Range constructor for `std::span`

Document #: D1394R0
Date: 2019-01-17
Project: Programming Language C++
Audience: LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

1 Abstract

This paper proposes that `span` be constructible from any forwarding contiguous range of its value type. The idea was extracted from P1206.

2 Tony tables

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td>std::vector<int> v(42);
std::span foo = v | view::take(3); //ill-formed</td><td>std::vector<int> v(42);
std::span<int> foo = v | view::take(3); //valid</td></tr>
<tr><td>std::vector<int> v(42);
std::span foo = v; // ill-valid
std::span bar(v.begin(), 3); // ill-formed</td><td>std::vector<int> v(42);
std::span foo = v; // valid
std::span bar(v.begin(), 3); // valid</td></tr>
<tr><td>std::vector<int> get_vector();
void foo(std::experimental::span<int>);
void bar(std::experimental::span<const int>);
bar(get_vector()); //valid
foo(get_vector()); //ill-formed</td><td>std::vector<int> get_vector();
void foo(std::experimental::span<int>);
void bar(std::experimental::span<const int>);
bar(get_vector()); //ill-formed
foo(get_vector()); //ill-formed</td></tr>
</table>

3 Motivation

Span is specified to be constructible from `Container` types. However, while defined, `Container` is not a concept and as such `ContiguousRange` is more expressive. Furthermore, there exist some non-container ranges that would otherwise be valid ranges to construct span from. As such span as currently specified fits poorly with the iterators / ranges model of the rest of the standard library.

4 Design considerations

Currently, a `rvalue-ref Container<T>` binds to `span<const T>`. This behavior is surprising, dangerous and fits poorly with the `forwarding-range` model introduced with ranges. We therefore propose that span should only be constructible from `forwarding-range`s.

We propose to specify all constructors currently accepting a container or pointers in terms of `ContiguousRange` and `ContiguousIterator` respectively as well as to add or modify the relevant deduction guides for these constructors.

5 Future work

- We suggest that both the wording and the implementation of span would greatly benefit from a trait to detect whether a type has a static extent. Because `std::extent` equals to 0 for types without static extent, and because 0 is a valid extent for containers, `std::extent` proved too limited. However we do not propose a solution in the present paper.

6 Proposed wording

Change in **[views.span] 21.7.3**:

```
 // [span.cons], constructors, copy, and assignment
constexpr span() noexcept;
template <ContiguousIterator It>
requires ConvertibleTo<remove_reference_t<iter_reference_t<It>>(*)[], ElementType(*)[]>
constexpr span( ~~pointer ptr~~ It begin, index_type count);
~~constexpr span(pointer first, pointer last);~~
template <ContiguousIterator It, SizedSentinel<It> End>
requires ConvertibleTo<remove_reference_t<iter_reference_t<It>>(*)[], ElementType(*)[]>
constexpr span(It first, End last);

template<size_t N>
constexpr span(element_type (&arr)[N]) noexcept;
template<size_t N>
constexpr span(array<value_type, N>& arr) noexcept;
template<size_t N>
constexpr span(const array<value_type, N>& arr) noexcept;
~~template<class Container>~~
~~constexpr span(Container& cont);~~
~~template<class Container>~~
~~constexpr span(const Container& cont);~~
template <ranges::ContiguousRange R>
requires ranges::SizedRange<R> && forwarding-range<R> &&
ConvertibleTo<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>(*)[], ElementType(*)[]>
constexpr span(R&& r)
constexpr span(const span\& other) noexcept = default;
template<class OtherElementType, ptrdiff_t OtherExtent>
constexpr span(const span<OtherElementType, OtherExtent>\& s) noexcept;

...

}


template<class T, size_t N>
span(T (&)[N]) -> span<T, N>;
template<class T, size_t N>
span(array<T, N>&) -> span<T, N>;
template<class T, size_t N>
span(const array<T, N>&) -> span<const T, N>;
template <ContiguousIterator It, SizedSentinel<It> End>
span(It, End) -> span<remove_reference_t<iter_reference_t<It>>>;
template <ContiguousIterator It, size_t N>
span(It, N) -> span<remove_reference_t<iter_reference_t<It>>>;

template<class T, size_t N>
span(const array<T, N>&) -> span<const T, N>;
~~template<class Container>~~
~~constexpr span(Container& cont);~~
~~template<class Container>~~
~~constexpr span(const Container& cont);~~
~~constexpr span(const span& other) noexcept = default;~~
template<ranges::ContiguousRange>
requires ranges::SizedRange<R> && forwarding-range<R>
-> span<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>>;
```

