Document Number: P1148R0
Date: 2018-10-07
Audience: Library Working Group

Reply to: Tim Song
rs2740@gmail.com

Cleaning up Clause 20

Contents

1 Introduction [intro]

1.1 Summary [intro.summary]

1 This paper proposes a comprehensive cleanup of the specification of `char_traits`, `basic_string` and `basic_string_view`. Among other things, it resolves the following LWG issues:

(1.1) — 2151: by adding a *Expects:* to 20.3.2.6.8;

(1.2) — 2318: by rewriting the affected wording;

(1.3) — 2836: by respecifying to rely on `basic_string_view` throughout ("*Throws:* Nothing." was not added; it can be added to `basic_string_view`'s constructors in 20.4.2.1 if desired);

(1.4) — 2841: by updating the wording to use "equivalent to" whenever possible;

(1.5) — 2929: by avoiding default-constructing allocators for temporaries used in these and other functions;

(1.6) — 2994 (partially): by making `traits::char_type`/`charT` mismatch ill-formed;

(1.7) — 3111: by using the "`[s, s + n)` is a valid range" formulation throughout.

2 Some drive-by fixes are proposed for the introductory portions of Clause 27, primarily to define two *typedef-names* (`u16streampos` and `u32streampos`) used in `char_traits` specializations but not defined anywhere. While the specification of iostreams could likely use some improvement, doing so is beyond the scope of this paper.

1.2 Revision History [intro.history]

1.2.1 Revision 0 [intro.history.r0]

Initial version.

1.3 Style of presentation [intro.style]

1 The remainder of this document is a technical specification in the form of editorial instructions directing that changes be made to the text of the C++ working draft N4762. The formatting of the text suggests the origin of each portion of the wording.

Existing wording from the C++ working draft - included to provide context - is presented without decoration.

Entire blocks of wording to be added are presented in a distinct cyan color.

In-line additions of wording to the C++ working draft are presented in cyan with underline.

~~In-line bits of wording to be removed from the C++ working draft are presented in red with strike-through.~~

Entire blocks of wording to be removed form the C++ working draft are presented in red.

Drafting notes are presented in blue, like [*Drafting note*: this – *end drafting note*] .

20 Strings library [strings]

20.1 General [strings.general]

1 This Clause describes components for manipulating sequences of any non-array trivial standard-layout (6.7) type. Such types are called *char-like types*, and objects of char-like types are called *char-like objects* or simply *characters*.

2 The following subclauses describe a character traits class, string classes, and null-terminated sequence utilities, as summarized in Table 56.

Table 56 — Strings library summary

Subclause		Header(s)
20.2	Character traits	`<string>`
20.3	String classes	`<string>`
20.4	String view classes	`<string_view>`

20.5	Null-terminated sequence utilities	`<cctype>`
		`<cwctype>`
		`<cstring>`
		`<cwchar>`
		`<cstdlib>`
		`<cuchar>`

20.2 Character traits [char.traits]

1 This subclause defines requirements on classes representing *character traits*, and defines a class template `char_traits<charT>`, along with four specializations, `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`, that satisfy those requirements.

2 Most classes specified in 20.3, 20.4, and Clause 27 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member *typedef-name*s and functions in the template parameter `traits` used by each such template. This subclause defines the semantics of these members.

3 To specialize those templates to generate a string, string view, or iostream class to handle a particular character container type (15.3.3) ~~`CharT`~~`C`, that and its related character traits class ~~`Traits`~~`X` are passed as a pair of parameters to the string, string view, or iostream template as parameters `charT` and `traits`. If ~~`Traits`~~`X::char_type` is not~~shall be~~ the same type as ~~`CharT`~~`C`, the program is ill-formed.

4 This subclause specifies a class template, `char_traits<charT>`, and four explicit specializations of it, `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`, all of which appear in the header `<string>` and satisfy the requirements below.

20.2.1 Character traits requirements [char.traits.require]

1 In Table 57, `X` denotes a ~~T~~traits class defining types and functions for the character container type ~~`CharT`~~`C`; `c` and `d` denote values of type ~~`CharT`~~`C`; `p` and `q` denote values of type `const` ~~`CharT`~~`C*`; `s` denotes a value of type ~~`CharT`~~`C*`; `n`, `i` and `j` denote values of type `size_t`; `e` and `f` denote values of type `X::int_type`; `pos` denotes a value of type `X::pos_type`; ~~`state` denotes a value of type `X::state_type`;~~ and `r` denotes an lvalue of type ~~`CharT`~~`C`. Operations on ~~`Traits`~~`X` shall not throw exceptions.

Table 57 — Character traits requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::char_type`	~~`charT`~~`C`	~~(described in 20.2.2)~~	compile-time
`X::int_type`		(described in 20.2.2)	compile-time

Table 57 — Character traits requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::off_type`		(described in ~~20.2.2~~27.2.2 and 27.3)	compile-time
`X::pos_type`		(described in ~~20.2.2~~27.2.2 and 27.3)	compile-time
`X::state_type`		(described in 20.2.2)	compile-time
`X::eq(c,d)`	`bool`	*Returns:* whether `c` is to be treated as equal to `d`.	constant
`X::lt(c,d)`	`bool`	*Returns:* whether `c` is to be treated as less than `d`.	constant
`X::compare(p,q,n)`	`int`	*Returns:* `0` if for each `i` in `[0,n)`, `X::eq(p[i],q[i])` is `true`; else, a negative value if, for some `j` in `[0,n)`, `X::lt(p[j],q[j])` is `true` and for each `i` in `[0,j)` `X::eq(p[i],q[i])` is `true`; else a positive value.	linear
`X::length(p)`	`size_t`	*Returns:* the smallest `i` such that `X::eq(p[i],charT())` is `true`.	linear
`X::find(p,n,c)`	`const X::char_type*`	*Returns:* the smallest `q` in `[p,p+n)` such that `X::eq(*q,c)` is `true`, zero otherwise.	linear
`X::move(s,p,n)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`. Copies correctly even where the ranges `[p,p+n)` and `[s,s+n)` overlap. *Returns:* `s`.	linear
`X::copy(s,p,n)`	`X::char_type*`	~~*Requires:*~~ *Expects:* `p` not in `[s,s+n)`. *Returns:* `s`. for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`.	linear
`X::assign(r,d)`	(not used)	assigns `r=d`.	constant
`X::assign(s,n,c)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],c)`. *Returns:* `s`.	linear
`X::not_eof(e)`	`int_type`	*Returns:* `e` if `X::eq_int_type(e,X::eof())` is `false`, otherwise a value `f` such that `X::eq_int_type(f,X::eof())` is `false`.	constant
`X::to_char_type(e)`	`X::char_type`	*Returns:* if for some `c`, `X::eq_int_type(e,X::to_int_type(c))` is `true`, `c`; else some unspecified value.	constant
`X::to_int_type(c)`	`X::int_type`	*Returns:* some value `e`, constrained by the definitions of `to_char_type` and `eq_int_type`.	constant

Table 57 — Character traits requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::eq_int_type(e,f)`	`bool`	*Returns:* for all `c` and `d`, `X::eq(c,d)` is equal to `X::eq_int_type(X::to_int_type(c), X::to_int_type(d))`; otherwise, yields `true` if `e` and `f` are both copies of `X::eof()`; otherwise, yields `false` if one of `e` and `f` is a copy of `X::eof()` and the other is not; otherwise the value is unspecified.	constant
`X::eof()`	`X::int_type`	*Returns:* a value `e` such that `X::eq_int_type(e,X::to_int_type(c))` is `false` for all values `c`.	constant

2 The class template

```
template<class charT> struct char_traits;
```

~~shall be~~is provided in the header `<string>` as a basis for explicit specializations.

20.2.2 Traits typedefs [char.traits.typedefs]

```
using char_type = CHAR_T;
```

1 The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 20.3 and Clause 27.

`using int_type =` ~~`INT_T`~~*`see below`*`;`

2 ~~*Requires:*~~ *Expects:* ~~For a certain character container type `char_type`, a related container type `INT_T`~~`int_type` shall be ~~a type or class which can~~able to represent all of the valid characters converted from the corresponding `char_type` values, as well as an end-of-file value, `eof()`. ~~The type `int_type` represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions.~~[226]

[*Drafting note*: This subclause is specifying the requirements for all character traits, so it's not clear how the type can be implementation-defined. We can remove the level of indirection by changing the table above to point to the referenced subclauses directly. (Not that they impose any requirements beyond "the behavior is implementation-defined if you don't use `streamoff` and `streampos`".) – *end drafting note*]

```
using off_type = implementation-defined;
using pos_type = implementation-defined;
```

3 *Requires:* Requirements for `off_type` and `pos_type` are described in 27.2.2 and 27.3.

[*Drafting note*: This one is intentionally kept as a *Requires:* in this paper. When we eventually decide on the right way to handle named requirements with both syntactic and semantic components, it can be updated accordingly. – *end drafting note*]

`using state_type =` ~~`STATE_T`~~*`see below`*`;`

4 *Requires:* `state_type` shall ~~satisfy~~meet the *Cpp17Destructible* (Table 29), *Cpp17CopyAssignable* (Table 28), *Cpp17CopyConstructible* (Table 26), and *Cpp17DefaultConstructible* (Table 24) requirements.

20.2.3 `char_traits` specializations [char.traits.specializations]

```
namespace std {
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
```

[226] If `eof()` can be held in `char_type` then some iostreams operations ~~may~~can give surprising results.

```
    template<> struct char_traits<char32_t>;
    template<> struct char_traits<wchar_t>;
  }
```

1 The header `<string>` ~~shall define~~defines four specializations of the class template `char_traits`: `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`.

2 The requirements for the members of these specializations are given in 20.2.1.

20.2.3.1 `struct char_traits<char>` [char.traits.specializations.char]

```
  namespace std {
    template<> struct char_traits<char> {
      using char_type  = char;
      using int_type   = int;
      using off_type   = streamoff;
      using pos_type   = streampos;
      using state_type = mbstate_t;

      static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
      static constexpr bool eq(char_type c1, char_type c2) noexcept;
      static constexpr bool lt(char_type c1, char_type c2) noexcept;

      static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
      static constexpr size_t length(const char_type* s);
      static constexpr const char_type* find(const char_type* s, size_t n,
                                             const char_type& a);
      static char_type* move(char_type* s1, const char_type* s2, size_t n);
      static char_type* copy(char_type* s1, const char_type* s2, size_t n);
      static char_type* assign(char_type* s, size_t n, char_type a);

      static constexpr int_type not_eof(int_type c) noexcept;
      static constexpr char_type to_char_type(int_type c) noexcept;
      static constexpr int_type to_int_type(char_type c) noexcept;
      static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
      static constexpr int_type eof() noexcept;
    };
  }
```

1 The defined types for `int_type`, `pos_type`, `off_type`, and `state_type` shall be `int`, `streampos`, `streamoff`, and `mbstate_t` respectively.

2 The type `streampos` shall be an implementation-defined type that satisfies the requirements for `pos_type` in 27.2.2 and 27.3.

3 The type `streamoff` is an implementation-defined type that satisfies the requirements for `off_type` in 27.2.2 and 27.3.

4 The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

5 The two-argument member `assign` ~~shall be~~is defined identically to the built-in operator `=`. The two-argument members `eq` and `lt` ~~shall be~~are defined identically to the built-in operators `==` and `<` for type `unsigned char`.

6 The member `eof()` ~~shall return~~returns `EOF`.

20.2.3.2 `struct char_traits<char16_t>` [char.traits.specializations.char16_t]

```
  namespace std {
    template<> struct char_traits<char16_t> {
      using char_type  = char16_t;
      using int_type   = uint_least16_t;
      using off_type   = streamoff;
      using pos_type   = u16streampos;
      using state_type = mbstate_t;

      static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
      static constexpr bool eq(char_type c1, char_type c2) noexcept;
      static constexpr bool lt(char_type c1, char_type c2) noexcept;
      static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
      static constexpr size_t length(const char_type* s);
      static constexpr const char_type* find(const char_type* s, size_t n,
                                             const char_type& a);
      static char_type* move(char_type* s1, const char_type* s2, size_t n);
      static char_type* copy(char_type* s1, const char_type* s2, size_t n);
      static char_type* assign(char_type* s, size_t n, char_type a);

      static constexpr int_type not_eof(int_type c) noexcept;
      static constexpr char_type to_char_type(int_type c) noexcept;
      static constexpr int_type to_int_type(char_type c) noexcept;
      static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
      static constexpr int_type eof() noexcept;
    };
  }
```

1 The type `u16streampos` shall be an implementation-defined type that satisfies the requirements for `pos_type` in 27.2.2 and 27.3.

2 The two-argument members `assign`, `eq`, and `lt` ~~shall be~~are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

3 The member `eof()` ~~shall return~~returns an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

20.2.3.3 `struct char_traits<char32_t>` [char.traits.specializations.char32__t]

```
namespace std {
  template<> struct char_traits<char32_t> {
    using char_type  = char32_t;
    using int_type   = uint_least32_t;
    using off_type   = streamoff;
    using pos_type   = u32streampos;
    using state_type = mbstate_t;

    static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
    static constexpr size_t length(const char_type* s);
    static constexpr const char_type* find(const char_type* s, size_t n,
                                           const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

1 The type `u32streampos` shall be an implementation-defined type that satisfies the requirements for `pos_type` in 27.2.2 and 27.3.

2 The two-argument members `assign`, `eq`, and `lt` ~~shall be~~are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

3 The member `eof()` ~~shall return~~returns an implementation-defined constant that cannot appear as a Unicode code point.

20.2.3.4 `struct char_traits<wchar_t>` [char.traits.specializations.wchar.t]

```
namespace std {
  template<> struct char_traits<wchar_t> {
    using char_type  = wchar_t;
    using int_type   = wint_t;
    using off_type   = streamoff;
    using pos_type   = wstreampos;
    using state_type = mbstate_t;

    static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;

    static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
    static constexpr size_t length(const char_type* s);
    static constexpr const char_type* find(const char_type* s, size_t n,
                                           const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

1 The defined types for `int_type`, `pos_type`, and `state_type` shall be `wint_t`, `wstreampos`, and `mbstate_t` respectively.

2 The type `wstreampos` shall be an implementation-defined type that satisfies the requirements for `pos_type` in 27.2.2 and 27.3.

