Movability of Single-pass Iterators

Document #:	D1207R2
Date:	2019-06-01
Project:	Programming Language C++
Audience:	LWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

I want to move(it), move(it), y'all want to move(it);

Changes

Revision 2

- Remove discussions about `ContiguousIterator`
- Use Latex
- Add wording

Revision 1

- Refine the impact on the standard
- Mention `ITER_CONCEPT`
- Remove the idea the `ranges::copy` could move from non-copyable iterators
- Replace Cpp17Iterator by LegacyIterator to reflect the standard
- Remove the very wrong idea of having view return `begin()` by reference.
- Fix some confusing phrasing

Introduction

Non-forward Input iterators and output iterators, also known as "Single-pass iterators" are semantically move-only. The standard states:

Note: For input iterators, a == b does not imply ++a == ++b (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms.

This means that once an iterator is copied, only one of the copies can be read-from if either one is incremented, which make the usefulness of such object questionable. Deferencing multiple copies of a single pass iterator often exposes undefined or invalid behavior if either one is incremented: the following example exposes Undefined behavior:

```
auto other = some_input_iterator;
std::cout << *(++other) << *some_input_iterator << '\n';
```

It would, therefore, make sense that classes satisfying the `InputIterator` concept shall only be required to be movable.

Alas, Single-pass iterators and many classes satisfying its requirements predate C++11, they do therefore have move only semantic with copy syntax. In that regard, they are similar to `auto_ptr`.

Terminology

This paper redefines the requirements of some concepts as specified in the Working Draft In the rest of this paper

- `InputIterator` designs the `InputIterator` concept as proposed by this paper
- `Cpp20InputIterator` designs the `InputIterator` concept as specified in the Working Draft.
- `OutputIterator` designs the `OutputIterator` concept as proposed by this paper
- `OutputIterator` designs the `OutputIterator` concept as proposed in the Working Draft

Scope

This paper proposes changes targeting C++20. Because the modifications proposed here changes some requirements and concepts introduced by Ranges, the authors strongly suggest they are considered for the inclusion in the same version of the standard. Indeed, Concepts introduced by ranges gives us a unique opportunity to make the modifications proposed, as they might, in some cases, break code, if introduced after the publication of C++20.

Non-Goal

As a large amount of code depends on the Input/Output iterators requirements as specified by C++17, this paper does not propose any modifications to the `LegacyInputIterator` or any class that depends on it. Specifically, we do not propose to change the requirements or wording of `istream_iterator`, `ostream_iterator`, `istreambuf_iterator` or `ostreambuf_iterator`. Furthermore, we do not propose modifications to algorithms in the namespace `std`. The new iterators we propose here are in fact mostly incompatible with existing algorithms. They are meant to be used in the `ranges` namespace and as basic building blocks of range-based views.

While the ability to use move-only iterators with the algorithms defined in the `std` namespace would certainly be welcomed, doing so would weaken the `Cpp20InputIterator` concept and leads to other issues (namely, `std` based algorithms require iterators to be `EqualityComparable`, which the `Cpp20InputIterator` does not require).

In practice, that means that types satisfying the `LegacyInputIterator` requirements continue to work unaffected with algorithms defined in the `std` namespace. They may not be compatible with algorithms defined in the ranges namespace, or with new code using non-movable types satisfying the `InputIterator` concept as proposed here.

Inversely, types satisfying the `InputIterator` concepts may not be compatible with algorithms in `std` as they may not be able to satisfy the `LegacyInputIterator` requirements **if** they are not copyable.

Because it hardly makes sense to copy an Input Iterator (more on that later), it would be possible to add support for move-only iterators to the `std` namespace without much change to the standard. However, because implementers may copy iterators within the implementation of the standard library, along with existing third-party libraries, a lot of code would need to be adapted. And there is little pressure to do so as existing iterators types cannot be changed.

Furthermore, while we propose to add support for movable non-forward iterators, the proposed design does not preclude, in any way, the existance of copyable non-forward iterators.

Motivation

Move-only state

It may be desirable for an iterator to hold a move-only object, becoming itself move-only, which is not possible with iterators modeling LegacyIterator. A real-world example of such iterator is described in [P0902]. While syntactically copyable in the current design, a `coroutine_handle` such as used by a `generator` input iterator ought to be move-only.

Implicitly destructive operations

Reading from an input sequence is a destructive operation. But that destruction is reflected nowhere in the API. Less experienced developers may not be aware of the destructive / single-pass nature of non-forward Iterators By making `InputIterator` move only, developers will have to explicitly move them, which both signals the invalidation of the move-from object, but, more importantly, that the underlying data will be destroyed.

What is a move-only iterator?

Unlike [P0902], we do not propose to introduce a new iterator category.

A move-only Iterator is a non-forward iterator (either input or output depending on whether is it writable). This means that a move-only iterator has *almost* the same semantic requirements as an `InputIterator`, and offers the same operations. In other words, everything that can be expressed and done with a `Cpp20InputIterator` can be equally expressed and done with a move-only/non-copyable `InputIterator`.

Therefore, this paper does not propose to introduce a new iterator category, new named-requirement, concept name or iterator tag.

Furthermore, there is no `ForwardIterator` that is only movable, as a `ForwardIterator` is by definition an iterator that can be copied. We will expand on this later.

A Holistic Approach to Iterators

While the first part of this paper focuses on making move-only iterators possible, as a means to get some code to compile, it is important to take a step back and to think about what movability means for Iterators, from first principles.

An iterator denotes a position into a sequence of elements (whether that sequence maps to memory or not is, for our purpose, irrelevant).

A most basic iterator can be incremented, which means it can move to the next position in the sequence. An iterator does not own the sequence iterated over (there are exceptions, ie: generators), which means the salient property of an iterator is its position in that sequence.

Iterators categories then represent the way an iterator can move along that sequence.

- Input and FordwardIterator: sequentially, one direction
- BidirectionalIterator: sequentially, both directions

- RandomAccess: both directions in O(1)

ContiguousIterator is an optimization of RandomAccessIterator specific to the C++ memory model that further, constrain the underlying sequence to be laid out contiguously in memory.

Stepanov theorized an additional category, "Index iterator", which has O(1) access but in a single direction.

