Usability Enhancements for `std::span`

Tristan Brindle (tcbrindle@gmail.com)

Document Number:	P1024r3
Date:	2019-02-22
Project:	Programming Language C++
Audience:	Library Working Group

1 Introduction

The class template `span<ElementType, Extent>` was recently added to the working draft of the C++ International Standard [N4750]. A `span` is a lightweight object providing a "view" of an underlying contiguous array, which does not own the elements it points to. It is intended as a new "vocabulary type" for contiguous ranges, replacing the use of `(pointer, length)` pairs and, in some cases, `vector<T, A>&` function parameters.

This paper identifies several opportunities to enhance the usability of `span` by improving consistency with existing container interfaces and removing potential points of confusion for users.

An implementation of `span` including the changes detailed in this paper is available at [Github].

1.1 Terminology

For the purposes of this paper, a *fixed-size span* is a `span` whose `Extent` is greater than or equal to zero. A *dynamically-sized span* is a `span` whose `Extent` is equal to `std::dynamic_extent`.

1.2 Revision History

Revision 3

— Further wording tweaks after LWG feedback

Revision 2

— Update wording to reflect LWG feedback

Revision 1

— Update to reflect Rapperwil straw polls:

  — Add `front()` and `back()` member function to `span`? 3 | 7 | 5 | 1 | 0

  — Add `at()` member function? 0 | 0 | 4 | 4 | 5

  — Mark `empty()` `[[nodiscard]]`? *Unanimous consent*

  — Add non-member subview operations? 0 | 0 | 5 | 5 | 2

- Remove operator()? 5 | 12 | 0 | 0 | 0
- Add structured binding support for fixed-size span? *Unanimous consent*

Accordingly, the proposals to add at() and non-member subview operations have been removed from this revision

Revision 0

- Initial revision

2 Proposals

2.1 Add front() and back() member functions

To improve consistency with standard library containers, we propose adding front() and back() member functions with their usual meanings (that is, returning references to the first and last elements respectively). The effect of calling these functions on an empty span is undefined.

2.2 Mark empty() as [[nodiscard]]

The empty() member functions of standard library containers are decorated with the [[nodiscard]] attribute, to make it clearer to users that this function is an observer and does not modify the container state [P0600R1]. For consistency, this paper adds the attribute to span::empty() as well.

2.3 Remove operator()

The current wording for span includes an overload of the function call operator, duplicating the behaviour of operator[]. We assume that this is a holdover from span's genesis as a multidimensional array_view.

Providing this operator for member access is inconsistent with other container types and with built-in language arrays. Furthermore, it provides the mistaken impression that it is possible to "invoke" a span. We therefore propose its removal.

2.4 Structured bindings support for fixed-size spans

Built-in arrays and std::arrays may be used with structured bindings, via core language and library support respectively. To allow function arguments of type T (&)[N] to be replaced by the more appealing span<T, N> with equal functionality, we propose adding support for structured bindings for fixed-size spans. Specifically, we propose a new overload of std::get<N>(), and specialisations of tuple_element and tuple_size for span.

Dynamically-sized spans cannot be decomposed. To prevent this, this proposal declares, but does not define, a partial specialization of tuple_size for dynamically-sized spans:

```
template <class ElementType>
  struct tuple_size<span<ElementType, dynamic_extent>>; // not defined
```

Under the wording for structured bindings ([dcl.struct.bind]/3), making this specialization an incomplete type prevents the language from attempting decomposition via library types.

3 Proposed wording

Changes are relative to [N4750].

In section 19.5.3.6 [tuple.helper], change

> 6. In addition to being available via inclusion of the `<tuple>` header, the three templates are available when any of the headers `<array>`, `<ranges>`, `<span>`, or `<utility>` are included.
>
> 7. ...
>
> 8. In addition to being available via inclusion of the `<tuple>` header, the three templates are available when any of the headers `<array>`, `<ranges>`, `<span>`, or `<utility>` are included.

In section 26.7.2 [span.syn], add

```
// 26.7.X Tuple interface
template<class T> class tuple_size;
template<size_t I, class T> class tuple_element;

template<class ElementType, ptrdiff_t Extent>
  struct tuple_size<span<ElementType, Extent>>;
template <class ElementType>
  struct tuple_size<span<ElementType, dynamic_extent>>;

template<size_t I, class ElementType, ptrdiff_t Extent>
  struct tuple_element<I, span<ElementType, Extent>>;

template<size_t I, class ElementType, ptrdiff_t Extent>
  constexpr ElementType& get(span<ElementType, Extent>) noexcept;
```

In section 26.7.3.1 [span.overview], change

```
// 26.7.3.4, observers
constexpr index_type size() const noexcept;
constexpr index_type size_bytes() const noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// 26.7.3.5, element access
constexpr reference operator[](index_type idx) const;
~~constexpr reference operator()(index_type idx) const;~~
constexpr reference front() const;
constexpr reference back() const;
constexpr pointer data() const noexcept;
```

In section 26.7.3.5 [span.elem], change

> `[[nodiscard]] constexpr bool empty() const noexcept;`
>
> *Effects:* Equivalent to: `return size() == 0;`
>
> `constexpr reference operator[](index_type idx) const;`
> ~~`constexpr reference operator()(index_type idx) const;`~~
>
> *Requires:* `0 <= idx && idx < size()`.
>
> *Effects:* Equivalent to: `return *(data() + idx);`
>
> `constexpr reference front() const`
>
> *Expects:* `empty()` is `false`.
>
> *Effects:* Equivalent to: `return *data();`

```
constexpr reference back() const
```

Expects: `empty()` is `false`.

Effects: Equivalent to: `return *(data() + (size() - 1));`

Add a new subsection [span.tuple]:

```
template <class ElementType, ptrdiff_t Extent>
  struct tuple_size<span<ElementType, Extent>>
      : integral_constant<size_t, static_cast<size_t>(Extent)> { };

template <class ElementType>
  struct tuple_size<span<ElementType, dynamic_extent>>; // not defined
```

```
tuple_element<I, span<ElementType, Extent>>::type
```

Mandates: `Extent != dynamic_extent && I < static_cast<size_t>(Extent)` is `true`.

Value: The type `ElementType`.

```
template <class ElementType, ptrdiff_t Extent>
  constexpr ElementType& get(span<ElementType, Extent> s) noexcept;
```

Mandates: `Extent != dynamic_extent && I < static_cast<size_t>(Extent)` is `true`.

Returns: A reference to the `I`th element of `s`, where indexing is zero-based.

References

[Github] Tristan Brindle. Implementation of C++20 `std::span`. https://github.com/tcbrindle/span, 2018.

[N4750] Richard Smith. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4750.pdf, 2018 (accessed 2018-06-24).

[P0600R1] Nicolai Josuttis. [[nodiscard]] in the library, rev1. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0600r1.pdf, 2017.