Project: Programming Language C++, Library Evolution Working Group
Document number: P0123R2
Date: 2016-05-26
Reply-to: Neil MacIntosh neilmac@microsoft.com

string_span: bounds-safe views for sequences of characters

Contents

Changelog

Changes from R0

- Changed title to reflect design changes.
- Renamed the proposed type from *basic_string_view* to *basic_string_span* following feedback from LEWG at the Kona meeting.
- Changed *basic_string_span* from a type alias for *span* to be a template class of its own, in order to be able to specify additional, string-specific construction and comparison behaviors.
- Added suggested overload for *to_string()*.
- Separated out convenience type aliases for fixed- and dynamic- size string spans.

Changes from R1

- Added *difference_type* typedef to *span* to better support use in template functions.
- Removed *const_iterator begin const()* and *const_iterator end const ()* members of *span* based on LEWG feedback.
- Removed the deletion of constructors that take rvalue-references based on LEWG feedback.
- Added support for construction from *const Container&*.
- Tightened overload resolution requirements for construction from *const* container---type references so that constructing a non-*const basic_string_span* from a *const* container reference is not possible.

Introduction

This paper presents a design for *basic_string_span* (similar to the *basic_string_view* proposed in N3762 [1]) that would have an interface consistent with the *span* type described in P0122 [2]. Doing so improves the generality of the *basic_string_span* type and allows it to offer bounds-safety guarantees like *span*.

It is worth noting that the *basic_string_span* type presented here largely matches the interface of the *span* type proposed in P0122 [2].

Motivation and Scope

basic_string_span is a "vocabulary type" that is proposed for inclusion in the standard library. It can be widely used in C++ programs, as a replacement for passing const *basic_string* objects or zero-terminated character arrays. The *basic_string_span* design supports high performance and bounds-safe access to contiguous sequences of characters. This type would also improve modularity, composability, and reuse by decoupling accesses to string data from the specific container types used to store that data.

It is desirable that the interface offered by *basic_string_span* is harmonized with *span*, given the similarity between the purposes and functionality of the two types. This has the positive benefit of also reducing the number of interfaces that need to be learned by C++ programmers who want to perform bounds-safe, high-performance access to sequences – whether they are sequences of characters, or objects.

basic_string_span is presented as complementary type to the *basic_string_view* of N3762 [1]. Each fulfills an important but different aim. *basic_string_view* focuses on compatibility with the interface of

basic_string and support for null-termination. *basic_string_span* is explicitly not null-terminated, and provides a simpler interface that is closer to a "view over a sequence of characters" model.

Impact on the Standard

basic_string_span is a pure library extension. It does not require any changes or extensions to the core language.

As described in the Design Decisions section, it would be convenient to overload *to_string()* for *basic_string_span* parameters.

basic_string_span as presented here has been implemented in standard C++ and successfully used within a commercial static analysis tool for C++ as well as commercial office productivity software. An open source reference implementation is available at https://github.com/Microsoft/GSL [3].

Design Decisions

Interface and naming similarity to *span*

The concept of a string is essentially a contiguous sequence of characters. A *span* is a vocabulary type that encapsulates access to a contiguous sequence of objects. A *basic_string_span* is a vocabulary type that encapsulates access to a contiguous sequence of characters. It is clear that at least conceptually (even if not necessarily in implementation) *basic_string_span* and *span* are related types.

In order to allow code that deals with contiguous sequences to look and behave uniformly (whether they are of characters or some other element type), *basic_string_span* consciously copies the interface of *span* (with some minor changes to construction and comparison). This design decision reduces the "surface area" that a C++ programmer must learn and remember to use each of these vocabulary types.

This, in turn, makes the requirement on string containers that *basic_string_span* can be a view over as simple as possible. The proposed form of *basic_string_span* can be used over a wide variety of string containers - such as *CString*, *const char**, *BSTR*, *QString* or any of the other myriad of string types that are commonly used in C++ today. That capacity – to decouple functions from the details of the string type being used – is a significant benefit that *basic_string_span* can bring to C++ programmers.

