Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P2276R0**
Date: 2021-01-15
Reply to: Nicolai Josuttis (nico@josuttis.de)

Audience: LEWG, LWG
Issues: lwg3320

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

Currently, if a class provides **`cbegin()`** and **`cend()`** members, these members are neither called by **`std::cbegin()`** and **`std::cend()`** nor by **`std::ranges::cbegin()`** and **`std::ranges::cend()`**. This means that these functions may not provide ready-only access to elements, which is clearly against the whole purpose of **`cbegin()`** and **`cend()`**.

This problem was revealed by http://wg21.link/lwg3320, which, however, caused a reaction that made the situation even worse: **`const_iterator`** and **`cbegin()`** support was removed from **`std::span`** so that programmers now no longer are able to iterate read-only over the elements (instead of calling **`sp.cbegin()`** they can only use **`std::cbegin(sp)`** or **`std::ranges::cbegin(sp)`**, which does not provide read-only access).

This paper proposes to fix the situation so that everything works as expected:

- Let **`std::cbegin()`** and **`std::ranges::cbegin()`** call **`cbegin()`** members if available (same for all other **`c`**... functions).
- Bring support for **`const_iterator`**, **`cbegin()`** members, etc. back to std::span so that it is possible to iterate read-only over its elements.

This paper does ***not*** propose in any form to bring **`const_iterator`** support to ranges or views. There is probably another paper fixing that.

The reason to fix std::span is because we have a special case here: it is a view, but it is it is not part of the ranges library (i.e., not in the ranges sub-namespace). std::span is more like std::string_view and should therefore provide the same API. Note that there are already bug reports because of the last-minute removal of const iterator support for std::span.

Rev0:

First initial version.

Tony Table:

Before	After

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

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

`// can't use cbegin() members for spans:`
`std::vector<int> coll{1, 2, 3, 4, 5};`
`std::span<int> sp{coll.begin(), 2};`
`for (auto it = sp.cbegin();  // ERROR`
`  it != sp.cend(); ++it) {`
`  ...`
`}`
	`std::vector<int> coll{1, 2, 3, 4, 5};`
	`std::span<int> sp{coll.begin(), 2};`
	`for (auto it = sp.cbegin();  // OK`
	`  it != sp.cend(); ++it) {`
	`  ... // read-only access to elements`
	`}`

`view v;  // some view with cbegin() members`
 	
`std::is_same_v<decltype(v.cbegin()),`
`  decltype(std::cbegin(v))>`
`  // if valid, may be false`
	`view v;  // some view with cbegin() members`
	
	`std::is_same_v<decltype(v.cbegin()),`
	`  decltype(std::cbegin(v))>`
	`  // if valid, always true`

`// generic code ensuring cbegin() members are called:`
`if constexpr (requires { rg.cbegin() }) {`
`  for (auto it = rg.cbegin();`
`  it != rg.cend(); ++it) {`
`  ...`
`  }`
`}`
`else {`
`  for (auto it = std::ranges::cbegin(rg);`
`  it != std::ranges::cend(rg);`
`  ++it)`
`  ...`
`  }`
`}`
	`// generic code ensuring that cbegin() members are called:`
	`for (auto it = std::ranges::cbegin(rg);`
	`  it != std::ranges::cend(rg);`
	`  ++it)`
	`  ...`
	`}`

History

Purpose of cbegin()

In the C++ standard, "*constant iterators*" are defined as nonmutable iterators that are **no output iterators** (being able to write). For their support we introduced:

- With C++98 **`const_iterator`** defined as:

  "iterator type pointing to const T"

- With C++11 we changed **`const_iterator`** to be:

  "constant iterator type whose value type is T"

`cbegin()` members were introduced in 2005 (formally supported since C++11) as follows:

Motivation:

> "when a container traversal is intended for inspection only, it is a generally preferred practice to use a const_iterator in order to permit the compiler to diagnose const-correctness violations".

Therefore **`cbegin()`** members were proposed so that

> "***a programmer can directly obtain a `const_iterator` from even a non-const container***."

It is an important common design goal for C++ that you can use APIs in slightly different contexts and if they serve the same purpose they should have the same behavior. Therefore de-facto, we introduced a concept that does not only apply to containers.

If provided, **`const_iterator`** and **`cbegin()`**/**`cend()`** members are a useful way for all sequences "***in order to permit the compiler to diagnose const-correctness violations***".

In the C++ standard we already have **`cbegin()`** members also for

- **`match_results`**
- **`basic_string_view`**
- **`tzdb_list`**

and we planned to have it for

- **`span`**

Purpose of std::cbegin()

`std::cbegin()` was added with LWG2128 (so it is formally supported since C++14).

Its goal always was that is has the same effect as calling the corresponding members if they are available:

> "Step 1: Implement **`std::cbegin`**/**`cend()`** by calling **`std::begin`**/**`end()`**.
> ...
> [container.requirements.general] guarantees that this is **equivalent to calling cbegin/cend() members**."