Further work was made on iterator categories, notably the Boost.Iterator library focused on separating traversal (how the iterator moves along the sequence) from access (whether dereferencing an iterator allows the pointed element to be read, written or both). While a very interesting concept, it falls outside the scope of this paper. Just keep in mind that everything that applies to non-forward `InputIterator` usually applies to OutputIterator - which are always non-Forward, the standard lacking that symmetry between read access and write access.

However, focusing on traversal, the set of iterators categories is actually rather closed, there are only so many ways a sequence can be traversed. An important point of Stepanov design is that each category is a refinement of the preceding one. `RandomAccessIterator` is a `BidirectionalIterator` which in turn is a `ForwardIterator`. Every algorithm applicable to a `ForwardIterator` can be equally applied to a `BidirectionalIterator`, etc.

So, what separates `InputIterator` from `ForwardIterator` if they are both "forward" in that they can both traverse a sequence in one direction?

`ForwardIterator` is defined as being "multi-pass". Meaning it can traverse a sequence multiple times. That, in turn, implies `ForwardIterator` is copyable, because if a sequence can be traversed multiple times, it can also be traversed multiple times at the same time and therefore there can be multiple `ForwardIterator` pointing at different elements in the sequence. ForwardIterator is also always `EqualityComparable`. Two `ForwardIterator` compare equal if they point to the same elements in the sequence (remember, that in the general case, the position of an iterator in a sequence is its sole salient property). And so `ForwardIterator`, being both `EqualityComparable` and `Copyable` is Regular.

The standard defines the "multi pass" guarantee by stating: > a == b implies ++a == ++b > Given X is a pointer type or the expression (void)++X(a), *a is equivalent to the expression* a.

In other words: Two identical objects to which is applied the same transformation are identical.

Copying a `FordwardIterator` copies the salient properties of that value and incrementing it does not modify the underlying sequence. So `ForwardIterator` is required to be a regular type behaving like a regular type.

Which bring us to `InputIterator`. InputIterator is a "single pass" iterator. The underlying sequence can on only be traversed once. The existence of an Iterator at the nth position in the sequence implies there can be no valid iterator at the position n-1 in that same sequence.

```
//Given an InputIterator a
auto b = a; a++;
b; // is invalid.
```

However, remember that the sole salient property of an iterator is its distance to the start of the sequence. Incrementing an iterator only mutates that property (again, conceptually, independently of implementation). And the only operation that mutates that property is the increment operation (which Stepanov calls `successor`).

This implies that as a non-forward iterator moves from one element of the sequence to the next, that element is destroyed.

All of this is well known and is basically rephrasing "Input iterators are single pass".

An important point to make is that how an iterator can traverse a sequence is derived from the nature of the sequence rather than from the iterator itself. The point could be made that there is no such thing as an "Input iterator" Or a "Forward Iterator" because what we really mean is "Iterator over an Input Sequence" or "Iterator over a Forward Sequence".

This is saying that, to be able to reason properly about iterators and traversal, we must assume that the iterator type associated with a sequence is the most specialized possible for that sequence.

The problem is, of course, that we do not have, in the general case, a more meaningful way to express the traversability of a sequence than by defining what type of iterator is used to iterate over it.

It is then the responsibility of the developer providing the sequence to define the most appropriate – the most specialized – iterator category for that sequence.

In practice, because `InputIterator` and `ForwardIterator` are syntactically identical and because of the single-pass / multi-passes guarantees are poorly taught, it is common for iterators to be mis-categorized. Other iterator categories do not have these problems as each subsequent refining category adds syntax requirements: BidirectionalIiterator require decrement operators, RandomAccessIterator has further requirements.

But then, is there a set of operations and semantic requirements, translating to actual C++ syntax, that could allow for InputIterator to be easily distinguished from each other? Can we avoid requiring a tag system? Is there a defining operation that distinguishes `InputIterator` from `ForwardIterator` in such a way that it would both not require an explicit category tagging while at the same time offering a better understanding of iterator categories as well as a less surprising and safer API for non-forward iterators?

In fact, there is. We established that `ForwardIterator`s are semantically copyable, while `InputIterator`s are not. So the requirement that promotes an `InputIterator` into a `ForwardIterator` is indeed copyability - which translate in C++ to a copy constructor. We can, therefore, consider that, in the absence of a tag, all non-copyable iterators are `InputIterator`, while all copyable iterators are `ForwardIterator`.

This model, however, deviates slightly from Stepanov's work and `LegacyInputIterator`:

Copying a `LegacyInputIterator` does not invalidate either copy. In fact, it is quite valid to deference multiple copies of a `LegacyInputIterator`.

Elements Of Programming has the notion of `Regular` types (and in Stepanov's work all Iterators are regular), but also the notion of regular transformations (aka pure functions) - which, given the same input, always give the same output. Given a `ForwardIterator fi`, there is a `successor` function returning an incremented copy of `fi` such as `sucessor(fi) == sucessor(fi)`. In C++, that regular `sucessor` function is `ForwardIterator::operator++(int);`, in that `(it++) == (it++)` for any given `ForwardIterator`.

For `InputIterator`, Stepanov specifies that the `successor` is a pseudo transformation or a non-regular transformation that look like a regular one. And therein lies the rub.

Like a pointer, `InputIterator` is Regular, up until the point a transformation of an instance affects all copies.

```
InputIterator i = /*...*/
*i    //ok
auto a = i //ok
*i    //ok
i++;  // a now invalid
```

This design accurately models the nature of iterators. Because an iterator represents a position in a sequence, it is natural that multiple iterators could point to the same position. After one copy is incremented, in Stepanov's model, other copies are in a partially formed state and cannot be used (but they can be assigned to, or destroyed).

Let's consider the case where we move from an iterator instead of copying it.

```
InputIterator i = /*...*/
*i           //ok
auto a = move(i); //ok
*i;          //invalid
a++;         //ok
i++;         //invalid
```

Moving from an iterator invalidates it early, albeit artificially. As per standard, the moved-from iterator is in a valid, but unspecified state, and cannot be used (but can be assigned to, or destroyed). Notice the similarity between "a valid, but unspecified state" and "a partially formed state".

The difference is slim. Notably, both models are equally expressive. References can be used, should multiple names be necessary. In Stepanov's model iterators are made invalid by the natural mutation of the sequence upon increment rather than by artificially preventing multiple copies.