Removing string-specific member functions

String-specific member functions (such as the overloads of *find()* on *basic_string*) are not offered on *basic_string_span*. This design decision follows the general approach of the standard library, which is to separate algorithms such as *find_first()* from the containers or views they operate over, by making them free functions. The lack of these string-specific member functions also makes it clearer to users of *string_view* objects that they cannot assume they are operating over a *basic_string*. *basic_string_span* is a type that should decouple users from the details of underlying string container types.

Zero termination

Historical conventions around zero-terminating the sequence of characters that form a string reflect implementation choices, rather than a fundamental aspect of the string concept. *basic_string_span* is completely agnostic of zero-termination requirements/promises in the string data it contains. This allows the view to be broadly adopted, as it can view over character sequences that are not zero-terminated, as well as those that happen to be.

basic_string_span does correctly initialize from string zero-terminated string constants (dropping the terminating zero from the view of the constructed *basic_string_span* object), for the sake of convenience.

Value Type Semantics

basic_string_span is designed as a value type – it is expected to be cheap to construct, copy, move, and use. Users are encouraged to use it as a pass-by-value parameter type wherever they would have passed a pointer to character by value or *basic_string* by reference.

Conceptually, *basic_string_span* is simply a pointer to some storage and a count of the elements accessible via that pointer. Those two values within a span can only be set via construction or assignment (i.e. all member functions other than constructors and assignment operators are *const*). This property makes it easy for users to reason about the values of a span through the course of a function body.

These value type characteristics also help provide compiler implementations with considerable scope for optimizing the use of *basic_string_span* within programs. For example, *basic_string_span* has a trivial destructor, so common ABI conventions allow it to be passed in registers.

Character traits

Although this proposal does not include character traits support in the proposed definition of *basic_string_span*, it is not prejudiced against such inclusion. It would certainly be possible to add an additional template parameter to the type alias if free functions that wanted to operate over *basic_string_span* would find a character traits template type argument helpful.

Static or dynamic length

basic_string_span objects are capable of being declared as either having a static-size (fixed at compile-time) or dynamic-size (provided at runtime). Conversions between the two varieties are allowed with limitations to ensure bounds-safety is always preserved. These conversions follow the same rules as for *span*. Fixed-size *basic_string_span* can be implemented with no size overhead when compared to passing a single pointer.

Mutable or const elements

basic_string_span as a type-alias can also support either read-only or mutable access to a sequence. To access read-only data, the user can declare a *basic_string_span<const char>* (for example), and access to mutable data would use a *basic_string_span<char>*. While it is acknowledged that the majority of *basic_string_span* usage would tend to be for read-only access, there are still uses for mutable access to an existing string. As an example, some programs deal with fixed-size strings for storage or communication protocols, and find it convenient to pass such a fixed-size string to functions that set or modify the elements prior to transmission or serialization.

Range-checking and bounds-safety

All accesses to the data encapsulated by a *basic_string_span* are conceptually range-checked to ensure they remain within the bounds of the *basic_string_span*. What actually happens as the result of a failure to meet *basic_string_span*'s bounds-safety constraints at runtime is undefined behavior. However, it should be considered effectively fatal to a program's ability to continue reliable execution. This is a

critical aspect of *basic_string_span*'s design, and allows users to rely on the guarantee that as long as a sequence is accessed via a correctly initialized *basic_string_span*, then its bounds cannot be overrun.

As an example, in the current reference implementation, violating a range-check results by default in a call to *terminate()* but can also be configured via build-time mechanisms to continue execution (albeit with undefined behavior from that point on).

Conversion between fixed-size and dynamic-size *basic_string_span* objects is allowed, but with strict constraints that ensure bounds-safety is always preserved. At least two of these cases can be checked statically by leveraging the type system. In each case, the following rules assume the element types of the *basic_string_span* objects are compatible for assignment.

