Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P2276R1**
Date: 2021-09-09
Reply to: Nicolai Josuttis (nico@josuttis.de)

Audience: LEWG, LWG
Issues: lwg3320

Fix `cbegin`, Rev1

This paper proposes a few simple changes to solve the problem that cbegin is currently broken. In contrast to Rev0 of this paper it does not propose to fix `cbegin()`. Instead, the goal is to disable broken constness and enable to implement workarounds.

Motivation

The only reason `cbegin()` and `cend()` are provided is to ensure that you can't (accidentally) modify the elements of a collection you iterate over.

With the current specification this was guaranteed assuming that the collection you iterate over has *deep constness*, which means that the collection propagates its constness to its elements (if the collection is `const` the elements are also `const`).

But since C++20 the C++ standard provides container/range types with *shallow constness*. For types such as `std::span` or views, you can still modify the elements even when the container/range is declared to be `const`.

For these types with shallow constness generic code `std::cbegin()`, `std::cend()`, etc. is broken: It always calls `begin()` and `end()` members (even if `cbegin()` and `cend()` members are provided) so that you still can modify elements although you iterate with `cbegin()` and `cend()`.

This is clearly a bug.

Ideally, we should fix this bug, but there is significant resistance to do so as e.g. proposed in P2276R0 and P2278R0 (it is not clear yet how to ideally fix it, some people they like to benefit from this buggy behavior, and some people don't think breaking constness is important to deal with).

For more background and motivation see http://wg21.link/P2276R0 and http://wg21.link/P2278R0.

Proposed Solution

For this reason, this paper proposes the best we can do to minimize the consequences of keeping the buggy behavior alive:

- **A)** Provide `cbegin()` and `cend()` members for `std::span`.
  There was already **strong consensus** to do this.
- **B)** Provide a way (two concepts) to find out whether containers do (not) provide deep constness.
- **C)** Disable `std::cbegin()`, `std::cend()`, etc. when broken due to shallow constness
  (ideally at compile time or at least by deprecating it in that case)
- **D)** Disable `std::ranges::cbegin()`, `std::ranges::cend()`, etc. when broken due to shallow constness

For C) and D) we can still enable any disabled behavior later when we know and agree how to fix it.

For each of these items there is an individual sub-proposal to be able to vote on them separately.

All proposals are intended as defects against C++20.

Proposal A:
Introduce cbegin() and cend() members for std::span

The first proposal is the fix to bring back cbegin() and cend() members to std::span. (The funny thing is that for std::string_view, the read-only “span” for character sequences, cbegin() and cend() is provided, while for std::span when it is really needed it was removed with lwg3320. This fixes this mistake.)

This was voted already with **strong consensus** on 2021-03-22:

Outcome: Strong consensus

Poll: span should have .cbegin() etc as member functions

SF	WF	N	WA	SA
8	15	2	0	1

Attendance: 35

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());

Proposal B:
Introduce a way to check for deep-const ranges

We propose to add two new concepts:

- One that is satisfied if a range is a deep-const range (propagates constness to its elements)
- One that is satisfied if a range is a shallow-const range (does not propagate constness to its elements)

The implementation is pretty forward (but not trivial for ordinary programmers):

```
template<typename T>
concept shallow_const_range
  = std::range<T> &&
  !std::is_const_v<std::remove_reference_t<
            decltype(*std::ranges::begin(std::declval<const T>()))>>;
```

This allows to fix code that is broken for shallow-const ranges such as:

```
template<typename T>
void foo(const T& coll) {
  bar(coll.begin(), coll.end());   // since C++20 bar() may modify elements of std ranges
}
```

either by modifying it as follows:

```
template<typename T>
requires (!std::shallow_const_range<T>)
void foo(const T& coll) {
  bar(coll.begin(), coll.end());
}
```

or by adding an overload as follows:

```
template<std::shallow_const_range T>
void foo(const T& coll) = delete;
```

Proposed Wording for the new concepts:

In **24.2 Header <ranges> synopsis [ranges.syn]**

after contiguous_range add:

```
// const range refinements
template<class T>
concept shallow_const_range = see below ;
```

in **24.4.5 Other range refinements [range.refinements]**

Before “The common_range concept...”

add

The shallow_const_range concept specifies requirements of a range type for which constness is not propagated to the elements.

```
template<class T>
concept shallow_const_range = std::range<T> &&
 !std::is_const_v<std::remove_reference_t<
            decltype(*std::ranges::begin(std::declval<const T>()))>>;
```

Proposal C:
Disable std::cbegin(), std::cend(), etc. when it is broken (i.e. for shallow-const ranges)

Provided we have the concept above it is easy to fix.
Otherwise we can make the concept above as an internal auxiliary concept.

Note that as alternative to change the existing declarations, we can add new overloads for these functions with the same behavior but marked with **`[[deprecated]]`**

Proposed Wording to fix std::cbegin() etc. for the moment:

In **27.3 Header <iterator> synopsis [iterator.synopsis]**

For the following declarations:

```
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));

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));
```

replace:

```
template <class C>
```

by

```
template <class C> requires (!shallow_const_range<C>)
```

Perform corresponding fixed in the definitions in **27.7 Range access [iterator.range].**

Proposal D:
Disable std::ranges::cbegin, std::ranges::cend, etc. when it is broken (i.e. for shallow-const ranges)

Provided we have the concept above it is easy to fix.
Otherwise we can make the concept above as an internal auxiliary concept.

Proposed Wording to fix std::ranges::cbegin etc. for the moment:

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

Replace

> The expression ranges::cbegin(E) for a subexpression E of type T is expression-equivalent to:

by:

> For a subexpression E of type T, the expression ranges::cbegin(E) is ill-formed if shallow_const_range<decltype(E)> is satisfied. Otherwise, it is expression-equivalent to:

Accordingly for **cend**, **crbegin** , **crend**, **cdata**.

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 and Walter E. Brown, Niall Douglas, Tomasz Kamiński, Alisdair Meredith, Barry Revzin, Tim Song, Ville Voutilainen, and Jonathan Wakely.

Forgive me if I forgot anybody.