Project: Programming Language C++, Library Working Group
Document number: P0122R6
Date: 2018-02-11
Reply-to: Neil MacIntosh neilmac@fb.com

span: bounds-safe views for sequences of objects

Contents

Changelog

Changes from R0

- Changed the name of the type being proposed from *array_view* to *span* following feedback from LEWG at the Kona meeting.
- Removed multidimensional aspects from the proposal. *span* is now always single-dimension and contiguous.
- Added details on potential interoperation with the multidimensional *view* type from P0009 [5].
- Removed functions to convert from *span<byte>* to *span<T>* as they are not compatible with type aliasing rules.
- Introduced dependency on P0257 [6] for definition of *byte* type, in order to support *span* as a method of accessing object representation.
- Added section containing proposed wording for inclusion in the standard.
- Simplified *span* interface based on reviewer feedback.

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. For a view type like *span*, the constness of the view is immaterial to the constness of the element type, the iterator interface of *span* now reflects that.
- Removed the deletion of constructors that take rvalue-references based on LEWG feedback.
- Added support for construction from *const Container&*.

Changes from R2

- Wording cleanup: removed *const* on non-member functions and inappropriate *noexcept* specifiers. Improved wording to be clear that the *reverse_iterator* is not contiguous. Removed *constexpr* from *as_bytes()* and *as_writeable_bytes()* as it would be illegal. Tidied up effects of *last()* overloads and of *array/std::array* constructors for cases when the array is empty.
- Added back *cbegin()* and *cend()* and *const_iterator* type based on LEWG feedback in Oulu.
- Improved colors.

Changes from R3

- Updated the wording to be differences against N4618.

Changes from R4

- Removed dependency on P0257 now that *byte* is part of the standard.
- Updated the wording to be differences against N4659.
- Added constructors from *unique_ptr*, *shared_ptr*.

- Removed unachievable *constexpr* from *as_bytes()* and *as_writeable_bytes()* functions.

Changes from R5

- Removed conversion constructors that took a unique_ptr/shared_ptr argument.
- Added constexpr qualifier to all iterator access functions on span.
- Removed length() and length_bytes() member functions from span. Length() is considered unnecessary as string_view offers it if users are looking for std::string interface compatibility.
- Removed constructor from span that took a nullptr_t (as per request from LEWG). It does not add any value beyond the default constructor and may bind in unexpected ways for users.
- Removed move constructor and move assignment operator. They are unnecessary as this is designed to be a copy-only type.
- Removed redundant "Effects" clause from descriptions of copy constructor and assignment operator in proposed wording.
- Simplified many member functions descriptions down to an "effects equivalent to" form in proposed wording.
- Corrected typo in description of as_writeable_bytes() function.
- Added covering statement to synopsis that marks all member functions as having constant time complexity and removed individual time complexity clauses to proposed wording.
- Added (accidentally-) missing description for cbegin()/cend()/crbegin()/crend() to proposed wording.
- Removed unnecessary std:: qualification from remove_cv_t() call in proposed wording.
- Corrected definitions of comparison operations to take arguments by-value rather than by-reference to reflect the design of span as a copy-only type.
- Removed incorrect italicization of byte in proposed wording.

Introduction

This paper presents a design for a fundamental vocabulary type *span*.

The *span* type is an abstraction that provides a view over a contiguous sequence of objects, the storage of which is owned by some other object. The design for *span* presented here provides bounds-safety guarantees through a combination of compile-time and (configurable) run-time constraints.

The design of the *span* type discussed in this paper is related to the *span* previously proposed in N3851 [1] and also draws on ideas in the *array_ref* and *string_ref* classes proposed in N3334 [2]. *span* is closely related to the generalized, multidimensional memory-access abstraction *array_ref* described in P0009 [5]. The *span* proposed here is sufficiently compatible with *array_ref* that interoperability between the two types would be simple and well-defined.

While *array_ref* is proposed by P0009 [5] as a generalized and highly configurable view type that can address needs for specialized domains such as scientific computing, *span* is proposed as a simple solution to the common need for a single-dimensional view over contiguous storage.

