p0506r2 - use string_view for library function parameters instead of const string &/const char *

Peter Sommerlad

2017-10-06

Document Number:	p0506r2
Date:	2017-10-06
Project:	Programming Language C++
Audience:	LWG/LEWG

1 Motivation

With `basic_string_view` there is no longer a reason to keep library APIs that have overloads taking `std::string const &` and `char const *` parameter types. Both should be replaced by a single version taking a std::string_view.

2 Introduction

A draft of this paper was discussed in LEWG in Issaquah but it was considered to rushed to push this into C++17. However it addresses issue CH 9:

"The standard library should provide string_view parameters instead or in addition for functions defined with `char const *` or `string const &` as parameter types. Most notably in cases where both such overloads exist or where an internal copy is expected anyway. It might be doubted that the non-null termination of `string_view` could be an issue with functions that pass the `char *` down to OS functions, such as `fstream_buf::open()` etc and those should not provide it and favour generating a `std::string` temporary instead in that case. However, `std::path` demonstrates it is usable to have `string_view` overloads and there might be many places where it can be handy, or even better."

Proposed change: "Provide the overloads for `std::regex`, the exception classes, `std::bitset`, `std::locale` and more."

by providing changes for library sections 22, 23.9, 24.3.4, 25 and 31. The "and more" part is in 30.7.1/30.7.6 with respect to the quoted output manipulator and the application of string view to string streams is given in p0408.

3 Acknowledgements

— LEWG in Issaquah for proposing me to write this paper, even when it can not make it into C++17.

— Alisdair Meredith for extensive review and suggestions by email in July 2017.

— Arthur O'Dwyer for suggesting numeric conversions to use `string_view` as well.

4 Changes from previous versions

4.1 p0506r1

— include section 24.3.4 numeric conversions to employ `string_view` instead of `const string&` as suggested by Arthur O. Dwyer.

— Among some things Alisdair pointed out several places where deduction is incomplete, or an `Allocator` template parameter was sticking around. After thinking about that, I figured that `bitset` requires to keep a `char const *` constructor overload and just should replace the `const string &` overload with a `string_view` overload

— Alisdair further pointed out, that quoted will not work with a basic_string_view overload only. Also the standard text changed in between the version when this paper was started wrt LWG2785 to add an additional quoted overload for `basic_string_view`. I tried to implement it and figured out, that template argument deduction won't work nicely with implicit conversions, without all 3 template versions. So I suggest to sidestep from templatizing `quoted` for output and provide deliberate overloads for the relevant stream character types (`char`, `wchar_t`) in the standard. With such an implementation one `string_view` overload is sufficient to cover arguments of all kinds of string representations (character pointers, string, string view). This results in only two overloads instead of three templated overloads for output. We won't have streams for other character types than `char` and `wchar_t` in the foreseeable future, I believe. Unfortuntely C++17 shipped with the three template overloads for quoted output (sadly).

— As Alisdair suggested I removed reference to construction from nullptr from the locale section.

— 22.5.7 system error and filesystem error needs to be adapted as well to reduce the number of ctor overloads (didn't have time to specify it yet).

4.2 p0506r0

— removed unnecessary Allocator template parameter

— change layout in regex adaptation to see changes easier piecewise judgement

— adjust latex to most current std.tex macros

— adjust to new standard chapter numbering

— make `regex_search` allocator aware again by taking the allocator from a string parameter.

5 Impact on the Standard

Using string view as parameter type instead of overloading for char pointers and string references has the potential to significantly shorten the specification. Most notable this happens in the regex library, where we get from six overloads down to two, for example, for `regex_replace`.

In my opinion such a change for a new C++ standard is an important simplification and sidesteps current "pessimizations" due to extra string objects created when passed as arguments.

A separate paper p0408 specifies the application of `basic_string_view` for `basic_stringbuf` and string streams to optimize access to their respective internal buffer.

The following is relative to the CD/current working draft.

5.1 22.2 Exception classes [std.exceptions]

For all subclasses of std::exception, std::logic_error, and std::runtime_error specified in section 22.2 [stdexcept.syn] apply the following changes respectively by replacing *`std_exception`* with the corresponding class and *`base_exception`* by its respective base class:

```
namespace std {
  class std_exception  : public base_exception  {
  public:
    explicit std_exception(string_view what_arg);
    explicit std_exception(const string & what_arg);
    explicit std_exception(const char* what_arg);
  };
}
```

1 The class *`std_exception`* ... *no change*.

```
std_exception(const string& what_arg);
std_exception(string_view what_arg);
```

2 *Effects:* Constructs an object of class *`std_exception`*.