The second model in which the iterator is moved from, the one we think should be the default way to handle non-forward iterators, is however a much better fit for the C++ model, and offers much stronger guarantees to both the human developer as well as static analysis tools.

In the "increment invalidates" model, objects are spiritually moved-from at a distance, which neither the theory of special relativity nor the C++ memory, model are equipped to handle. This makes it hard

for tools to detect invalid uses - although it might become possible with better tools (See Herb Sutter's CppCon2018 talk). But most concerning, there is no way for a developer to know that the iterators are entangled.

```
auto i = troubles.begin();
auto schrodingers_iterator = i;
i++;
auto nasal_demon = *schrodingers_iterator;
```

The code above might be perfectly fine. Indeed whether it is well defined or not depends on whether the iterator return by `troubles.begin();` is forward or not. It is undecidable in these 4 lines of slide-code. It is not much more obvious in a complex program that may pass iterators to other functions or store them in containers, etc. There are, after all, no theoretical limits to the distance in time and space over which entanglement perdures.

Even worse, should the type of `troubles.begin();` be changed from Forward to Input, the code would change from perfectly fine to UB, with no warning.

Moving non-forward iterators, therefore, better expresses intent, is safer and less surprising. Move-only non-forward Iterators also express the destructive nature of incrementation and give a better sense of the difference between `InputIterator` and `ForwardIterator`.

An Holistic Approach to Iterator Tags and Iterator Concepts

Missing the notion of movability pre-c++11 and lacking concepts, `LegacyIterator`s are syntactically distinguished by tags. a `LegacyInputIterator` is one which has an `input_iterator_tag` tag, while a `LegacyForwardIterator` is one which has a `forward_iterator_tag` tag. This creates a sort of circular, self-referential definition. This has carried over to the `Iterator` concepts definitions.

Iterators concepts then :

- Have semantic requirements not expressed through syntax and therefore not enforceable at compile time
- Need syntax to artificially subscribe to the correct, most refined concept

Of course, it is not always possible to express all of a type's semantic requirements through syntax, and in some cases, tags are an unfortunate necessity. However, they should be the mechanism of last recourse, and whenever possible, the semantic requirements should be reflected in the syntax. The idea is that hidden requirements not expressed as code lead to easier-to-misuse types, which inevitably translates to runtime bugs. **Ultimately, requirements that can neither be checked at compile time (concepts) or runtime (contracts) are bound to be ignored**. Rooted in the belief that not all birds quack like a duck, this proposal leverages meaningful syntactic requirements to increase the type safety of the iterator taxonomy.

In the case of iterators, all requirements of all iterators categories can be expressed syntactically:

```
template <class I> concept bool InputIterator =
Readable<I> &&
Iterator<I> ;

template <class I> concept bool ForwardIterator =
InputIterator<I> &&
Copyable<I> &&
EqualityComparable<I>;

template <class I> concept bool BidirectionalIterator =
ForwardIterator<I> &&
Decrementable<I>;

template <class I> concept bool RandomAccessIterator =
BidirectionalIterator<I> &&
RandomAccessIncrementable<I>;
```

This is of course simplified but shows that each iterator category subsumes the last and adds a single, cohesive set of requirement enforceable at compile-time. In this design, there is no risk of a type satisfying the wrong concept because of a poorly chosen tag.

Tags as an opt-in opt-out mechanism

Iterators concepts already support semantic-only checking of iterator requirements for types that do not define either iterator_category or iterator concept. Currently, this machinery will identify categories from ForwardIterator to RandomAccessIterator. With this proposal, non-copyable tagless types that otherwise meet the requirements of InputIterator be correctly identified as non-forward InputIterator, which is always the correct assumption. Copyable tagless iterators will remain categorized as ForwardIterator by that machinery.

Q/A

Non-regular iterators, really?

This proposal advocates for Non-Regular Iterators, and weakens `WeaklyIncrementable` requirements to that effect. Non-Regularity is best avoided, so this might feel like going backward.

However, **non-regular types are easier to reason about than types that just pretend to be regular**. Because `InputIterator` is meant to iterate over a non-regular sequence, it is not regular (whether we like it or not), and the best we can do is make sure the syntax matches the semantic. It would

be accurate to say that `InputIterator` is locally regular, but this doesn't help much in the context of the c++ memory model. This paper is in part motivated by the conviction that exposing **a false sense of (Semi-)regularity is much more detrimental to code robustness than non-regularity**.

What about Equality of Input Iterators?

A first, misguided, version of this paper attempted to prevent comparability of types meeting the `InputIterator` requirements. `InputIterator` should, in general, not be `EqualityComparable`, since they cannot be copied and a fundamental idea in Stepanov's teachings is that copy and equality are two sides of the same coin.

However, preventing `Equality` requires dramatic changes to the design and the author was reminded that negative-requirements are in general a terrible idea.

Early feedback suggested a desire to be able to compare non-forward iterators. Consider the following:

```
auto a = stream.begin();
auto b = stream.begin();
if(a == b) {
}
```

This code will inevitably lead to suffering at some point. However, we cannot prevent people from constructing multiple non-forward iterators, and these iterators will compare equal until one of them invalidate the other.

Two non-forward iterators compare equal if-and-only-if they point to the same position of the same sequence (and only one such position can be referred to at any given time).

Allowing `EqualityComparable` on non-forward iterators also simplify the interoperability of `std::` and `ranges::` iterators. However, the author would like to recommend that all future non-forward iterators introduced in the standard be *not* `EqualityComparable`. Instead, non-forward iterator should compare to a Sentinel, which is a much better model. `common_iterator` can be used to ease migration and interoperability.

But... Moved-from objects are still objects!

Sure, moving-from leaves a trail of objects in an unspecified state. However, it is much more easy for tools and humans alike to understand that moved-from objects should not be used, and in fact, all majors compilers can warn about these patterns. We think that for the case at hand, focusing on the proper handling of values – as opposed to objects – is a sufficient approximation to reduce the potential for iterators misuse while not weakening the stronger mathematical underpinning of the STL.

Does iterators default-constructability needs revisiting?

Default-constructability of iterator seems to have been added, removed and added back to the Ranges TS and the One Ranges Proposal several times. To the best of my knowledge, this was done for the sake of Semiregularity. Given that this proposal strikes semi-regularity, should this question be revisited?

