Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **D2276R0**
Date: 2020-12-20 12:00CET
Reply to: Nicolai Josuttis (nico@josuttis.de)

Audience: LEWG, LWG
Issues: lwg3320

Fix std::cbegin(), std::ranges::cbegin, provide const_iterator support for std::span, Rev0

This paper fixes **`std::cbegin()`** and **`std::cend()`** to avoid that **`std::cbegin(coll)`** could yield something different than **`coll.cbegin()`**.

In the same way, the paper fixes the corresponding ranges customization points **`std::ranges:cbegin`** and so on.

With this fix, we no longer have the problem of http://wg21.link/lwg3320, which caused to remove **`const_iterator`** and **`cbegin()`** support from **`std::span`**. So, this paper also introduces missing **`const_iterator`** and **`cbegin()`**/**`cend()`** support back to std::span.

Note that the paper does not provide missing **`cbegin()`** members for several range factories and range adaptors. For them, using **`std::cbegin()`** and **`std::ranges::cbegin`** might still provide write access to elements: The problem is not getting worse with all the fixes of this paper, but needs careful consideration and should be handled in a separate paper.

Tony Table:

Before	After

`std::vector<int> coll{1, 2, 3, 4, 5};`
`std::span<int> sp{coll, 2};`
`for (auto pos = std::cbegin(sp);`
`pos != std::cend(sp); ++pos) {`
`*pos = 42;` // is no error (but it should be)
`}`
	`std::vector<int> coll{1, 2, 3, 4, 5};`
	`std::span<int> sp{coll, 2};`
	`for (auto pos = std::cbegin(sp);`
	`pos != std::cend(sp); ++pos) {`
	`*pos = 42;` // is error
	`}`

`std::vector<int> coll{1, 2, 3, 4, 5};`
`std::view<int> v{coll, 2};`
 	
`std::is_same_v<decltype(v.cbegin()),`
`decltype(std::cbegin(v))>`
// if valid, may be false
	`std::vector<int> coll{1, 2, 3, 4, 5};`
	`std::view<int> v{coll, 2};`
	
	`std::is_same_v<decltype(v.cbegin()),`
	`decltype(std::cbegin(v))>`
	// if valid, always true

`std::span coll{…};`
// read-only iteration over elements:
`typename std::decay_t<decltype(coll)>`
`::const_iterator cpos;`
`for (cpos = coll.begin();`
`cpos != coll.end(); ++cpos) {`
	`std::span coll{…};`
	// read-only iteration over elements:
	`for (auto cpos = coll.cbegin();`
	`cpos != coll.cend(); ++cpos) {`
	`process(*pos);`
	`}`

`  process(*pos);` `}`	

Rev0:

First initial version.

Motivation to fix std::cbegin() etc.

The current specification of **`std::cbegin()`** calls **`std::begin()`** for a const object, which by default calls the member function **`begin()`**. If the member function **`begin()`** for const objects returns the same as the member function **`cbegin()`** (usually a **`const_iterator`**), everything works fine:

```
std::vector<int> coll;

assert(( std::is_same_v<decltype(coll.cbegin()),
                        decltype(std::cbegin(coll))> )); // does not fail
```

However, for classes where members **`cbegin()`** return a different type than **`begin()`** for const objects, **`std::cbegin(c)`** does not yield the same type as **`c.cbegin()`**.

Containers, ranges, and sequences that have reference semantics fall into this problem. And this was reveals when proposing std::span for C++20:

- A **`span<const int>`** means that the span can be modified but refers to const elements.
- A **`const span<int>`** means that the span can't be modified but the elements are not const.

This works according to the reference semantics of pointers and iterators. And to deal with that, we have introduced `const_iterator`'s to express that the iterator can be modified while the element it refers to can't.