3 *Postconditions:* ~~`strcmp(what(), what_arg.c_str()) == 0.`~~
`what_arg.compare(what()) == 0.`

```
std_exception(const char* what_arg);
```

4 *Effects:* Constructs an object of class `std_exception`.

5 *Postconditions:* `strcmp(what(), what_arg) == 0.`

5.2 22.5.7 Class `system_error` [syserr.syserr]

Change the class definition in 22.5.7.1 [syserr.syserr.overview] as follows:

```
namespace std {
  class system_error : public runtime_error {
  public:
    system_error(error_code ec, const string_view& what_arg);
    system_error(error_code ec, const char* what_arg);
    system_error(error_code ec);
    system_error(int ev, const error_category& ecat, const string_view& what_arg);
```

```
    system_error(int ev, const error_category& ecat, const char* what_arg);
    system_error(int ev, const error_category& ecat);
    const error_code& code() const noexcept;
    const char* what() const noexcept override;
  };
}
```

Change the constructor definitions in 22.5.7.2 [syserr.syserr.members] Class `system_error` members accordingly:

`system_error(error_code ec, const string_view& what_arg);`

Effects: Constructs an object of class `system_error`.

Postconditions: `code() == ec` and `string(what()).find(what_arg) != string::npos`.

`system_error(error_code ec, const char* what_arg);`

Effects: Constructs an object of class `system_error`.

Postconditions: `code() == ec` and `string(what()).find(what_arg) != string::npos`.

`system_error(error_code ec);`

Effects: Constructs an object of class `system_error`.

Postconditions: `code() == ec`.

`system_error(int ev, const error_category& ecat, const string_view& what_arg);`

Effects: Constructs an object of class `system_error`.

Postconditions: `code() == error_code(ev, ecat)` and `string(what()).find(what_arg) != string::npos`.

`system_error(int ev, const error_category& ecat, const char* what_arg);`

Effects: Constructs an object of class `system_error`.

Postconditions: `code() == error_code(ev, ecat)` and `string(what()).find(what_arg) != string::npos`.

`system_error(int ev, const error_category& ecat);`

Effects: Constructs an object of class `system_error`.

Postconditions: `code() == error_code(ev, ecat)`.

5.3 30.10.8 Class filesystem_error [fs.class.filesystem_error]

Change the const string reference parameters of `filesystem_error` constructors to `string_view` to sidestep extra string temporaries. LEWG should discuss if this is beneficial enough to warrant an ABI breaking change.

Change the definition of class filesystem_error as follows:

```
namespace std::filesystem {
  class filesystem_error : public system_error {
  public:
```

```
    filesystem_error(const string_view& what_arg, error_code ec);
    filesystem_error(const string_view& what_arg,
                     const path& p1, error_code ec);
    filesystem_error(const string_view& what_arg,
                     const path& p1, const path& p2, error_code ec);

    const path& path1() const noexcept;
    const path& path2() const noexcept;
    const char* what() const noexcept override;
  };
}
```

1 The class `filesystem_error` defines the type of objects thrown as exceptions to report file system errors from functions described in this subclause.

5.3.1 30.10.8.1 [fs.filesystem_error.members]`filesystem_error` members

In section 30.10.8.1 [fs.filesystem_error.members]`filesystem_error` members change the constructors accordingly:

1 Constructors are provided that store zero, one, or two paths associated with an error.

`filesystem_error(const string_view& what_arg, error_code ec);`

2 *Postconditions:* The postconditions of this function are indicated in Table 1.

Table 1 — `filesystem_error(const string_view& , error_code)` effects

Expression	Value
`what_arg.compare(runtime_error::what())`	`what_arg.c_str()0`
`code()`	`ec`
`path1().empty()`	`true`
`path2().empty()`	`true`

`filesystem_error(const string_view& what_arg, const path& p1, error_code ec);`

3 *Postconditions:* The postconditions of this function are indicated in Table 2.

Table 2 — `filesystem_error(const string_view& , const path&, error_code)` effects

Expression	Value
`what_arg.compare(runtime_error::what())`	`what_arg.c_str()0`
`code()`	`ec`
`path1()`	Reference to stored copy of `p1`
`path2().empty()`	`true`

`filesystem_error(const string_view& what_arg, const path& p1, const path& p2, error_code ec);`

4 *Postconditions:* The postconditions of this function are indicated in Table 3.

