Range constructor for `std::span`

1 Abstract

This paper proposes that `span` be constructible from any contiguous forwarding-range with a compatible element type. The idea was extracted from P1206.

2 Tony tables

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::vector&lt;int&gt; v(42);</code></td>
<td><code>std::vector&lt;int&gt; v(42);</code></td>
</tr>
<tr>
<td>`std::span&lt;int&gt; foo = v</td>
<td>view::take(3); //ill-formed`</td>
</tr>
<tr>
<td><code>std::vector&lt;int&gt; v(42);</code></td>
<td><code>std::vector&lt;int&gt; v(42);</code></td>
</tr>
<tr>
<td><code>std::span bar(v.begin(), 3); // ill-formed</code></td>
<td><code>std::span bar(v.begin(), 3); // valid</code></td>
</tr>
<tr>
<td><code>std::vector&lt;int&gt; get_vector();</code></td>
<td><code>std::vector&lt;int&gt; get_vector();</code></td>
</tr>
<tr>
<td><code>void foo(std::span&lt;int&gt;);</code></td>
<td><code>void foo(std::span&lt;int&gt;);</code></td>
</tr>
<tr>
<td><code>void bar(std::span&lt;const int&gt;);</code></td>
<td><code>void bar(std::span&lt;const int&gt;);</code></td>
</tr>
<tr>
<td><code>bar(get_vector()); //valid</code></td>
<td><code>bar(get_vector()); //valid</code></td>
</tr>
<tr>
<td><code>foo(get_vector()); //ill-formed</code></td>
<td><code>foo(get_vector()); //ill-formed</code></td>
</tr>
</tbody>
</table>

3 Motivation

`std::span` is specified to be constructible from `Container` types. However, while defined, `Container` is not a concept and as such `contiguous_range` 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.

The intent of span was always to be constructible from a wide number of compatible types, whether standard contiguous containers, non-standard equivalent types, or views. This proposal ensure that
span, especially when used as parameter of a function will be constructible from all compatible
types while offering stronger and more consistent (in regard to Range) lifetime guarantees.

4 Design considerations

We propose to specify all constructors currently accepting a container or pointers in terms of
contiguous\_range and contiguous\_iterator 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 <class It>
constexpr span(pointer ptr It first, size_type count);
constexpr span(pointer first, pointer last);
template <class It, class 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);
template <class R>
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;
```
...
**Effects:** Constructs a span that is a view over the range \([ptr, ptr + count)\). Initializes data_ with to_address(first) and size_ with first + count.

**Ensures:** \(\text{size()} == \text{count} \&\& \text{data()} == \text{ptr}\).

**Throws:** Nothing. When and what to_address(first) throws.

```cpp
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:** \(\text{size()} == \text{last} - \text{first} \&\& \text{data()} == \text{first}\).

**Throws:** Nothing.

```cpp
template <class It, class End>
cconstexpr span(It first, End last);
```

**Constraints:**
- is_convertible_v<remove_reference_t<iter_reference_t<It>>(*)[], element_type(*)[]> is true, [Note: The intent is to allow only qualification conversions of the iterator reference type to element_type — end note],
- It satisfies contiguous_iterator,
- End satisfies sized_sentinel_for<It>, and
- is_convertible_v<End, size_t> is false.

**Expects:**
- If extent is not equal to dynamic_extent, then last - first is equal to extent,
- \([first, last)\) is a valid range,
- It models contiguous_iterator, and
- End models sized_sentinel_for<It>.

**Effects:** Initializes data_ with to_address(first) and size_ with last - first.

**Throws:** When and what to_address(first) throws.

```cpp
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:** \(\text{size()} == \text{N} \&\& \text{data()} == \text{data(arr)}\).

**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 element_type(*)(*)[].

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

**Constraints:**

- extent == dynamic_extent is true,
- 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(*)(*)[].

**Expects:** [data(cont), data(cont) + size(cont)) is a valid range.

**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.

```cpp
template <class R>
constexpr span(R&& r);
```

**Constraints:**

- extent == dynamic_extent is true,
- R satisfies ranges::contiguous_range and ranges::sized_range,
- either R satisfies forwarding-range or is_const_v<element_type> is true,
- remove_cvref_t<R> is not a specialization of span,
- remove_cvref_t<R> is not a specialization of array,
- is_array_v<remove_cvref_t<R>> is false, and
- is_convertible_v<remove_reference_t<ranges::range_reference_t<R>>>(*)[], element_type(*)(*)[] is true [Note: The intent is to allow only qualification conversions of the iterator reference type to element_type — end note].

**Expects:**

- R models ranges::contiguous_range and ranges::sized_range, and
- If is_const_v<element_type> is false, R models forwarding-range.
**Effects:** Initializes `data_` with `ranges::data(r)` and `size_` with `ranges::size(r)`.

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

```cpp
costexpr span(const span& other) noexcept = default;

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

Add a new section `[span.deduction]` to describe the following deduction guides:

```cpp
template <class It, class End>
span(It, End) -> span<remove_reference_t<iter_reference_t<It>>>;

**Constraints:** It satisfies contiguous_iterator.

template<class R>
span(R&&) -> span<remove_reference_t<ranges::range_reference_t<R>>>;

**Constraints:** R satisfies ranges::contiguous_range.
```

## 7 References

[P1419] Casey Carter, Corentin Jabot *A SFINAE-friendly trait to determine the extent of statically sized containers*
https://wg21.link/P1419

[P1391] Corentin Jabot *Range constructor for std::string_view*
https://wg21.link/P1391

[P1474] Casey Carter *Helpful pointers for contiguous_iterator*
https://wg21.link/P1474