The authors want to point out that default-constructed iterators are almost never in a specified state and are almost always unsafe to use. Moreover, DefaultConstructible is not a requirement of any algorithm using ranges and ultimately, we think enforcing DefaultConstructibility weakens the better `Sentinel` model introduced by ranges.

What about [P0902]?

Andrew Hunter's "Move-only iterators" paper proposes a design to introduce Move-Only iterators in the taxonomy of `LegacyIterator`. However, this design does not offer a solution to use these move-only iterators with existing algorithms, limiting their usefulness. The iterators proposed by P0902 are additionally `EqualityComparable`. The advantage of that is that they are compatible with algorithms designed with C++17 downward. That's, however, a potential source of bugs and confusion.

However, if LEWG feels strongly about a solution compatible with existing algorithms it would be possible to relax the requirements of concerned algorithms to accept move-only iterators. along with the introduction of a new `move_iterator_tag` trait.

Such algorithms would then be compatible with types satisfying `InputIterator` (as proposed by this paper) through a `common_iterator` adaptor.

If proven with enough confidence that requirements of existing algorithms in the `std` namespace can be relaxed to handle move-only iterator, the necessary modifications can be applied in a subsequent standard version.

So while there would definitively be value in supporting move-only iterators everywhere it makes sense, and the potential for breakage is relatively low, we do not propose it for lack of visibility on the consequences of such changes.

Why do you want to take my Copyable InputIterators away from me, I like them?!

We do not propose anything of the sort. But, we propose that

- Any `InputIterator` that happens to be `Copyable` is also a `ForwardIterator`.
- It remains possible to opt-out of that behavior by defining `iterator_concept` to be `input_iterator_tag`.

Non-copyable Iterator	Copyable Iterator	Copyable Iterator with a tag

struct It {
It(It&&) = default;
It(const It&) = delete;
//...
};
	struct It {
	It(const It&) = default;
	//...
	};
		struct It {
		It(const It&) = default;
		using iterator_concept = input_iterator_tag;
		//...
		};

static_assert(InputIterator<It>); static_assert(!ForwardIterator<It>);	static_assert(InputIterator<It>); static_assert(ForwardIterator<It>);	static_assert(InputIterator<It>); static_assert(!ForwardIterator<It>);

Will this break existing code ?!

We want to reiterate(!) that all the changes proposed in this paper are only applicable to concepts, types, and requirements that were added to the standard by the Ranges proposal. They do not, in any way, impact code depending on types, requirements or algorithms as defined by the C++17 standard

Won't that implicit categorization lead to miss-categorization?

The only valid use cases for `InputIterator` are streams or other input devices, and iterators that own a non-copyable generator. Most views and iterators are Forward. It turns out that C++ types are `Copyable` by default, therefore, Iterators will be categorized as `ForwardIterator` by default, which is correct in most cases.

This proposal is also a teaching opportunity because the nature of `InputIterator` is often poorly understood and misconstrued. We suspect that these tweaks to the taxonomy of Iterator will make them easier to teach.

Post Increment on non-copyable iterators

Post-incrementing move-only iterators would obviously be incorrect. However, a satisfying solution was offered by [P0541]

Implementation experience

We validated the design in cmstl2. However, cmcstl2 deviates from the Working Draft as it doesn't have the same Deep Integration system and therefore lacks the `ITER_CONCEPT` machinery. Furthermore, we have not yet completed this work. Some algorithms, like `count`, proved to require specialization for `InputIterator` because of implementation specific details.

Acknowledgments

The authors like to thank Connor Waters, Tony Van Eerd, Eric Niebler, Casey Carter, Christopher Di Bella, Sean Parent and Arthur O'Dwyer who gave tremendously helpful feedbacks during the writing of this paper.

Wording

🔲 Iterator concepts [iterator.concepts]

🔲 Concept `WeaklyIncrementable` [iterator.concept.winc]

The `WeaklyIncrementable` concept specifies the requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are not required to be equality-preserving, nor is the type required to be `EqualityComparable`.

```
template<class I>
concept WeaklyIncrementable =
Semiregular Movable<I> &&
requires(I i) {
        typename iter_difference_t<I>;
        requires SignedIntegral<iter_difference_t<I>>;
        { ++i } -> Same<I&>; // not required to be equality-preserving
        i++; // not required to be equality-preserving
};
```

Let `i` be an object of type `I`. When `i` is in the domain of both pre- and post-increment, `i` is said to be *incrementable*. `I` models `WeaklyIncrementable<I>` only if

- The expressions `++i` and `i++` have the same domain.
- If `i` is incrementable, then both `++i` and `i++` advance `i` to the next element.
- If `i` is incrementable, then `addressof(++i)` is equal to `addressof(i)`.

[*Note:* For `CopyConstructible` `WeaklyIncrementable` types, `a` equals `b` does not imply that `++a` equals `++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on weakly incrementable types should never attempt to pass through the same incrementable value twice. They should be single-pass algorithms. These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

Concept `Incrementable` [iterator.concept.inc]

The `Incrementable` concept specifies requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be `EqualityComparable`. [*Note:* This supersedes the annotations on the increment expressions in the definition of `WeaklyIncrementable`. — *end note*]

```
template<class I>
concept Incrementable =
Regular<I> &&
WeaklyIncrementable<I> &&
requires(I i) {
        { i++ } -> Same<I>;
};
```

Let `a` and `b` be incrementable objects of type `I`. `I` models `Incrementable` only if

- If `bool(a == b)` then `bool(a++ == b)`.
- If `bool(a == b)` then `bool(((void)a++, a) == ++b)`.

[*Note:* The requirement that `a` equals `b` implies `++a` equals `++b` (which is not true for weakly incrementable types) allows the use of multi-pass one-directional algorithms with types that model `Incrementable`. — *end note*]

Concept `Iterator` [iterator.concept.iterator]

The `Iterator` concept forms the basis of the iterator concept taxonomy; every iterator models `Iterator`. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels, to read or write values, or to provide a richer set of iterator movements (**??**, **??**, **??**).

```
template<class I>
concept Iterator =
requires(I i) {
        { *i } -> can-reference;
} &&
WeaklyIncrementable<I>;
```

[*Note:* Unlike the *Cpp17Iterator* requirements, the `Iterator` concept does not requires copyability of single-pass iterators. — *end note*]

Concept `ForwardIterator` [iterator.concept.forward]

The `ForwardIterator` concept adds copyability, equality comparison and the multi-pass guarantee, specified below.

```
template<class I>
concept ForwardIterator =
InputIterator<I> &&
DerivedFrom<ITER_CONCEPT(I), forward_iterator_tag> &&
Incrementable<I> &&
Sentinel<I, I>;
```

The domain of `==` for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators of the same type may be compared and shall compare equal to other value-initialized iterators of the same type. [*Note:* Value-initialized iterators behave as if they refer past the end of the same empty sequence. — *end note*]

Pointers and references obtained from a forward iterator into a range `[i, s)` shall remain valid while `[i, s)` continues to denote a range.