Table 3 — `filesystem_error(~~const~~ string_view~~&~~, const path&, const path&, error_code)` effects

Expression	Value
what_arg.compare(runtime_error::what())	~~what_arg.c_str()~~0
code()	ec
path1()	Reference to stored copy of p1
path2()	Reference to stored copy of p2

5.4 23.9 Class template `bitset` [template.bitset]

Note to the reviewers: It should be considered if this new constructor could be made constexpr.

In p1 replace the constructors taking `string const &` and `char const *` by one taking `string_view` by applying the following changes:

```
namespace std {
  template<size_t N> class bitset {
  public:
//...
    // ?? constructors:
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;

    template<class charT, class traits>
      explicit bitset(
        basic_string_view<charT, traits> sv,
        typename basic_string_view<charT, traits>::size_type pos = 0,
        typename basic_string_view<charT, traits>::size_type n =
          basic_string_view<charT, traits>::npos,
          charT zero = charT('0'), charT one = charT('1'));
    template<class charT, class traits, class Allocator>
      explicit bitset(
        const basic_string<charT, traits, Allocator>& str,
        typename basic_string<charT, traits, Allocator>::size_type pos = 0,
        typename basic_string<charT, traits, Allocator>::size_type n =
          basic_string<charT, traits, Allocator>::npos,
          charT zero = charT('0'), charT one = charT('1'));
    template <class charT>
      explicit bitset(
        const charT* str,
        typename basic_string_view<charT>::size_type n = basic_string_view<charT>::npos,
        charT zero = charT('0'), charT one = charT('1'));
        //...
        };
```

In 23.9.1 replace p3 to p6 defining the removed constructor by the following:

```
template <class charT, class traits>
```

```
explicit
bitset(basic_string_view<charT, traits> sv,
       typename basic_string_view<charT, traits>::size_type pos = 0,
       typename basic_string_view<charT, traits>::size_type n =
         basic_string_view<charT, traits>::npos,
         charT zero = charT('0'), charT one = charT('1'));
```

1 *Throws:* `out_of_range` if `pos > str.size()` or `invalid_argument` if an invalid character is found (see below).

2 *Effects:* Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`.

The function then throws`invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values.

Otherwise, the function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`.

3 An element of the constructed object has value zero if the corresponding character in `str`, beginning at position `pos`, is `zero`. Otherwise, the element has the value one. Character position `pos + M - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

4 If `M < N`, remaining bit positions are initialized to zero.

In 23.9.1 change p7 defining the changed constructor as follows:

```
template <class charT>
  explicit bitset(
    const charT* str,
    typename basic_string_view<charT>::size_type n = basic_string_view<charT>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

5 *Effects:* Constructs an object of class `bitset<N>` as if by:

```
bitset(n == basic_string_view<charT>::npos
          ? basic_string_view<charT>(str)
          : basic_string_view<charT>(str, n),
       0, n, zero, one)
```

5.5 24.3.4 Numeric conversions [string.confersions]

Change the section as follows, by replacing all `const string &` with `string_view` instead and all `const wstring &` with `wstring_view`. Accordingly change the header in 24.3.1 accordingly and LEWG should consider moving the `sto`*x* `wsto`*x* functions to the `<string_view>` header 24.4.1 instead and splitting 24.3.4 into two sections and put the "to" functions to a new section in 24.4. I wait for that guidance before taking everything into account.

```
int stoi(string_view~~const string&~~ str, size_t* idx = nullptr, int base = 10);
long stol(string_view~~const string&~~ str, size_t* idx = nullptr, int base = 10);
```

```
unsigned long stoul(string_view~~const string&~~ str, size_t* idx = nullptr, int base = 10);
long long stoll(string_view~~const string&~~ str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(string_view~~const string&~~ str, size_t* idx = nullptr, int base = 10);
```

1 *Effects:* The functions behave as if the character sequence denoted by `str` is null-terminated while the following calls happen. The first two functions call `strtol(str.data()~~.c_str()~~, ptr, base)`, `strtoul(str.data()~~.c_str()~~, ptr, base)`, `strtoll(str.data()~~.c_str()~~, ptr, base)`, and `strtoull(str.data()~~.c_str()~~, ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

2 *Returns:* The converted result.

3 *Throws:* `invalid_argument` if `strtol`, `strtoul`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if `strtol`, `strtoul`, `strtoll` or `strtoull` sets `errno` to `ERANGE`, or if the converted value is outside the range of representable values for the return type.