1. A fixed-size *basic_string_span* may be constructed or assigned from another fixed-size *basic_string_span* of equal length.
2. A dynamic-size *basic_string_span* may always be constructed or assigned from a fixed-size *basic_string_span*.
3. A fixed-size *basic_string_span* may always be constructed or assigned from a dynamic-size *basic_string_span*. Undefined behavior will result if the construction or assignment is not bounds-safe. In the reference implementation, for example, this is achieved via a runtime check that results in *terminate()* on failure.

Construction

Construction is one place where *basic_string_span* differs from its "relative" *span*.

To simplify use of *basic_string_pan* as a simple parameter, *basic_string_span* offers a number of constructors for common string container types that store contiguous sequences of elements. As *basic_string_span* does not zero-terminate string data, it does a little extra work in some cases to avoid inadvertently including a terminator.

Most of the constructors for *basic_string_span* are equivalent to the constructors for span. However, the key differences are in the following summarized extract from the specification:

```
template <size_t N>
  constexpr basic_string_span(element_type(&arr)[N]);

template <size_t N>
  constexpr basic_string_span(array<remove_const_t<element_type>, N>& arr);

template <size_t N>
  constexpr basic_string_span(const array<remove_const_t<element_type>, N>&
arr);
```

These three constructors check for a terminating zero in the character sequence provided as input. If one is found, then it is not included in the *basic_string_span* being constructed. This allows code such as the following to behave in a least-surprise fashion:

```
// ss.size() returns 5. It has been constructed without the terminating ‘\0’
basic_string_span<const char, dynamic_extent> ss = “Hello”;
```

There are also specific constructors that take *basic_string*, for convenience.

Convenience aliases

There are a number of “convenience” aliases provided for the various combinations of character types, *const*-ness, fixed- and dynamic-size that are commonly useful with *basic_string_span*:

```
using string_span = basic_string_span<char, dynamic_extent>;

using cstring_span = basic_string_span<const char, dynamic_extent>;

using wstring_span = basic_string_span<wchar_t, dynamic_extent>;

using cwstring_span = basic_string_span<const wchar_t, dynamic_extent>;

template<ptrdiff_t Extent>
  using fixed_string_span = basic_string_span<char, Extent>;

template<ptrdiff_t Extent>
  using fixed_cstring_span = basic_string_span<const char, Extent>;

template<size_t Extent>
  using fixed_wstring_span = basic_string_span<wchar_t, Extent>;

template<size_t Extent>
  using fixed_cwstring_span = basic_string_span<const wchar_t, Extent>;
```

to_string conversion

In usage, it is often convenient to take a *basic_string_span* parameter, but then wish to copy the sequence it views into a new *basic_string* container for further processing and storage. To support this scenario, it is proposed to overload the existing *to_string()* free function in the standard library so that it will construct a *basic_string* from a *basic_string_span*.

```
template<class CharT, ptrdiff_t Extent>
  basic_string<remove_const_t<CharT>> to_string(basic_string_span<CharT,
Extent> s);
```

Proposed Wording Changes

The following proposed wording changes against the working draft of the standard are relative to N4567 [4].

In these changes,

Yellow highlight is used to indicate modified text or sections.

Red highlight is used to indicate deleted text.

Green highlight is used to indicate newly added text.

17.6.1.2 Headers [headers]

2 The C++ standard library provides 54 *C++ library headers*, as shown in Table 14.

Table 14 – C++ library headers

<algorithm>	<fstream>	<list>	<regex>	<thread>
<array>	<functional>	<locale>	<scoped_allocator>	<tuple>
<atomic>	<future>	<map>	<set>	<type_traits>
<bitset>	<initializer_list>	<memory>	<sstream>	<typeindex>
<chrono>	<iomanip>	<mutex>	<stack>	<typeinfo>
<codecvt>	<ios>	<new>	<stdexcept>	<unordered_map>
<complex>	<iosfwd>	<numeric>	<streambuf>	<unordered_set>
<condition_variable>	<iostream>	<ostream>	<string>	<utility>
<deque>	<istream>	<queue>	<string_span>	<valarray>
<exception>	<iterator>	<random>	<strstream>	<vector>
<forward_list>	<limits>	<ratio>	<system_error>	

23 Contains library [containers]

23.1 General [containers.general]

Edit paragraph 2:

The following subclauses describe container requirements, and components for sequence containers, associative containers, and views as summarized in Table 94.