Two dereferenceable iterators `a` and `b` of type `X` offer the *multi-pass guarantee* if:

- `a == b` implies `++a == ++b` and
- The expression `((void)[](X x){++x;}(a), *a)` is equivalent to the expression `*a`.

[*Note:* The requirement that `a == b` implies `++a == ++b` and the removal of the restrictions on the number of assignments through a mutable iterator (which applies to output iterators) allow the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

Iterator primitives [iterator.primitives]

Iterator operations [iterator.operations]

Since only random access iterators provide `+` and `-` operators, the library provides two function templates `advance` and `distance`. These function templates use `+` and `-` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations.

```
template<class InputIterator, class Distance>
constexpr void advance(InputIterator& i, Distance n);
```

Expects: `n` is negative only for bidirectional iterators.

Effects: Increments `i` by `n` if `n` is non-negative, and decrements `i` by `-n` otherwise.

```
template<class InputIterator>
constexpr typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Expects: `last` is reachable from `first`, or `InputIterator` meets the *Cpp17RandomAccessIterator* requirements and `first` is reachable from `last`.

Effects: If `InputIterator` meets the *Cpp17RandomAccessIterator* requirements, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

```
template<class InputIterator>
constexpr InputIterator next(InputIterator x,
typename iterator_traits<InputIterator>::difference_type n = 1);
```

Effects: Equivalent to: `advance(x, n); return x;`

```
template<class BidirectionalIterator>
constexpr BidirectionalIterator prev(BidirectionalIterator x,
typename iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

Effects: Equivalent to: `advance(x, -n); return x;`

Range iterator operations [range.iter.ops]

`ranges::next` [range.iter.op.next]

```
template<ForwardIterator I>
constexpr I ranges::next(I x);
```

Effects: Equivalent to: `++x; return x;`

```
template<ForwardIterator I>constexpr I ranges::next(I x, iter_difference_t<I> n);
```

Effects: Equivalent to: `ranges::advance(x, n); return x;`

```
template<ForwardIterator I, Sentinel<I> S>constexpr I ranges::next(I x, S bound);
```

Effects: Equivalent to: `ranges::advance(x, bound); return x;`

```
template<ForwardIterator I, Sentinel<I> S>constexpr I ranges::next(I x, iter_difference_-
t<I> n, S bound);
```

Effects: Equivalent to: `ranges::advance(x, n, bound); return x;`

Move iterators and sentinels [move.iterators]

Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator's indirection operator to an rvalue. Some generic algorithms can be called with move iterators to replace copying with moving.

[*Example:*

```
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end());          // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
make_move_iterator(s.end())); // moves strings into v2
```

— *end example*]

Class template `move_iterator` [move.iterator]

```
namespace std {
        template<class Iterator>
        class move_iterator {
                public:
                using iterator_type     = Iterator;
                using iterator_concept  = input_iterator_tag;
                using iterator_category = see below;
                using value_type        = iter_value_t<Iterator>;
                using difference_type   = iter_difference_t<Iterator>;
                using pointer           = Iterator;
                using reference         = iter_rvalue_reference_t<Iterator>;

                constexpr move_iterator();
                constexpr explicit move_iterator(Iterator i);
                template<class U> constexpr move_iterator(const move_iterator<U>& u);
                template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);

                constexpr iterator_type base() const;
                constexpr iterator_type base() &&;
                constexpr reference operator*() const;

                constexpr move_iterator& operator++();
                constexpr auto operator++(int);
                constexpr move_iterator& operator--();
                constexpr move_iterator operator--(int);
```

```
        constexpr move_iterator operator+(difference_type n) const;
        constexpr move_iterator& operator+=(difference_type n);
        constexpr move_iterator operator-(difference_type n) const;
        constexpr move_iterator& operator-=(difference_type n);
        constexpr reference operator[](difference_type n) const;

        template<Sentinel<Iterator> S>
        friend constexpr bool
        operator==(const move_iterator& x, const move_sentinel<S>& y);
        template<Sentinel<Iterator> S>
        friend constexpr bool
        operator==(const move_sentinel<S>& x, const move_iterator& y);
        template<Sentinel<Iterator> S>
        friend constexpr bool
        operator!=(const move_iterator& x, const move_sentinel<S>& y);
        template<Sentinel<Iterator> S>
        friend constexpr bool
        operator!=(const move_sentinel<S>& x, const move_iterator& y);
        template<SizedSentinel<Iterator> S>
        friend constexpr iter_difference_t<Iterator>
        operator-(const move_sentinel<S>& x, const move_iterator& y);
        template<SizedSentinel<Iterator> S>
        friend constexpr iter_difference_t<Iterator>
        operator-(const move_iterator& x, const move_sentinel<S>& y);
        friend constexpr iter_rvalue_reference_t<Iterator>
        iter_move(const move_iterator& i)
        noexcept(noexcept(ranges::iter_move(i.current)));
        template<IndirectlySwappable<Iterator> Iterator2>
        friend constexpr void
        iter_swap(const move_iterator& x, const move_iterator<Iterator2>& y)
        noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

        private:
        Iterator current;   // exposition only
    };
}
```

The member *typedef-name* `iterator_category` denotes

- `random_access_iterator_tag` if the type `iterator_traits<Iterator>::iterator_category` models `DerivedFrom<random_access_iterator_tag>`, and
- `iterator_traits<Iterator>::iterator_category` otherwise.

Requirements [move.iter.requirements]

The template parameter `Iterator` shall either meet the *Cpp17InputIterator* requirements or model `InputIterator`. Additionally, if any of the bidirectional traversal functions are instantiated, the template parameter shall either meet the *Cpp17BidirectionalIterator* requirements or model `BidirectionalIterator`. If any of the random access traversal functions are instantiated, the template parameter shall either meet the *Cpp17RandomAccessIterator* requirements or model `RandomAccessIterator`.