```
float stof(string_view~~const string&~~ str, size_t* idx = nullptr);
double stod(string_view~~const string&~~ str, size_t* idx = nullptr);
long double stold(string_view~~const string&~~ str, size_t* idx = nullptr);
```

4 *Effects:* Theses functions behave as if the character sequence denoted by `str` is null-terminated while the following calls happen. These functions call `strtof(str.data()~~.c_str()~~, ptr)`, `strtod(str.data()~~.c_str()~~, ptr)`, and `strtold(str.data()~~.c_str()~~, ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

5 *Returns:* The converted result.

6 *Throws:* `invalid_argument` if `strtof`, `strtod`, or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtof`, `strtod`, or `strtold` sets `errno` to `ERANGE` or if the converted value is outside the range of representable values for the return type.

```
int stoi(wstring_view~~const wstring&~~ str, size_t* idx = nullptr, int base = 10);
long stol(wstring_view~~const wstring&~~ str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(wstring_view~~const wstring&~~ str, size_t* idx = nullptr, int base = 10);
long long stoll(wstring_view~~const wstring&~~ str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(wstring_view~~const wstring&~~ str, size_t* idx = nullptr, int base = 10);
```

7 *Effects:* Theses functions behave as if the character sequence denoted by `str` is null-terminated while the following calls happen. The first two functions call `wcstol(str.data()~~.c_str()~~, ptr, base)`, and the last three functions call `wcstoul(str.data()~~.c_str()~~, ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.data()~~.c_str()~~, ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the

index of the first unconverted element of `str`.

8 *Returns:* The converted result.

9 *Throws:* `invalid_argument` if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

```
float stof(wstring_view~~const wstring&~~ str, size_t* idx = nullptr);
double stod(wstring_view~~const wstring&~~ str, size_t* idx = nullptr);
long double stold(wstring_view~~const wstring&~~ str, size_t* idx = nullptr);
```

10 *Effects:* Theses functions behave as if the character sequence denoted by `str` is null-terminated while the following calls happen. These functions call `wcstof(str.data()~~.c_str()~~, ptr)`, `wcstod(str.data()~~.c_str()~~, ptr)`, and `wcstold(str.data()~~.c_str()~~, ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

11 *Returns:* The converted result.

12 *Throws:* `invalid_argument` if `wcstof`, `wcstod`, or `wcstold` reports that no conversion could be performed. Throws `out_of_range` if `wcstof`, `wcstod`, or `wcstold` sets `errno` to `ERANGE`.

5.6 25.3 Locale

The following things could be adapted:

— locale's ctors

— locale's call operator

— `wstring_convert` (not proposed)

— all `xxx_byname` template class constructors.

Change the synopsis of class `locale` as follows:

```
namespace std {
  class locale {
  public:
    // types:
    class facet;
    class id;
    using category = int;
    static const category    // values assigned here are for exposition only
      none     = 0,
      collate  = 0x010, ctype    = 0x020,
      monetary = 0x040, numeric  = 0x080,
      time     = 0x100, messages = 0x200,
      all = collate | ctype | monetary | numeric | time  | messages;

    // construct/copy/destroy:
    locale() noexcept;
```

```
    locale(const locale& other) noexcept;
    explicit locale(const char* std_name);
    explicit locale(const string_view& std_name);
    locale(const locale& other, const char* std_name, category);
    locale(const locale& other, const string_view& std_name, category);
    template <class Facet> locale(const locale& other, Facet* f);
    locale(const locale& other, const locale& one, category);
    ~locale();                  // not virtual
    const locale& operator=(const locale& other) noexcept;
    template <class Facet> locale combine(const locale& other) const;

    // locale operations:
    basic_string<char>                  name() const;

    bool operator==(const locale& other) const;
    bool operator!=(const locale& other) const;

// note: this change might require 3 overloads for charT pointers, basic_string and basic_string_view
// because template argument deduction will fail. is that used at all somewhere?
    template <class charT, class traits, class Allocator>
      bool operator()(const basic_string_view<charT,traits,Allocator>& s1,
                      const basic_string_view<charT,traits,Allocator>& s2) const;