Add an extra row to Table 94:

Table 94 – Containers library summary

Subclause	Header(s)
23.7 Views	 <string_span>

23 Containers library [containers]

23.1 General [containers.general]

2 The following subclauses describe container requirements, and components for sequence containers, associative containers, and views as summarized in Table 94.

Table 94 – Containers library summary

Subclause	Header(s)
23.2 Requirements	
23.3 Sequence containers	<array>

	<deque> <forward_list> <list> <vector>
23.4 Associative containers	<map> <set>
23.5 Unordered associative containers	<unordered_map> <unordered_set>
23.6 Container adaptors	<queue> <stack>
23.7 Views	<string_span>

23.7 Views [views]

23.7.1 General [views.general]

1 The header <string_span> defines the view string_span. A span is a view over a contiguous sequence of characters, the storage of which is owned by some other object.

Header <string_span> synopsis

```
namespace std {

// [views.constants], constants
constexpr ptrdiff_t dynamic_extent = -1;

// [basic_string_span], class template basic_string_span
template <class CharT, ptrdiff_t Extent = dynamic_extent>
class basic_string_span;

// [basic_string_span.comparison], basic_string_span comparison operators
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator==(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator!=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator<(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator<=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator>(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff t Extent>
  constexpr bool operator>=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

```
// [basic_string_span.basic_string], basic_string_span to basic_string
conversion
template<class CharT, ptrdiff_t Extent>
  basic_string<remove_const_t<CharT>> to_string(basic_string_span<CharT,
Extent> s);

// [string_span.aliases], convenience aliases for basic_string_span
using string_span = basic_string_span<char, dynamic_extent>;

using cstring_span = basic_string_span<const char, dynamic_extent>;

using wstring_span = basic_string_span<wchar_t, dynamic_extent>;
q`
using cwstring_span = basic_string_span<const wchar_t, dynamic_extent>;

template<ptrdiff_t Extent>
  using fixed_string_span = basic_string_span<char, Extent>;

template<ptrdiff_t Extent>
  using fixed_cstring_span = basic_string_span<const char, Extent>;

template<ptrdiff_t Extent>
  using fixed_wstring_span = basic_string_span<wchar_t, Extent>;

template<ptrdiff_t Extent>
  using fixed_cwstring_span = basic_string_span<const wchar_t, Extent>;

} // namespace std
```

23.7.2 Class template basic_string_span [basic_string_span]

1 A `basic_string_span` is a view over a contiguous sequence of characters, the storage of which is owned by some other object.

2 `CharT` is required to be either a narrow character type (3.9.1/1) or `wchar_t` (3.9.1/5).

2 Throughout this section, whenever a requirement fails to be met, the result is considered undefined behavior. It may – for example – cause immediate termination via a call to `terminate()`, or cause an exception to be thrown.

3 The iterators for `basic_string_span` are all random access iterators and contiguous iterators.

4 For a `basic_string_span<const T>`, the `iterator` and `const_iterator` types are allowed to be synonyms.

```
namespace std {

// A view over a contiguous, single-dimension sequence of characters
template <class CharT, ptrdiff_t Extent = dynamic_extent>
class basic_string_span {
public:
  // constants and types
  using element_type = CharT;
  using index_type = ptrdiff_t;
  using difference_type = ptrdiff_t;
```

```
  using pointer = element_type*;
  using reference = element_type&;
  using iterator = /* implementation-defined */;
  using reverse_iterator = reverse_iterator<iterator>;

  constexpr static index_type extent = Extent;