Construction and assignment [move.iter.cons]

```
constexpr move_iterator();
```

Effects: Constructs a `move_iterator`, value-initializing `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `Iterator`.

```
constexpr explicit move_iterator(Iterator i);
```

Effects: Constructs a `move_iterator`, initializing `current` with `i`.

```
template<class U> constexpr move_iterator(const move_iterator<U>& u);
```

Mandates: `U` is convertible to `Iterator` and `Iterator` is `CopyConstructible`.

Effects: Constructs a `move_iterator`, initializing `current` with `u.base()`.

```
template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);
```

Mandates: `U` is convertible to `Iterator`.

Effects: Assigns `u.base()` to `current`.

Conversion [move.iter.op.conv]

```
constexpr Iterator base() const;
```

Constraints: `CopyConstructible<Iterator>`

Returns: `current`.

```
constexpr Iterator base() &&;
```

Returns: `std::move(current);`

Element access [move.iter.elem]

```
constexpr reference operator*() const;
```

Effects: Equivalent to: `return ranges::iter_move(current);`

```
constexpr reference operator[](difference_type n) const;
```

Effects: Equivalent to: `ranges::iter_move(current + n);`

Navigation [move.iter.nav]

```
constexpr move_iterator& operator++();
```

Effects: As if by `++current`.

Returns: `*this`.

```
constexpr auto operator++(int);
```

Mandates: `CopyConstructible<Iterator>`

Effects: If `Iterator` models `ForwardIterator`, equivalent to:

```
move_iterator tmp = *this;
++current;
return tmp;
```

Otherwise, equivalent to `++current`.

```
constexpr move_iterator& operator--();
```

Effects: As if by `--current`.

Returns: `*this`.

```
constexpr move_iterator operator--(int);
```

Effects: As if by:

```
move_iterator tmp = *this;
--current;
return tmp;
```

```
constexpr move_iterator operator+(difference_type n) const;
```

Returns: `move_iterator(current + n)`.

```
constexpr move_iterator& operator+=(difference_type n);
```

Effects: As if by: `current += n;`

Returns: `*this`.

```
constexpr move_iterator operator-(difference_type n) const;
```

Returns: `move_iterator(current - n)`.

```
constexpr move_iterator& operator-=(difference_type n);
```

Effects: As if by: `current -= n;`

Returns: `*this`.

Common iterators [iterators.common]

Class template `common_iterator` [common.iterator]

Class template `common_iterator` is an iterator/sentinel adaptor that is capable of representing a non-common range of elements (where the types of the iterator and sentinel differ) as a common range (where they are the same). It does this by holding either an iterator or a sentinel, and implementing the equality comparison operators appropriately.

[*Note:* The `common_iterator` type is useful for interfacing with legacy code that expects the begin and end of a range to have the same type. — *end note*]

[*Example:*

```
template<class ForwardIterator>
void fun(ForwardIterator begin, ForwardIterator end);

list<int> s;
// populate the list s
using CI = common_iterator<counted_iterator<list<int>::iterator>, default_sentinel_t>;
// call fun on a range of 10 ints
fun(CI(counted_iterator(s.begin(), 10)), CI(default_sentinel));
```

— *end example*]

```
namespace std {
        template<Iterator I, Sentinel<I> S>
        requires (!Same<I, S>)
        class common_iterator {
                public:
```

```
constexpr common_iterator() = default;
constexpr common_iterator(I i);
constexpr common_iterator(S s);
template<class I2, class S2>
requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S>
&& CopyConstructible<I>
constexpr common_iterator(const common_iterator<I2, S2>& x);

template<class I2, class S2>
requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S> &&
Assignable<I&, const I2&> && Assignable<S&, const S2&>
common_iterator& operator=(const common_iterator<I2, S2>& x);

decltype(auto) operator*();
decltype(auto) operator*() const
requires dereferenceable<const I>;
decltype(auto) operator->() const
requires see below;

common_iterator& operator++();
decltype(auto) operator++(int);

template<class I2, Sentinel<I> S2>
requires Sentinel<S, I2>
friend bool operator==(
const common_iterator& x, const common_iterator<I2, S2>& y);
template<class I2, Sentinel<I> S2>
requires Sentinel<S, I2> && EqualityComparableWith<I, I2>
friend bool operator==(
const common_iterator& x, const common_iterator<I2, S2>& y);
template<class I2, Sentinel<I> S2>
requires Sentinel<S, I2>
friend bool operator!=(
const common_iterator& x, const common_iterator<I2, S2>& y);

template<SizedSentinel<I> I2, SizedSentinel<I> S2>
requires SizedSentinel<S, I2>
friend iter_difference_t<I2> operator-(
const common_iterator& x, const common_iterator<I2, S2>& y);

friend iter_rvalue_reference_t<I>
iter_move(const common_iterator& i)
noexcept(noexcept(ranges::iter_move(declval<const I&>())))
requires InputIterator<I>;
template<IndirectlySwappable<I> I2, class S2>
friend void
```

```
        iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
        noexcept(noexcept(ranges::iter_swap(declval<const I&>(), declval<const I2&>())));

        private:
        variant<I, S> v_;   // exposition only
    };

    template<class I, class S>
    struct incrementable_traits<common_iterator<I, S>> {
        using difference_type = iter_difference_t<I>;
    };

    template<InputIterator I, class S>
    struct iterator_traits<common_iterator<I, S>> {
        using iterator_concept = see below;
        using iterator_category = see below;
        using value_type = iter_value_t<I>;
        using difference_type = iter_difference_t<I>;
        using pointer = see below;
        using reference = iter_reference_t<I>;
    };
}
```

🔲 Associated types [common.iter.types]

The nested *typedef-name* s of the specialization of `iterator_traits` for `common_iterator<I, S>` are defined as follows.

- `iterator_concept` denotes `forward_iterator_tag` if `I` models `ForwardIterator`; otherwise it denotes `input_iterator_tag`.
- `iterator_category` denotes `forward_iterator_tag` if `iterator_traits<I>::iterator_category` models `DerivedFrom<forward_iterator_tag>`; otherwise it denotes `input_iterator_tag`.
- If the expression `a.operator->()` is well-formed, where `a` is an lvalue of type `const common_iterator<I, S>`, then `pointer` denotes the type of that expression. Otherwise, `pointer` denotes `void`.