3 The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

4 The two-argument members `assign`, `eq`, and `lt` ~~shall be~~are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

5 The member `eof()` ~~shall return~~returns `WEOF`.

20.3 String classes [string.classes]

1 The header `<string>` defines the `basic_string` class template for manipulating varying-length sequences of char-like objects and four *typedef-names*, `string`, `u16string`, `u32string`, and `wstring`, that name the specializations `basic_string<char>`, `basic_string<char16_t>`, `basic_string<char32_t>`, and `basic_string<wchar_t>`, respectively.

20.3.1 Header `<string>` synopsis [string.syn]

```
#include <initializer_list>

namespace std {
  // 20.2, character traits
  template<class charT> struct char_traits;
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;

  // 20.3.2, basic_string
  template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
    class basic_string;

  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs);
```

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              basic_string<charT, traits, Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              basic_string<charT, traits, Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const charT* lhs,
              const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const charT* lhs,
              basic_string<charT, traits, Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(charT lhs,
              const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(charT lhs,
              basic_string<charT, traits, Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              const charT* rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              const charT* rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              charT rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              charT rhs);

template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs,
                  const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs,
                  const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT, traits, Allocator>& lhs,
                  const charT* rhs);
```

```
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT, traits, Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs,
                  const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT, traits, Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs,
                  const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT, traits, Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs,
                  const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT, traits, Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT, traits, Allocator>& rhs);

// 20.3.3.9, swap
template<class charT, class traits, class Allocator>
  void swap(basic_string<charT, traits, Allocator>& lhs,
            basic_string<charT, traits, Allocator>& rhs)
    noexcept(noexcept(lhs.swap(rhs)));

// 20.3.3.10, inserters and extractors
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is,
               basic_string<charT, traits, Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const basic_string<charT, traits, Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str,
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str,
            charT delim);
```

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str);

// basic_string typedef names
using string    = basic_string<char>;
using u16string = basic_string<char16_t>;
using u32string = basic_string<char32_t>;
using wstring   = basic_string<wchar_t>;

// 20.3.4, numeric conversions
int stoi(const string& str, size_t* idx = nullptr, int base = 10);
long stol(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const string& str, size_t* idx = nullptr, int base = 10);
long long stoll(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = nullptr, int base = 10);
float stof(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
long double stold(const string& str, size_t* idx = nullptr);
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);

int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);
float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

namespace pmr {
  template<class charT, class traits = char_traits<charT>>
    using basic_string = std::basic_string<charT, traits, polymorphic_allocator<charT>>;

  using string    = basic_string<char>;
  using u16string = basic_string<char16_t>;
  using u32string = basic_string<char32_t>;
  using wstring   = basic_string<wchar_t>;
}

// 20.3.5, hash support
template<class T> struct hash;
```

```
  template<> struct hash<string>;
  template<> struct hash<u16string>;
  template<> struct hash<u32string>;
  template<> struct hash<wstring>;
  template<> struct hash<pmr::string>;
  template<> struct hash<pmr::u16string>;
  template<> struct hash<pmr::u32string>;
  template<> struct hash<pmr::wstring>;

  inline namespace literals {
  inline namespace string_literals {
    // 20.3.6, suffix for basic_string literals
    string    operator""s(const char* str, size_t len);
    u16string operator""s(const char16_t* str, size_t len);
    u32string operator""s(const char32_t* str, size_t len);
    wstring   operator""s(const wchar_t* str, size_t len);
  }
  }
}
```

20.3.2 Class template `basic_string` [basic.string]

1 The class template `basic_string` describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a “string” if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a `basic_string` object is designated by `charT`.

[*Drafting note*: This paragraph duplicates the requirements in 20.3.2.1. – *end drafting note*]

2 ~~The member functions of `basic_string` use an object of the `Allocator` class passed as a template parameter to allocate and free storage for the contained char-like objects.~~[227]

3 A specialization of `basic_string` is a contiguous container (21.2.1).

4 In all cases, `[data(), data() + size()]` is a valid range, `data() + size()` points at an object with value `charT()` (a “null terminator”), and `size() <= capacity()` is `true`.

[*Drafting note*: This appears to have no normative effect. We don’t use the term "length error" or "out-of-range error" anywhere. – *end drafting note*]

5 The functions described in this Clause can report two kinds of errors, each associated with an exception type:

(5.1) — a *length* error is associated with exceptions of type `length_error` (18.2.5);

(5.2) — an *out-of-range* error is associated with exceptions of type `out_of_range` (18.2.6).

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_string {
  public:
    // types
    using traits_type            = traits;
    using value_type             = charT;
    using allocator_type         = Allocator;
    using size_type              = typename allocator_traits<Allocator>::size_type;
    using difference_type        = typename allocator_traits<Allocator>::difference_type;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;

    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    static const size_type npos  = -1;
```

[227] ~~`Allocator::value_type` must name the same type as `charT` (20.3.2.1).~~

```
// 20.3.2.2, construct/copy/destroy
basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
explicit basic_string(const Allocator& a) noexcept;
basic_string(const basic_string& str);
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string& str, size_type pos, const Allocator& a = Allocator());
basic_string(const basic_string& str, size_type pos, size_type n,
             const Allocator& a = Allocator());
template<class T>
  basic_string(const T& t, size_type pos, size_type n, const Allocator& a = Allocator());
template<class T>
  explicit basic_string(const T& t, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
  basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
basic_string(const basic_string&, const Allocator&);
basic_string(basic_string&&, const Allocator&);

~basic_string();
basic_string& operator=(const basic_string& str);
basic_string& operator=(basic_string&& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value);
template<class T>
  basic_string& operator=(const T& t);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(initializer_list<charT>);

// 20.3.2.3, iterators
iterator       begin() noexcept;
const_iterator begin() const noexcept;
iterator       end() noexcept;
const_iterator end() const noexcept;

reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator         cbegin() const noexcept;
const_iterator         cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 20.3.2.4, capacity
size_type size() const noexcept;
size_type length() const noexcept;
size_type max_size() const noexcept;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const noexcept;
void reserve(size_type res_arg);
void shrink_to_fit();
void clear() noexcept;
[[nodiscard]] bool empty() const noexcept;

// 20.3.2.5, element access
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
const_reference at(size_type n) const;
```

```
reference       at(size_type n);

const charT& front() const;
charT&       front();
const charT& back() const;
charT&       back();

// 20.3.2.6, modifiers
basic_string& operator+=(const basic_string& str);
template<class T>
  basic_string& operator+=(const T& t);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& operator+=(initializer_list<charT>);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
  basic_string& append(const T& t);
template<class T>
  basic_string& append(const T& t, size_type pos, size_type n = npos);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(initializer_list<charT>);

void push_back(charT c);

basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value);
basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
  basic_string& assign(const T& t);
template<class T>
  basic_string& assign(const T& t, size_type pos, size_type n = npos);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);

basic_string& insert(size_type pos, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n = npos);
template<class T>
  basic_string& insert(size_type pos, const T& t);
template<class T>
  basic_string& insert(size_type pos1, const T& t, size_type pos2, size_type n = npos);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);
```

```
void pop_back();

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2 = npos);
template<class T>
  basic_string& replace(size_type pos1, size_type n1, const T& t);
template<class T>
  basic_string& replace(size_type pos1, size_type n1, const T& t,
                        size_type pos2, size_type n2 = npos);
basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);

basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
template<class T>
  basic_string& replace(const_iterator i1, const_iterator i2, const T& t);
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                        InputIterator j1, InputIterator j2);
basic_string& replace(const_iterator, const_iterator, initializer_list<charT>);

size_type copy(charT* s, size_type n, size_type pos = 0) const;

void swap(basic_string& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
           allocator_traits<Allocator>::is_always_equal::value);

// 20.3.2.7, string operations
const charT* c_str() const noexcept;
const charT* data() const noexcept;
charT* data() noexcept;
operator basic_string_view<charT, traits>() const noexcept;
allocator_type get_allocator() const noexcept;

template<class T>
  size_type find (const T& t, size_type pos = 0) const noexcept(see below);
size_type find (const basic_string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const noexcept;
template<class T>
  size_type rfind(const T& t, size_type pos = npos) const noexcept(see below);
size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(charT c, size_type pos = npos) const noexcept;

template<class T>
  size_type find_first_of(const T& t, size_type pos = 0) const noexcept(see below);
size_type find_first_of(const basic_string& str, size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const noexcept;
template<class T>
  size_type find_last_of (const T& t, size_type pos = npos) const noexcept(see below);
size_type find_last_of (const basic_string& str, size_type pos = npos) const noexcept;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT c, size_type pos = npos) const noexcept;
```

```
    template<class T>
      size_type find_first_not_of(const T& t, size_type pos = 0) const noexcept(see below);
    size_type find_first_not_of(const basic_string& str, size_type pos = 0) const noexcept;
    size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
    size_type find_first_not_of(const charT* s, size_type pos = 0) const;
    size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
    template<class T>
      size_type find_last_not_of (const T& t, size_type pos = npos) const noexcept(see below);
    size_type find_last_not_of (const basic_string& str, size_type pos = npos) const noexcept;
    size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
    size_type find_last_not_of (const charT* s, size_type pos = npos) const;
    size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;

    basic_string substr(size_type pos = 0, size_type n = npos) const;
    template<class T>
      int compare(const T& t) const noexcept(see below);
    template<class T>
      int compare(size_type pos1, size_type n1, const T& t) const;
    template<class T>
      int compare(size_type pos1, size_type n1, const T& t,
                  size_type pos2, size_type n2 = npos) const;
    int compare(const basic_string& str) const noexcept;
    int compare(size_type pos1, size_type n1, const basic_string& str) const;
    int compare(size_type pos1, size_type n1, const basic_string& str,
                size_type pos2, size_type n2 = npos) const;
    int compare(const charT* s) const;
    int compare(size_type pos1, size_type n1, const charT* s) const;
    int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

    bool starts_with(basic_string_view<charT, traits> x) const noexcept;
    bool starts_with(charT x) const noexcept;
    bool starts_with(const charT* x) const;
    bool ends_with(basic_string_view<charT, traits> x) const noexcept;
    bool ends_with(charT x) const noexcept;
    bool ends_with(const charT* x) const;
  };

  template<class InputIterator,
           class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
    basic_string(InputIterator, InputIterator, Allocator = Allocator())
      -> basic_string<typename iterator_traits<InputIterator>::value_type,
                      char_traits<typename iterator_traits<InputIterator>::value_type>,
                      Allocator>;

  template<class charT,
           class traits,
           class Allocator = allocator<charT>>
    explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
      -> basic_string<charT, traits, Allocator>;

  template<class charT,
           class traits,
           class Allocator = allocator<charT>>
    basic_string(basic_string_view<charT, traits>,
                 typename see below::size_type, typename see below::size_type,
                 const Allocator& = Allocator())
      -> basic_string<charT, traits, Allocator>;
}
```

6 A `size_type` parameter type in a `basic_string` deduction guide refers to the `size_type` member type of the type deduced by the deduction guide.

20.3.2.1 General requirements [string.require]

1 If any operation would cause size() to exceed max_size(), that operation ~~shall throw~~throws an exception object of type length_error.

2 If any member function or operator of basic_string throws an exception, that function or operator ~~shall have~~has no other effect on the basic_string object.

3 In every specialization basic_string<charT, traits, Allocator>, the type allocator_traits<Allocator>::value_type shall name the same type as charT. Every object of type basic_string<charT, traits, Allocator> ~~shall use~~uses an object of type Allocator to allocate and free storage for the contained charT objects as needed. The Allocator object used ~~shall be~~is obtained as described in 21.2.1. In every specialization basic_string<charT, traits, Allocator>, the type traits shall satisfy the character traits requirements (20.2) ~~and the type traits::char_type shall name the same type as charT~~. [*Note:* The program is ill-formed if traits::char_type is not the same type as charT. — *end note*]

4 References, pointers, and iterators referring to the elements of a basic_string sequence may be invalidated by the following uses of that basic_string object:

(4.1) — as an argument to any standard library function taking a reference to non-const basic_string as an argument.[228]

(4.2) — Calling non-const member functions, except operator[], at, data, front, back, begin, rbegin, end, and rend.

20.3.2.2 Constructors and assignment operators [string.cons]

```
explicit basic_string(const Allocator& a) noexcept;
```

1 *Effects:* Constructs an object of class basic_string.

2 *Ensures:* size() ~~is~~== 0 ~~and capacity() is an unspecified value~~.

```
basic_string(const basic_string& str);
basic_string(basic_string&& str) noexcept;
```

3 *Effects:* Constructs an object ~~of class basic_string~~ whose value is that of str prior to this call.

4 ~~*Ensures:*~~ *Remarks:* ~~data() points at the first element of an allocated copy of the array whose first element is pointed at by the original value str.data(), size() is equal to the original value of str.size(), and capacity() is a value at least as large as size().~~ In the second form, str is left in a valid but unspecified state ~~with an unspecified value~~.

```
basic_string(const basic_string& str, size_type pos,
             const Allocator& a = Allocator());
basic_string(const basic_string& str, size_type pos, size_type n,
             const Allocator& a = Allocator());
```

[*Drafting note*: The *Throws:* part is covered by the specification of basic_string_view::substr. – *end drafting note*]

5 *Throws:* out_of_range if pos > str.size().

6 *Effects:* Constructs an object of class basic_string and determines the effective length rlen of the initial string value as str.size() - pos in the first form and as the smaller of str.size() - pos and n in the second form.

7 *Ensures:* data() points at the first element of an allocated copy of rlen consecutive elements of the string controlled by str beginning at position pos, size() is equal to rlen, and capacity() is a value at least as large as size().

8 *Effects:* Let n be npos for the first overload. Equivalent to:

```
basic_string(basic_string_view<charT, traits>(str).substr(pos, n), a)
```

```
template<class T>
  basic_string(const T& t, size_type pos, size_type n, const Allocator& a = Allocator());
```

9 *Constraints:* is_convertible_v<const T&, basic_string_view<charT, traits>> is true.

[228] For example, as an argument to non-member functions swap() (20.3.3.9), operator>>() (20.3.3.10), and getline() (20.3.3.10), or as an argument to basic_string::swap().

10 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then behaves the same as:

```
basic_string(sv.substr(pos, n), a);
```

11 *Remarks:* This constructor shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true`.