   // [basic_string_span.cons], basic_string_span constructors, copy,
assignment and destructor
  constexpr basic_string_span();
  constexpr basic_string_span(nullptr_t);
   constexpr basic_string_span(pointer ptr);
   constexpr basic_string_span(pointer ptr, index_type count);
  constexpr basic_string_span(pointer firstElem, pointer lastElem);
  template <size_t N>
    constexpr basic_string_span(element_type (&arr)[N]);
  template <size_t N>
     constexpr basic_string_span(array<remove_const_t<element_type>, N>&
arr);
  template <size_t N>
    constexpr basic_string_span(const array<remove_const_t<element_type>,
N>& arr);
  template <class Container>
    constexpr basic_string_span(Container& cont);
  template <class Container>
    basic_string_span(const Container& cont);
  constexpr basic_string_span(const basic_string_span&) noexcept =
default;
  constexpr basic_string_span(basic_string_span&&) noexcept = default;
  template <class OtherCharT, ptrdiff_t OtherExtent>
    constexpr basic_string_span(const basic_string_span<OtherCharT,
OtherExtent>& other);
  template <class OtherElementType, ptrdiff_t OtherExtent>
    constexpr basic_string_span(basic_string_span<OtherElementType,
OtherExtent>&& other);
    constexpr
basic_string_span(basic_string<remove_const_t<element_type>>& s);
    constexpr basic_string_span(const
basic_string<remove_const_t<element_type>>& s);
  ~basic_string_span() noexcept = default;
  basic_string_span& operator=(const basic_string_span& other) noexcept =
default;
  basic_string_span& operator=(basic_string_span&& other) noexcept =
default;

  // [basic_string_span.sub], basic_string_span subviews
  template <ptrdiff_t Count>
    constexpr basic_string_span<element_type, Count> first() const;
  template <ptrdiff_t Count>
    constexpr basic_string_span<element_type, Count> last() const;
  template <ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
    constexpr basic_string_span<element_type, Count> subspan() const;
  constexpr basic_string_span<element_type, dynamic_extent>
first(index_type count) const;
  constexpr basic_string_span<element_type, dynamic_extent>
last(index_type count) const;
```

```
  constexpr basic_string_span<element_type, dynamic_extent>
subspan(index_type offset, index_type count = dynamic_extent) const;

  // [basic_string_span.obs], basic_string_span observers
  constexpr index_type length() const noexcept;
  constexpr index_type size() const noexcept;
  constexpr index_type length_bytes() const noexcept;
  constexpr index_type size_bytes() const noexcept;
  constexpr bool empty() const noexcept;

  // [basic_string_span.elem], basic_string_span element access
  constexpr reference operator[](index_type idx) const;
  constexpr reference operator()(index_type idx) const;
  constexpr pointer data() const noexcept;

  // [basic_string_span.iter], basic_string_span iterator support
  iterator begin() const noexcept;
  iterator end() const noexcept;

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

private:
  // exposition only
  pointer data_;
  index_type size_;
};

// [basic_string_span.comparison], basic_string_span comparison operators
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator==(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator!=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator<(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator<=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator>(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

template <class CharT, ptrdiff_t Extent>
  constexpr bool operator>=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;

// [basic_string_span.basic_string], basic_string_span to basic_string
conversion
template<class CharT, ptrdiff_t Extent>
```

```
  basic_string<remove_const_t<CharT>> to_string(basic_string_span<CharT,
Extent> s);

} // namespace std
```

23.7.2.1 `basic_string_span` constructors, copy, assignment, and destructor [basic_string_span.cons]

```
  constexpr basic_string_span();
  constexpr basic_string_span(nullptr_t);
```

Requires: `extent == dynamic_extent || extent == 0`

Effects: Constructs an empty `basic_string_span`.

Postconditions: `size() == 0 && data() == nullptr`

Complexity: Constant.

```
  constexpr basic_string_span(pointer s);
```

Requires: `char_traits<CharT>::length(s) <= PTRDIFF_MAX`

Returns: `basic_string(s, char_traits<CharT>::length(s))`

Complexity: Linear.