Motivation and Scope

The evolution of the standard library has demonstrated that it is possible to design and implement abstractions in Standard C++ that improve the reliability of C++ programs without sacrificing either performance or portability. This proposal identifies a new "vocabulary type" for inclusion in the standard library that enables both high performance and bounds-safe access to contiguous sequences of elements. This type would also improve modularity, composability, and reuse by decoupling accesses to array data from the specific container types used to store that data.

These characteristics lead to higher quality programs. Some of the bounds and type safety constraints of *span* directly support "correct-by-construction" programming methodology – where errors simply do not compile. One of the major advantages of *span* over the common idiom of a "pointer plus length" pair of parameters is that it provides clearer semantics hints to analysis tools looking to help detect and prevent defects early in a software development cycle.

Impact on the Standard

This proposal is a pure library extension. It does not require any changes to standard classes, functions, or headers.

However – if adopted – it may be useful to overload some standard library functions for this new type (an example would be *copy()*).

span has been implemented in standard C++ (C++11) and is being successfully used within a commercial static analysis tool for C++ code as well as commercial office productivity software. An open source, reference implementation is available at https://github.com/Microsoft/GSL [3].

Design Decisions

View not container

span is simply a view over another object's contiguous storage – but unlike *array* or *vector* it does not "own" the elements that are accessible through its interface. An important observation arises from this: *span* never performs any free store allocations.

While span is a view, it is not an iterator. You cannot perform increment or decrement operations on it, nor dereference it.

No configurable view properties

In the related *array_ref* type described in P0009 [5], properties are used to control policies such as memory layout (column-major, row-major) and location (on heterogenous memory architectures) for specific specializations of *array_ref*. *span* does not require properties as it is always a simple view over contiguous storage. Its memory layout and access characteristics are equivalent to those of a built-in array. This difference should not prevent conversions between *array_ref* and *span* instances, it merely constrains that they could only be available in cases where *array_ref* properties are compatible with the characteristics of *span*.

View length and measurement

The general usage protocol of the *span* class template supports both static-size (fixed at compile time) and dynamic-size (provided at runtime) views. The *Extent* template parameter to *span* is used to provide the extent of the *span*.

```
constexpr ptrdiff_t dynamic_extent = -1;
```

The default value for *Extent* is *dynamic_extent*: a unique value outside the normal range of lengths (0 to *PTRDIFF_MAX* inclusive) reserved to indicate that the length of the sequence is only known at runtime and must be stored within the *span*. A dynamic-size *span* is, conceptually, just a pointer and size field (this is not an implementation requirement, however).

```
int* somePointer = new int[someLength];

// Declaring a dynamic-size span
// s will have a dynamic-size specified by someLength at construction
span<int> s { somePointer, someLength };
```

The type used for measuring and indexing into span is *ptrdiff_t*. Using a signed index type helps avoid common mistakes that come from implicit signed to unsigned integer conversions when users employ integer literals (which are nearly always signed). The use of *ptrdiff_t* is natural as it is the type used for pointer arithmetic and array indexing – two operations that *span* explicitly aims to replace but that an implementation of *span* would likely rely upon.

A fixed-size *span* provides a value for *Extent* that is between 0 and PTRDIFF_MAX (inclusive). A fixed-size *span* requires no storage size overhead beyond a single pointer – using the type system to carry the fixed-length information. This allows *span* to be an extremely efficient type to use for access to fixed-length buffers.

```
int arr[10];

// deduction of size from arrays means that span size is always correct
span<int, 10> s2 { arr }; // fixed-size span of 10 ints
span<int, 20> s3 { arr }; // error: will fail compilation
span<int> s4 { arr }; // dynamic-size span of 10 ints
```

Value Type Semantics

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 by value or a container type by reference, such as *array* or *vector*.

Conceptually, *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 *span* within programs. For example, *span* has a trivial destructor, so common ABI conventions allow it to be passed in registers.

Range-checking and bounds-safety

All accesses to the data encapsulated by a span are conceptually range-checked to ensure they remain within the bounds of the *span*. What actually happens as the result of a failure to meet *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 *span*'s design, and allows users to rely on the guarantee that as long as a sequence is accessed via a correctly initialized *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 *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 *span* objects are compatible for assignment.

