Range constructor for \texttt{std::span}

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

\section{Tony tables}

\begin{tabular}{|l|l|}
\hline
Before & After \\
\hline
\texttt{std::vector<int> v(42);} & \texttt{std::vector<int> v(42);} \\
\texttt{std::span<int> foo =} & \texttt{std::span foo = v | view::take(3); //valid} \\
\texttt{v | view::take(3); //ill-formed} & \texttt{v | view::take(3); //ill-formed} \\
\hline
\texttt{std::vector<int> v(42);} & \texttt{std::vector<int> v(42);} \\
\texttt{std::span bar(v.begin(), 3); // ill-formed} & \texttt{std::span bar(v.begin(), 3); // valid} \\
\hline
\texttt{std::vector<int> get_vector();} & \texttt{std::vector<int> get_vector();} \\
\texttt{void foo(std::span<int>);} & \texttt{void foo(std::span<int>);} \\
\texttt{void bar(std::span<const int>);} & \texttt{void bar(std::span<const int>);} \\
\texttt{bar(get_vector()); //valid} & \texttt{bar(get_vector()); //valid} \\
\texttt{foo(get_vector()); //ill-formed} & \texttt{foo(get_vector()); //ill-formed} \\
\hline
\end{tabular}

\section{Motivation}

\texttt{std::span} is specified to be constructible from \texttt{Container} types. However, while defined, \texttt{Container} is not a concept and as such \texttt{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.

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 `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

The following wording assumes `std::to_address` will be specialized for `ContiguousIterator` as proposed in [P1474].

Change in [views.span] 21.7.3:

```cpp
// [span.cons], constructors, copy, and assignment
constexpr span() noexcept;
constexpr span(pointer ptr It begin, index_type count);
constexpr span(pointer first, pointer last);
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(R&& r);
```
constexpr span(const span& other) noexcept = default;

template<class OtherElementType, ptrdiff_t OtherExtent>
constexpr span(const span<OtherElementType, OtherExtent>& s) noexcept;

...

}  

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.

cconstexpr span(pointer ptr, index_type count);

template <class It>
constexpr span(It first, index_type count);

Constraints:

  • ConvertibleTo<remove_reference_t<iter_reference_t<It>>(*)[], element_type(*)[]>() is true. [Note: The intent is to allow qualification conversions of the iterator reference type to element_type — end note]

Expects: [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]
Ensures: `size() == count && data() == ptr to_address(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 <class It, class End>`
`constexpr span(It first, End last);`

Constraints:

- `ConvertibleTo<remove_reference_t<iter_reference_t<It>>(*)[], element_type(*)[]>` is true, [Note: The intent is to allow qualification conversions of the iterator reference type to `element_type` — end note],

- `End models SizedSentinel<It>`.

Expects:

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

- `[first, end)` shall be a valid range.

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

Ensures: `size() == last - first` && `data() == to_address(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 `element_type(*)[]`. 
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.

template <class R>
constexpr span(R&& r)

Constraints:

- extent == dynamic_extent is true,
- R models ranges::ContiguousRange and ranges::SizedRange,
- either R models forwarding-range or is_const_v<element_type> is true,
- R is not a specialization of span,
- R is not a specialization of array,
- is_array_v<R> is false,
-ConvertibleTo<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>(*)[], element_type(*)[]> is true [Note: The intent is to allow qualification conversions of the iterator reference type to element_type — end note].

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

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

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

constexpr 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 models `ranges::ContiguousIterator`,
- End models `SizedSentinel<It>`.

```cpp
template <class It, size_t N>
span(It) -> span<remove_reference_t<iter_reference_t<It>>, N>
```

*Constraints: It models `ranges::ContiguousIterator`.*

```cpp
template<class R>
span(R&&) -> span<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>>
```

*Constraints: R models `ranges::ContiguousRange`.*

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 ContiguousIterator*  
https://wg21.link/P1474