So with std::span<> as it was originally proposed for C++20 we got the following problem:

```
std::vector<int> coll{1, 2, 3, 4, 5};

std::span<int> sp{coll, 2};


assert(( std::is_same_v<decltype(sp.cbegin()),
                        decltype(std::cbegin(sp))> )); // fails
```

Because a const span does not delegate constness to its elements.

The problem was raised with http://wg21.link/lwg3320 and the consequence was that **`const_iterator`** and **`cbegin()`** support was (temporarily) removed from std::span (see the resolution of the issue).

However, **`std::cbegin(c)`** should never yield something different than **`c.cbegin()`**.

Proposed fix for std::cbegin() and std::ranges::cbegin() etc.

Fix for std::cbegin() and std::cend()

The first fix proposed in this paper is to modify the current definition of **`std::cbegin(c)`**:

- If **`c`** supports **`std::begin(c)`**, we call it

It should first try to call a cbegin() meber before it falls back to the current behavior:

- If **`c`** supports **`c.cbegin()`**, we call it
- Otherwise, if **`c`** supports **`std::begin(c)`**, we call that

This means that std::cbegin(c) always does the same as c.cbegin() if the member function is provided.

`std::cend()` should be fixed accordingly.

Fix for std::crbegin() and std::crend()

The current definition of **`std::crbegin(c)is as follows`**:

- If **`c`** supports **`std::rbegin(c)`**, we call it

However, here we have we have the following options for a fix:

a) According to std::crbegin() prefer to call a crbegin() member function:

- If **`c`** supports **`c.crbegin()`**, we call it
- Otherwise, if **`c`** supports **`std::rbegin(c)`**, we call that

b) Prefer also to call make_reverse_iterator() using cbegin() and cend() members:

- If **`c`** supports **`c.crbegin()`**, we call it
- Otherwise, if **`c.cend()`** is valid (and a bidirectional iterator),
  `call make_reverse_iterator(c.cend())`
- Otherwise, if **`c`** supports **`std::rbegin(c)`**, we call that

c) Prefer also to call make_reverse_iterator() using std::cbegin() and std::cend():

- If **`c`** supports **`c.crbegin()`**, we call it
- Otherwise, if **`std::cend(c)`** is valid (and a bidirectional iterator),
  `call make_reverse_iterator(std::cend(c))`
- Otherwise, if **`c`** supports **`std::rbegin(c)`**, we call that

d) Also strike the fallback to std::rbegin():

- If **`c`** supports **`c.crbegin()`**, we call it
- Otherwise, if **`std::cend(c)`** is valid (and a bidirectional iterator),
  `call make_reverse_iterator(std::cend(c))`

~~- If **c** supports **`std::rbegin(c)`**, we call that~~

Again, in all cases, if a class provides a member crbegin(), std::crbegin() uses it.

However, b), c), and c) would also provide reverse iterators with constness for the elements if classes with reference semantics provide cbegin() but not crbegin().

The difference is that

`std::crend()` should be fixed accordingly.

Fix for std::ranges::cbegin(), std::ranges::crbegin() etc.

We propose also to fix **`std::ranges::cbegin`**, **`std::ranges::cend`**, **`std::ranges::crbegin`**, and **`std::ranges::crend`** accordingly.

Bringing back const_iterator support to std::span

With that fix we propose to bring back const_iterator support to std::span<>.

That means that we in fact revert the proposed resolution of http://wg21.link/lwg3320 and add the following members back to std::span:

- Type **`const_iterator`**
- Type **`const_reverse_iterator`**
- **`cbegin() const`**
- **`cend() const`**
- **`crbegin() const`**
- **`crend() const`**

Q&A

Do we have evidence that this is a major problem in practice?