```
template<class T>
  explicit basic_string(const T& t, const Allocator& a = Allocator());
```

12 *Constraints:*

(12.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(12.2) — `is_convertible_v<const T&, const charT*>` is `false`.

13 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then behaves the same as `basic_string(sv.data(), sv.size(), a)`.

14 *Remarks:* This constructor shall not participate in overload resolution unless

(14.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(14.2) — `is_convertible_v<const T&, const charT*>` is `false`.

```
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

15 ~~*Requires:* `s` points to an array of at least `n` elements of `charT`.~~ *Expects:* `[s, s + n)` is a valid range.

16 *Effects:* Constructs an object ~~of class `basic_string` and determines its initial string value from~~whose initial value is the ~~array of `charT` of length `n` whose first element is designated by `s`~~ range `[s, s + n)`.

17 *Ensures:* ~~`data()` points at the first element of an allocated copy of the array whose first element is pointed at by `s`,~~ `size()` is equal to `n`, and ~~`capacity()` is a value at least as large as `size()`~~ `traits::compare(data(), s, n) == 0`.

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

18 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

19 *Effects:* Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `traits::length(s)` whose first element is designated by `s`.

20 *Ensures:* `data()` points at the first element of an allocated copy of the array whose first element is pointed at by `s`, `size()` is equal to `traits::length(s)`, and `capacity()` is a value at least as large as `size()`.

21 ~~*Remarks:* Shall not participate in overload resolution if~~*Constraints:* `Allocator` is a type that ~~does not qualify~~qualifies as an allocator (21.2.1). [*Note:* This affects class template argument deduction. — *end note*]

22 *Effects:* Equivalent to: `basic_string(s, traits::length(s), a)`.

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

23 *Requires:* `n < npos`.

24 ~~*Remarks:* Shall not participate in overload resolution if~~*Constraints:* `Allocator` is a type that ~~does not qualify~~qualifies as an allocator (21.2.1). [*Note:* This affects class template argument deduction. — *end note*]

25 *Effects:* Constructs an object ~~of class `basic_string` and determines its initial string value by repeating the char-like object `c` for all `n` elements.~~whose value consists of `n` copies of `c`.

26 *Ensures:* `data()` points at the first element of an allocated array of `n` elements, each storing the initial value `c`, `size()` is equal to `n`, and `capacity()` is a value at least as large as `size()`.

```
template<class InputIterator>
  basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
```

27 *Constraints:* `InputIterator` is a type that qualifies as an input iterator (21.2.1).

28 *Effects:* ~~If `InputIterator` is an integral type, equivalent to:~~

~~`basic_string(static_cast<size_type>(begin), static_cast<value_type>(end), a);`~~

~~Otherwise c~~Constructs a string from the values in the range [begin, end), as indicated in ~~the Sequence Requirements table (see 21.2.3)~~Table 68.

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

29 *Effects:* ~~Same as~~Equivalent to basic_string(il.begin(), il.end(), a).

```
basic_string(const basic_string& str, const Allocator& alloc);
basic_string(basic_string&& str, const Allocator& alloc);
```

30 *Effects:* Constructs an object ~~of class basic_string~~ whose value is that of str prior to this call. The stored allocator is constructed from alloc. In the second form, str is left in a valid but unspecified state.

31 *Ensures:* data() points at the first element of an allocated copy of the array whose first element is pointed at by the original value of str.data(), size() is equal to the original value of str.size(), capacity() is a value at least as large as size(), and get_allocator() is equal to alloc. In the second form, str is left in a valid state with an unspecified value.

32 *Throws:* The second form throws nothing if alloc == str.get_allocator().

```
template<class InputIterator,
         class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
  basic_string(InputIterator, InputIterator, Allocator = Allocator())
    -> basic_string<typename iterator_traits<InputIterator>::value_type,
                    char_traits<typename iterator_traits<InputIterator>::value_type>,
                    Allocator>;
```

33 ~~*Remarks:* Shall not participate in overload resolution if~~*Constraints:* InputIterator is a type that ~~does not qualify~~qualifies as an input iterator, ~~or if~~and Allocator is a type that ~~does not qualify~~qualifies as an allocator (21.2.1).

```
template<class charT,
         class traits,
         class Allocator = allocator<charT>>
  explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
    -> basic_string<charT, traits, Allocator>;

template<class charT,
         class traits,
         class Allocator = allocator<charT>>
  basic_string(basic_string_view<charT, traits>,
               typename see below::size_type, typename see below::size_type,
               const Allocator& = Allocator())
    -> basic_string<charT, traits, Allocator>;
```

34 ~~*Remarks:* Shall not participate in overload resolution if~~*Constraints:* Allocator is a type that ~~does not qualify~~qualifies as an allocator (21.2.1).

```
basic_string& operator=(const basic_string& str);
```

35 *Effects:* If *this and str are the same object, has no effect. Otherwise, replaces the value of *this with a copy of str.

36 *Returns:* *this.

37 *Ensures:* If *this and str are the same object, the member has no effect. Otherwise, data() points at the first element of an allocated copy of the array whose first element is pointed at by str.data(), size() is equal to str.size(), and capacity() is a value at least as large as size().

```
basic_string& operator=(basic_string&& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value);
```

38 *Effects:* Move assigns as a sequence container (21.2), except that iterators, pointers and references may be invalidated.

39 *Returns:* *this.

```
template<class T>
  basic_string& operator=(const T& t);
```

40 *Constraints:*

(40.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(40.2) — `is_convertible_v<const T&, const charT*>` is `false`.

41 *Effects:* Equivalent to:

~~{~~

```
  basic_string_view<charT, traits> sv = t;
  return assign(sv);
```

~~}~~

42 ~~*Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.~~

```
basic_string& operator=(const charT* s);
```

43 ~~*Returns:*~~ *Effects:* Equivalent to `return *this = basic_string_view<charT, traits>(s);`~~.~~

44 ~~*Remarks:* Uses `traits::length()`.~~

```
basic_string& operator=(charT c);
```

45 ~~*Returns:*~~ *Effects:* Equivalent to:

`return *this =` ~~`basic_string(1, c)`~~`basic_string_view<charT, traits>(addressof(c), 1);`

```
basic_string& operator=(initializer_list<charT> il);
```

46 *Effects:* ~~As if by: `*this = basic_string(il);`~~Equivalent to:

```
return *this = basic_string_view<charT, traits>(il.begin(), il.size());
```

47 ~~*Returns:* `*this`.~~

20.3.2.3 Iterator support [string.iterators]

```
iterator       begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
```

1 *Returns:* An iterator referring to the first character in the string.

```
iterator       end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
```

2 *Returns:* An iterator which is the past-the-end value.

```
reverse_iterator       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;
```

3 *Returns:* An iterator which is semantically equivalent to `reverse_iterator(end())`.

```
reverse_iterator       rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;
```

4 *Returns:* An iterator which is semantically equivalent to `reverse_iterator(begin())`.

20.3.2.4 Capacity [string.capacity]

```
size_type size() const noexcept;
size_type length() const noexcept;
```

1 *Returns:* A count of the number of char-like objects currently in the string.

2 *Complexity:* Constant time.

```
size_type length() const noexcept;
```

3 *Returns:* `size()`.

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

4 *Returns:* The largest possible number of char-like objects that can be stored in a `basic_string`.

5 *Complexity:* Constant time.

```
void resize(size_type n, charT c);
```

6 ~~*Throws:* `length_error` if `n > max_size()`.~~ [*Drafting note*: This is covered by the front matter (20.3.2.1 p1). – *end drafting note*]

7 *Effects:* Alters ~~the length of the string designated by~~ the value of `*this` as follows:

(7.1) — If `n <= size()`, ~~the function replaces the string designated by `*this` with a string of length `n` whose elements are a copy of the initial elements of the original string designated by `*this`.~~ erases the last `size() - n` elements.

(7.2) — If `n > size()`, ~~the function replaces the string designated by `*this` with a string of length `n` whose first `size()` elements are a copy of the original string designated by `*this`, and whose remaining elements are all initialized to `c`.~~ appends `n - size()` copies of `c`.

```
void resize(size_type n);
```

8 *Effects:* ~~As if by~~ Equivalent to `resize(n, charT())`.

```
size_type capacity() const noexcept;
```

9 *Returns:* The size of the allocated storage in the string.

10 *Complexity:* Constant time.

```
void reserve(size_type res_arg);
```

11 *Effects:* A directive that informs a `basic_string` of a planned change in size, so that the storage allocation can be managed accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`.

12 *Throws:* `length_error` if `res_arg > max_size()` or any exceptions thrown by `allocator_traits<Allocator>::allocate`.[229]

```
void shrink_to_fit();
```

13 *Effects:* `shrink_to_fit` is a non-binding request to reduce `capacity()` to `size()`. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*] It does not increase `capacity()`, but may reduce `capacity()` by causing reallocation.

14 *Complexity:* Linear in the size of the sequence.

15 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence as well as the past-the-end iterator. If no reallocation happens, they remain valid.

```
void clear() noexcept;
```

16 *Effects:* ~~Behaves as if the function calls:~~ Equivalent to: `erase(begin(), end());`

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

17 ~~*Returns:*~~ *Effects:* Equivalent to: `return size() == 0`;~~.~~

20.3.2.5 Element access [string.access]

```
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
```

1 ~~*Requires:*~~ *Expects:* `pos <= size()`.

[229] ~~`reserve()` uses `allocator_traits<Allocator>::allocate()` which may throw an appropriate exception.~~

2 *Returns:* *(begin() + pos) if pos < size(). Otherwise, returns a reference to an object of type charT with value charT(), where modifying the object to any value other than charT() leads to undefined behavior.

3 *Throws:* Nothing.

4 *Complexity:* Constant time.

```
const_reference at(size_type pos) const;
reference       at(size_type pos);
```

5 *Throws:* out_of_range if pos >= size().

6 *Returns:* operator[](pos).

```
const charT& front() const;
charT& front();
```

7 ~~*Requires:*~~ *Expects:* !empty().

8 *Effects:* Equivalent to: return operator[](0);

```
const charT& back() const;
charT& back();
```

9 ~~*Requires:*~~ *Expects:* !empty().

10 *Effects:* Equivalent to: return operator[](size() - 1);

20.3.2.6 Modifiers [string.modifiers]

20.3.2.6.1 basic_string::operator+= [string.op+=]

```
basic_string& operator+=(const basic_string& str);
```

1 *Effects:* ~~Calls~~Equivalent to: return append(str);~~.~~

2 ~~*Returns:* *this.~~

```
template<class T>
  basic_string& operator+=(const T& t);
```

3 *Effects:* Creates a variable, sv, as if by basic_string_view<charT, traits> sv = t; and then calls append(sv).

4 *Returns:* *this.

5 *Remarks:* This function shall not participate in overload resolution unless is_convertible_v<const T&, basic_string_view<charT, traits>> is true and is_convertible_v<const T&, const charT*> is false.

6 *Constraints:*

(6.1) — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and

(6.2) — is_convertible_v<const T&, const charT*> is false.

7 *Effects:* Equivalent to:

```
basic_string_view<charT, traits> sv = t;
return append(sv);
```

```
basic_string& operator+=(const charT* s);
```

8 *Effects:* ~~Calls~~Equivalent to: return append(s);~~.~~

9 ~~*Returns:* *this.~~

```
basic_string& operator+=(charT c);
```

10 *Effects:* ~~Calls push_back(c);~~Equivalent to: return append(size_type{1}, c);

11 ~~*Returns:* *this.~~

```
basic_string& operator+=(initializer_list<charT> il);
```

12 *Effects:* ~~Calls~~Equivalent to: return append(il);~~.~~

13 ~~*Returns:* *this.~~

20.3.2.6.2 basic_string::append [string.append]

```
basic_string& append(const basic_string& str);
```

1 *Effects:* ~~Calls~~Equivalent to: `return append(str.data(), str.size());`~~.~~

2 ~~*Returns:* `*this`.~~

```
basic_string& append(const basic_string& str, size_type pos, size_type n = npos);
```

3 *Throws:* `out_of_range` if `pos > str.size()`.

4 *Effects:* Determines the effective length `rlen` of the string to append as the smaller of `n` and `str.size() - pos` and calls `append(str.data() + pos, rlen)`.

5 *Returns:* `*this`.

6 *Effects:* Equivalent to: `return append(basic_string_view<charT, traits>(str).substr(pos, n));`

```
template<class T>
  basic_string& append(const T& t);
```

7 *Constraints:*

(7.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(7.2) — `is_convertible_v<const T&, const charT*>` is `false`.

8 *Effects:* Equivalent to:

```
  ~~{~~
    basic_string_view<charT, traits> sv = t;
    return append(sv.data(), sv.size());
  ~~}~~
```

9 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

```
template<class T>
  basic_string& append(const T& t, size_type pos, size_type n = npos);
```

10 *Throws:* `out_of_range` if `pos > sv.size()`.

11 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t`. Determines the effective length `rlen` of the string to append as the smaller of `n` and `sv.size() - pos` and calls `append(sv.data() + pos, rlen)`.

12 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

13 *Returns:* `*this`.

14 *Constraints:*

(14.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(14.2) — `is_convertible_v<const T&, const charT*>` is `false`.

15 *Effects:* Equivalent to:

```
    basic_string_view<charT, traits> sv = t;
    return append(sv.substr(pos, n));
```

```
basic_string& append(const charT* s, size_type n);
```

16 *Requires:* `s` points to an array of at least `n` elements of `charT`.

17 *Throws:* `length_error` if `size() + n > max_size()`.

18 *Effects:* The function replaces the string controlled by `*this` with a string of length `size() + n` whose first `size()` elements are a copy of the original string controlled by `*this` and whose remaining elements are a copy of the initial `n` elements of `s`.

19 *Expects:* `[s, s + n)` is a valid range.

20 *Effects:* Appends a copy of the range [s, s + n) to the string.

21 *Returns:* *this.

```
basic_string& append(const charT* s);
```

22 ~~*Requires:* s points to an array of at least traits::length(s) + 1 elements of charT.~~

23 *Effects:* ~~Calls~~Equivalent to: return append(s, traits::length(s));~~.~~

24 ~~*Returns:* *this.~~

```
basic_string& append(size_type n, charT c);
```

25 *Effects:* ~~Equivalent to: return append(basic_string(n, c));~~ Appends n copies of c to the string.

