P2836R1
`std::basic_const_iterator` should follow its underlying type's convertibility

Author: Christopher Di Bella

Presentation: Tomasz Kamiński

Disclaimer:
This deck was prepared by me (Tomasz Kamiński) for LEWG presentation purposes, and all error or mistakes are mine.

Examples

```
void oldApi(std::vector<int>::const_iterator f, std::vector<int>::const_iterator l);


std::vector<int> v;

oldApi(v.begin(), v.end()); //  OK, v.begin() is std::vector<int>::iterator
```

Examples

```
void oldApi(std::vector<int>::const_iterator f, std::vector<int>::const_iterator l);


std::vector<int> v;

oldApi(v.begin(), v.end()); // OK, v.begin() is std::vector<int>::iterator


auto cv = v | std::views::as_const;

oldApi(cv.begin(), cv.end()); // OK, cv.begin() is std::vector<int>::const_iterator
```

Examples

```
void oldApi(std::vector<int>::const_iterator f, std::vector<int>::const_iterator l);


std::vector<int> v;

oldApi(v.begin(), v.end()); // OK, v.begin() is std::vector<int>::iterator


auto cv = v | std::views::as_const;

oldApi(cv.begin(), cv.end()); // OK, cv.begin() is std::vector<int>::const_iterator


std::ranges::subrange sv = v;

oldApi(sv.begin(), sv.end()); // OK, sv.begin() is std::vector<int>::iterator
```

Before

```
void oldApi(std::vector<int>::const_iterator f, std::vector<int>::const_iterator l);


std::vector<int> v;

oldApi(v.begin(), v.end()); // OK, v.begin() is std::vector<int>::iterator


auto cv = v | std::views::as_const;

oldApi(cv.begin(), cv.end()); // OK, cv.begin() is std::vector<int>::const_iterator


std::ranges::subrange sv = v;

oldApi(sv.begin(), sv.end()); // OK, sv.begin() is std::vector<int>::iterator


auto csv = sv | std::views::as_const;

oldApi(csv.begin(), csv.end()); // ILL-FORMED, sv.begin() is std::basic_const_iterator<...>
                                // not convertible to std::vector<int>::const_iterator
```

After

```
void oldApi(std::vector<int>::const_iterator f, std::vector<int>::const_iterator l);


std::vector<int> v;

oldApi(v.begin(), v.end()); // OK, v.begin() is std::vector<int>::iterator


auto cv = v | std::views::as_const;

oldApi(cv.begin(), cv.end()); // OK, cv.begin() is std::vector<int>::const_iterator


std::ranges::subrange sv = v;

oldApi(sv.begin(), sv.end()); // OK, sv.begin() is std::vector<int>::iterator


auto csv = sv | std::views::as_const;

oldApi(csv.begin(), csv.end()); // OK, sv.begin() is std::basic_const_iterator<...>
                                // converts to std::vector<int>::const_iterator
```

C++20 vs C++23 behavior

```
std::vector<int> v;
std::ranges::subrange sv = v;

std::vector<int>::const_iterator ci = std::ranges::cbegin(sv);
//   OK in C++20, std::ranges::cbegin(sv) returns mutable std::vector<int>::iterator,
//      that is convertible to std::vector<int>::const_iterator
```

C++20 vs C++23 behavior

```
std::vector<int> v;
std::ranges::subrange sv = v;

std::vector<int>::const_iterator ci = std::ranges::cbegin(sv);

//   OK in C++20, std::ranges::cbegin(sv) returns mutable std::vector<int>::iterator,
//      that is convertible to std::vector<int>::const_iterator
//   ILL-FORMED in C++23, std::ranges::cbegin(sv) returns
//      std::basic_const_iterator<std::vector<int>::iterator>,
//      that cannot be converted std::vector<int>::const_iterator
```

C++20 vs C++23 behavior

```
std::vector<int> v;
std::ranges::subrange sv = v;

std::vector<int>::const_iterator ci = std::ranges::cbegin(sv);

//   OK in C++20, std::ranges::cbegin(sv) returns mutable std::vector<int>::iterator,
//     that is convertible to std::vector<int>::const_iterator
//   ILL-FORMED in C++23, std::ranges::cbegin(sv) returns
//     std::basic_const_iterator<std::vector<int>::iterator>,
//     that cannot be converted std::vector<int>::const_iterator
//   OK with this proposal, std::basic_const_iterator<std::vector<int>::iterator>,
//     is now convertible to std::vector<int>::const_iterator
```

C++20 vs C++23 behavior

```
std::vector<int> v;
std::ranges::subrange sv = v;

std::vector<int>::const_iterator ci = std::ranges::cbegin(sv);
//   OK in C++20, std::ranges::cbegin(sv) returns mutable std::vector<int>::iterator,
//     that is convertible to std::vector<int>::const_iterator
//   ILL-FORMED in C++23, std::ranges::cbegin(sv) returns
//     std::basic_const_iterator<std::vector<int>::iterator>,
//     that cannot be converted std::vector<int>::const_iterator
//   OK with this proposal, std::basic_const_iterator<std::vector<int>::iterator>,
//     is now convertible to std::vector<int>::const_iterator

auto ci2 = std::ranges::cbegin(sv);
// std::vector<int>::iterator in C++20
// std::basic_const_iterator<std::vector<int>::iterator> in C++23
```