```
  constexpr basic_string_span(pointer ptr, index_type count);
```

Requires: When `ptr` is null pointer then `count` shall be `0`. When `ptr` is not null pointer, then it shall point to the beginning of a valid sequence of objects of at least `count` length. `count` shall always be `>= 0`. If `extent` is not `dynamic_extent`, then `count` shall be equal to `extent`.

Effects: Constructs a `basic_string_span` that is a view over the sequence of objects pointed to by `ptr`. If `ptr` is null pointer or `count` is `0` then an empty `basic_string_span` is constructed.

Postconditions: `size() == count && data() == ptr`

Complexity: Constant.

```
  constexpr basic_string_span(pointer firstElem, pointer lastElem);
```

Requires: `[firstElem, lastElem)` is a valid range and `distance(firstElem, lastElem) >= 0`. If `extent` is not equal to `dynamic_extent`, then `distance(firstElem, lastElem)` shall be equal to `extent`.

Effects: Constructs a `basic_string_span` that is a view over the range `[firstElem, lastElem)`. If `distance(firstElem, lastElem)` then an empty `span` is constructed.

Postconditions: `size() == distance(first, last) && data() == firstElem`

Complexity: The same as for `distance(first, last)`

```
  template <size_t N>
    constexpr basic_string_span(element_type (&arr)[N]);
  template <size_t N>
    constexpr basic_string_span(array<remove_const_t<element_type>, N>&
arr);
  template <size_t N>
    constexpr basic_string_span(const array<remove_const_t<element_type>,
N>& arr);
```

Requires: Unless `extent == dynamic_extent`, then either a trailing element with value `element_type{}` shall be found and the count of elements up to and excluding that element shall equal `extent`, or there shall be no trailing element with value `element_type{}` and `N == extent`.

The second constructor shall not participate in overload resolution unless `is_const<element_type>::value` is `true`.

Effects: Constructs a `basic_string_span` that is a view over the supplied array, excluding any zero-termination that may exist.

Complexity: Linear in the size of `arr`.

```
  template <class Container>
    constexpr basic_string_span(Container& cont);
  template <class Container>
    constexpr basic_string_span(const Container& cont);
```

Requires: The constructor shall not participate in overload resolution unless:

- `Container` meets the requirements of both a contiguous container (defined in 23.2.1/13) and a sequence container (defined in 23.2.3).
- The `Container` implements the optional sequence container requirement of `operator[]` (defined in Table 100).
- `Container::value_type` is the same as `remove_const_t<element_type>`.

The constructor shall not participate in overload resolution if `Container` is a `basic_string_span` or `array` or `basic_string`.

The second constructor shall not participate in overload resolution unless `std::is_const<element_type>::value` is `true`.

If `extent` is not equal to `dynamic_extent`, then `cont.size()` shall be equal to `extent`.

Effects: Constructs a `basic_string_span` that is a view over the sequence owned by `cont`.

Postconditions: `size() == cont.size() && data() == addressof(cont[0])`

Complexity: Constant.

```
  constexpr basic_string_span(const basic_string_span& other) noexcept =
default;
  constexpr basic_string_span(basic_string_span&& other) noexcept =
default;
```

Effects: Constructs a `basic_string_span` by copying the implementation data members of another `basic_string_span`.

Postconditions: `other.size() == size() && other.data() == data()`

Complexity: Constant.

```
  template <class OtherCharT, ptrdiff_t OtherExtent>
    constexpr basic_string_span(const basic_string_span<OtherCharT,
OtherExtent>& other);

  template <class OtherCharT, ptrdiff_t OtherExtent>
    constexpr basic_string_span(basic_string_span<OtherCharT,
OtherExtent>&& other);
```

Requires: These constructors shall not participate in overload resolution unless `OtherCharT` differs from `CharT` only by cv-qualifiers. If `extent` is not equal to `dynamic_extent`, then `other.size()` shall be equal to `extent`.

Effects: Constructs a `basic_string_span` by copying the implementation data members of another `basic_string_span`, performing suitable conversions.

Postconditions: `size() == other.size() && data() == reinterpret_cast<pointer>(other.data())`

Complexity: Constant.