Without **`const_iterator`** support we can't iterate safely over a non-const span/view having the guarantee that we don't modify the elements:

```
template<typename T>
void foo1(T&& coll)
{
  // read-only iteration over elements:
  for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    process(*pos);   // may modify elements
  }
}

template<typename T>
void foo2(T&& coll)
{
  // read-only iteration over elements:
  for (auto pos = std::cbegin(coll); pos != std::end(coll); ++pos) {
    process(*pos);   // may modify elements
  }
}

template<typename T>
void foo3(T&& coll)
{
  // read-only iteration over elements:
  for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
    process(*pos);   // OK, but requires cbegin() and cend() support
  }
}

template<typename T>
void foo4(T&& coll)
{
  // read-only iteration over elements:
  for (typename std::decay_t<decltype(coll)>::const_iterator
       pos = coll.begin(); pos != coll.end(); ++pos) {
    process(*pos);   // OK, but requires const_iterator support
  }
```

```
}
```

Note that especially foo2() is a severe violation of the principles and naive understanding of what using cbegin() and cend() does (it is breaking logical const correctness).

Also note that **`std::as_const()`** does not help here, because again it only makes the container/iterator const, not the elements.

Proposed Wording

(All against N4861)

Proposed Wording for std::cbegin etc.

In **23.2 Header <iterator> synopsis [iterator.synopsis]:**

Change

```
template<class C>
constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
-> decltype(std::begin(c));
template<class C>
constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));
```

and

```
template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));
```

To

```
template<class C> constexpr requires see below auto cbegin(const C& c) noexcept(see below)
  -> see below;
template<class C> constexpr requires see below auto cend(const C& c) noexcept(see below)
  -> see below;
```

and

```
template<class C> constexpr requires see below auto crbegin(const C& c)
  -> see below;
template<class C> constexpr requires see below auto crend(const C& c)
  -> see below;
```

In **23.7 Range access [iterator.range]:**

Change:

template<class C> constexpr auto cbegin(const C& c)
noexcept(noexcept(std::begin(c))) -> decltype(std::begin(c));

6 *Returns:* std::begin(c).

template<class C> constexpr auto cend(const C& c)
noexcept(noexcept(std::end(c))) -> decltype(std::end(c));

7 *Returns:* std::end(c).

And:

template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));

14 *Returns:* std::rbegin(c).

template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));

15 *Returns:* std::rend(c).

To:

template<class C> `requires` *see below*
constexpr auto cbegin(const C& c)

noexcept(*see below*) -> *see below*;

6 *Effects:*
- If c.cbegin() is a valid expression then expression-equivalent to c.cbegin()
- Otherwise, if begin(c) is a valid expression then expression-equivalent to begin(c)
- Otherwise, ill-formed.

template<class C> `requires` *see below*
constexpr auto cend(const C& c)

noexcept(*see below*) -> *see below*;

7 *Effects:*
- If c.cend() is a valid expression then expression-equivalent to c.cend()
- Otherwise, if end(c) is a valid expression then expression-equivalent to end(c)
- Otherwise, ill-formed.

And:

template<class C> `requires` *see below*

constexpr auto crbegin(const C& c) -> *see below*;

14 *Effects:*
- If c.crbegin() is a valid expression then expression-equivalent to c.crbegin()
- Otherwise, if rbegin(c) is a valid expression then expression-equivalent to rbegin(c)
- Otherwise, ill-formed.

template<class C> `requires` *see below*

constexpr auto crend(const C& c) -> *see below*;

15 *Effects:*
- If c.crend() is a valid expression then expression-equivalent to c.crend()
- Otherwise, if rend(c) is a valid expression then expression-equivalent to rend(c)
- Otherwise, ill-formed.

Proposed Wording for std::ranges::cbegin etc.

In **24.3.3 ranges::cbegin [range.access.cbegin]:**

Fix as follows:

The name ranges::cbegin denotes a customization point object (16.4.2.2.6).
The expression ranges::cbegin(E) for a subexpression E of type T is expression-equivalent to:
(1.1) — If *decay-copy*(t.cbegin()) is a valid expression whose type models input_or_output_iterator, ranges::cbegin(E) is expression-equivalent to *decay-copy*(t.cbegin()).

(1.2) — Otherwise, ranges::begin(static_cast<const T&>(E)) if E is an lvalue.