In 21.7.3.2 [span.cons]

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

Ensures: size() == 0 && data() == nullptr.

Remarks: This constructor shall not participate in overload resolution unless Extent <= 0 is true.

```
~~constexpr span(pointer ptr, index_type count);~~

template <ContiguousIterator It>
requires ConvertibleTo<remove_reference_t<iter_reference_t<It>>(*)[], ElementType(*)[]>
constexpr span(It first, index_type count);
```

Requires: [~~ptr~~ first, ~~ptr~~ first + count) shall be a valid range. If extent is not equal to dynamic_extent, then count shall be equal to extent.

Effects: Constructs a span that is a view over the range [~~ptr~~ first , ~~ptr~~ first + count).

Ensures: size() == count && data() == ~~ptr~~ adressof(*first).

Throws: Nothing.

```
constexpr span(pointer first, pointer last);
```

Requires: `[first, last)` shall be a valid range. If `extent` is not equal to `dynamic_extent`, then `last - first` shall be equal to `extent`.

Effects: Constructs a span that is a view over the range `[first, last)`.

Ensures: `size() == last - first && data() == first`.

Throws: Nothing.

```
template <ContiguousIterator It, SizedSentinel<It> End>
requires ConvertibleTo<remove_reference_t<iter_reference_t<It>>(*)[], ElementType(*)[]>
constexpr span(It first, End last);
```

Expects: If `extent` is not equal to `dynamic_extent`, then `last - first` shall be equal to `extent`.

Effects: Constructs a span that is a view over the range `[first, last)`.

Ensures: `size() == last - first && data() == adressof(*first)`.

Throws: Nothing.

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

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

Ensures: `size() == N && data() == data(arr)`.

Remarks: These constructors shall not participate in overload resolution unless:

- `extent == dynamic_extent || N == extent` is `true`, and
- `remove_pointer_t<decltype(data(arr))>(*)[]` is convertible to `ElementType(*)[]`.

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

Requires: `[data(cont), data(cont) + size(cont))` shall be a valid range. If `extent` is not equal to `dynamic_extent`, then `size(cont)` shall be equal to `extent`.

Effects: Constructs a `span` that is a view over the range `[data(cont), data(cont) + size(cont))`.

Ensures: `size() == size(cont) && data() == data(cont)`.

Throws: What and when `data(cont)` and `size(cont)` throw.

Remarks: These constructors shall not participate in overload resolution unless:

- `Container` is not a specialization of `span`,
- `Container` is not a specialization of `array`,

- `is_array_v<Container>` is `false`,
- `data(cont)` and `size(cont)` are both well-formed, and
- `remove_pointer_t<decltype(data(cont))>(*)[]` is convertible to `ElementType(*)[]`.

```
template <ranges::ContiguousRange R>
requires ranges::SizedRange<R> && forwarding-range<R> &&
ConvertibleTo<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>(*)[], ElementType(*)[]>
constexpr span(R&& r)
```

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

Effects: Constructs a `span` that is a view over the range `r`.

Ensures: `ranges::size() == ranges::size(r) && ranges::data() == ranges::data(r)`.

Throws: What and when `ranges::data(r)` and `ranges::size(r)` throw.

Constraints:

- `R` is not a specialization of `span`,
- `R` is not a specialization of `array`,
- `is_array_v<R>` is `false`,

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

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