1. A fixed-size *span* may be constructed or assigned from another fixed-size span of equal length.
2. A dynamic-size *span* may always be constructed or assigned from a fixed-size *span*.
3. A fixed-size *span* may always be constructed or assigned from a dynamic-size *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.

Element types and conversions

span must be configured with its element type via the template parameter *ValueType*, which is required to be a complete object type that is not an abstract class type. *span* supports either read-only or mutable access to the sequence it encapsulates. To access read-only data, the user can declare a *span<const T>*, and access to mutable data would use a *span<T>*.

Construction or assignment between *span* objects with different element types is allowed whenever it can be determined statically that the element types are exactly storage-size equivalent (so there is no difference in the extent of memory being accessed), and that the types can legally be aliased.

As a result of these rules, it is always possible to convert from a *span<T>* to a *span<const T>*. It is not allowed to convert in the opposite direction, from *span<const T>* to *span<T>*. This property is extremely convenient for calling functions that take *span* parameters.

Element access and iteration

span's interface for accessing elements is largely similar to that of *array*. It overloads *operator[]* for element access, and offers random access iterators, making it adoptable with a minimum of source changes in code that previously used an array, an *array* object, or a pointer to access more than one object. span also overloads *operator()* for element access, to provide compatibility with code written to operate against *view*.

span provides random-access iterators over its data, comparable to *vector* and *array*. All accesses to elements made through these iterators are range-checked (subject to configuration as previously described), just as if they had been performed via the subscript operator on *span*. There is no difference in the mutability of the iterators returned from a *const* or non-*const span* as the constness of the element type is already determined when the *span* is created. As is appropriate for a view, whether the *span* itself is *const* does not affect the element type, and this is reflected in the simplicity of the iterator model.

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

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

constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;

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

constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;
```

Construction

The *span* class is expected to become a frequently used vocabulary type in function interfaces (as a safer replacement of "(pointer, length)" idioms), as it specifies a minimal set of requirements for safely accessing a sequence of objects and decouples a function that needs to access a sequence from the details of the storage that holds such elements.

To simplify use of *span* as a simple parameter, *span* offers a number of constructors for common container types that store contiguous sequences of elements. A summarized extract from the specification illustrates this:

```
 // [span.cons], span constructors, copy, assignment, and destructor
constexpr span();
constexpr span(pointer ptr, index_type count);
constexpr span(pointer firstElem, pointer lastElem);
template <size_t N>
  constexpr span(element_type (&arr)[N]);
template <size_t N>
   constexpr span(array<remove_const_t<element_type>, N>& arr);
template <size_t N>
  constexpr span(const array<remove_const_t<element_type>, N>& arr);
```

```
  template <class Container>
    constexpr span(Container& cont);
  template <class Container>
    constexpr span(const Container& cont);
  constexpr span(const span& other) noexcept = default;
  template <class OtherElementType, ptrdiff_t OtherExtent>
    constexpr span(const span<OtherElementType, OtherExtent>& other);
```

It is allowed to construct a span from the null pointer, and this creates an object with *.size() == 0*. Any attempt to construct a span with a null pointer value and a non-zero length is considered a range-check error.

Byte representations and conversions

A span of any element type that is a standard-layout type can be converted to a span<*const byte*> or a span<*byte*> via the free functions *as_bytes()* and *as_writeable_bytes()* respectively. These operations are considered useful for systems programming where byte-oriented access for serialization and data transmission is essential.

```
// [span.objectrep], views of object representation
template <class ElementType, ptrdiff_t Extent>
  span<const byte, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_bytes(span<ElementType, Extent> s)
noexcept;

template <class ElementType, ptrdiff_t Extent>
  span<byte, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_writeable_bytes(span<ElementType, Extent>
) noexcept;
```

These byte-representation conversions still preserve const-correctness, however. It is not possible to convert from a *span<const T>* be converted to a span<*byte*> (through SFINAE overload restriction).

Comparisons

span supports all the same comparison operations as a sequential standard library container: element-wise comparison and a total ordering by lexicographical comparison. This helps make it an effective replacement for existing uses of sequential contiguous container types like *array* or *vector*.