26 *Returns:* *this.

```
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
```

27 *Constraints:* InputIterator is a type that qualifies as an input iterator (21.2.1).

28 *Requires:* [first, last) is a valid range.

29 *Effects:* Equivalent to: return append(basic_string(first, last, get_allocator()));

```
basic_string& append(initializer_list<charT> il);
```

30 *Effects:* ~~Calls~~Equivalent to: return append(il.begin(), il.size());~~.~~

31 ~~*Returns:* *this.~~

```
void push_back(charT c);
```

32 *Effects:* Equivalent to append(~~static_cast<size_type>(~~size_type{1}~~)~~, c).

20.3.2.6.3 basic_string::assign [string.assign]

```
basic_string& assign(const basic_string& str);
```

1 *Effects:* Equivalent to: return *this = str;

```
basic_string& assign(basic_string&& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value);
```

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

```
basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);
```

3 *Throws:* out_of_range if pos > str.size().

4 *Effects:* Determines the effective length rlen of the string to assign as the smaller of n and str.size() - pos and calls assign(str.data() + pos, rlen).

5 *Returns:* *this.

6 *Effects:* Equivalent to:

```
return assign(basic_string_view<charT, traits>(str).substr(pos, n));
```

```
template<class T>
  basic_string& assign(const T& t);
```

7 *Constraints:*

(7.1) — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and

(7.2) — is_convertible_v<const T&, const charT*> is false.

8 *Effects:* Equivalent to:

```
~~{~~
  basic_string_view<charT, traits> sv = t;
  return assign(sv.data(), sv.size());
~~}~~
```

9 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

```
template<class T>
  basic_string& assign(const T& t, size_type pos, size_type n = npos);
```

10 *Throws:* `out_of_range` if `pos > sv.size()`.

11 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t`. Determines the effective length `rlen` of the string to assign as the smaller of `n` and `sv.size() - pos` and calls `assign(sv.data() + pos, rlen)`.

12 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

13 *Returns:* `*this`.

14 *Constraints:*

(14.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(14.2) — `is_convertible_v<const T&, const charT*>` is `false`.

15 *Effects:* Equivalent to:

```
basic_string_view<charT, traits> sv = t;
return assign(sv.substr(pos, n));
```

```
basic_string& assign(const charT* s, size_type n);
```

16 ~~*Requires:* `s` points to an array of at least `n` elements of `charT`.~~*Expects:* `[s, s + n)` is a valid range.

17 ~~*Throws:* `length_error` if `n > max_size()`.~~

18 *Effects:* Replaces the string controlled by `*this` with ~~a string of length `n` whose elements are a copy of those pointed to by `s`.~~a copy of the range `[s, s + n)`.

19 *Returns:* `*this`.

```
basic_string& assign(const charT* s);
```

20 ~~*Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.~~

21 *Effects:* ~~Calls~~Equivalent to: `return assign(s, traits::length(s));`~~.~~

22 ~~*Returns:* `*this`.~~

```
basic_string& assign(initializer_list<charT> il);
```

23 *Effects:* ~~Calls~~Equivalent to: `return assign(il.begin(), il.size());`~~.~~

24 ~~*Returns:* `*this`.~~

```
basic_string& assign(size_type n, charT c);
```

25 *Effects:* Equivalent to: ~~`return assign(basic_string(n, c));`~~

```
clear();
resize(n, c);
return *this;
```

```
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
```

26 *Constraints:* `InputIterator` is a type that qualifies as an input iterator (21.2.1).

27 *Effects:* Equivalent to: `return assign(basic_string(first, last, get_allocator()));`

20.3.2.6.4 `basic_string::insert` [string.insert]

```
basic_string& insert(size_type pos, const basic_string& str);
```

1 *Effects:* Equivalent to: `return insert(pos, str.data(), str.size());`

```
basic_string& insert(size_type pos1, const basic_string& str, size_type pos2, size_type n = npos);
```

2 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > str.size()`.

3 *Effects:* Determines the effective length `rlen` of the string to insert as the smaller of `n` and `str.size() - pos2` and calls `insert(pos1, str.data() + pos2, rlen)`.

4 *Returns:* `*this`.

5 *Effects:* Equivalent to:

```
return insert(pos1, basic_string_view<charT, traits>(str), pos2, n);
```

```
template<class T>
  basic_string& insert(size_type pos, const T& t);
```

6 *Constraints:*

(6.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(6.2) — `is_convertible_v<const T&, const charT*>` is `false`.

7 *Effects:* Equivalent to:

```
~~{~~
  basic_string_view<charT, traits> sv = t;
  return insert(pos, sv.data(), sv.size());
~~}~~
```

8 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

```
template<class T>
  basic_string& insert(size_type pos1, const T& t, size_type pos2, size_type n = npos);
```

9 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > sv.size()`.

10 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t`. Determines the effective length `rlen` of the string to assign as the smaller of `n` and `sv.size() - pos2` and calls `insert(pos1, sv.data() + pos2, rlen)`.

11 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

12 *Returns:* `*this`.

13 *Constraints:*

(13.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(13.2) — `is_convertible_v<const T&, const charT*>` is `false`.

14 *Effects:* Equivalent to:

```
basic_string_view<charT, traits> sv = t;
return insert(pos1, sv.substr(pos2, n));
```

```
basic_string& insert(size_type pos, const charT* s, size_type n);
```

15 ~~*Requires:* `s` points to an array of at least `n` elements of `charT`.~~*Expects:* `[s, s + n)` is a valid range.

16 *Throws:*

(16.1) — `out_of_range` if `pos > size()`, ~~or~~

(16.2) — `length_error` if ~~`size() + n > max_size()`.~~`n > max_size() - size()`, or

(16.3) — any exceptions thrown by `allocator_traits<Allocator>::allocate`.

17 *Effects:* ~~Replaces the string controlled by `*this` with a string of length `size() + n` whose first `pos` elements are a copy of the initial elements of the original string controlled by `*this` and whose next `n` elements are a copy of the elements in `s` and whose remaining elements are a copy of the remaining elements of the original string controlled by `*this`.~~ Inserts a copy of the range `[s, s + n)` immediately before the character at position `pos` if `pos < size()`, or otherwise at the end of the string.

18 *Returns:* `*this`.

```
basic_string& insert(size_type pos, const charT* s);
```

19 ~~*Requires:* s points to an array of at least traits::length(s) + 1 elements of charT.~~

20 *Effects:* Equivalent to: return insert(pos, s, traits::length(s));

```
basic_string& insert(size_type pos, size_type n, charT c);
```

21 *Throws:*

(21.1) — out_of_range if pos > size(),

(21.2) — length_error if n > max_size() - size(), or

(21.3) — any exceptions thrown by allocator_traits<Allocator>::allocate.

22 *Effects:* ~~Equivalent to: return insert(pos, basic_string(n, c));~~ Inserts n copies of c before the character at position pos if pos < size(), or otherwise at the end of the string.

```
iterator insert(const_iterator p, charT c);
```

23 ~~*Requires:*~~ *Expects:* p is a valid iterator on *this.

24 *Effects:* Inserts a copy of c ~~before the character referred to by~~at the position p. [*Drafting note*: This avoids suggesting that p needs to refer to a character. – *end drafting note*]

25 *Returns:* An iterator which refers to ~~the copy of~~ the inserted character.

```
iterator insert(const_iterator p, size_type n, charT c);
```

26 ~~*Requires:*~~ *Expects:* p is a valid iterator on *this.

27 *Effects:* Inserts n copies of c ~~before the character referred to by~~at the position p.

28 *Returns:* An iterator which refers to ~~the copy of~~ the first inserted character, or p if n == 0.

```
template<class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
```

29 *Constraints:* InputIterator is a type that qualifies as an input iterator (21.2.1).

30 ~~*Requires:*~~ *Expects:* p is a valid iterator on *this. ~~[first, last) is a valid range.~~

31 *Effects:* Equivalent to insert(p - begin(), basic_string(first, last, get_allocator())).

32 *Returns:* An iterator which refers to ~~the copy of~~ the first inserted character, or p if first == last.

```
iterator insert(const_iterator p, initializer_list<charT> il);
```

33 *Effects:* ~~As if by~~Equivalent to: return insert(p, il.begin(), il.end());~~.~~

34 ~~*Returns:* An iterator which refers to the copy of the first inserted character, or p if il is empty.~~

20.3.2.6.5 basic_string::erase [string.erase]

```
basic_string& erase(size_type pos = 0, size_type n = npos);
```

1 *Throws:* out_of_range if pos > size().

2 *Effects:* Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos. Removes the characters in the range [begin() + pos, begin() + pos + xlen).

3 ~~The function then replaces the string controlled by *this with a string of length size() - xlen whose first pos elements are a copy of the initial elements of the original string controlled by *this, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.~~

4 *Returns:* *this.

```
iterator erase(const_iterator p);
```

5 *Expects:* p is a valid dereferenceable iterator on *this.

6 *Throws:* Nothing.

7 *Effects:* Removes the character referred to by p.

8 *Returns:* An iterator which points to the element immediately following p prior to the element being erased. If no such element exists, end() is returned.

```
iterator erase(const_iterator first, const_iterator last);
```

9 ~~*Requires:*~~ *Expects:* `first` and `last` are valid iterators on `*this`~~, defining a range `[first, last)`~~. `[first, last)` is a valid range.

10 *Throws:* Nothing.

11 *Effects:* Removes the characters in the range `[first, last)`.

12 *Returns:* An iterator which points to the element pointed to by `last` prior to the other elements being erased. If no such element exists, `end()` is returned.

```
void pop_back();
```

13 ~~*Requires:*~~ *Expects:* `!empty()`.

14 *Throws:* Nothing.

15 *Effects:* Equivalent to `erase(`~~`size`~~`end() - 1`~~`, 1`~~`)`.

20.3.2.6.6 `basic_string::replace` [string.replace]

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
```

1 *Effects:* Equivalent to: `return replace(pos1, n1, str.data(), str.size());`

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2 = npos);
```

2 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > str.size()`.

3 *Effects:* Determines the effective length `rlen` of the string to be inserted as the smaller of `n2` and `str.size() - pos2` and calls `replace(pos1, n1, str.data() + pos2, rlen)`.

4 *Returns:* `*this`.

5 *Effects:* Equivalent to:

```
  return replace(pos1, n1, basic_string_view<charT, traits>(str).substr(pos2, n2));
```

```
template<class T>
  basic_string& replace(size_type pos1, size_type n1, const T& t);
```

6 *Constraints:*

(6.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(6.2) — `is_convertible_v<const T&, const charT*>` is `false`.

7 *Effects:* Equivalent to:

```
~~{~~
  basic_string_view<charT, traits> sv = t;
  return replace(pos1, n1, sv.data(), sv.size());
~~}~~
```

8 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

```
template<class T>
  basic_string& replace(size_type pos1, size_type n1, const T& t,
                        size_type pos2, size_type n2 = npos);
```

9 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > sv.size()`.

10 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t`. Determines the effective length `rlen` of the string to be inserted as the smaller of `n2` and `sv.size() - pos2` and calls `replace(pos1, n1, sv.data() + pos2, rlen)`.

11 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

12 *Returns:* `*this`.

13 *Constraints:*

(13.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(13.2) — `is_convertible_v<const T&, const charT*>` is `false`.

14 *Effects:* Equivalent to:

```
basic_string_view<charT, traits> sv = t;
return replace(pos1, n1, sv.substr(pos2, n2));
```

```
basic_string& replace(size_type pos1, size_type n1, const charT* s, size_type n2);
```

15 ~~*Requires:* `s` points to an array of at least `n2` elements of `charT`.~~ *Expects:* `[s, s + n2)` is a valid range.

16 *Throws:*

(16.1) — `out_of_range` if `pos1 > size()`, ~~or~~

(16.2) — `length_error` if the length of the resulting string would exceed `max_size()` (see below)~~.~~, or

(16.3) — any exceptions thrown by `allocator_traits<Allocator>::allocate`.

17 *Effects:* Determines the effective length `xlen` of the string to be removed as the smaller of `n1` and `size() - pos1`. If `size() - xlen >= max_size() - n2` throws `length_error`. Otherwise, the function replaces ~~the string controlled by `*this` with a string of length `size() - xlen + n2` whose first `pos1` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `n2` elements are a copy of the initial `n2` elements of `s`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `pos + xlen`~~the characters in the range `[begin() + pos1, begin() + pos1 + xlen)` with a copy of the range `[s, s + n2)`.

18 *Returns:* `*this`.

```
basic_string& replace(size_type pos, size_type n, const charT* s);
```

19 ~~*Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.~~

20 *Effects:* Equivalent to: `return replace(pos, n, s, traits::length(s));`

```
basic_string& replace(size_type pos1, size_type n1, size_type n2, charT c);
```

21 *Throws:*

(21.1) — `out_of_range` if `pos1 > size()`,

(21.2) — `length_error` if the length of the resulting string would exceed `max_size()` (see below), or

(21.3) — any exceptions thrown by `allocator_traits<Allocator>::allocate`.

22 *Effects:* ~~Equivalent to: `return replace(pos1, n1, basic_string(n2, c));`~~ Determines the effective length `xlen` of the string to be removed as the smaller of `n1` and `size() - pos1`. If `size() - xlen >= max_size() - n2` throws `length_error`. Otherwise, the function replaces the characters in the range `[begin() + pos1, begin() + pos1 + xlen)` with `n2` copies of `c`.

23 *Returns:* `*this`.

```
basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
```

24 *Requires:* `[begin(), i1)` and `[i1, i2)` are valid ranges.

25 *Effects:* Calls `replace(i1 - begin(), i2 - i1, str)`.

26 *Returns:* `*this`.

27 *Effects:* Equivalent to: `return replace(i1, i2, basic_string_view<charT, traits>(str));`

```
template<class T>
  basic_string& replace(const_iterator i1, const_iterator i2, const T& t);
```

28 *Constraints:*

(28.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(28.2) — `is_convertible_v<const T&, const charT*>` is `false`.

29 ~~*Requires:*~~ *Expects:* `[begin(), i1)` and `[i1, i2)` are valid ranges.

30 *Effects:* ~~Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then calls `replace(i1 - begin(), i2 - i1, sv)`.~~Equivalent to:

```
basic_string_view<charT, traits> sv = t;
return replace(i1 - begin(), i2 - i1, sv.data(), sv.size());
```

31 *Returns:* `*this`.

