(44.4) — a value x of a type convertible to const S::value_type*, in which case STR(x) is basic_string_view<S::value_type>(x);

(44.5) — a value x of type convertible to S::value_type, in which case STR(x) is basic_string_view<S::value_type>(&x, 1).

Table 128 — sub_match comparisons

Expression	Return type	Pre/post-condition
s == t	bool	STR(s).compare(STR(t)) == 0
s != t	bool	STR(s).compare(STR(t)) != 0
s < t	bool	STR(s).compare(STR(t)) < 0
s > t	bool	STR(s).compare(STR(t)) > 0
s <= t	bool	STR(s).compare(STR(t)) <= 0
s >= t	bool	STR(s).compare(STR(t)) >= 0

29.10 Class template match_results [re.results]

```
namespace std {
  template<class BidirectionalIterator,
           class Allocator = allocator<sub_match<BidirectionalIterator>>>
    class match_results {
    public:
      using value_type       = sub_match<BidirectionalIterator>;
      ...
      using string_type      = basic_string<char_type>;
      using string_view_type = basic_string_view<char_type>;
      ...

      // 29.10.4, 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;
    string_view_type view(size_type sub = 0) const;
    const_reference operator[](size_type n) const;
    ...
    // 29.10.5, format
    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;
    template<class OutputIter, class ST, class SA>
      OutputIter
        format(OutputIter out,
               const basic_string<char_type, ST, SA>& fmt,
               regex_constants::match_flag_type flags = regex_constants::format_default) const;
    template<class OutputIter, class ST>
      OutputIter
        format(OutputIter out,
               basic_string_view<char_type, ST> fmt,
               regex_constants::match_flag_type flags = regex_constants::format_default) const;
    template<class ST, class SA>
      basic_string<char_type, ST, SA>
        format(const basic_string<char_type, ST, SA>& fmt,
               regex_constants::match_flag_type flags = regex_constants::format_default) const;
    template<class ST>
      basic_string<char_type, ST>
        format(basic_string_view<char_type, ST> fmt,
               regex_constants::match_flag_type flags = regex_constants::format_default) const;
    string_type
      format(const char_type* fmt,
             regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

29.10.4 Element access [re.results.acc]

...

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

5 *Requires:* `ready() == true`.

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

```
string_view_type view(size_type sub = 0) const;
```

7 *Requires:* `ready() == true`.

8 *Returns:* `string_view_type((*this)[sub])`.

29.10.5 Formatting [re.results.form]

...

```
template<class OutputIter, class ST, class SA>
  OutputIter format(
      OutputIter out,
      const basic_string<char_type, ST, SA>& fmt,
      regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

4 *Effects:* Equivalent to:

```
return format(out, fmt.data(), fmt.data() + fmt.size(), flags);
```

```
template<class OutputIter, class ST>
  OutputIter format(
      OutputIter out,
      basic_string_view<char_type, ST> fmt,
      regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

5 *Effects:* Equivalent to:

```
return format(out, fmt.data(), fmt.data() + fmt.size(), flags);
```

```
template<class ST, class SA>
  basic_string<char_type, ST, SA> format(
      const basic_string<char_type, ST, SA>& fmt,
      regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

6 *Requires:* `ready() == true`.

7 *Effects:* Constructs an empty string `result` of type `basic_string<char_type, ST, SA>` and calls:

```
format(back_inserter(result), fmt, flags);
```

8 *Returns:* `result`.

```
template<class ST>
  basic_string<char_type, ST> format(
      basic_string_view<char_type, ST> fmt,
      regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

9 *Requires:* `ready() == true`.

10 *Effects:* Constructs an empty string `result` of type `basic_string<char_type, ST>` and calls:

```
format(back_inserter(result), fmt, flags);
```

11 *Returns:* `result`.

29.11 Regular expression algorithms [re.alg]

29.11.2 `regex_match` [re.alg.match]

...

```
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);
```

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

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

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

...

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

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

```
template<class ST, class charT, class traits>
  bool regex_match(basic_string_view<charT, ST> sv,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
```

9 *Returns:* `regex_match(sv.begin(), sv.end(), e, flags)`.

29.11.3 `regex_search` [re.alg.search]

...

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

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

```
template<class ST, class Allocator, class charT, class traits>
  bool regex_search(basic_string_view<charT, ST> sv,
                    match_results<typename basic_string_view<charT, ST>::const_iterator,
                                  Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
```

6 *Returns:* `regex_search(sv.begin(), sv.end(), m, 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);
```

8 *Returns:* `regex_search(s.begin(), s.end(), e, flags)`.

```
template<class ST, class charT, class traits>
  bool regex_search(basic_string_view<charT, ST> sv,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
```

9 *Returns:* `regex_search(sv.begin(), sv.end(), e, flags)`.

29.11.4 `regex_replace` [re.alg.replace]

```
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, class ST>
  OutputIterator
    regex_replace(OutputIterator out,
                  BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  basic_string_view<charT, ST> 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);
```

1 *Effects:* Constructs a `regex_iterator` object `i` as if by

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

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

```
out = copy(first, last, out)
```

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

(1.1) — If `!(flags & regex_constants::format_no_copy)`, calls

```
out = copy(m.prefix().first, m.prefix().second, out)
```

(1.2) — Then calls

```
out = m.format(out, fmt, flags)
```

for the first and second form of the function and

```
out = m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)
```

for the ~~second~~third.

Finally, if such a match is found and `!(flags & regex_constants::format_no_copy)`, calls

```
out = copy(last_m.suffix().first, last_m.suffix().second, out)
```

where `last_m` is a copy of the last match found. If `flags & regex_constants::format_first_only` is nonzero, then only the first match found is replaced.

2 *Returns:* `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, class FST>
  basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  basic_string_view<charT, FST> 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);

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

3 *Effects:* Constructs an empty string `result` of type `basic_string<charT, ST, SA>` or basic_string<charT, ST> and calls:

```
regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags);
```

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

5 *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);
```

6 *Returns:* `result`.