Range constructor for \texttt{std::span}

Document #: P1394R1  
Date: 2019-03-11  
Project: Programming Language C++  
Audience: LWG  
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>  
Casey Carter <casey@carter.net>

1 Abstract

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

2 History

2.1 Revision 1

- Make \texttt{span<const T>} constructible from rvalue ref ranges of T as per LEWG request in Kona
- Typos and Improved wording

3 Tony tables

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{std::vector&lt;int&gt; v(42);}</td>
<td>\texttt{std::vector&lt;int&gt; v(42);}</td>
</tr>
<tr>
<td>\texttt{std::span&lt;int&gt; foo = v</td>
<td>view::take(3); //ill-formed}</td>
</tr>
<tr>
<td>\texttt{std::vector&lt;int&gt; v(42);}</td>
<td>\texttt{std::vector&lt;int&gt; v(42);}</td>
</tr>
<tr>
<td>\texttt{std::span bar(v.begin(), 3); //ill-formed}</td>
<td>\texttt{std::span bar(v.begin(), 3); //valid}</td>
</tr>
<tr>
<td>\texttt{std::vector&lt;int&gt; get_vector();}</td>
<td>\texttt{std::vector&lt;int&gt; get_vector();}</td>
</tr>
<tr>
<td>\texttt{void foo(std::span&lt;int&gt;);}</td>
<td>\texttt{void foo(std::span&lt;int&gt;);}</td>
</tr>
<tr>
<td>\texttt{void bar(std::span&lt;\texttt{const int}&gt;);}</td>
<td>\texttt{void bar(std::span&lt;\texttt{const int}&gt;);}</td>
</tr>
<tr>
<td>\texttt{bar(get_vector()); //valid}</td>
<td>\texttt{bar(get_vector()); //valid}</td>
</tr>
<tr>
<td>\texttt{foo(get_vector()); //ill-formed}</td>
<td>\texttt{foo(get_vector()); //ill-formed}</td>
</tr>
</tbody>
</table>
4  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.

5  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.

6  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.

7  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>
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> || std::is_const_v<ElementType>)
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;
...

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 std::to_address(first).

    Throws: Nothing.

castexpr 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() == std::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 ElementType(*)().

template<class Container> constexpr span(Container& cont);
template<class Container> constexpr span(const Container& cont);
Requires: \([\text{data}\text{(cont)}, \text{data}\text{(cont)} + \text{size}\text{(cont)})\) 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 \([\text{data}\text{(cont)}, \text{data}\text{(cont)} + \text{size}\text{(cont)})\).

Ensures: \(\text{size}() \equiv \text{size}\text{(cont)} \&\& \text{data}() \equiv \text{data}\text{(cont)}\).

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

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

- Container is not a specialization of span,
- Container is not a specialization of array,
- \(\text{is\_array\_v<Container>}\) is false,
- \(\text{data}\text{(cont)}\) and \(\text{size}\text{(cont)}\) are both well-formed, and
- \(\text{remove\_pointer_t<decltype(\text{data}\text{(cont)})>}(\star)\,\square\) is convertible to \(\text{ElementType}(\star)\,\square\).

```
template <ranges::ContiguousRange R>
requires ranges::SizedRange<R> &&
(forwarding-range<R> || std::is_const_v<ElementType>) &&
ConvertibleTo<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>(\star)\,\square, ElementType(\star)\,\square>
constexpr span(R&& r)
```

**Expects:** If extent is not equal to dynamic_extent, then \(\text{size}\text{(r)}\) shall be equal to extent.

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

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

**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 span,
- \(\text{R}\) is not a specialization of array,
- \(\text{is\_array\_v<R>}\) is false,

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

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

8 References

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

5
[P1391] Corentin Jabot *Range constructor for std::string_view*  
https://wg21.link/P1391

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