```
  constexpr basic_string_span(basic_string<remove_const_t<element_type>>&
s);
  constexpr basic_string_span(const
basic_string<remove_const_t<element_type>>& s);
  constexpr
basic_string_span(basic_string<remove_const_t<element_type>>&& s) =
delete;
```

```
```

Requires: If `extent` is not equal to `dynamic_extent`, then `s.length()` shall be equal to `extent`.

The second constructor shall not participate in overload resolution unless `std::is_const<element_type>::value` is `true`.

Effects: Constructs a `basic_string_span` that is a view over the character sequence owned by the supplied `basic_string`.

Postconditions: `size() == s.length() && data() == reinterpret_cast<pointer>(addressof(s[0]))`

Complexity: Constant.

```
  basic_string_span& operator=(const basic_string_span& other) noexcept =
default;
  basic_string_span& operator=(basic_string_span&& other) noexcept =
default;
```

Effects: Assigns the implementation data of one `basic_string_span` into another.

Postconditions: `size() == other.size() && data() == other.data()`

Complexity: Constant.

23.7.2.2 `basic_string_span` subviews [basic_string_span.sub]

```
  template <ptrdiff_t Count>
    constexpr basic_string_span<element_type, Count> first() const;
```

Requires: `Count >= 0 && Count <= size()`

Effects: Returns a new `basic_string_span` that is a view over the initial `Count` elements of the current `basic_string_span`.

Returns: `basic_string_span(data(), Count);`

Complexity: Constant.

```
  template <ptrdiff_t Count>
    constexpr basic_string_span<element_type, Count> last() const;
```

Requires: `Count >= 0 && Count <= size()`

Effects: Returns a new `basic_string_span` that is a view over the final `Count` elements of the current `basic_string_span`.

Returns: `basic_string_span(Count == 0 ? data() : data() + (size() - Count), Count)`

Complexity: Constant.

```
template <ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
    constexpr basic_string_span<element_type, Count> subspan() const;
```

Requires: `(Offset == 0 || Offset > 0 && Offset < size()) && (Count == dynamic_extent || Count >= 0 && Offset + Count <= size())`

Effects: Returns a new `basic_string_span` that is a view over `Count` elements of the current `basic_string_span` starting at element `Offset`. If `Count` is equal to `dynamic_extent`, then a `basic_string_span` over all elements from `Offset` onwards is returned.

Returns: `basic_string_span(data() + Offset, Count == dynamic_extent ? size() - Offset : Count)`

Complexity: Constant

```
constexpr basic_string_span<element_type, dynamic_extent>
first(index_type count) const;
```

Requires: `count >= 0 && count <= size()`

Effects: Returns a new `basic_string_span` that is a view over the initial `count` elements of the current `basic_string_span`.

Returns: `basic_string_span(data(), count);`

Complexity: Constant.

```
constexpr basic_string_span<element_type, dynamic_extent>
last(index_type count) const;
```

Requires: `Count >= 0 && Count <= size()`

Effects: Returns a new `basic_string_span` that is a view over the final `Count` elements of the current `basic_string_span`.

Returns: `basic_string_span(Count == 0 ? data() : data() + (size() - Count), Count)`

Complexity: Constant.

```
constexpr basic_string_span<element_type, dynamic_extent>
subview(index_type offset, index_type count = dynamic_extent) const;
```

Requires: `(Offset == 0 || Offset > 0 && Offset < size()) && (Count == dynamic_extent || Count >= 0 && Offset + Count <= size())`

Effects: Returns a new `basic_string_span` that is a view over `Count` elements of the current `basic_string_span` starting at element `Offset`. If `Count` is equal to `dynamic_extent`, then a `basic_string_span` over all elements from `Offset` onwards is returned.

Returns: `basic_string_span(data() + Offset, Count == dynamic_extent ? size() - Offset : Count)`

Complexity: Constant

23.7.2.2 `basic_string_span` observers [basic_string_span.obs]

```
constexpr index_type length() const noexcept;
```

Effects: Equivalent to `size()`.