    // global locale objects:
    static       locale  global(const locale&);
    static const locale& classic();
  };
}
```

5.6.1 25.3.1.1.2 Class `locale::facet` [locale.facet]

Change p4 as follows:

1 For some standard facets a standard "`..._byname`" class, derived from it, implements the virtual function semantics equivalent to that facet of the locale constructed by `locale(const char*string_view)` with the same name. Each such facet provides a constructor that takes a `const char*string_view` argument, which names the locale, and a `refs` argument, which is passed to the base class constructor. Each such facet also provides a constructor that takes a `string` argument `str` and a `refs` argument, which has the same effect as calling the first constructor with the two arguments `str.c_str()` and `refs`. If there is no "`..._byname`" version of a facet, the base class implements named locale semantics itself by reference to other facets.

5.6.2 23.3.1.2 `locale` constructors and destructor [locale.cons]

Change p4 to p11 as follows. Note: since locale must store the passed string_view string value internally (as a string), it is not an issue, if it is not a NTBS.:

```
explicit locale(const char*string_view std_name);
```

1 *Effects:* Constructs a locale using standard C locale names, e.g., `"POSIX"`. The resulting locale implements semantics defined to be associated with that name.

2 *Throws:* `runtime_error` if the argument is not valid~~, or is null~~.

3 *Remarks:* The set of valid string argument values is `"C"`sv, `""`sv, and any implementation-defined values.

```
explicit locale(const string& std_name);
```

4 *Effects:* The same as `locale(std_name.c_str())`.

```
locale(const locale& other, ~~const char*~~string_view std_name, category);
```

5 *Effects:* Constructs a locale as a copy of `other` except for the facets identified by the `category` argument, which instead implement the same semantics as `locale(std_name)`.

6 *Throws:* `runtime_error` if the argument is not valid~~, or is null~~.

7 *Remarks:* The locale has a name if and only if `other` has a name.

```
locale(const locale& other, const string& std_name, category cat);
```

8 *Effects:* The same as `locale(other, std_name.c_str(), cat)`.

5.6.3 25.3.1.3 `locale` members [locale.members]

Change p5 as follows:

```
basic_string<char> name() const;
```

1 *Returns:* The name of `*this`, if it has one; otherwise, the string `"*"`. If `*this` has a name, then `locale(name()~~.c_str()~~)` is equivalent to `*this`. Details of the contents of the resulting string are otherwise implementation-defined.

5.6.4 25.3.1.4 `locale` operators [locale.operators]

Change the definition of `operator()` p3-p5 as follows:

```
template <class charT, class traitsdel, class Allocator>
  bool operator()(~~const~~ basic_string_view<charT,traits~~,Allocator~~>~~&~~ s1,
                  ~~const~~ basic_string_view<charT,traits~~,Allocator~~>~~&~~ s2) const;
```

1 *Effects:* Compares two strings according to the `collate<charT>` facet.

2 *Remarks:* This member operator template (and therefore `locale` itself) satisfies requirements for a comparator predicate template argument (Clause **??**) applied to strings.

3 *Returns:* The result of the following expression:

```
use_facet< collate<charT> >(*this).compare
  (s1.data(), s1.data()+s1.size(), s2.data(), s2.data()+s2.size()) < 0;
```

5.6.5 25.3.3.2.2 string conversions [conversions.string]

While there is potential to remove some of the overloads of `wstring_convert`'s member functions, I refrain from proposing a change, because I feel not be able to judge the potential impact. At least the reduction from 4 to 3 overloads each, seems to be achievable.

5.6.6 "..._byname" class templates

For each of the following class templates referred in the following as `xxx_byname` with its corresponding base class template referred to as `xxx_base`

- `ctype_byname` (25.4.1.2 [locale.ctype.byname])
- `codecvt_byname` (25.4.1.5 [locale.codecvt.byname])
- `numpunct_byname` (25.4.3.2 [locale.numpunct.byname])
- `collate_byname` (25.4.4.2 [locale.collate.byname])
- `time_get_byname` (25.4.5.2 [locale.time.get.byname])
- `time_put_byname` (25.4.5.4 [locale.time.put.byname])
- `moneypunct_byname` (25.4.6.4 [locale.moneypunct.byname])
- `messages_byname` (25.4.7.2 [locale.messages.byname])

replace the overloaded explicit constructors as follows

```
namespace std {
  template <...>
  class xxx_byname : public xxx_base {
  public:
    // other members, if any
    explicit xxx_byname(const char*, size_t refs = 0);
    explicit xxx_byname(const string_view&, size_t refs = 0);
  protected:
   ~xxx_byname();
  };
}
```

5.7 30.7.1 Overview [iostream.format.overview]

Change the header `<iomanip>`'s synopsis as follows:

```
namespace std {
  // types T1, T2, ... are unspecified implementation types
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags  (ios_base::fmtflags mask);
  T3 setbase(int base);
  template<charT> T4 setfill(charT c);
  T5 setprecision(int n);
  T6 setw(int n);
  template <class moneyT> T7 get_money(moneyT& mon, bool intl = false);
  template <class moneyT> T8 put_money(const moneyT& mon, bool intl = false);
  template <class charT> T9 get_time(struct tm* tmb, const charT* fmt);
  template <class charT> T10 put_time(const struct tm* tmb, const charT* fmt);

  template <class charT>
    T11 quoted(const charT* s, charT delim = charT('"'), charT escape = charT('\\'));
  template <class charT, class traits, class Allocator>
    T12 quoted(const basic_string<charT, traits, Allocator>& s,
               charT delim = charT('"'), charT escape = charT('\\'));
```

```
    T11 quoted(string_view s,
               char delim = '"', char escape = '\\');

    T12 quoted(wstring_view s,
               wchar_t delim = L'"', wchar_t escape = L'\\');
  template <class charT, class traits, class Allocator>
    T13 quoted(basic_string<charT, traits, Allocator>& s,
               charT delim = charT('"'), charT escape = charT('\\'));
  template <class charT, class traits>
    T14 quoted(basic_string_view<charT, traits> s,
               charT delim = charT('"'), charT escape = charT('\\'));
}
```

5.7.1 30.7.6 Quoted manipulators[quoted.manip]

Change the specification for the output manipulator as follows:

```
template <class charT>
  unspecified quoted(const charT* s, charT delim = charT('"'), charT escape = charT('\\'));
template <class charT, class traits, class Allocator>
  unspecified quoted(const basic_string<charT, traits, Allocator>& s,
                     charT delim = charT('"'), charT escape = charT('\\'));
template <class charT, class traits>
  unspecified quoted(basic_string_view<charT, traits> s,
                     charT delim = charT('"'), charT escape = charT('\\'));

  unspecified quoted(string_view s,
                     char delim = '"', char escape = '\\');
  unspecified quoted(wstring_view s,
                     wchar_t delim = L'"', wchar_t escape = L'\\');
```

1 *Returns:* An object of unspecified type such that if `out` is an instance of `basic_ostream` with corresponding member types `char_type` ~~the same as `charT`~~ and ~~with member type~~ `traits_type` as the type of parameter `s`~~, which in the second and third form is the same as `traits`~~, then the expression `out << quoted(s, delim, escape)` behaves as a formatted output function (??) of `out`. This forms a character sequence `seq`, initially consisting of the following elements:

(1.1) — `delim`.

(1.2) — Each character in `s`. If the character to be output is equal to `escape` or `delim`, as determined by `traits_type::eq`, first output `escape`.

(1.3) — `delim`.

Let `x` be the number of elements initially in `seq`. Then padding is determined for `seq` as described in ??, `seq` is inserted as if by calling `out.rdbuf()->sputn(seq, n)`, where `n` is the larger of `out.width()` and `x`, and `out.width(0)` is called. The expression `out << quoted(s, delim, escape)` shall have type `basic_ostream<charT, traits>&` and value `out`.

5.8 31 Regular Expression library [re]

Changes here get a bit more involved, so it might not be wise to adopt this for C++17, unless there is implementation experience. However, I try my best to go ahead with it.

My analysis brought up the following areas that could be simplified by using `string_view` instead of 2 or more overloads with `char const *` and `basic_string<C,T,A> const &`:

— 31.4. and 31.9.2 `sub_match` comparison operators (including `char` only, 3 overloads to 1 overload)

— 31.4 and 31.11.2 `regex_match` constructor overloads

— 31.4 and 31.11.3 `regex_search` overloads

— 31.4 and 31.11.4 `regex_replace` overloads

— 31.8 `basic_regex` constructor overloads, `operator=()` and `assign()` overloads

— 31.10 `match_results::format()` overloads

5.8.1 31.4 Header <regex> synopsis [re.syn]

In the namespace std of the header synopsis, apply the following marked changes. Note, unchanged parts are deliberately omitted to keep this paper of a manageable size.

Adjust the declarations of the free function sub_match comparison operators after the comment `// 31.9.2 sub_match non-member operators:` as follows.

For each comparison operator function template taking a `basic_string` by const reference as one of its parameter types, replace it with one taking a `basic_string_view` parameter in the same position instead. Remove all comparison operator function taking a `typename iterator_traits<BiIter>::value_type const*` parameter. The latter will be subsumed by the new string_view overloads. There is no change to the free comparison operator function templates taking a `typename iterator_traits<BiIter>::value_type const &`. This should result in the following changes:

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

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

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

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

In the code section following the comment `// 31.11.2 function template regex_match` apply the following changes:

```
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);
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 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_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 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);
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);

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 charT, class traits>
  bool regex_match(basic_string_view<charT, ST> s,
                   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);
```

In the code section following the comment `// 31.11.3 function template regex_search` apply the following changes:

```
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);
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);

template <class ST, class charT, class traits>
  bool regex_search(basic_string_view<charT, ST> s,
                    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> 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 Allocator, class traits>
  bool regex_search(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);
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);
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);
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);
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;
```

In the code section following the comment `// 31.11.4 function template regex_replace` apply the following changes (Note, that here we might use a feature, specifying the allocator for the returned string to be given by the string's allocator. We might consider only replacing the character pointer versions):

```
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, 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);
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); // optional to allow specifying allocator

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, 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);
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);
```

5.8.2 31.8 Class template `basic_regex` [re.regex]

In the class definition in p3 apply the following changes after the comment `//31.8.2, 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_view<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();
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 basic_string_view<charT, ST, SA>& p);

// 31.8.3, 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);
basic_regex& assign(const charT* p, size_t len, flag_type f);
template <class string_traits, class A>
  basic_regex& assign(const basic_string_view<charT, string_traits, A>& s,
                      flag_type f = regex_constants::ECMAScript);
template <class InputIterator>
  basic_regex& assign(InputIterator first, InputIterator last,
                      flag_type f = regex_constants::ECMAScript);
basic_regex& assign(initializer_list<charT>,
                    flag_type = regex_constants::ECMAScript);
```

5.8.3 31.8.2 `basic_regex` constructors[re.regex.construct]

strike p2 to p5 defining the `charT` pointer constructor overload. Change p14 to p16 as follows:

```
template <class ST, class SA>
  explicit basic_regex(const basic_string_view<charT, ST, SA>& s,flag_type f = regex_constants::ECMAScr
```

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

2 *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 `s`, and interpreted according to the flags specified in `f`.

3 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked subexpressions within the expression.

5.8.4 31.8.3 `basic_regex` assign[re.regex.assign]

Strike p5 and p6 defining operator= for `charT` pointer and replace p8 as follows:

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

1 *Requires:* `ptr` shall not be a null pointer.

2 *Effects:* Returns `assign(ptr)`.

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

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

Strike p13 and modify p15-18 as follows (note we keep the charT pointer, length assign function overload):

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

4 *Returns:* `assign(string_type(ptr), f)`.

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

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

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

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

7 *Returns:* `*this`.

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

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

5.9 31.9 Class template `sub_match` [re.submatch]

Change the member function compare overloads to take a `string_view` instead of a pointer or string const reference. For convenience I suggest adding a type alias for that as follows.

```
namespace std {
  template <class BidirectionalIterator>
  class sub_match : public std::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;

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

5.9.1 31.9.1 sub_match members [re.submatch.members]

Change p6 as follows and strike p7:

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

1 *Returns:* `str().compare(s)`.

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

2 *Returns:* `str().compare(s)`.

5.9.2 31.9.2 sub_match non-member operators [re.submatch.op]

Change the section as follows:

```
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 sub_match<BiIter>& rhs);
```

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

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

3 *Returns:* `lhs.compare(rhs) < 0`.

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

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

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

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

```
template <class BiIter>
```

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

6 *Returns:* `lhs.compare(rhs) > 0`.

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

7 *Returns:* `rhs.compare(typename sub_match<BiIter>::string_view_type(lhs.data(), lhs.size())) == 0`.

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

8 *Returns:* `!(lhs == rhs)`.

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

9 *Returns:* `rhs.compare(typename sub_match<BiIter>::string_view_type(lhs.data(), lhs.size())) > 0`.

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

10 *Returns:* `rhs < lhs`.

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

11 *Returns:* `!(lhs < rhs)`.

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

12 *Returns:* `!(rhs < lhs)`.

```
template <class BiIter, class ST>
```

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

13 *Returns:* `lhs.compare(typename sub_match<BiIter>::string_view_type(rhs.data(), rhs.size())) == 0`.

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

14 *Returns:* `!(lhs == rhs)`.

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

15 *Returns:* `lhs.compare(typename sub_match<BiIter>::string_view_type(rhs.data(), rhs.size())) < 0`.

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

16 *Returns:* `rhs < lhs`.

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

17 *Returns:* `!(lhs < rhs)`.

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

18 *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);
```

19 *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);
```

20 *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);
```

21 *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);
```

22 *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);
```

23 *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);
```

24 *Returns:* `!(rhs < lhs)`.