```
// [span.comparison], span comparison operators
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator==(span<ElementType, Extent> l, span<ElementType,
Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator!=(span<ElementType, Extent> l, span<ElementType,
Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<(span<ElementType, Extent> l, span<ElementType,
Extent> r);

template <class ElementType, ptrdiff_t Extent>
```

```
  constexpr bool operator<=(span<ElementType, Extent> l, span<ElementType,
Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>(span<ElementType, Extent> l, span<ElementType,
Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>=(span<ElementType, Extent> l, span<ElementType,
Extent> r);
```

Regardless of whether they contain a valid pointer or null pointer, zero-length *span*s are all considered equal. This is considered a useful property when writing library code. If users wish to distinguish between a zero-length *span* with a valid pointer value and a *span* containing the null pointer, then they can do so by calling the *data()* member function and examining the pointer value directly.

Creating sub-spans

span offers convenient member functions for generating a new *span* that is a reduced view over its sequence. In each case, the newly constructed *span* is returned by value from the member function. As the design requires bounds-safety, these member functions are guaranteed to either succeed and return a valid *span*, or fail with undefined behavior (e.g. calling *terminate())* if the parameters were not within range.

```
  // [span.sub], span subviews
  constexpr span<element_type, dynamic_extent> first(index_type count) const;
  constexpr span<element_type, dynamic_extent> last(index_type count) const;
  constexpr span<element_type, dynamic_extent> subspan(index_type offset,
index_type count = dynamic_extent) const;
```

first() returns a new *span* that is limited to the first N elements of the original sequence. Conversely, *last()* returns a new *span* that is limited to the last N elements of the original sequence. *subspan()* allows an arbitrary sub-range within the sequence to be selected and returned as a new *span*.

All three member functions are overloaded in forms that accept their parameters as template parameters, rather than function parameters. These overloads are helpful for creating fixed-size *span* objects from an original input *span,* whether fixed- or dynamic-size.

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

Multidimensional *span*

span as presented here only supports a single-dimension view of a sequence. This covers the most common usage of contiguous sequences in C++. *span* has convenience (such as iterators, *first()*, *last(),* and *subspan()*) and default behaviors that make most sense in a single-dimension.

Adding support for multidimensional and noncontiguous (strided) views of data is deferred to a separate type not described here. One such candidate would be the more general *array_ref* facility described in P0009 [5]. The interface of *span* is sufficiently compatible with that of *array_ref*, that users should not feel any significant discontinuity between the two. In fact, it is entirely possible to implement a *span* using *array_ref*.

Proposed Wording Changes

The following proposed wording changes against the working draft of the standard are relative to N4659 [6].

20.5.1.2 Headers [headers]

2 The C++ standard library provides the *C++ library headers*, as shown in Table 16.

Table 16 – C++ library headers

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

26 Containers library [containers]

26.1 General [containers.general]

2 The following subclauses describe container requirements, and components for sequence containers, associative containers, and views as summarized in Table 82.

Table 82 – Containers library summary

Subclause	**Header(s)**
26.2 Requirements	
26.3 Sequence containers	<array> <deque> <forward_list> <list> <vector>

26.4 Associative containers	<map> <set>
26.5 Unordered associative containers	<unordered_map> <unordered_set>
26.6 Container adaptors	<queue> <stack>
26.7 Views	<span>

26.7 Views [views]

26.7.1 General [views.general]

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

Header `<span>` synopsis

```
namespace std {

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


// [span], class template span
template <class ElementType, ptrdiff_t Extent = dynamic_extent>
class span;

// [span.comparison], span comparison operators
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator==(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator!=(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<=(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>=(span<ElementType, Extent> l,
span<ElementType, Extent> r);

// [span.objectrep], views of object representation
```

```
template <class ElementType, ptrdiff_t Extent>
  span<const char, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_bytes(span<ElementType, Extent> s)
noexcept;

template <class ElementType, ptrdiff_t Extent>
  span<char, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_writeable_bytes(span<ElementType,
Extent>) noexcept;

} // namespace std
```

23.7.2 Class template `span` [views.span]

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

2 `ElementType` is required to be a complete object type that is not an abstract class type.

3 The `iterator` type for `span` is a random access iterator and contiguous iterator. The `reverse_iterator` type is a random access iterator.

4 All member functions of span have a constant time complexity.