```
constexpr index_type size() const noexcept;
```

Effects: Returns the number of elements accessible through the `basic_string_span`.

Returns: `>= 0`

Complexity: Constant

```
constexpr index_type length_bytes() const noexcept;
```

Effects: Equivalent to size_bytes().

```
constexpr index_type size_bytes() const noexcept;
```

Effects: Returns the number of bytes used for the object representation of all elements accessible through the `basic_string_span`.

Returns: `size() * sizeof(element_type)`

Complexity: Constant

```
constexpr bool empty() const noexcept;
```

```
```

Effects: Equivalent to `size() == 0`.

Returns: `size() == 0`

Complexity: Constant

23.7.2.3 `basic_string_span` element access [basic_string_span.elem]

```
  constexpr reference operator[](index_type idx) const;
  constexpr reference operator()(index_type idx) const;
```

Requires: `idx >= 0 && idx < size()`

Effects: Returns a reference to the element at position `idx`.

Returns: `*(data() + idx)`

Complexity: Constant

```
  constexpr pointer data() const noexcept;
```

Effects: Returns either a pointer to the first element in the sequence accessible via the `basic_string_span` or the null pointer if that was the value used to construct the `basic_string_span`.

Returns: (for exposition) `data_`

Complexity: Constant

23.7.2.4 `basic_string_span` iterator support [basic_string_span.iterators]

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

Returns: An iterator referring to the first element in the `basic_string_span`.

Complexity: Constant

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

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

Complexity: Constant

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

Returns: An iterator that is semantically equivalent to `reverse_iterator(end())`.

Complexity: Constant

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

Returns: An iterator that is semantically equivalent to `reverse_iterator(begin())`.

Complexity: Constant

23.7.2.5 `basic_string_span` comparison operators [basic_string_span.comparison]

```
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator==(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

Effects: Equivalent to `equal(l.begin(), l.end(), r.begin(), r.end())`.

```
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator!=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

Effects: Equivalent to `!(l == r)`.

```
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator<(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

Effects: Equivalent to `lexicographical_compare(l.begin(), l.end(), r.begin(), r.end())`.

```
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator>(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

Effects: Equivalent to `(r < l)`.

```
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator<=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

Effects: Equivalent to `!(l > r)`.

```
template <class CharT, ptrdiff_t Extent>
  constexpr bool operator>=(const basic_string_span<CharT, Extent>& l,
const basic_string_span<CharT, Extent>& r) const noexcept;
```

Effects: Equivalent to `!(l < r)`.

23.7.2.6 `basic_string_span` to `basic_string` conversion [basic_string_span.basic_string]

```
template<class CharT, ptrdiff_t Extent>
  basic_string<remove_const_t<CharT>> to_string(basic_string_span<CharT,
Extent> s);
```

Returns: `basic_string(s.data(), s.size())`

Acknowledgements

basic_string_span was designed to support the C++ Core Coding Guidelines [5] and as such, the current version reflects the input of Herb Sutter, Jim Springfield, Gabriel Dos Reis, Chris Hawblitzel, Gor Nishanov, and Dave Sielaff. Łukasz Mendakiewicz, Bjarne Stroustrup, Eric Niebler and Artur Laksberg provided helpful review during development. Anna Gringauze provided many useful insights and design fixes and wrote an initial implementation.

Many thanks to Gabriel Dos Reis and Stephan T. Lavavej for their valuable input to this document.

References

[1] J. Yasskin, "string_view: a non-owning reference to a string, revision 5" 09 January 2013, [Online], Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3762.html

[2] N. MacIntosh, "span: bounds-safe views for sequences of objects" P0122R2, 26 May 2016.

[3] Microsoft, "Guideline Support Library reference implementation: basic_string_span", 2015, [Online], Available: https://github.com/Microsoft/GSL

[4] Richard Smith, "Working Draft: Standard For Programming Language C++", N4567, 2015, [Online], Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4567.pdf

[5] Bjarne Stroustrup, Herb Sutter, "C++ Core Coding Guidelines", 2015, [Online], Available: https://github.com/isocpp/CppCoreGuidelines