Usability Enhancements for `std::span`

Tristan Brindle (tcbrindle@gmail.com)

Document Number:	P1024r0
Date:	2018-05-06
Project:	Programming Language C++
Audience:	Library Evolution Working Group

1 Introduction

The class template `span<ElementType, Extent>` was recently added to the working draft of the C++ International Standard [N4741]. 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`.

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 Add `at()` member function

The `span` proposal paper [P0122R7] makes it clear that implementations are encouraged to add bounds-checking to `span`'s interfaces, and indeed the reference implementation in the Microsoft GSL [GSL] does exactly this. However, bounds-checking is not required by the wording, and it seems highly likely that for effiency reasons such checks (if present) will be disabled in optimised builds in eventual standard library implementations.

In common with other containers then, we propose to add an `at()` member function with guaranteed bounds checking on all implementations even in optimised builds, throwing `std::out_of_range` if the supplied argument is invalid.

2.3 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.

2.4 Add non-member subview operations

The current wording for `span` defines the subview operations (namely `first()`, `last()` and `subspan()`) as member functions. Unfortunately, this introduces a major usability issue. As member templates of a class template, the fixed-size overloads of these functions require the use of the highly unusual "dot template" syntax in cases where the `span` itself is a dependent type. For example:

```
template <typename T>
std::span<T, 3> first_three(std::span<T> s)
{
    return s.template first<3>(); // Urgh!
}
```

Without the keyword `template` here, compilers are required to parse the return expression as an attempted use of less-than, leading to extremely unhelpful error messages in current implementations.

As a vocabulary type, `span` will be used by programmers with varying levels of C++ knowledge, including beginners. It is our experience that the "dot template" syntax is unfamiliar to all but experts. We hesitate to think of the questions that would be generated on Stack Overflow and elsewhere if this syntax were required. For this reason, we propose adding non-member versions of `first()`, `last()` and `subspan()`, which avoids the need for the `template` keyword.

(We note that `std::get<N>()` for `tuple` and `variant`, which conceptually behaves like a member of those classes, is defined as a non-member, likely for exactly this reason.)

A pleasant side-effect of adding non-member versions of these operations is that is becomes possible to create subviews of contiguous containers directly, without first having to construct an intermediate `span` over the whole container. For example:

```
std::vector vec{1, 2, 3, 4, 5};

auto ugly = std::span{vec}.first<3>(); // Current syntax
auto nicer = std::first<3>(vec); // This proposal
```

We have not proposed the removal of the member subview operations in this paper; indeed the wording below forwards the non-member calls to the member versions, using `decltype` returns for SFINAE and to avoid restating the complex `subspan<Offset, Count>()` return type. We do note however that the member versions are specified only to call public `span` constructors, and so in principle the non-member versions could be made "standalone" and the member versions removed if LEWG prefers avoiding the duplication.

2.5 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.

2.6 Structured bindings support for fixed-size spans

Built-in arrays and `std::array`s 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 [N4741].

In section 26.7.2 [span.syn], add

```
// 26.7.X Non-member subview operations
template<ptrdiff_t Count, class Cont>
  constexpr auto first(Cont& cont)
      -> decltype(span{cont}.template first<Count>());
template<ptrdiff_t Count, class Cont>
  constexpr auto last(Cont& cont)
      -> decltype(span{cont}.template last<Count>());
template<ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent, class Cont>
  constexpr auto subspan(Cont cont)
      -> decltype(span{cont}.template subspan<Offset, Count>());

template<class Cont>
  constexpr auto first(Cont& cont, ptrdiff_t count)
      -> decltype(span{cont}.first(count));
template<class Cont>
  constexpr auto last(Cont& cont, ptrdiff_t count)
      -> decltype(span{cont}.last(count));
template<class Cont>
  constexpr auto subspan(Cont& cont, ptrdiff_t offset,
                         ptrdiff_t count = dynamic_extent)
      -> decltype(span{cont}.subspan(offset, count));

// 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 at(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 at(index_type idx) const;
```

Returns: `*(data() + idx);`

Throws: `out_of_range` if `idx < 0 || idx >= size()`

```
constexpr reference front() const
```

Requires: `empty() == false`

Effects: Equivalent to `return *data();`

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

Requires: `empty() == false`

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

Add a new subsection [span.sub.nonmember]:

```
template<ptrdiff_t Count, class Cont>
  constexpr auto first(Cont& cont)
      -> decltype(span{cont}.template first<Count>());
```

Remarks: This function shall participate in overload resolution only if `span{cont}` is well-formed

Requires: `0 <= Count && Count <= span{cont}.size()`

Effects: Equivalent to `return span{cont}.template first<Count>();`

```
template<ptrdiff_t Count, class Cont>
  constexpr auto last(Cont& cont)
      -> decltype(span{cont}.template last<Count>());
```

Remarks: This function shall participate in overload resolution only if `span{cont}` is well-formed

Requires: `0 <= Count && Count <= span{cont}.size()`

Effects: Equivalent to `return span{cont}.template last<Count>();`

```
template<ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent, class Cont>
  constexpr auto subspan(Cont cont)
      -> decltype(span{cont}.template subspan<Offset, Count>());
```

Remarks: This function shall participate in overload resolution only if `span{cont}` is well-formed

Requires:

```
  (0 <= Offset && Offset <= span{cont}.size())
  && (Count == dynamic_extent || Count >= 0 && Offset + Count <= span{cont}.size())
```

Effects: Equivalent to `return span{cont}.template subspan<Offset, Count>();`

```
template<class Cont>
  constexpr auto first(Cont& cont, ptrdiff_t count)
      -> decltype(span{cont}.first(count));
```

Remarks: This function shall participate in overload resolution only if `span{cont}` is well-formed

Requires: `0 <= count && count <= span{cont}.size()`

Effects: Equivalent to `return span{cont}.first(count);`

```
template<class Cont>
  constexpr auto last(Cont& cont, ptrdiff_t count)
      -> decltype(span{cont}.last(count));
```

Remarks: This function shall participate in overload resolution only if `span{cont}` is well-formed

Requires: `0 <= count && count <= span{cont}.size()`

Effects: Equivalent to `return span{cont}.last(count);`

```
template<class Cont>
  constexpr auto subspan(Cont cont, ptrdiff_t offset,
                         ptrdiff_t count = dynamic_extent)
      -> decltype(span{cont}.subspan(offset, count));
```

Remarks: This function shall participate in overload resolution only if `span{cont}` is well-formed

Requires:

```
    (0 <= offset && offset <= span{cont}.size())
    && (count == dynamic_extent || count >= 0 && offset + count <= span{cont}.size())
```

Effects: Equivalent to `return span{cont}.subspan(offset, count);`

Add a new subsection [span.tuple]:

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

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

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

Requires: `Extent != dynamic_extent && I < static_cast<size_t>(Extent)`. The program is ill-formed if `I` is out of bounds.

Value: The type ElementType.

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

Requires: `Extent != dynamic_extent && I < static_cast<size_t>(Extent)`. The program is ill-formed if `I` is out of bounds.

Returns: A reference to the $\texttt{I}^{\text{th}}$ element of `s`, where indexing is zero-based.

Throws: Nothing

References

[GSL] Microsoft Corporation. Guidelines Support Library. https://github.com/Microsoft/GSL, 2018.

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

[N4741] Richard Smith. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4741.pdf, 2018 (accessed 2018-04-20).

[P0122R7] Neil MacIntosh and Stephan T. Lavavej. span: bounds-safe views for sequences of objects. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0122r7.pdf, 2018.

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