32 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

```
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);
```

33 *Requires:* `[begin(), i1)` and `[i1, i2)` are valid ranges and `s` points to an array of at least `n` elements of `charT`.

34 *Effects:* Calls `replace(i1 - begin(), i2 - i1, s, n)`.

35 *Returns:* `*this`.

36 *Effects:* Equivalent to: `return replace(i1, i2, basic_string_view<charT, traits>(s, n));`

```
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
```

37 *Requires:* `[begin(), i1)` and `[i1, i2)` are valid ranges and `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

38 *Effects:* Calls `replace(i1 - begin(), i2 - i1, s, traits::length(s))`.

39 *Returns:* `*this`.

40 *Effects:* Equivalent to: `return replace(i1, i2, basic_string_view<charT, traits>(s));`

```
basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);
```

41 ~~*Requires:*~~ *Expects:* `[begin(), i1)` and `[i1, i2)` are valid ranges.

42 *Effects:* Calls `replace(i1 - begin(), i2 - i1, basic_string(n, c))`.

43 *Returns:* `*this`.

44 *Effects:* Equivalent to: `return replace(i1 - begin(), i2 - i1, n, c);`

```
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2);
```

45 *Requires:* `[begin(), i1)`, `[i1, i2)` and `[j1, j2)` are valid ranges.

46 *Effects:* Calls `replace(i1 - begin(), i2 - i1, basic_string(j1, j2, get_allocator()))`.

47 *Returns:* `*this`.

48 *Constraints:* `InputIterator` is a type that qualifies as an input iterator (21.2.1).

49 *Effects:* Equivalent to: `return replace(i1, i2, basic_string(j1, j2, get_allocator()));`

```
basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);
```

50 *Requires:* `[begin(), i1)` and `[i1, i2)` are valid ranges.

51 *Effects:* Calls `replace(i1 - begin(), i2 - i1, il.begin(), il.size())`.

52 *Returns:* `*this`.

53 *Effects:* Equivalent to: `return replace(i1, i2, il.begin(), il.size());`

20.3.2.6.7 `basic_string::copy` [string.copy]

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

1 Let `rlen` be the smaller of `n` and `size() - pos`.

2 *Throws:* `out_of_range` if `pos > size()`.

3 *Requires:* `[s, s + rlen)` is a valid range.

4 *Effects:* Equivalent to `traits::copy(s, data() + pos, rlen)`. [*Note:* This does not terminate `s` with a null object. — *end note*]

5 *Returns:* `rlen`.

6 *Effects:* Equivalent to: `return basic_string_view<charT, traits>(*this).copy(s, n, pos);` [*Note:* This does not terminate `s` with a null object. — *end note*]

20.3.2.6.8 `basic_string::swap` [string.swap]

```
void swap(basic_string& s)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
           allocator_traits<Allocator>::is_always_equal::value);
```

1 *Expects:* `allocator_traits<Allocator>::propagate_on_container_swap::value` is `true` or `get_allocator() == s.get_allocator()`.

2 *Ensures:* `*this` contains the same sequence of characters that was in `s`, `s` contains the same sequence of characters that was in `*this`.

3 *Throws:* Nothing.

4 *Complexity:* Constant time.

20.3.2.7 String operations [string.ops]

20.3.2.7.1 Accessors [string.accessors]

```
const charT* c_str() const noexcept;
const charT* data() const noexcept;
```

1 *Returns:* A pointer `p` such that `p + i == &operator[](i)` for each `i` in `[0, size()]`.

2 *Complexity:* Constant time.

3 ~~*Requires:*~~ *Remarks:* The program shall not ~~alter~~modify any of the values stored in the character array; otherwise, the behavior is undefined.

```
charT* data() noexcept;
```

4 *Returns:* A pointer `p` such that `p + i == &operator[](i)` for each `i` in `[0, size()]`.

5 *Complexity:* Constant time.

6 ~~*Requires:*~~ *Remarks:* The program shall not ~~alter~~modify the value stored at `p + size()` to any value other than `charT()`; otherwise, the behavior is undefined. [*Drafting note*: This makes the function consistent with `operator[]` after LWG issue 2475. – *end drafting note*]

```
operator basic_string_view<charT, traits>() const noexcept;
```

7 *Effects:* Equivalent to: `return basic_string_view<charT, traits>(data(), size());`

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

8 *Returns:* A copy of the `Allocator` object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

20.3.2.7.2 Searching [string.find]

1 Let *F* be one of `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, and `find_last_not_of`.

(1.1) — Each member function of the form

```
size_type F(const basic_string& str, size_type pos) const noexcept;
```

has effects equivalent to: `return F(basic_string_view<charT, traits>(str), pos);`

(1.2) — Each member function of the form

```
size_type F(const charT* s, size_type pos) const;
```

has effects equivalent to: `return F(basic_string_view<charT, traits>(s), pos);`

(1.3) — Each member function of the form

```
size_type F(const charT* s, size_type pos, size_type n) const;
```

has effects equivalent to: `return F(basic_string_view<charT, traits>(s, n), pos);`

(1.4) — Each member function of the form

```
size_type F(charT c, size_type pos) const noexcept;
```

has effects equivalent to: `return F(basic_string_view<charT, traits>(addressof(c), 1), pos);`

```
template<class T>
  size_type find(const T& t, size_type pos = 0) const noexcept(see below);
template<class T>
  size_type rfind(const T& t, size_type pos = npos) const noexcept(see below);
template<class T>
  size_type find_first_of(const T& t, size_type pos = 0) const noexcept(see below);
template<class T>
  size_type find_last_of(const T& t, size_type pos = npos) const noexcept(see below);
template<class T>
  size_type find_first_not_of(const T& t, size_type pos = 0) const noexcept(see below);
template<class T>
  size_type find_last_not_of(const T& t, size_type pos = npos) const noexcept(see below);
```

2 *Constraints:*

(2.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(2.2) — `is_convertible_v<const T&, const charT*>` is `false`.

3 *Effects:* Let *G* be the name of the function. Equivalent to:

```
        basic_string_view<charT, traits> s = *this, sv = t;
        return s.G(sv, pos);
```

4 *Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_convertible_v<const T&, basic_string_view<charT, traits>>`.

[*Drafting note*: The conditional `noexcept` restores these functions to the position before LWG issue 2946. – *end drafting note*]

20.3.2.7.3 `basic_string::find` [string.find]

```
template<class T>
  size_type find(const T& t, size_type pos = 0) const;
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the lowest position `xpos`, if possible, such that both of the following conditions hold:

(1.1) — `pos <= xpos` and `xpos + sv.size() <= size();`

(1.2) — `traits::eq(at(xpos + I), sv.at(I))` for all elements `I` of the data referenced by `sv`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

4 *Throws:* Nothing unless the initialization of `sv` throws an exception.

```
size_type find(const basic_string& str, size_type pos = 0) const noexcept;
```

5 *Effects:* Equivalent to: `return find(basic_string_view<charT, traits>(str), pos);`

```
size_type find(const charT* s, size_type pos, size_type n) const;
```

6 *Returns:* `find(basic_string_view<charT, traits>(s, n), pos)`.

```
size_type find(const charT* s, size_type pos = 0) const;
```

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

7 *Returns:* `find(basic_string_view<charT, traits>(s), pos)`.

```
size_type find(charT c, size_type pos = 0) const;
```

8 *Returns:* `find(basic_string(1, c), pos)`.

20.3.2.7.4 `basic_string::rfind` [string.rfind]

```
template<class T>
  size_type rfind(const T& t, size_type pos = npos) const;
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the highest position `xpos`, if possible, such that both of the following conditions hold:

(1.1) — `xpos <= pos` and `xpos + sv.size() <= size()`;

(1.2) — `traits::eq(at(xpos + I), sv.at(I))` for all elements `I` of the data referenced by `sv`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

4 *Throws:* Nothing unless the initialization of `sv` throws an exception.

```
size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
```

5 *Effects:* Equivalent to: `return rfind(basic_string_view<charT, traits>(str), pos);`

```
size_type rfind(const charT* s, size_type pos, size_type n) const;
```

6 *Returns:* `rfind(basic_string_view<charT, traits>(s, n), pos)`.

```
size_type rfind(const charT* s, size_type pos = npos) const;
```

7 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

8 *Returns:* `rfind(basic_string_view<charT, traits>(s), pos)`.

```
size_type rfind(charT c, size_type pos = npos) const;
```

9 *Returns:* `rfind(basic_string(1, c), pos)`.

20.3.2.7.5 `basic_string::find_first_of` [string.find.first.of]

```
template<class T>
  size_type find_first_of(const T& t, size_type pos = 0) const;
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the lowest position `xpos`, if possible, such that both of the following conditions hold:

(1.1) — `pos <= xpos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), sv.at(I))` for some element `I` of the data referenced by `sv`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

4 *Throws:* Nothing unless the initialization of `sv` throws an exception.

```
size_type find_first_of(const basic_string& str, size_type pos = 0) const noexcept;
```

5 *Effects:* Equivalent to: `return find_first_of(basic_string_view<charT, traits>(str), pos);`

```
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
```

6 *Returns:* `find_first_of(basic_string_view<charT, traits>(s, n), pos)`.

```
size_type find_first_of(const charT* s, size_type pos = 0) const;
```

7 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

8 *Returns:* `find_first_of(basic_string_view<charT, traits>(s), pos)`.

```
size_type find_first_of(charT c, size_type pos = 0) const;
```

9 *Returns:* `find_first_of(basic_string(1, c), pos)`.

20.3.2.7.6 `basic_string::find_last_of` [string.find.last.of]

```
template<class T>
  size_type find_last_of(const T& t, size_type pos = npos) const;
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the highest position `xpos`, if possible, such that both of the following conditions hold:

(1.1) — `xpos <= pos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), sv.at(I))` for some element `I` of the data referenced by `sv`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

3 *Throws:* Nothing unless the initialization of `sv` throws an exception.

4 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
size_type find_last_of(const basic_string& str, size_type pos = npos) const noexcept;
```

5 *Effects:* Equivalent to: `return find_last_of(basic_string_view<charT, traits>(str), pos);`

```
size_type find_last_of(const charT* s, size_type pos, size_type n) const;
```

6 *Returns:* `find_last_of(basic_string_view<charT, traits>(s, n), pos)`.

```
size_type find_last_of(const charT* s, size_type pos = npos) const;
```

7 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

8 *Returns:* `find_last_of(basic_string_view<charT, traits>(s), pos)`.

```
size_type find_last_of(charT c, size_type pos = npos) const;
```

9 *Returns:* `find_last_of(basic_string(1, c), pos)`.

20.3.2.7.7 `basic_string::find_first_not_of` [string.find.first.not.of]

```
template<class T>
  size_type find_first_not_of(const T& t, size_type pos = 0) const;
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the lowest position `xpos`, if possible, such that both of the following conditions hold:

(1.1) — `pos <= xpos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), sv.at(I))` for no element `I` of the data referenced by `sv`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

3 *Throws:* Nothing unless the initialization of `sv` throws an exception.

4 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
size_type find_first_not_of(const basic_string& str, size_type pos = 0) const noexcept;
```

5 *Effects:* Equivalent to:

```
  return find_first_not_of(basic_string_view<charT, traits>(str), pos);
```

```
size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
```

6 *Returns:* `find_first_not_of(basic_string_view<charT, traits>(s, n), pos)`.

```
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
```

7 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

8 *Returns:* `find_first_not_of(basic_string_view<charT, traits>(s), pos)`.

```
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

9 *Returns:* `find_first_not_of(basic_string(1, c), pos)`.

20.3.2.7.8 `basic_string::find_last_not_of` [string.find.last.not.of]

```
template<class T>
  size_type find_last_not_of(const T& t, size_type pos = npos) const;
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the highest position `xpos`, if possible, such that both of the following conditions hold:

(1.1) — `xpos <= pos` and `xpos < size()`;

(1.2) — `traits::eq(at(xpos), sv.at(I))` for no element `I` of the data referenced by `sv`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

3 *Throws:* Nothing unless the initialization of `sv` throws an exception.

4 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
size_type find_last_not_of(const basic_string& str, size_type pos = npos) const noexcept;
```

5 *Effects:* Equivalent to:

```
  return find_last_not_of(basic_string_view<charT, traits>(str), pos);
```

```
size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
```

6 *Returns:* `find_last_not_of(basic_string_view<charT, traits>(s, n), pos)`.

```
size_type find_last_not_of(const charT* s, size_type pos = npos) const;
```

7 *Requires:* `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

8 *Returns:* `find_last_not_of(basic_string_view<charT, traits>(s), pos)`.

```
size_type find_last_not_of(charT c, size_type pos = npos) const;
```

9 *Returns:* `find_last_not_of(basic_string(1, c), pos)`.

20.3.2.7.9 `basic_string::substr` [string.substr]

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

1 *Throws:* `out_of_range` if `pos > size()`.

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

3 *Returns:* `basic_string(data()+pos, rlen)`.

20.3.2.7.10 `basic_string::compare` [string.compare]

```
template<class T>
  int compare(const T& t) const noexcept(see below);
```

1 *Effects:* Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then determines the effective length `rlen` of the strings to compare as the smaller of `size()` and `sv.size()`. The function then compares the two strings by calling `traits::compare(data(), sv.data(), rlen)`.

2 *Returns:* The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 58.

[*Drafting note*: Remove Table 58. – *end drafting note*]

Table 58 — `compare()` results

Condition	Return Value
`size() <  sv.size()`	`< 0`
`size() == sv.size()`	`0`
`size() >  sv.size()`	`> 0`

3 *Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.

4 *Throws:* Nothing unless the initialization of `sv` throws an exception.

5 *Constraints:*

(5.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(5.2) — `is_convertible_v<const T&, const charT*>` is `false`.

6 *Effects:* Equivalent to: `return basic_string_view<charT, traits>(*this).compare(t);`

[*Drafting note*: There are only two overloads of `basic_string_view::compare` taking one argument, and only the intended overload taking a `basic_string_view` is viable given the constraints above. – *end drafting note*]

7 *Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_convertible_v<const T&, basic_string_view<charT, traits>>`.

```
template<class T>
  int compare(size_type pos1, size_type n1, const T& t) const;
```

8 *Constraints:*

(8.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(8.2) — `is_convertible_v<const T&, const charT*>` is `false`.