🔲 Constructors and conversions [common.iter.const]

```
constexpr common_iterator(I i);
```

Effects: Initializes `v_` as if by `v_{in_place_type<I>, std::move(i)}`.

```
constexpr common_iterator(S s);
```

Effects: Initializes `v_` as if by `v_{in_place_type<S>, std::move(s)}`.

```
template<class I2, class S2>
requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S>
&& CopyConstructible<I>
constexpr common_iterator(const common_iterator<I2, S2>& x);
```

Expects: `x.v_.valueless_by_exception()` is `false`.

Effects: Initializes `v_` as if by `v_{in_place_index<`*i*`>, get<`*i*`>(x.v_)}`, where *i* is `x.v_.index()`.

```
template<class I2, class S2>
requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S> &&
Assignable<I&, const I2&> && Assignable<S&, const S2&>
common_iterator& operator=(const common_iterator<I2, S2>& x);
```

Expects: `x.v_.valueless_by_exception()` is `false`.

Effects: Equivalent to:

- If `v_.index() == x.v_.index()`, then `get<`*i*`>(v_) = get<`*i*`>(x.v_)`.
- Otherwise, `v_.emplace<`*i*`>(get<`*i*`>(x.v_))`.

where *i* is `x.v_.index()`.

Returns: `*this`

🔲 Accessors [common.iter.access]

```
decltype(auto) operator*();
decltype(auto) operator*() const
requires dereferenceable<const I>;
```

Expects: `holds_alternative<I>(v_)`.

Effects: Equivalent to: `return *get<I>(v_);`

```
decltype(auto) operator->() const
requires see below;
```

The expression in the requires clause is equivalent to:

```
Readable<const I> &&
(requires(const I& i) { i.operator->(); } ||
is_reference_v<iter_reference_t<I>> ||
Constructible<iter_value_t<I>, iter_reference_t<I>>)
```

Expects: `holds_alternative<I>(v_)`.

Effects:

- If `I` is a pointer type or if the expression `get<I>(v_).operator->()` is well-formed, equivalent to: `return get<I>(v_);`
- Otherwise, if `iter_reference_t<I>` is a reference type, equivalent to:

```
auto&& tmp = *get<I>(v_);
return addressof(tmp);
```

- Otherwise, equivalent to: `return` *`proxy`*`(*get<I>(v_));` where *`proxy`* is the exposition-only class:

```
class proxy {
        iter_value_t<I> keep_;
        proxy(iter_reference_t<I>&& x)
        : keep_(std::move(x)) {}
        public:
        const iter_value_t<I>* operator->() const {
                return addressof(keep_);
        }
};
```

🔗 Navigation [common.iter.nav]

```
common_iterator& operator++();
```

Expects: `holds_alternative<I>(v_)`.

Effects: Equivalent to `++get<I>(v_)`.

Returns: `*this`.

```
decltype(auto) operator++(int);
```

Mandates: `CopyConstructible<I>`

Expects: `holds_alternative<I>(v_)`.

Effects: If `I` models `ForwardIterator`, equivalent to:

```
common_iterator tmp = *this;
++*this;
return tmp;
```

Otherwise, equivalent to: `return get<I>(v_)++;`

🔲 Counted iterators [iterators.counted]

🔲 Class template `counted_iterator` [counted.iterator]

Class template `counted_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of the distance to the end of its range. It can be used together with `default_sentinel` in calls to generic algorithms to operate on a range of N elements starting at a given position without needing to know the end position a priori.

[*Example:*

```
list<string> s;
// populate the list s with at least 10 strings
vector<string> v;
// copies 10 strings into v:
ranges::copy(counted_iterator(s.begin(), 10), default_sentinel, back_inserter(v));
```

— *end example*]

Two values `i1` and `i2` of types `counted_iterator<I1>` and `counted_iterator<I2>` refer to elements of the same sequence if and only if `next(i1.base(), i1.count())` and `next(i2.base(), i2.count())` refer to the same (possibly past-the-end) element.

```
namespace std {
  template<Iterator I>
  class counted_iterator {
    public:
    using iterator_type = I;

    constexpr counted_iterator() = default;
    constexpr counted_iterator(I x, iter_difference_t<I> n);
    template<class I2>
    requires ConvertibleTo<const I2&, I>
    constexpr counted_iterator(const counted_iterator<I2>& x);

    template<class I2>
    requires Assignable<I&, const I2&>
    constexpr counted_iterator& operator=(const counted_iterator<I2>& x);

    constexpr I base() const;
    constexpr I base() &&;
    constexpr iter_difference_t<I> count() const noexcept;
```

```
constexpr decltype(auto) operator*();
constexpr decltype(auto) operator*() const
requires dereferenceable<const I>;

constexpr counted_iterator& operator++();
decltype(auto) operator++(int);
constexpr counted_iterator operator++(int)
requires ForwardIterator<I>;
constexpr counted_iterator& operator--()
requires BidirectionalIterator<I>;
constexpr counted_iterator operator--(int)
requires BidirectionalIterator<I>;

constexpr counted_iterator operator+(iter_difference_t<I> n) const
requires RandomAccessIterator<I>;
friend constexpr counted_iterator operator+(
iter_difference_t<I> n, const counted_iterator& x)
requires RandomAccessIterator<I>;
constexpr counted_iterator& operator+=(iter_difference_t<I> n)
requires RandomAccessIterator<I>;

constexpr counted_iterator operator-(iter_difference_t<I> n) const
requires RandomAccessIterator<I>;
template<Common<I> I2>
friend constexpr iter_difference_t<I2> operator-(
const counted_iterator& x, const counted_iterator<I2>& y);
friend constexpr iter_difference_t<I> operator-(
const counted_iterator& x, default_sentinel_t);
friend constexpr iter_difference_t<I> operator-(
default_sentinel_t, const counted_iterator& y);
constexpr counted_iterator& operator-=(iter_difference_t<I> n)
requires RandomAccessIterator<I>;

constexpr decltype(auto) operator[](iter_difference_t<I> n) const
requires RandomAccessIterator<I>;

template<Common<I> I2>
friend constexpr bool operator==(
const counted_iterator& x, const counted_iterator<I2>& y);
friend constexpr bool operator==(
const counted_iterator& x, default_sentinel_t);
friend constexpr bool operator==(
default_sentinel_t, const counted_iterator& x);

template<Common<I> I2>
friend constexpr bool operator!=(
```

```
        const counted_iterator& x, const counted_iterator<I2>& y);
        friend constexpr bool operator!=(
        const counted_iterator& x, default_sentinel_t y);
        friend constexpr bool operator!=(
        default_sentinel_t x, const counted_iterator& y);

        template<Common<I> I2>
        friend constexpr bool operator<(
        const counted_iterator& x, const counted_iterator<I2>& y);
        template<Common<I> I2>
        friend constexpr bool operator>(
        const counted_iterator& x, const counted_iterator<I2>& y);
        template<Common<I> I2>
        friend constexpr bool operator<=(
        const counted_iterator& x, const counted_iterator<I2>& y);
        template<Common<I> I2>
        friend constexpr bool operator>=(
        const counted_iterator& x, const counted_iterator<I2>& y);

        friend constexpr iter_rvalue_reference_t<I> iter_move(const counted_iterator& i)
        noexcept(noexcept(ranges::iter_move(i.current)))
        requires InputIterator<I>;
        template<IndirectlySwappable<I> I2>
        friend constexpr void
        iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
        noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

        private:
        I current = I();                    // exposition only
        iter_difference_t<I> length = 0;    // exposition only
    };

    template<class I>
    struct incrementable_traits<counted_iterator<I>> {
        using difference_type = iter_difference_t<I>;
    };

    template<InputIterator I>
    struct iterator_traits<counted_iterator<I>> : iterator_traits<I> {
        using pointer = void;
    };
}
```

