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

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-ranges.

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;
template <ContiguousIterator It>
requiresConvertibleTo<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>
requiresConvertibleTo<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<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>>>(*)[], ElementType(*)[]>
template<ContiguousIterator It, size_t N>
span(It, N) -> span<remove_\_reference_\_t<iter_\_reference_\_t<It>>>(*)[], ElementType(*)[]>

In 21.7.3.2 [span.cons]

constexpr span() noexcept;

\textit{Ensures:} size() == 0 && data() == \texttt{nullptr}.

\textit{Remarks:} This constructor shall not participate in overload resolution unless \texttt{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: \([\text{ptr } \text{first}, \text{ptr } \text{first} + \text{count})\) shall be a valid range. If extent is not equal to \text{dynamic}_\text{extent}, then \text{count} shall be equal to \text{extent}.

Effects: Constructs a \text{span} that is a view over the range \([\text{ptr } \text{first}, \text{ptr } \text{first} + \text{count})\).

Ensures: \text{size()} == \text{count} && \text{data()} == \text{ptr} \text{std::to_address(first)}.

Throws: Nothing.

\text{constexpr span(pointer first, pointer last);}

Requires: \([\text{first}, \text{last})\) shall be a valid range. If extent is not equal to \text{dynamic}_\text{extent}, then \text{last} - \text{first} shall be equal to \text{extent}.

Effects: Constructs a span that is a view over the range \([\text{first}, \text{last})\).

Ensures: \text{size()} == \text{last} - \text{first} && \text{data()} == \text{first}.

Throws: Nothing.

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

\text{Expects: If extent is not equal to \text{dynamic}_\text{extent}, then \text{last} - \text{first} shall be equal to \text{extent}.

\text{Effects: Constructs a span that is a view over the range \([\text{first}, \text{last})\).

\text{Ensures: size()} == \text{last} - \text{first} && \text{data()} == \text{std::to_address(first)}.

\text{Nothing.}

\text{Throws: Nothing.}

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

\text{Effects: Constructs a span that is a view over the supplied array.

\text{Ensures: size()} == N \&\& \text{data()} == \text{data(arr)}.

\text{Remarks: These constructors shall not participate in overload resolution unless:

- extent == \text{dynamic}_\text{extent} \mid \mid N == \text{extent} is true, and
- remove_pointer_t<\text{decltype(data(arr))\*>(\*)[] is convertible to ElementType(*)[].

\text{template<\text{class Container}> \text{constexpr span(Container& cont);}
\text{template<\text{class Container}> \text{constexpr span(const Container& cont);}

\text{Requires: \([\text{data(cont)}, \text{data(cont)} + \text{size(cont)})\) shall be a valid range. If extent is not equal to \text{dynamic}_\text{extent}, then \text{size(cont)} shall be equal to \text{extent}.

\text{Effects: Constructs a span that is a view over the range \([\text{data(cont)}, \text{data(cont)} + \text{size(cont)})\).}
Ensures: \( \text{size()} == \text{size}(\text{cont}) \) \&\& \( \text{data()} == \text{data}(\text{cont}) \).

Throws: What and when \( \text{data}(\text{cont}) \) and \( \text{size}(\text{cont}) \) throw.

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

- \text{Container} is not a specialization of \text{span},
- \text{Container} is not a specialization of \text{array},
- \( \text{is}_\text{array}_\text{v}<\text{Container}> \) is false,
- \( \text{data}(\text{cont}) \) and \( \text{size}(\text{cont}) \) are both well-formed, and
- \( \text{remove\_pointer\_t}<\text{decltype}(\text{data}(\text{cont}))\>(\*)[] \) is convertible to \( \text{ElementType}(\*)[] \).

```cpp
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 \text{extent} is not equal to \text{dynamic\_extent}, then \( \text{size}(\text{r}) \) shall be equal to \text{extent}.

Effects: Constructs a \text{span} that is a view over the range \( \text{r} \).

Ensures: \( \text{ranges}:\text{size()} == \text{ranges}:\text{size}(\text{r}) \) \&\& \( \text{ranges}:\text{data()} == \text{ranges}:\text{data}(\text{r}) \).

Throws: What and when \( \text{ranges}:\text{data}(\text{r}) \) and \( \text{ranges}:\text{size}(\text{r}) \) throw.

Constraints:

- \( \text{R} \) is not a specialization of \text{span},
- \( \text{R} \) is not a specialization of \text{array},
- \( \text{is}_\text{array}_\text{v}<\text{R}> \) is false,

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

Ensures: \( \text{other}:\text{size()} == \text{size()} \) \&\& \( \text{other}:\text{data()} == \text{data()} \).

7 References

[P1419] Casey Carter, Corentin Jabot \textit{A SFINAE-friendly trait to determine the extent of statically sized containers}

\url{https://wg21.link/P1419}

[P1391] Corentin Jabot \textit{Range constructor for std::string\_view}

\url{https://wg21.link/P1391}

[P1474] Casey Carter \textit{Helpful pointers for ContiguousIterator}

\url{https://wg21.link/P1474}