(1.3) — Otherwise, ranges::begin(static_cast<const T&&>(E)).

In **24.3.4 ranges::cend [range.access.cend]:**

Fix as follows:

> The name ranges::cend denotes a customization point object (16.4.2.2.6).
> The expression ranges::cend(E) for a subexpression E of type T is expression-equivalent to:
>
> (1.1) —If *decay-copy*(t.cend()) is a valid expression whose type models sentinel_for<iterator_t<T>> then ranges::cend(E) is expression-equivalent to *decay-copy*(t.cend()).
>
> (1.2) — Otherwise, ranges::end(static_cast<const T&>(E)) if E is an lvalue.
>
> (1.3) — Otherwise, ranges::end(static_cast<const T&&>(E)).

In **24.3.7 ranges::crbegin [range.access.crbegin]:**

Fix as follows:

> 1 The name ranges::crbegin denotes a customization point object (16.4.2.2.6).
> The expression ranges::crbegin(E) for a subexpression E of type T is expression-equivalent to:
>
> (1.1) — If *decay-copy*(t.crbegin()) is a valid expression whose type models input_or_output_iterator, ranges::crbegin(E) is expression-equivalent to *decay-copy*(t.crbegin()).
>
> (1.3) — Otherwise, ranges::rbegin(static_cast<const T&>(E)) if E is an lvalue.
>
> (1.3) — Otherwise, ranges::rbegin(static_cast<const T&&>(E)).

In **24.3.8 ranges::crend [range.access.crend]:**

Fix as follows:

> 1 The name ranges::crend denotes a customization point object (16.4.2.2.6).
> The expression ranges::crend(E) for a subexpression E of type T is expression-equivalent to:
>
> (1.1) — If *decay-copy*(t.crend()) is a valid expression whose type models sentinel_for<decltype(ranges::rbegin(E))> then ranges::crend(E) is expression-equivalent to *decay-copy*(t.crend()).
>
> (1.2) — Otherwise, ranges::rend(static_cast<const T&>(E)) if E is an lvalue.
>
> (1.3) — Otherwise, ranges::rend(static_cast<const T&&>(E)).

Proposed Wording for std::span

This fix reverts the overload resolution of http://wg21.link/lwg3320

In **22.7.3.1 Overview [span.overview]:**

Fix as follows:

```
namespace std {
 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 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 22.7.3.7
using const_iterator = implementation-defined;
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
static constexpr size_type extent = Extent;
…
// 22.7.3.7, 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;
```

In **22.7.3.7 Iterator support [span.iterators]:**

Modify as follows:

using iterator = *implementation-defined* ;

using const_iterator = *implementation-defined* ;

1 The types model~~s~~ contiguous_iterator (23.3.4.14), meet~~s~~ the *Cpp17RandomAccessIterator* requirements (23.3.5.6), and meet~~s~~ the requirements for constexpr iterators (23.3.1). All requirements on container iterators (22.2) apply to span::iterator and span::const_iterator as well.

constexpr const_iterator cbegin() const noexcept;

-6- Returns: A constant iterator referring to the first element in the span. If empty() is true, then it returns the same value as cend().

constexpr const_iterator cend() const noexcept;

-7- Returns: A constant iterator which is the past-the-end value.

constexpr const_reverse_iterator crbegin() const noexcept;

-8- Effects: Equivalent to: return const_reverse_iterator(cend());

constexpr const_reverse_iterator crend() const noexcept;

-9- Effects: Equivalent to: return const_reverse_iterator(cbegin());

Feature Test Macro

New macro or do we have a versioned macro?

One or multiple feature test macros (cbegin fix, span fix, ranges fix)?

Acknowledgements

Thanks to a all the people who discussed the issue, proposed information, and helped with possible wording. Especially: The people in the C++ library (evolution) working group, Niall Douglas, Ville Voutilainen.

Forgive me if I forgot anybody.