```
namespace std {

// A view over a contiguous, single-dimension sequence of objects
template <class ElementType, ptrdiff_t Extent = dynamic_extent>
class span {
public:
  // constants and types
  using element_type = ElementType;
  using value_type = remove_cv_t<ElementType>;
  using index_type = ptrdiff_t;
  using difference_type = ptrdiff_t;
  using pointer = element_type*;
  using reference = element_type&;
  using iterator = /*implementation-defined */;
  using const_iterator = /* implementation-defined */;
  using reverse_iterator = reverse_iterator<iterator>;
  using const_reverse_iterator = reverse_iterator<const_iterator>;

  constexpr static index_type extent = Extent;

   // [span.cons], span constructors, copy, assignment, and destructor
  constexpr span() noexcept;
  constexpr span(pointer ptr, index_type count);
  constexpr span(pointer firstElem, pointer lastElem);
  template <size_t N>
    constexpr span(element_type (&arr)[N]);
  template <size_t N>
     constexpr span(array<remove_const_t<element_type>, N>& arr);
  template <size_t N>
    constexpr span(const array<remove_const_t<element_type>, N>& arr);
  template <class Container>
```

```
    constexpr span(Container& cont);
  template <class Container>
    span(const Container&);
  constexpr span(const span& other) noexcept = default;
  template <class OtherElementType, ptrdiff_t OtherExtent>
    constexpr span(const span<OtherElementType, OtherExtent>& other);
  ~span() noexcept = default;
  constexpr span& operator=(const span& other) noexcept = default;

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

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

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

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

  constexpr const_iterator cbegin() const noexcept;
  constexpr const_iterator cend() const noexcept;

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

  constexpr const_reverse_iterator crbegin() const noexcept;
  constexpr const_reverse_iterator crend() const noexcept;

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

// [span.comparison], span comparison operators
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator==(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator!=(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<=(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>(span<ElementType, Extent> l,
span<ElementType, Extent> r);

template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>=(span<ElementType, Extent> l,
span<ElementType, Extent> r);

// [span.objectrep], views of object representation
template <class ElementType, ptrdiff_t Extent>
  span<const byte, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_bytes(span<ElementType, Extent> s);

template <class ElementType, ptrdiff_t Extent>
  span<byte, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_writeable_bytes(span<ElementType,
Extent> );

} // namespace std
```

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

```
  constexpr span() noexcept;
```

Remarks: if `Extent != dynamic_extent || Extent != 0` then the program is ill-formed.

Effects: Constructs an empty `span`.

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

```
  constexpr 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 `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 `span` is constructed.

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

Throws: Nothing

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

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

Effects: Constructs a `span` that is a view over the range `[firstElem, lastElem)`. If `distance(firstElem, lastElem) == 0` then an empty `span` is constructed.

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

Throws: Nothing

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

Remarks: If `extent != dynamic_extent` && `N != extent`, then the program is ill-formed.

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

Effects: Constructs a `span` that is a view over the supplied array.

Postconditions: `size() == N && data() == addressof(arr[0])`

Throws: Nothing

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

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

- `Container` meets the requirements of both a contiguous container (defined in [container.requirements.general]/13) and a sequence container (defined in [sequence.reqmts]).

- The `Container` implements the optional sequence container requirement of `operator[]`.
- `Container::value_type` is the same as `remove_const_t<element_type>`.

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

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

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

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

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

Throws: Nothing

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

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

```
template <class OtherElementType, ptrdiff_t OtherExtent>
  constexpr span(const span<OtherElementType, OtherExtent>& other);
```

Remarks: This constructor shall not participate in overload resolution unless trying to access `OtherElementType` through an `ElementType*` would meet the rules for well-defined object access defined in [basic.lval]/8.

