Range constructor for `std::span`

Document #:	D1394R0
Date:	2019-01-14
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 contiguous range of its value type. The idea was extracted from P1206.

2 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 is overly constrained.

3 Design considerations

- Like the current design we propose that span can be constructed from lvalue references
- We further propose that it can be constructed from any `forwarding-range` or `View` as they don't own the underlying data.
- We believe allowing rvalue-reference of containers would be needlessly and surprisingly dangerous.

4 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.

5 Proposed wording

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

```
  // [span.cons], constructors, copy, and assignment
constexpr span() noexcept;
constexpr span(pointer ptr, index_type count);
~~constexpr span(pointer first, pointer last);~~
template <ranges::ContiguousIterator It, ranges::Sentinel<It> End>
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);~~
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 <ranges::ContiguousIterator It, ranges::Sentinel<It> End>
span(It, End) -> span<It, End>;

template<class T, size_t N>
span(const array<T, N>&) -> span<const T, N>;

template<~~Container~~ ContiguousRange>
span(~~Container&~~ ranges::ContiguousRange&)
-> span<~~typename Container&~~ ranges::iter_value_t<ranges::iterator_t<R>>>;
template<~~Container~~ ContiguousRange>
span(~~const Container&~~ const ranges::ContiguousRange&)
-> span<~~typename Container&~~ ranges::iter_value_t<ranges::iterator_t<R>>>;

template<anges::ContiguousRange>
```

```
requires ranges::View<R> || forwarding-range<R>
-> span<ranges::iter_value_t<ranges::iterator_t<R>>>;
```

In 21.7.3.2 [span.cons]

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

Requires: `[ptr, ptr + 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, ptr + count)`.

Ensures: `size() == count && data() == ptr`.

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 <ranges::ContiguousIterator It, ranges::Sentinel<It> End>
constexpr span(It first, End 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() == adressof(*it)`.

Throws: Nothing.

Remark: This constructor shall not participate in overload resolution unless:

- `ranges::iter_value_t<It>(*)[] is convertible to ElementType(*)[];`

```
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>
constexpr span(R & r);
template <ranges::ContiguousRange R>
constexpr span(const R & r)
template <ranges::ContiguousRange R>
requires ranges::View<R> || forwarding-range<R>
constexpr span(R&&)
```

Requires: 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: `size() == size(r) && data() == data(r)`.

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

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

- `R` is not a specialization of `span`,
- `R` is not a specialization of `array`,
- `is_array_v<R>` is `false`,
- `remove_pointer_t<decltype(data(r))>(*)[]` is convertible to `ElementType(*)[]`.

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

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