9 *Effects:* Equivalent to:

```
~~{~~
  ~~basic_string_view<charT, traits> sv = t;~~
  return basic_string_view<charT, traits>(~~data(), size()~~*this).substr(pos1, n1).compare(~~sv~~t);
~~}~~
```

10 ~~*Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.~~

```
template<class T>
  int compare(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos) const;
```

11 *Constraints:*

(11.1) — `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and

(11.2) — `is_convertible_v<const T&, const charT*>` is `false`.

12 *Effects:* Equivalent to:

```
basic_string_view<charT, traits> s = *this, sv = t;
return ~~basic_string_view<charT, traits>(data(),~~
    ~~size())~~s.substr(pos1, n1).compare(sv.substr(pos2, n2));
```

13 ~~*Remarks:* This function shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and `is_convertible_v<const T&, const charT*>` is `false`.~~

```
int compare(const basic_string& str) const noexcept;
```

14 *Effects:* Equivalent to: `return compare(basic_string_view<charT, traits>(str));`

```
int compare(size_type pos1, size_type n1, const basic_string& str) const;
```

15 *Effects:* Equivalent to: `return compare(pos1, n1, basic_string_view<charT, traits>(str));`

```
int compare(size_type pos1, size_type n1, const basic_string& str,
            size_type pos2, size_type n2 = npos) const;
```

16 *Effects:* Equivalent to:

```
return compare(pos1, n1, basic_string_view<charT, traits>(str), pos2, n2);
```

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

17 ~~*Returns:*~~ *Effects:* Equivalent to: `return compare(basic_string_view<charT, traits>(s));`~~.~~

```
int compare(size_type pos, size_type n1, const charT* s) const;
```

18 ~~*Returns:* `basic_string(*this, pos, n1).compare(basic_string(s))`.~~

19 *Effects:* Equivalent to: `return compare(pos, n1, basic_string_view<charT, traits>(s));`

```
int compare(size_type pos, size_type n1, const charT* s, size_type n2) const;
```

20 ~~*Returns:* `basic_string(*this, pos, n1).compare(basic_string(s, n2))`.~~

21 *Effects:* Equivalent to: `return compare(pos, n1, basic_string_view<charT, traits>(s, n2));`

20.3.2.7.11 `basic_string::starts_with` [string.starts.with]

```
bool starts_with(basic_string_view<charT, traits> x) const noexcept;
bool starts_with(charT x) const noexcept;
bool starts_with(const charT* x) const;
```

1 *Effects:* Equivalent to:

```
  return basic_string_view<charT, traits>(data(), size()).starts_with(x);
```

20.3.2.7.12 `basic_string::ends_with` [string.ends.with]

```
bool ends_with(basic_string_view<charT, traits> x) const noexcept;
bool ends_with(charT x) const noexcept;
bool ends_with(const charT* x) const;
```

1 *Effects:* Equivalent to:

```
  return basic_string_view<charT, traits>(data(), size()).ends_with(x);
```

20.3.3 Non-member functions [string.nonmembers]

20.3.3.1 `operator+` [string.op+]

[*Drafting note*: The way these function templates are currently specified to handle allocators is haphazard at best. One effectively uses `select_on_container_copy_construction` on one of the operands; seven move constructs the allocator from an rvalue operand; four use a default constructed allocator. A complete fix to this problem is deferred to a separate paper, P1165R0. The changes below only fixes two obvious issues: a pointless extra copy and a bogus note. – *end drafting note*]

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              const basic_string<charT, traits, Allocator>& rhs);
```

1 *Returns:* `std::move(basic_string<charT, traits, Allocator>(lhs).append(rhs))`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              const basic_string<charT, traits, Allocator>& rhs);
```

2 *Returns:* `std::move(lhs.append(rhs))`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              basic_string<charT, traits, Allocator>&& rhs);
```

3 *Returns:* `std::move(rhs.insert(0, lhs))`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              basic_string<charT, traits, Allocator>&& rhs);
```

4 *Returns:* `std::move(lhs.append(rhs))`. [*Note:* ~~Or equivalently, `std::move(rhs.insert(0, lhs))`.~~ — *end note*]

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
```

5 *Returns:* `basic_string<charT, traits, Allocator>(lhs) + rhs`.

6 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const charT* lhs, basic_string<charT, traits, Allocator>&& rhs);
```

7 *Returns:* `std::move(rhs.insert(0, lhs))`.

8 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(charT lhs, const basic_string<charT, traits, Allocator>& rhs);
```

9 *Returns:* `basic_string<charT, traits, Allocator>(1, lhs) + rhs`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(charT lhs, basic_string<charT, traits, Allocator>&& rhs);
```

10 *Returns:* `std::move(rhs.insert(0, 1, lhs))`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);
```

11 *Returns:* `lhs + basic_string<charT, traits, Allocator>(rhs)`.

12 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs, const charT* rhs);
```

13 *Returns:* `std::move(lhs.append(rhs))`.

14 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs, charT rhs);
```

15 *Returns:* `lhs + basic_string<charT, traits, Allocator>(1, rhs)`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs, charT rhs);
```

16 *Returns:* `std::move(lhs.append(1, rhs))`.

20.3.3.2 Non-member comparison functions [string.comparison]

```
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
```

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

template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT, traits, Allocator>& lhs,
                  const basic_string<charT, traits, Allocator>& rhs) noexcept;
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);
```

1 *Effects:* Let *op* be the operator. Equivalent to:

```
return basic_string_view<charT, traits>(lhs) op basic_string_view<charT, traits>(rhs);
```

20.3.3.3 operator== [string.operator==]

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

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

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

2 *Returns:* `rhs == lhs`.

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

3 *Requires:* `rhs` points to an array of at least `traits::length(rhs) + 1` elements of `charT`.

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

20.3.3.4 operator!= [string.op!=]

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

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

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

2 *Returns:* `rhs != lhs`.

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

3 *Requires:* `rhs` points to an array of at least `traits::length(rhs) + 1` elements of `charT`.

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

20.3.3.5 `operator<` [string.op<]

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

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

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

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

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

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

20.3.3.6 `operator>` [string.op>]

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

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

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

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

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

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

20.3.3.7 `operator<=` [string.op<=]

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

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

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

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

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

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

20.3.3.8 `operator>=` [string.op>=]

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

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

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

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

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

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

20.3.3.9 `swap` [string.special]

```
template<class charT, class traits, class Allocator>
  void swap(basic_string<charT, traits, Allocator>& lhs,
            basic_string<charT, traits, Allocator>& rhs)
    noexcept(noexcept(lhs.swap(rhs)));
```

1 *Effects:* Equivalent to `lhs.swap(rhs)`.

20.3.3.10 Inserters and extractors [string.io]

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, basic_string<charT, traits, Allocator>& str);
```

1 *Effects:* Behaves as a formatted input function (27.7.4.2.1). After constructing a `sentry` object, if the sentry converts to `true`, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)`. If `is.width()` is greater than zero, the maximum number `n` of characters appended is `is.width()`; otherwise `n` is `str.max_size()`. Characters are extracted and appended until any of the following occurs:

(1.1) — *n* characters are stored;

(1.2) — end-of-file occurs on the input sequence;

(1.3) — `isspace(c, is.getloc())` is `true` for the next available input character *c*.

2 After the last character (if any) is extracted, `is.width(0)` is called and the `sentry` object is destroyed.

3 If the function extracts no characters, it calls `is.setstate(ios::failbit)`, which may throw `ios_base::failure` (27.5.5.4).

4 *Returns:* `is`.

```
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const basic_string<charT, traits, Allocator>& str);
```

5 *Effects:* Equivalent to: `return os << basic_string_view<charT, traits>(str);`

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str,
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str,
            charT delim);
```

6 *Effects:* Behaves as an unformatted input function (27.7.4.3), except that it does not affect the value returned by subsequent calls to `basic_istream<>::gcount()`. After constructing a `sentry` object, if the sentry converts to `true`, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)` until any of the following occurs:

(6.1) — end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate( ios_base::eofbit)`).

(6.2) — `traits::eq(c, delim)` for the next available input character *c* (in which case, *c* is extracted but not appended) (27.5.5.4)

(6.3) — `str.max_size()` characters are stored (in which case, the function calls `is.setstate(ios_base::failbit))` (27.5.5.4)

7 The conditions are tested in the order shown. In any case, after the last character is extracted, the `sentry` object is destroyed.

8 If the function extracts no characters, it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` (27.5.5.4).

9 *Returns:* `is`.

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str);
```

10 *Returns:* `getline(is, str, is.widen('\n'))`.

20.3.4 Numeric conversions [string.conversions]

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

1 *Effects:* The first two functions call `strtol(str.c_str(), ptr, base)`, and the last three functions call `strtoul(str.c_str(), ptr, base)`, `strtoll(str.c_str(), ptr, base)`, and `strtoull(str.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(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
long double stold(const string& str, size_t* idx = nullptr);
```

4 *Effects:* These functions call `strtof(str.c_str(), ptr)`, `strtod(str.c_str(), ptr)`, and `strtold( str.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.

```
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

7 *Returns:* Each function returns a `string` object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of `"%d"`, `"%u"`, `"%ld"`, `"%lu"`, `"%lld"`, `"%llu"`, `"%f"`, `"%f"`, or `"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size.

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

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

8 *Effects:* The first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.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`.

9 *Returns:* The converted result.

10 *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(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);
```

11 *Effects:* These functions call `wcstof(str.c_str(), ptr)`, `wcstod(str.c_str(), ptr)`, and `wcstold( str.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`.

12 *Returns:* The converted result.

13 *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`.

```
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);
```

14 *Returns:* Each function returns a `wstring` object holding the character representation of the value of its argument that would be generated by calling `swprintf(buf, buffsz, fmt, val)` with a format specifier of `L"%d"`, `L"%u"`, `L"%ld"`, `L"%lu"`, `L"%lld"`, `L"%llu"`, `L"%f"`, `L"%f"`, or `L"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size `buffsz`.

20.3.5 Hash support [basic.string.hash]

```
template<> struct hash<string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;
template<> struct hash<pmr::string>;
template<> struct hash<pmr::u16string>;
template<> struct hash<pmr::u32string>;
template<> struct hash<pmr::wstring>;
```

1 If `S` is one of these string types, `SV` is the corresponding string view type, and `s` is an object of type `S`, then `hash<S>()(s) == hash<SV>()(SV(s))`.

20.3.6 Suffix for `basic_string` literals [basic.string.literals]

```
string operator""s(const char* str, size_t len);
```

1 *Returns:* `string{str, len}`.

```
u16string operator""s(const char16_t* str, size_t len);
```

2 *Returns:* `u16string{str, len}`.

```
u32string operator""s(const char32_t* str, size_t len);
```

3 *Returns:* `u32string{str, len}`.

```
wstring operator""s(const wchar_t* str, size_t len);
```

4 *Returns:* `wstring{str, len}`.

5 [*Note:* The same suffix `s` is used for `chrono::duration` literals denoting seconds but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

20.4 String view classes [string.view]

1 The class template `basic_string_view` describes an object that can refer to a constant contiguous sequence of char-like (20.1) objects with the first element of the sequence at position zero. In the rest of this subclause, the type of the char-like objects held in a `basic_string_view` object is designated by `charT`.

2 [*Note:* The library provides implicit conversions from `const charT*` and `std::basic_string<charT, ...>` to `std::basic_string_view<charT, ...>` so that user code can accept just `std::basic_string_view<charT>` as a non-templated parameter wherever a sequence of characters is expected. User-defined types should define their own implicit conversions to `std::basic_string_view` in order to interoperate with these functions. — *end note*]

3 The complexity of `basic_string_view` member functions is $\mathscr{O}(1)$ unless otherwise specified.

20.4.1 Header <string_view> synopsis [string.view.synop]

```
namespace std {
  // 20.4.2, class template basic_string_view
  template<class charT, class traits = char_traits<charT>>
  class basic_string_view;

  // 20.4.3, non-member comparison functions
  template<class charT, class traits>
    constexpr bool operator==(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
  template<class charT, class traits>
    constexpr bool operator!=(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
  template<class charT, class traits>
    constexpr bool operator< (basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
  template<class charT, class traits>
    constexpr bool operator> (basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
  template<class charT, class traits>
    constexpr bool operator<=(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
  template<class charT, class traits>
    constexpr bool operator>=(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
  // see 20.4.3, sufficient additional overloads of comparison functions

  // 20.4.4, inserters and extractors
  template<class charT, class traits>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os,
                 basic_string_view<charT, traits> str);

  // basic_string_view typedef names
  using string_view    = basic_string_view<char>;
  using u16string_view = basic_string_view<char16_t>;
  using u32string_view = basic_string_view<char32_t>;
  using wstring_view   = basic_string_view<wchar_t>;
  // 20.4.5, hash support
  template<class T> struct hash;
  template<> struct hash<string_view>;
  template<> struct hash<u16string_view>;
  template<> struct hash<u32string_view>;
  template<> struct hash<wstring_view>;

  inline namespace literals {
  inline namespace string_view_literals {
    // 20.4.6, suffix for basic_string_view literals
    constexpr string_view    operator""sv(const char* str, size_t len) noexcept;
    constexpr u16string_view operator""sv(const char16_t* str, size_t len) noexcept;
    constexpr u32string_view operator""sv(const char32_t* str, size_t len) noexcept;
    constexpr wstring_view   operator""sv(const wchar_t* str, size_t len) noexcept;
  }
  }
}
```

1 The function templates defined in 19.2.2 and 22.7 are available when `<string_view>` is included.

20.4.2 Class template basic_string_view [string.view.template]

```
template<class charT, class traits = char_traits<charT>>
class basic_string_view {
public:
  // types
  using traits_type            = traits;
  using value_type             = charT;
  using pointer                = value_type*;
  using const_pointer          = const value_type*;
  using reference              = value_type&;
  using const_reference        = const value_type&;
  using const_iterator         = implementation-defined; // see 20.4.2.2
  using iterator               = const_iterator;[230]
  using const_reverse_iterator = reverse_iterator<const_iterator>;
  using reverse_iterator       = const_reverse_iterator;
  using size_type              = size_t;
  using difference_type        = ptrdiff_t;
  static constexpr size_type npos = size_type(-1);

  // 20.4.2.1, construction and assignment
  constexpr basic_string_view() noexcept;
  constexpr basic_string_view(const basic_string_view&) noexcept = default;
  constexpr basic_string_view& operator=(const basic_string_view&) noexcept = default;
  constexpr basic_string_view(const charT* str);
  constexpr basic_string_view(const charT* str, size_type len);

  // 20.4.2.2, iterator support
  constexpr const_iterator begin() const noexcept;
  constexpr const_iterator end() const noexcept;
  constexpr const_iterator cbegin() const noexcept;
  constexpr const_iterator cend() const noexcept;
  constexpr const_reverse_iterator rbegin() const noexcept;
  constexpr const_reverse_iterator rend() const noexcept;
  constexpr const_reverse_iterator crbegin() const noexcept;
  constexpr const_reverse_iterator crend() const noexcept;

  // 20.4.2.3, capacity
  constexpr size_type size() const noexcept;
  constexpr size_type length() const noexcept;
  constexpr size_type max_size() const noexcept;
  [[nodiscard]] constexpr bool empty() const noexcept;
```

[230] Because `basic_string_view` refers to a constant sequence, `iterator` and `const_iterator` are the same type.