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

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

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

Throws: Nothing

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

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

23.7.2.2 span subviews [span.sub]

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

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

Effects: Equivalent to: `return span(data(), Count);`

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

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

Effects: Equivalent to: `return span(data() + (size() - Count), Count);`

```
  template <ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
    constexpr 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 `span` that is a view over `Count` elements of the current `span` starting at element `Offset`. If `Count` is equal to `dynamic_extent`, then a `span` over all elements from `Offset` onwards is returned.

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

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

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

Effects: Equivalent to: `return span(data(), count);`

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

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

Effects: Equivalent to: `return span(data() + (size() - count), count);`

```
  constexpr span<element_type, dynamic_extent> subspan(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: Equivalent to: `return span(data() + offset, count == dynamic_extent ? size() - offset : count);`

23.7.2.2 `span` observers [span.obs]

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

Effects: Equivalent to: `return size_;`

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

Effects: Equivalent to: `return size() * sizeof(element_type;`

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

Effects: Equivalent to: `return size() == 0;`

23.7.2.3 `span` element access [span.elem]

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

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

Effects: Equivalent to: `return *(data() + idx)`

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

Effects: Equivalent to: `return data_;`

23.7.2.4 `span` iterator support [span.iterators]

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

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

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

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

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

Effects: Equivalent to `return reverse_iterator(end())`.

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

Returns: Equivalent to: `return reverse_iterator(begin());`

```
constexpr const_iterator cbegin() const noexcept;
```

Returns: A const iterator referring to the first element in the `span`.

```
constexpr const_iterator cend() const noexcept;
```

Returns: A const iterator which is the past-the-end value.

```
constexpr const_reverse_iterator crbegin() const noexcept;
```

Effects: Equivalent to `return reverse_iterator(cend())`.

```
constexpr const_reverse_iterator crend() const noexcept;
```

Returns: Equivalent to: `return reverse_iterator(cbegin());`

23.7.2.5 span comparison operators [span.comparison]

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator==(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

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

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator!=(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

Effects: Equivalent to: `return !(l == r);`

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

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

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator<=(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

Effects: Equivalent to: `return !(l > r);`

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

Effects: Equivalent to: `return (r < l);`

```
template <class ElementType, ptrdiff_t Extent>
  constexpr bool operator>=(span<ElementType, Extent> l,
span<ElementType, Extent> r);
```

Effects: Equivalent to: `return !(l < r);`

23.7.2.6 views of object representation [span.objectrep]

```
template <class ElementType, ptrdiff_t Extent>
```

```
  span<const byte, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_bytes(span<ElementType, Extent> s)
noexcept;
```

Effects: Equivalent to: `return { reinterpret_cast<const byte*>(s.data()), sizeof(ElementType) * s.size() };`

```
template <class ElementType, ptrdiff_t Extent>
  span<byte, ((Extent == dynamic_extent) ? dynamic_extent :
(sizeof(ElementType) * Extent))> as_writeable_bytes(span<ElementType,
Extent>) noexcept;
```

Remarks: This function will not participate in overload resolution when `is_const<ElementType>::value` is `true`.

Effects: Equivalent to: `return { reinterpret_cast<byte*>(s.data()), sizeof(ElementType) * s.size() };`

Acknowledgements

This work has been heavily informed by N3851 (an *array_view* proposal) and previous discussion amongst committee members regarding that proposal. Gabriel Dos Reis, Titus Winters and Stephan T. Lavavej provided invaluable feedback on this document. Thanks to Casey Carter and Daniel Krügler for detailed feedback on the wording.

This version of *span* was designed to support the C++ Core Coding Guidelines [4] 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 of this version of *span* during its development.

The authors of P0009 were invaluable in discussing how *span* and *array_ref* can be compatible and by doing so support a programming model that is safe and consistent as users move between a single dimension and multiple dimensions.

References

[1] Łukasz Mendakiewicz, Herb Sutter, "Multidimensional bounds, index and span", N3851, 2014, [Online], Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3851.pdf.

[2] J. Yasskin, "Proposing array_ref<T> and string_ref", N3334 14 January 2012, [Online], Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3334.html.

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

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

[5] H. Carter Edwards et al., “Polymorphic Multidimensional Array View”, P0009, 2015, [Online], Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0009r0.html

[6] Richard Smith, “Working Draft: Standard For Programming Language C++”, N4659, 2017, [Online], Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/n4659.pdf