And it also was a **clear intention of std::cbegin() not only to work for containers**:

> "It automatically works with arrays,
>
> ...
>
> It works with initializer_list,"

Why std::cbegin() and std::ranges::cbegin are broken

Until C++17, everything was fine, because as long as the members

begin() const

and

cbegin() const

yield the same type, the design goal of **`std::cbegin(c)`** matching any **`c.cbegin()`** was fulfilled.

But there are useful sequence types where providing **`const_iterator`** and **`cbegin()`** might makes sense, but design goal would be broken.

In general, for sequences with **reference semantics and shallow constness** it is useful to specify:

```
iterator begin() const;          // const container doesn't mean const elements
const_iterator cbegin() const;   // but cbegin() still provides constant iterators
```

Since C++20, we started to have sequence types with reference semantics and shallow constness in the standard:

- std::span
- Several views

Realizing that the design goal of **`cbegin()`** and **`std::cbegin()`** was broken, http://wg21.link/lwg3320 was raised.

Unfortunately the resolution didn't solve the problem. The situation got even worse.

Now in C++20 as specified, if a programmer wants to use a **`const_iterator`** in order to permit the compiler to diagnose const-correctness violations, he/she can no longer use **`cbegin()`** members. And using **`std::cbegin()`** or **`std::ranges::cbegin()`** still does not work. Making the container **`const`** does also not work and a **`const_iterator`** we could convert to is also not defined.

Thus, **there is no easy way to iterate read-only over the elements of a span.**

Therefore, this paper proposes to ensure that for any collection/container/range/view if the programmer provides **`cbegin()`** members, **`std::cbegin()`** and **`std::ranges::cbegin()`** call them or do the equivalent thing.

Everything else leads to significant confusion of application programmers.

Why not only providing a mechanism to yield const iterators automatically?

There is another solution proposed by Barry Revzin.

It provides a solution for all situations, where **`cbegin()`** members are not provided and should be adopted independent from this paper.

However, if a container provides **`cbegin()`** members (for whatever reason) it still breaks the design goal of **`std::cbegin()`** being equivalent to calling the member, because it might yield a constant iterator but of different type. This especially applies to all sequences with reference semantics and shallow constness currently providing **`cbegin()`** (which they have to to work properly).

This for sure creates confusion. But even worse: If the types are different then code that uses **`std::cbegin(c)`** is no longer equivalent to code that uses **`c.cbegin()`**.

This has the following consequences:

a) Functions that use both API's to return something no longer compile:

```
auto foo() {            // ERROR: can't deduce return type
  return std::cbegin(c);  // may happen indirectly
  ...
  return c.cbegin();      // may happen indirectly
}
```

b) Functions that require the same type, no longer compile:

```
auto pos1 = cfind1(coll);     // might use std::cbegin(c) or use std::ranges::cbegin(c)
auto pos2 = cfind2(coll);     // might use c.cbegin()
std::distance(pos1, pos2);    // ERROR
pos1 - pos2                   // ERROR
```

c) If for whatever reason a **`const_iterator`** provides a different API than an iterator:

```
coll.cbegin().foo();          // might compile
std::cbegin(coll).foo();      // might not compile
```

or vice versa.

a) and b) is likely to happen in practice (we would have it with std::span supporting const_iterator again).

c) might not be on the agenda right now, but there are scenarios where this might be useful (e.g., proxy iterators providing different proxy types for the element access).

So, the obvious proposal is to always call corresponding members if available.

Otherwise any other solution (existing or proposed in other papers) can be used.

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 **`c.cbegin()`**, we call it

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

- If **`c`** supports **`c.cbegin()`**, we call it
- Otherwise, ...

Here, "..." might be the current wording or any other wording provided by other papers to automatically provide const iterators).

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~~

However, a possible fix for reverse iterators is not the purpose of this paper.

Other papers should do it.

So, I propose, that std::rbegin() should also first try to call a crbegin() member before it falls back to the current behavior:

- If **c** supports **c.crbegin()**, we call it
- Otherwise, ...

std::crend() should be fixed accordingly.

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

This paper proposes also to fix **std::ranges::cbegin**, **std::ranges::cend**, **std::ranges::crbegin**, and **std::ranges::crend** accordingly.

For example, for std::ranges::cbegin():

> The name ranges::cbegin denotes a customization point object (16.3.3.3.6). The expression ranges::cbegin(E) for a subexpression E of type T is expression-equivalent to:
>
> (1.1) — E.cbegin() if this is valid.
>
> (1.2) — ranges::begin(static_cast<const T&>(E)) if E is an lvalue.
>
> (1.3) — Otherwise, ranges::begin(static_cast<const T&&>(E)).

Bringing back const_iterator support to std::span

With that fix we propose to bring back constant 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?

We already get bug reports about this problem:

> "I've just received a bug report on one of my open source libraries
> caused by span<T>::const_iterator no longer existing."

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 models contiguous_iterator (23.3.4.14), meets the *Cpp17RandomAccessIterator* requirements (23.3.5.6), and meets 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 all the people who discussed the issue, proposed information, and helped with possible wording. Especially: The people in the C++ library (evolution) working group, Walter E. Brown, Niall Douglas, Alisdair Meredith, Barry Revzin, Ville Voutilainen.

Forgive me if I forgot anybody.