```
  // 20.4.2.4, element access
  constexpr const_reference operator[](size_type pos) const;
  constexpr const_reference at(size_type pos) const;
  constexpr const_reference front() const;
  constexpr const_reference back() const;
  constexpr const_pointer data() const noexcept;

  // 20.4.2.5, modifiers
  constexpr void remove_prefix(size_type n);
  constexpr void remove_suffix(size_type n);
  constexpr void swap(basic_string_view& s) noexcept;

  // 20.4.2.6, string operations
  size_type copy(charT* s, size_type n, size_type pos = 0) const;

  constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;

  constexpr int compare(basic_string_view s) const noexcept;
  constexpr int compare(size_type pos1, size_type n1, basic_string_view s) const;
  constexpr int compare(size_type pos1, size_type n1, basic_string_view s,
                        size_type pos2, size_type n2) const;
  constexpr int compare(const charT* s) const;
  constexpr int compare(size_type pos1, size_type n1, const charT* s) const;
  constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

  constexpr bool starts_with(basic_string_view x) const noexcept;
  constexpr bool starts_with(charT x) const noexcept;
  constexpr bool starts_with(const charT* x) const;
  constexpr bool ends_with(basic_string_view x) const noexcept;
  constexpr bool ends_with(charT x) const noexcept;
  constexpr bool ends_with(const charT* x) const;

  // 20.4.2.7, searching
  constexpr size_type find(basic_string_view s, size_type pos = 0) const noexcept;
  constexpr size_type find(charT c, size_type pos = 0) const noexcept;
  constexpr size_type find(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find(const charT* s, size_type pos = 0) const;
  constexpr size_type rfind(basic_string_view s, size_type pos = npos) const noexcept;
  constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
  constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
  constexpr size_type rfind(const charT* s, size_type pos = npos) const;

  constexpr size_type find_first_of(basic_string_view s, size_type pos = 0) const noexcept;
  constexpr size_type find_first_of(charT c, size_type pos = 0) const noexcept;
  constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
  constexpr size_type find_last_of(basic_string_view s, size_type pos = npos) const noexcept;
  constexpr size_type find_last_of(charT c, size_type pos = npos) const noexcept;
  constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;
  constexpr size_type find_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
  constexpr size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
  constexpr size_type find_first_not_of(const charT* s, size_type pos,
                                        size_type n) const;
  constexpr size_type find_first_not_of(const charT* s, size_type pos = 0) const;
  constexpr size_type find_last_not_of(basic_string_view s,
                                       size_type pos = npos) const noexcept;
  constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
  constexpr size_type find_last_not_of(const charT* s, size_type pos,
                                       size_type n) const;
  constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;

private:
  const_pointer data_; // exposition only
```

```
    size_type size_;       // exposition only
  };
```

1 In every specialization `basic_string_view<charT, traits>`, the type `traits` shall satisfy the character traits requirements (20.2)~~, and the type `traits::char_type` shall name the same type as `charT`~~. [*Note:* The program is ill-formed if `traits::char_type` is not the same type as `charT`. — *end note*]

2 The type `iterator` satisfies the constexpr iterator requirements (22.2.1).

20.4.2.1 Construction and assignment [string.view.cons]

```
constexpr basic_string_view() noexcept;
```

1 *Effects:* Constructs an empty `basic_string_view`.

2 *Ensures:* `size_ == 0` and `data_ == nullptr`.

```
constexpr basic_string_view(const charT* str);
```

3 ~~*Requires:*~~ *Expects:* `[str, str + traits::length(str))` is a valid range.

4 *Effects:* Constructs a `basic_string_view`, initializing `data_` with `str` and `size_` with `traits::length(str)` ~~with the postconditions in Table 59~~.

[*Drafting note*: Remove Table 59. – *end drafting note*]

Table 59 — `basic_string_view(const charT*)` effects

Element	Value
`data_`	`str`
`size_`	`traits::length(str)`

5 *Complexity:* $\mathcal{O}$(`traits::length(str)`).

```
constexpr basic_string_view(const charT* str, size_type len);
```

6 ~~*Requires:*~~ *Expects:* `[str, str + len)` is a valid range.

7 *Effects:* Constructs a `basic_string_view`, initializing `data_` with `str` and `size_` with `len` ~~with the postconditions in Table 60~~.

[*Drafting note*: Remove Table 60. – *end drafting note*]

Table 60 — `basic_string_view(const charT*, size_type)` effects

Element	Value
`data_`	`str`
`size_`	`len`

20.4.2.2 Iterator support [string.view.iterators]

```
using const_iterator = implementation-defined;
```

1 A type that meets the requirements of a constant random access iterator (22.2.7) and of a contiguous iterator (22.2.1) whose `value_type` is the template parameter `charT`.

2 For a `basic_string_view str`, any operation that invalidates a pointer in the range `[str.data(), str.data() + str.size())` invalidates pointers, iterators, and references returned from `str`'s member functions.

3 All requirements on container iterators (21.2) apply to `basic_string_view::const_iterator` as well.

```
constexpr const_iterator begin() const noexcept;
constexpr const_iterator cbegin() const noexcept;
```

4 *Returns:* An iterator such that

(4.1) — if `!empty()`, `&*begin() == data_`,

(4.2) — otherwise, an unspecified value such that `[begin(), end())` is a valid range.

```
constexpr const_iterator end() const noexcept;
constexpr const_iterator cend() const noexcept;
```

5 *Returns:* `begin() + size()`.

```
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
```

6 *Returns:* `const_reverse_iterator(end())`.

```
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;
```

7 *Returns:* `const_reverse_iterator(begin())`.

20.4.2.3 Capacity [string.view.capacity]

```
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
```

1 *Returns:* `size_`.

```
constexpr size_type length() const noexcept;
```

2 *Returns:* `size_`.

```
constexpr size_type max_size() const noexcept;
```

3 *Returns:* The largest possible number of char-like objects that can be referred to by a `basic_string_view`.

```
[[nodiscard]] constexpr bool empty() const noexcept;
```

4 *Returns:* `size_ == 0`.

20.4.2.4 Element access [string.view.access]

```
constexpr const_reference operator[](size_type pos) const;
```

1 ~~*Requires:*~~ *Expects:* `pos < size()`.

2 *Returns:* `data_[pos]`.

3 *Throws:* Nothing.

4 [*Note:* Unlike `basic_string::operator[]`, `basic_string_view::operator[](size())` has undefined behavior instead of returning `charT()`. — *end note*]

```
constexpr const_reference at(size_type pos) const;
```

5 *Throws:* `out_of_range` if `pos >= size()`.

6 *Returns:* `data_[pos]`.

```
constexpr const_reference front() const;
```

7 ~~*Requires:*~~ *Expects:* `!empty()`.

8 *Returns:* `data_[0]`.

9 *Throws:* Nothing.

```
constexpr const_reference back() const;
```

10 ~~*Requires:*~~ *Expects:* `!empty()`.

11 *Returns:* `data_[size() - 1]`.

12 *Throws:* Nothing.

```
constexpr const_pointer data() const noexcept;
```

13 *Returns:* `data_`.

14 [*Note:* Unlike `basic_string::data()` and string literals, `data()` may return a pointer to a buffer that is not null-terminated. Therefore it is typically a mistake to pass `data()` to a function that takes just a `const charT*` and expects a null-terminated string. — *end note*]

20.4.2.5 Modifiers [string.view.modifiers]

```
constexpr void remove_prefix(size_type n);
```

1 ~~*Requires:*~~ *Expects:* `n <= size()`.

2 *Effects:* Equivalent to: `data_ += n; size_ -= n;`

```
constexpr void remove_suffix(size_type n);
```

3 ~~*Requires:*~~ *Expects:* `n <= size()`.

4 *Effects:* Equivalent to: `size_ -= n;`

```
constexpr void swap(basic_string_view& s) noexcept;
```

5 *Effects:* Exchanges the values of `*this` and `s`.

20.4.2.6 String operations [string.view.ops]

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

1 Let `rlen` be the smaller of `n` and `size() - pos`.

2 *Throws:* `out_of_range` if `pos > size()`.

3 ~~*Requires:*~~ *Expects:* `[s, s + rlen)` is a valid range.

4 *Effects:* Equivalent to `traits::copy(s, data() + pos, rlen)`.

5 *Returns:* `rlen`.

6 *Complexity:* $\mathcal{O}$(`rlen`).

```
constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;
```

7 Let `rlen` be the smaller of `n` and `size() - pos`.

8 *Throws:* `out_of_range` if `pos > size()`.

9 *Effects:* Determines `rlen`, the effective length of the string to reference.

10 *Returns:* `basic_string_view(data() + pos, rlen)`.

```
constexpr int compare(basic_string_view str) const noexcept;
```

11 Let `rlen` be the smaller of `size()` and `str.size()`.

12 *Effects:* Determines `rlen`, the effective length of the strings to compare. The function then compares the two strings by calling `traits::compare(data(), str.data(), rlen)`.

13 *Complexity:* $\mathcal{O}$(`rlen`).

14 *Returns:* The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 61.

Table 61 — `compare()` results

Condition	Return Value
`size() < str.size()`	`< 0`
`size() == str.size()`	`0`
`size() > str.size()`	`> 0`

```
constexpr int compare(size_type pos1, size_type n1, basic_string_view str) const;
```

15 *Effects:* Equivalent to: `return substr(pos1, n1).compare(str);`

```
constexpr int compare(size_type pos1, size_type n1, basic_string_view str,
                      size_type pos2, size_type n2) const;
```

16 *Effects:* Equivalent to: `return substr(pos1, n1).compare(str.substr(pos2, n2));`

```
constexpr int compare(const charT* s) const;
```

17 *Effects:* Equivalent to: `return compare(basic_string_view(s));`

```
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;
```

18 *Effects:* Equivalent to: `return substr(pos1, n1).compare(basic_string_view(s));`

```
constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;
```

19 *Effects:* Equivalent to: `return substr(pos1, n1).compare(basic_string_view(s, n2));`

```
constexpr bool starts_with(basic_string_view x) const noexcept;
```

20 *Effects:* Equivalent to: `return compare(0, npos, x) == 0;`

```
constexpr bool starts_with(charT x) const noexcept;
```

21 *Effects:* Equivalent to: `return starts_with(basic_string_view(&x, 1));`

```
constexpr bool starts_with(const charT* x) const;
```

22 *Effects:* Equivalent to: `return starts_with(basic_string_view(x));`

```
constexpr bool ends_with(basic_string_view x) const noexcept;
```

23 *Effects:* Equivalent to:

```
return size() >= x.size() && compare(size() - x.size(), npos, x) == 0;
```

```
constexpr bool ends_with(charT x) const noexcept;
```

24 *Effects:* Equivalent to: `return ends_with(basic_string_view(&x, 1));`

```
constexpr bool ends_with(const charT* x) const;
```

25 *Effects:* Equivalent to: `return ends_with(basic_string_view(x));`

20.4.2.7 Searching [string.view.find]

1 This subclause specifies the `basic_string_view` member functions named `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, and `find_last_not_of`.

2 Member functions in this subclause have complexity $\mathcal{O}$(`size() * str.size()`) at worst, although implementations should do better.

3 Let *F* be one of `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, and `find_last_not_of`. [*Drafting note*: Turn the next three paragraphs into a bulleted list: – *end drafting note*]

(3.1) — Each member function of the form

```
constexpr return-type F(const charT* s, size_type pos) const;
```

~~is~~has effects equivalent to: `return F(basic_string_view(s), pos);`

(3.2) — Each member function of the form

```
constexpr return-type F(const charT* s, size_type pos, size_type n) const;
```

~~is~~has effects equivalent to: `return F(basic_string_view(s, n), pos);`

(3.3) — Each member function of the form

```
constexpr return-type F(charT c, size_type pos) const noexcept;
```

~~is~~has effects equivalent to: `return F(basic_string_view(&c, 1), pos);`

```
constexpr size_type find(basic_string_view str, size_type pos = 0) const noexcept;
```

4 Let `xpos` be the lowest position, if possible, such that the following conditions hold:

(4.1) — `pos <= xpos`

(4.2) — `xpos + str.size() <= size()`

(4.3) — `traits::eq(at(xpos + I), str.at(I))` for all elements `I` of the string referenced by `str`.

5 *Effects:* Determines `xpos`.

6 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
constexpr size_type rfind(basic_string_view str, size_type pos = npos) const noexcept;
```

7 Let `xpos` be the highest position, if possible, such that the following conditions hold:

(7.1) — `xpos <= pos`

(7.2) — `xpos + str.size() <= size()`

(7.3) — `traits::eq(at(xpos + I), str.at(I))` for all elements `I` of the string referenced by `str`.

8 *Effects:* Determines `xpos`.

9 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
constexpr size_type find_first_of(basic_string_view str, size_type pos = 0) const noexcept;
```

10 Let `xpos` be the lowest position, if possible, such that the following conditions hold:

(10.1) — `pos <= xpos`

(10.2) — `xpos < size()`

(10.3) — `traits::eq(at(xpos), str.at(I))` for some element `I` of the string referenced by `str`.

11 *Effects:* Determines `xpos`.

12 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
constexpr size_type find_last_of(basic_string_view str, size_type pos = npos) const noexcept;
```

13 Let `xpos` be the highest position, if possible, such that the following conditions hold:

(13.1) — `xpos <= pos`

(13.2) — `xpos < size()`

(13.3) — `traits::eq(at(xpos), str.at(I))` for some element `I` of the string referenced by `str`.