Proposal: Allow conversion to `basic_const_iterator<Iterator>`

```
template<not-a-const-iterator CI>
  requires constant-iterator<CI>
        && convertible_to<Iterator const&, CI>
constexpr operator CI() const& { return _current; }

template<not-a-const-iterator CI>
   requires constant-iterator<CI>
         && convertible_to<Iterator, CI>
constexpr operator CI() && { return std::move(_current); }
```

`basic_const_iterator<Iterator>` will convert to any iterator `CI`, such that:

- `Iterator` can be converted to it (`convertible_to<Iterator, CI>`)
- `CI` is an constant iterator (`constant_iterator<CI>`)
- `CI` is not `basic_const_iterator` (`not-a-const-iterator CI`)

The second requirement implies that the we prevent silently dropping constness. Third prevents ambiguity with constructor.

Why not std::vector<int>::const_iterator

```
std::vector<int> v; std::subrange sv(v);

std::ranges::cbegin(sv);  // just return std::vector<int>::const_iterator
```

Pros:

- Intuitive, meets expectation of most of the developers
- Prevents code bloat. Currently programs may specialize algorithms and views for both:
  - `std::vector<int>::const_iterator`
  - `std::basic_const_iterator<std::vector<int>::iterator>`

Why not std::vector<int>::const_iterator

```
std::vector<int> v; std::subrange sv(v);

std::ranges::cbegin(sv);  // just return std::vector<int>::const_iterator
```

Pros:

- Intuitive, meets expectation of most of the developers
- Prevents code bloat. Currently programs may specialization of algorithms/views for both:
  - `std::vector<int>::const_iterator`
  - `std::basic_const_iterator<std::vector<int>::iterator>`

Cons:

- Requires a new customization point for mapping between mutable and constant iterator.
- **May produce a non-range, i.e `std::ranges::cbegin(r)`, `std::ranges::end(r)` will no longer be iterable.**

Why not std::vector<int>::const_iterator?

```
template<typename IT>
struct my_range_adaptor {
  using iterator = IT;
  struct sentinel;
  iterator begin() const;
  sentinel end() const;
};
```

```
template<typename It> // usually hidden friend of sentinel
bool operator==(typename my_range_adaptor<It>::iterator,
                typename my_range_adaptor<It>::sentinel);
```

```
my_range_adaptor<std::vector<int>::iterator> m = /* ... */;
m.begin() == m.end(); // OK, picks above operator
```

Why not std::vector<int>::const_iterator?

```
template<typename IT>
struct my_range_adaptor {
  using iterator = IT;
  struct sentinel;
  iterator begin() const;
  sentinel end() const;
};


template<typename It> // usually friend of sentinel
bool operator==(typename my_range_adaptor<It>::iterator,
                typename my_range_adaptor<It>::sentinel);


my_range_adaptor<std::vector<int>::iterator> m = /* ... */;
m.begin() == m.end(); // OK, picks above operator

auto cb = std::ranges::cbegin(m); // basic_const_iterator<....>
cb == m.end(); // OK, basic_const_iterator<It> can be compared with everything
               // that It can be compared, i.e. invokes above operator
```

Why not std::vector<int>::const_iterator?

```
template<typename IT>
struct my_range_adaptor {
  using iterator = IT;
  struct sentinel;
  iterator begin() const;
  sentinel end() const;
};


template<typename It> // usually friend of sentinel
bool operator==(typename my_range_adaptor<It>::iterator,
                typename my_range_adaptor<It>::sentinel);


my_range_adaptor<std::vector<int>::iterator> m = /* ... */;
m.begin() == m.end(); // OK, picks operator above

auto cb = std::ranges::cbegin(m); // basic_const_iterator<....>
cb == m.end(); // OK, basic_const_iterator<It> can be compared with everything
               // that It can be compared, i.e. invokes above operator

std::vector<int>::const_iterator ci = m.begin(); // ok, there is conversion
ci == m.end(); // ILL-FORMED, no matching overload for equality
```

Why not std::vector<int>::const_iterator?

- Some standard views (e.g. `std::take_while_view`) would be affected.
- Unknown number of user provided views, i.e. coming from program or third party libraries.

LWG3946: Fix for const_iterator_t, const_sentinel_t

Previous revision of the paper pointed out, that following may be ill-formed:

```
std::ranges::const_iterator_t<R> f = std::ranges::cbegin(r);
std::ranges::const_sentinel_t<R> l = std::ranges::end(r);
```

This problem was addressed by library issue 3946 (LWG3946) that adjusted definition of the `const_iterator_t` and `const_sentinel_t` to:

```
template<range R>
  using const_iterator_t = decltype(ranges::cbegin(declval<R&>()));
template<range R>
  using const_sentinel_t = decltype(ranges::cend(declval<R&>()));
```