Constructors and conversions **[counted.iter.const]**

```
constexpr counted_iterator(I i, iter_difference_t<I> n);
```

Expects: `n >= 0`.

Effects: Initializes `current` with `i` and `length` with `n`.

```
template<class I2>
requires ConvertibleTo<const I2&, I>
constexpr counted_iterator(const counted_iterator<I2>& x);
```

Mandates: `CopyConstructible<I>`

Effects: Initializes `current` with `x.current` and `length` with `x.length`.

```
template<class I2>
requires Assignable<I&, const I2&>
constexpr counted_iterator& operator=(const counted_iterator<I2>& x);
```

Effects: Assigns `x.current` to `current` and `x.length` to `length`.

Returns: `*this`.

Accessors [counted.iter.access]

```
constexpr I base() const;
```

Constraints: `CopyConstructible<I>`

Returns: `current`.

```
constexpr I base() &&;
```

Returns: `std::move(current);`

```
constexpr iter_difference_t<I> count() const noexcept;
```

Effects: Equivalent to: `return length;`

Element access [counted.iter.elem]

```
constexpr decltype(auto) operator*();
constexpr decltype(auto) operator*() const
requires dereferenceable<const I>;
```

Effects: Equivalent to: `return *current;`

```
constexpr decltype(auto) operator[](iter_difference_t<I> n) const
requires RandomAccessIterator<I>;
```

Expects: `n < length`.

Effects: Equivalent to: `return current[n];`

🔲 Navigation [counted.iter.nav]

```
constexpr counted_iterator& operator++();
```

Expects: `length > 0`.

Effects: Equivalent to:

```
++current;
--length;
return *this;
```

```
decltype(auto) operator++(int);
```

Mandates: `CopyConstructible<I>`

Expects: `length > 0`.

Effects: Equivalent to:

```
--length;
try { return current++; }
catch(...) { ++length; throw; }
```

```
constexpr counted_iterator operator++(int)
requires ForwardIterator<I>;
```

Effects: Equivalent to:

```
counted_iterator tmp = *this;
++*this;
return tmp;
```

```
constexpr counted_iterator& operator--();
requires BidirectionalIterator<I>
```

Effects: Equivalent to:

```
--current;
++length;
return *this;
```

```
constexpr counted_iterator operator--(int)
requires BidirectionalIterator<I>;
```

Effects: Equivalent to:

```
counted_iterator tmp = *this;
--*this;
return tmp;
```

```
constexpr counted_iterator operator+(iter_difference_t<I> n) const
requires RandomAccessIterator<I>;
```

Effects: Equivalent to: `return counted_iterator(current + n, length - n);`

```
friend constexpr counted_iterator operator+(
iter_difference_t<I> n, const counted_iterator& x)
requires RandomAccessIterator<I>;
```

Effects: Equivalent to: `return x + n;`

```
constexpr counted_iterator& operator+=(iter_difference_t<I> n)
requires RandomAccessIterator<I>;
```

Expects: `n <= length`.

Effects: Equivalent to:

```
current += n;
length -= n;
return *this;
```

```
constexpr counted_iterator operator-(iter_difference_t<I> n) const
requires RandomAccessIterator<I>;
```

Effects: Equivalent to: `return counted_iterator(current - n, length + n);`

```
template<Common<I> I2>
friend constexpr iter_difference_t<I2> operator-(
const counted_iterator& x, const counted_iterator<I2>& y);
```

Expects: `x` and `y` refer to elements of the same sequence.

Effects: Equivalent to: `return y.length - x.length;`

```
friend constexpr iter_difference_t<I> operator-(
const counted_iterator& x, default_sentinel_t);
```

Effects: Equivalent to: `return -x.length;`

```
friend constexpr iter_difference_t<I> operator-(
default_sentinel_t, const counted_iterator& y);
```

Effects: Equivalent to: `return y.length;`

```
constexpr counted_iterator& operator-=(iter_difference_t<I> n)
requires RandomAccessIterator<I>;
```

Expects: `-n <= length`.

Effects: Equivalent to:

```
current -= n;
length += n;
return *this;
```

Reference

References

[P0541] Eric Niebler *Ranges TS: Post-Increment on Input and Output Iterators*
https://wg21.link/P0541

[P0896] Eric Niebler, Casey Carter, Christopher Di Bella. *The One Ranges Proposal*
https://wg21.link/P0896

[P0902] Andrew Hunter *Move-only iterators*
https://wg21.link/P0902

[P1035] Christopher Di Bella *Input range adaptors*
https://wg21.link/P1035