

span should have size_type, not index_type

Document number: P1872R0

Date: 2019-09-16

Project: Programming Language C++, Library Evolution Working Group

Reply-to: Nevin “☺” Liber, nliber@anl.gov

Table of Contents

Introduction	1
Motivation and Scope	1
Impact on the Standard	2
Design Decisions	2
Technical Specifications	2
[span.overview]	2
[span.cons].....	4
[span.sub].....	4
[span.obs].....	4
[span.elem].....	5
Acknowledgements	5
References	5

Introduction

[P1227R2](#) changed the size and indexing operations in span from the signed type `ptrdiff_t` to the unsigned type `size_t`. The typedef should be changed from `index_type` to `size_type` to be consistent and interoperable with the rest of the standard library.

Motivation and Scope

This paper is the proposed change for a US NB comment pertaining to the [C++20CD](#).

`index_type` was invented for span back when its size and indexing operations were the signed `ptrdiff_t`. Now that they are the *unsigned* `size_t`, the typedef should be changed from `index_type` to `size_type` to be consistent and interoperable with the rest of the standard library.

Impact on the Standard

This proposal alters the specification of `span`, which is in the [C++20CD](#), but it has not yet been published.

Design Decisions

Another possibility is to add `size_type` as an additional typedef. However, having two names is strictly worse than having one name, as developers will use them interchangeably. There is no reason to keep `index_type`, as it is not used by anything else in the standard library.

Technical Specifications

All changes are relative to the [C++20CD](#):

[\[span.overview\]](#)

```
template<class ElementType, size_t Extent = dynamic_extent>
class span {
public:
    // constants and types
    using element_type = ElementType;
    using value_type = remove_cv_t<ElementType>;
    using index_type size_type = size_t;
    using difference_type = ptrdiff_t;
    using pointer = element_type*;
    using const_pointer = const element_type*;
    using reference = element_type&;
    using const_reference = const element_type&;
    using iterator = implementation-defined; // see [span.iterators]
    using const_iterator = implementation-defined;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    static constexpr index_type size_type extent = Extent;

    // [span.cons], constructors, copy, and assignment
    constexpr span() noexcept;
    constexpr span(pointer ptr, index_type size_type count);
    constexpr span(pointer first, pointer 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(const span& other) noexcept = default;
template<class OtherElementType, size_t OtherExtent>
    constexpr span(const span<OtherElementType, OtherExtent>& s)
noexcept;

~span() noexcept = default;

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

// [span.sub], subviews
template<size_t Count>
    constexpr span<element_type, Count> first() const;
template<size_t Count>
    constexpr span<element_type, Count> last() const;
template<size_t Offset, size_t Count = dynamic_extent>
    constexpr span<element_type, see below> subspan() const;

    constexpr span<element_type, dynamic_extent>
first(index_type size_type count) const;
    constexpr span<element_type, dynamic_extent>
last(index_type size_type count) const;
    constexpr span<element_type, dynamic_extent> subspan(
        index_type size_type offset, index_type size_type count =
dynamic_extent) const;

// [span.obs], observers
constexpr index_type size_type size() const noexcept;
constexpr index_type size_type size_bytes() const noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// [span.elem], element access
constexpr reference operator[](index_type size_type idx) const;
constexpr reference front() const;
constexpr reference back() const;
constexpr pointer data() const noexcept;

// [span.iterators], iterator support
constexpr iterator begin() const noexcept;
constexpr iterator end() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

friend constexpr iterator begin(span s) noexcept { return
s.begin(); }
friend constexpr iterator end(span s) noexcept { return s.end();
}

```

```
private:
    pointer data_; // exposition only
    index_type size_; // exposition only
};
```

[\[span.cons\]](#)

```
constexpr span(pointer ptr, index_type count);
```

Expects: [ptr, ptr + count) is a valid range.

If extent is not equal to dynamic_extent, then count is equal to extent.

Effects: Constructs a span that is a view over the range [ptr, ptr + count).

Ensures: size() == count && data() == ptr.

Throws: Nothing.

[\[span.sub\]](#)

```
constexpr span<element_type, dynamic_extent> first(index_type
count) const;
```

Expects: count <= size() is true.

Effects: Equivalent to: return {data(), count};

```
constexpr span<element_type, dynamic_extent> last(index_type
count) const;
```

Expects: count <= size() is true.

Effects: Equivalent to: return {data() + (size() - count), count};

```
constexpr span<element_type, dynamic_extent> subspan(
    index_type offset, index_type count =
dynamic_extent) const;
```

Expects:

```
offset <= size() && (count == dynamic_extent || offset + count <=
size())
```

is true.

Effects: Equivalent to:

```
return {data() + offset, count == dynamic_extent ? size() - offset :
count};
```

[\[span.obs\]](#)

```
constexpr index_type size() const noexcept;
```

Effects: Equivalent to: return size_;

```
constexpr index_type size_bytes() const noexcept;
```

Effects: Equivalent to: return size() * sizeof(element_type);

[\[span.elem\]](#)

```
constexpr reference operator[] (index_type size_type idx) const;
```

Expects: `idx < size()` is `true`.

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

Acknowledgements

Thanks to JeanHeyd Meneide for bringing up this issue as well as reviewing this along with Niall Douglas.

This was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Additionally, this research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[N4830](#) (C++20CD) – Working Draft, Standard for Programming Language C++

[P1227R2](#) - Signed `ssize()` functions, unsigned `size()` functions (Jörg Brown)