14 *Effects:* Determines `xpos`.

15 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
constexpr size_type find_first_not_of(basic_string_view str, size_type pos = 0) const noexcept;
```

16 Let `xpos` be the lowest position, if possible, such that the following conditions hold:

(16.1) — `pos <= xpos`

(16.2) — `xpos < size()`

(16.3) — `traits::eq(at(xpos), str.at(I))` for no element `I` of the string referenced by `str`.

17 *Effects:* Determines `xpos`.

18 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```
constexpr size_type find_last_not_of(basic_string_view str, size_type pos = npos) const noexcept;
```

19 Let `xpos` be the highest position, if possible, such that the following conditions hold:

(19.1) — `xpos <= pos`

(19.2) — `xpos < size()`

(19.3) — `traits::eq(at(xpos), str.at(I))` for no element `I` of the string referenced by `str`.

20 *Effects:* Determines `xpos`.

21 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

20.4.3 Non-member comparison functions [string.view.comparison]

1 Let `S` be `basic_string_view<charT, traits>`, and `sv` be an instance of `S`. Implementations shall provide sufficient additional overloads marked `constexpr` and `noexcept` so that an object `t` with an implicit conversion to `S` can be compared according to Table 62.

[*Example:* A sample conforming implementation for `operator==` would be:

```
~~template<class T> using __identity = decay_t<T>;~~
template<class charT, class traits>
  constexpr bool operator==(basic_string_view<charT, traits> lhs,
                            basic_string_view<charT, traits> rhs) noexcept {
    return lhs.compare(rhs) == 0;
  }
```

Table 62 — Additional `basic_string_view` comparison overloads

Expression	Equivalent to
`t == sv`	`S(t) == sv`
`sv == t`	`sv == S(t)`
`t != sv`	`S(t) != sv`
`sv != t`	`sv != S(t)`
`t < sv`	`S(t) < sv`
`sv < t`	`sv < S(t)`
`t > sv`	`S(t) > sv`
`sv > t`	`sv > S(t)`
`t <= sv`	`S(t) <= sv`
`sv <= t`	`sv <= S(t)`
`t >= sv`	`S(t) >= sv`
`sv >= t`	`sv >= S(t)`

```
template<class charT, class traits>
  constexpr bool operator==(basic_string_view<charT, traits> lhs,
                            ~~__identity~~type_identity_t<basic_string_view<charT, traits>> rhs) noexcept {
    return lhs.compare(rhs) == 0;
  }
template<class charT, class traits>
  constexpr bool operator==(~~__identity~~type_identity_t<basic_string_view<charT, traits>> lhs,
                            basic_string_view<charT, traits> rhs) noexcept {
    return lhs.compare(rhs) == 0;
  }
```

— *end example*]

```
template<class charT, class traits>
  constexpr bool operator==(basic_string_view<charT, traits> lhs,
                            basic_string_view<charT, traits> rhs) noexcept;
```

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

```
template<class charT, class traits>
  constexpr bool operator!=(basic_string_view<charT, traits> lhs,
                            basic_string_view<charT, traits> rhs) noexcept;
```

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

```
template<class charT, class traits>
  constexpr bool operator<(basic_string_view<charT, traits> lhs,
                           basic_string_view<charT, traits> rhs) noexcept;
```

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

```
template<class charT, class traits>
  constexpr bool operator>(basic_string_view<charT, traits> lhs,
                           basic_string_view<charT, traits> rhs) noexcept;
```

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

```
template<class charT, class traits>
  constexpr bool operator<=(basic_string_view<charT, traits> lhs,
                            basic_string_view<charT, traits> rhs) noexcept;
```

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

```
template<class charT, class traits>
  constexpr bool operator>=(basic_string_view<charT, traits> lhs,
                            basic_string_view<charT, traits> rhs) noexcept;
```

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

20.4.4 Inserters and extractors [string.view.io]

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, basic_string_view<charT, traits> str);
```

1 *Effects:* Behaves as a formatted output function (27.7.5.2.1) of `os`. Forms a character sequence `seq`, initially consisting of the elements defined by the range `[str.begin(), str.end())`. Determines padding for `seq` as described in 27.7.5.2.1. Then inserts `seq` as if by calling `os.rdbuf()->sputn(seq, n)`, where `n` is the larger of `os.width()` and `str.size()`; then calls `os.width(0)`.

2 *Returns:* `os`

20.4.5 Hash support [string.view.hash]

```
template<> struct hash<string_view>;
template<> struct hash<u16string_view>;
template<> struct hash<u32string_view>;
template<> struct hash<wstring_view>;
```

1 The specialization is enabled (19.14.16). [*Note:* The hash value of a string view object is equal to the hash value of the corresponding string object (20.3.5). — *end note*]

20.4.6 Suffix for `basic_string_view` literals [string.view.literals]

```
constexpr string_view operator""sv(const char* str, size_t len) noexcept;
```

1 *Returns:* `string_view{str, len}`.

```
constexpr u16string_view operator""sv(const char16_t* str, size_t len) noexcept;
```

2 *Returns:* `u16string_view{str, len}`.

```
constexpr u32string_view operator""sv(const char32_t* str, size_t len) noexcept;
```

3 *Returns:* `u32string_view{str, len}`.

```
constexpr wstring_view operator""sv(const wchar_t* str, size_t len) noexcept;
```

4 *Returns:* `wstring_view{str, len}`.

27 Input/output library [input.output]

27.1 General [input.output.general]

1 This Clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 104.

Table 104 — Input/output library summary

Subclause		Header(s)
27.2	Requirements	
27.3	Forward declarations	`<iosfwd>`
27.4	Standard iostream objects	`<iostream>`
27.5	Iostreams base classes	`<ios>`
27.6	Stream buffers	`<streambuf>`

27.7	Formatting and manipulators	`<istream>`
		`<ostream>`
		`<iomanip>`

27.8	String streams	`<sstream>`
27.9	File streams	`<fstream>`
27.10	Synchronized output streams	`<syncstream>`
27.11	File systems	`<filesystem>`

27.12	C library files	`<cstdio>`
		`<cinttypes>`

3 Figure 7 illustrates relationships among various types described in this clause. A line from **A** to **B** indicates that **A** is an alias (e.g., a typedef) for **B** or that **A** is defined in terms of **B**.

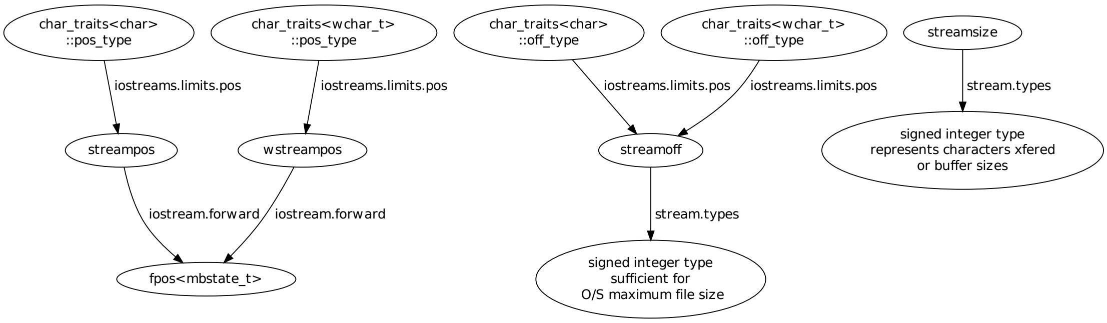
char_traits<char> ::pos_type
char_traits<wchar_t> ::pos_type
char_traits<char> ::off_type
char_traits<wchar_t> ::off_type
streamsize
iostreams.limits.pos
iostreams.limits.pos
iostreams.limits.pos
iostreams.limits.pos
stream.types
streampos
wstreampos
streamoff
signed integer type represents characters xfered or buffer sizes
iostream.forward
iostream.forward
stream.types
fpos<mbstate_t>
signed integer type sufficient for O/S maximum file size

Figure 7 — Stream position, offset, and size types [non-normative]

27.2 Iostreams requirements [iostreams.requirements]

27.2.1 Imbue limitations [iostream.limits.imbue]

1 No function described in Clause 27 except for `ios_base::imbue` and `basic_filebuf::pubimbue` causes any instance of `basic_ios::imbue` or `basic_streambuf::imbue` to be called. If any user function called from a function declared in Clause 27 or as an overriding virtual function of any class declared in Clause 27 calls `imbue`, the behavior is undefined.

27.2.2 Positioning type limitations [iostreams.limits.pos]

1 The classes of Clause 27 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos_type and traits::off_type are other types is implementation-defined.

2 In the classes of Clause 27, a template parameter with name charT represents a member of the set of types containing char, wchar_t, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated.

27.2.3 Thread safety [iostreams.threadsafety]

1 Concurrent access to a stream object (27.8, 27.9), stream buffer object (27.6), or C Library stream (27.12) by multiple threads may result in a data race (6.8.2) unless otherwise specified (27.4). [*Note:* Data races result in undefined behavior (6.8.2). — *end note*]

2 If one thread makes a library call *a* that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call *b* such that this does not result in a data race, then *a*'s write synchronizes with *b*'s read.

27.3 Forward declarations [iostream.forward]

27.3.1 Header <iosfwd> synopsis [iosfwd.syn]

```
namespace std {
  template<class charT> class char_traits;
  template<> class char_traits<char>;
  template<> class char_traits<char16_t>;
  template<> class char_traits<char32_t>;
  template<> class char_traits<wchar_t>;

  template<class T> class allocator;

  template<class charT, class traits = char_traits<charT>>
    class basic_ios;
  template<class charT, class traits = char_traits<charT>>
    class basic_streambuf;
  template<class charT, class traits = char_traits<charT>>
    class basic_istream;
  template<class charT, class traits = char_traits<charT>>
    class basic_ostream;
  template<class charT, class traits = char_traits<charT>>
    class basic_iostream;

  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_stringbuf;
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_istringstream;
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_ostringstream;
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_stringstream;

  template<class charT, class traits = char_traits<charT>>
    class basic_filebuf;
  template<class charT, class traits = char_traits<charT>>
    class basic_ifstream;
  template<class charT, class traits = char_traits<charT>>
    class basic_ofstream;
  template<class charT, class traits = char_traits<charT>>
    class basic_fstream;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
      class basic_syncbuf;
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
      class basic_osyncstream;

    template<class charT, class traits = char_traits<charT>>
      class istreambuf_iterator;
    template<class charT, class traits = char_traits<charT>>
      class ostreambuf_iterator;

    using ios  = basic_ios<char>;
    using wios = basic_ios<wchar_t>;

    using streambuf = basic_streambuf<char>;
    using istream   = basic_istream<char>;
    using ostream   = basic_ostream<char>;
    using iostream  = basic_iostream<char>;

    using stringbuf     = basic_stringbuf<char>;
    using istringstream = basic_istringstream<char>;
    using ostringstream = basic_ostringstream<char>;
    using stringstream  = basic_stringstream<char>;

    using filebuf  = basic_filebuf<char>;
    using ifstream = basic_ifstream<char>;
    using ofstream = basic_ofstream<char>;
    using fstream  = basic_fstream<char>;

    using syncbuf = basic_syncbuf<char>;
    using osyncstream = basic_osyncstream<char>;

    using wstreambuf = basic_streambuf<wchar_t>;
    using wistream   = basic_istream<wchar_t>;
    using wostream   = basic_ostream<wchar_t>;
    using wiostream  = basic_iostream<wchar_t>;

    using wstringbuf     = basic_stringbuf<wchar_t>;
    using wistringstream = basic_istringstream<wchar_t>;
    using wostringstream = basic_ostringstream<wchar_t>;
    using wstringstream  = basic_stringstream<wchar_t>;

    using wfilebuf  = basic_filebuf<wchar_t>;
    using wifstream = basic_ifstream<wchar_t>;
    using wofstream = basic_ofstream<wchar_t>;
    using wfstream  = basic_fstream<wchar_t>;

    using wsyncbuf = basic_syncbuf<wchar_t>;
    using wosyncstream = basic_osyncstream<wchar_t>;

    template<class state> class fpos;
    using streampos  = fpos<char_traits<char>::state_type>;
    using wstreampos = fpos<char_traits<wchar_t>::state_type>;
    using u16streampos = fpos<char_traits<char16_t>::state_type>;
    using u32streampos = fpos<char_traits<char32_t>::state_type>;
  }
```

1 Default template arguments are described as appearing both in `<iosfwd>` and in the synopsis of other headers but it is well-formed to include both `<iosfwd>` and one or more of the other headers.[289]

[289] It is the implementation's responsibility to implement headers so that including `<iosfwd>` and other headers does not violate the rules about multiple occurrences of default arguments.

27.3.2 Overview [iostream.forward.overview]

1 The class template specialization `basic_ios<charT, traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iostream` is a class template derived from both `basic_istream<charT, traits>` and `basic_ostream<charT, traits>`.

2 The class template specialization `basic_streambuf<charT, traits>` serves as a base class for class templates `basic_stringbuf`~~ and~~, `basic_filebuf`, and `basic_syncbuf`.

3 The class template specialization `basic_istream<charT, traits>` serves as a base class for class templates `basic_istringstream` and `basic_ifstream`.

4 The class template specialization `basic_ostream<charT, traits>` serves as a base class for class templates `basic_ostringstream`~~ and~~, `basic_ofstream`, and `basic_osyncstream`.

5 The class template specialization `basic_iostream<charT, traits>` serves as a base class for class templates `basic_stringstream` and `basic_fstream`.

6 [*Note:* For each of the class templates above, the program is ill-formed if `traits::char_type` is not the same type as `charT` (20.2). — *end note*]

7 Other *typedef-names* define instances of class templates specialized for `char` or `wchar_t` types.

8 Specializations of the class template `fpos` are used for specifying file position information.

9 The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively.

10 [*Note:* This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be

```
template<class stateT> class fpos { /* ... */ };       // depends on nothing
using _STATE = /* ... */ ;                 // implementation private declaration of stateT

using streampos = fpos<_STATE>;

template<> struct char_traits<char> {
  using pos_type = streampos;
}
```

— *end note*]

[*Drafting note*: Given that we defined `char_traits<char>::state_type` to be exactly `mbstate_t`, the example seems pointless. – *end drafting note*]