```
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);
```

25 *Returns:* `lhs.compare(typename sub_match<BiIter>::string_type(rhs.data(), rhs.size())) == 0`.

```
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);
```

26 *Returns:* `!(lhs == 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);
```

27 *Returns:* `lhs.compare(typename sub_match<BiIter>::string_type(rhs.data(), rhs.size())) < 0`.

```
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);
```

28 *Returns:* `rhs < lhs`.

```
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);
```

29 *Returns:* `!(lhs < 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);
```

30 *Returns:* `!(rhs < lhs)`.

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

31 *Returns:* `rhs.compare(lhs) == 0`.

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

32 *Returns:* `!(lhs == rhs)`.

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

33 *Returns:* `rhs.compare(lhs) > 0`.

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

34 *Returns:* `rhs < lhs`.

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

35 *Returns:* `!(lhs < rhs)`.

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

36 *Returns:* `!(rhs < lhs)`.

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

37 *Returns:* `lhs.compare(rhs) == 0`.

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

38 *Returns:* `!(lhs == rhs)`.

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

39 *Returns:* `lhs.compare(rhs) < 0`.

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

40 *Returns:* `rhs < lhs`.

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

41 *Returns:* `!(lhs < rhs)`.

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

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

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

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

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

44 *Returns:* `!(lhs == rhs)`.

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

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

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

46 *Returns:* `rhs < lhs`.

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

47 *Returns:* `!(lhs < rhs)`.

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

48 *Returns:* `!(rhs < lhs)`.

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

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

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

50 *Returns:* `!(lhs == rhs)`.

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

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

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

52 *Returns:* `rhs < lhs`.

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

53 *Returns:* `!(lhs < rhs)`.

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

54 *Returns:* `!(rhs < lhs)`.

```
template <class charT, class ST, class BiIter>
```

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

55 *Returns:* `(os << m.str())`.

5.10 31.10 Class template `match_results` [re.results]

Change the code in the class template after the comment `//31.10.5, format:` as follows. (Note, I propose a slight semantic change to the existing format returning a string. In the original version this would take the allocator type from the passed in fmt parameter instead of the default allocator, which is used with the character pointer version):

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

5.10.1 31.10.5 `match_results` formatting [re.results.form]

I suggest that we lose the option to specify the allocator indirectly by the string type used. Change p4 to p10 as follows:

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

1 *Effects:* Equivalent to:

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

```
template <class ST~~, class SA~~>
  ~~basic_string<char_type, ST, SA>~~string_type
  format(~~const~~ basic_string_view<char_type, ST~~, SA~~>& fmt,
         regex_constants::match_flag_type flags =
```

```
      regex_constants::format_default) const;
```

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

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

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

4 *Returns:* `result`.

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

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

6 *Effects:* Constructs an empty string `result` of type `string_type` and calls:

```
format(back_inserter(result),
       fmt, fmt + char_traits<char_type>::length(fmt), flags);
```

7 *Returns:* `result`.

5.11 31.11 Regular expression algorithms [re.alg]

Adjust the changed function apis from the synopsis accordingly.

5.11.1 31.11.2 `regex_match`[re.alg.match]

Change p5 to p8 as follows:

```
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);
```

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

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

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);
```

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

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

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

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

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

5.11.2 31.11.3 `regex_search`[re.alg.search]

Change p4 to p5 and p7 to p8 as follows:

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

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

```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<
                      typename basic_string<charT, ST, SA>::const_iterator,
                      Allocator>& m,
                    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> 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);
```

2 *Returns:* The result of `regex_search(s.begin(), s.end(), m, e, flags)`.

```
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);
```

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

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

4 *Returns:* `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);

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

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

5.11.3 31.11.4 `regex_replace`[re.alg.replace]

Change the section as follows:

```
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_view<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);
```

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

(1.2) — Then calls

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

~~for the first form of the function and~~

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

~~for the second.~~

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

```
out = std::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 non-zero, 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_view<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);
```

3 *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);
```

4 *Returns:* `result`.

```
template <class traits, class charT, class ST, class SAFST>
  basic_string<charT>
  regex_replace(const basic_string_view<charT,ST>* s,
                const basic_regex<charT, traits>& e,
                const basic_string_view<charT, FSTST, 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);
```

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

```
regex_replace(back_inserter(result),
              s, s + char_traits<charT>::length(s), e, fmt, flags);
```

6 *Returns:* `result`.