Chapter 5

Standard Concepts in Detail

This chapter describes all *concepts* of the C++20 standard library in detail.

5.1 Overview of all Standard Concepts

Table *Basic Concepts for Types and Objects* lists the basic concepts for types and objects in general.

Table *Concepts for Ranges, Iterators, and Algorithms* lists the concepts for ranges, views, iterators, and algorithms.

Table *Auxiliary Concepts* lists the concepts that are mainly used as building blocks for other concepts and usually not directly used by application programmers.

5.1.1 Header Files and Namespaces

Standard concepts are defined in different header files:

- Many basic concepts are defined in header `<concepts>`, which is included by `<ranges>` and `<iterator>`.
- Concepts for iterators are defined in header `<iterator>`.
- Concepts for ranges are defined in header `<ranges>`.
- `three_way_comparable` concepts are defined in `<compare>` (which is included by almost every other header file)
- `uniform_random_bit_generator`, which is defined in `<random>`

Almost all concepts are defined in namespace `std`. The only exception are the ranges concepts, which are defined in namespace `std::ranges`.

Concept	Constraint
integral	integral type
signed_integral	signed integral type
unsigned_integral	unsigned integral type
floating_point	floating-point type
movable	Supports move initialization/assignment and swaps
copyable	Supports move and copy initialization/assignment and swaps
semiregular	Supports default initialization, copies, moves, and swaps
regular	Supports default initialization, copies, moves, swaps, and equality comparisons
same_as	Same types
convertible_to	Type convertible to another
derived_from	Type convertible from another
constructible_from	Type constructible from others
assignable_from	Type assignable from another
swappable_with	Type swappable with another
common_with	Two types have a common type
common_reference_with	Two types have a common reference type
equality_comparable	Type supports checks for equality
equality_comparable_with	Can check two types for equality
totally_ordered	Types supports a strict weak ordering
totally_ordered_with	Can check two types for strict weak ordering
three_way_comparable	Can apply all comparison operators (including operator <=>)
three_way_comparable_with	Can compare two types with all comparison operators (incl. operator <=>)
invocable	Type is a callable for specified arguments
regular_invocable	Type is a callable for specified arguments (no modifications)
predicate	Type is a predicate (callable that returns a Boolean value)
relation	A callable type defines a relation between two types
equivalence_relation	A callable type defines an equality relation between two types
strict_weak_order	A callable type defines an ordering relation between two types
uniform_random_bit_generator	A callable type can be used as a random-number generator

Table 5.1. Basic Concepts for Types and Objects

Concept	Constraint
default_initializable	Type is default initializable
move_constructible	Type supports move initializations
copy_constructible	Type supports copy initializations
destructible	Type is destructible
swappable	Type is swappable
weakly_incrementable	Type supports the increment operators
incrementable	Type supports equality-preserving increment operators

Table 5.2. Auxiliary Concepts

Concept	Constraint
range	Type is a range
output_range	Type is a range to write to
input_range	Type is a range to read from
forward_range	Type is a range to read from multiple times
bidirectional_range	Type is a range to read forward and backwards from
random_access_range	Type is a range that supports jumping around over elements
contiguous_range	Type is a range with elements in contiguous memory
sized_range	Type is a range with cheap size support
common_range	Type is a range with iterators and sentinels having the same type
borrowed_range	Type is lvalue or borrowed range
view	Type is a view
viewable_range	Type is or can be converted to a view
indirectly_writable	Type can be used to write to where it refers
indirectly_readable	Type can be used to read from where it refers
indirectly_movable	Type refers to movable objects
indirectly_movable_storable	Type refers to movable objects with support for temporaries
indirectly_copyable	Type refers to copyable objects
indirectly_copyable_storable	Type refers to copyable objects with support for temporaries
indirectly_swappable	Type refers to swappable objects
indirectly_comparable	Type refers to comparable objects
input_output_iterator	Type is an iterator
output_iterator	Type is an output iterator
input_iterator	Type is (at least) an input iterator
forward_iterator	Type is (at least) a forward iterator
bidirectional_iterator	Type is (at least) a bidirectional iterator
random_access_iterator	Type is (at least) a random-access iterator
contiguous_iterator	Type is an iterator to elements in contiguous memory
sentinel_for	Type can be used as sentinel for an iterator type
sized_sentinel_for	Type can be used as sentinel for an iterator type with cheap computation of distances
permutable	Type is a (at least) forward iterator that can reorder elements
mergeable	Two types can be used to merge sorted elements to a third type
sortable	A type is sortable (according to a comparison and projection)
indirectly_unary_invocable	Operation can be used to call it with the value of an iterator type
indirectly_regular_unary_invocable	Stateless operation can be used to call it with the value of an iterator type
indirect_unary_predicate	Unary predicate can be used to call it with the value of an iterator type
indirect_binary_predicate	Binary predicate can be used to call it with the values of two iterators types
indirect_equivalence_relation	Predicate can be used to check two values of the passed iterator(s) for equality
indirect_strict_weak_order	Predicate can be used to order two values of the passed iterator(s)

Table 5.3. Concepts for Ranges, Iterators, and Algorithms

5.1.2 Standard Concepts Subsume

The concepts provided by the C++ standard library are carefully designed to subsume other concepts when it makes sense. In fact, they build a pretty complex subsumption graph. Figure 5.1 gives some impression of how complex it is.

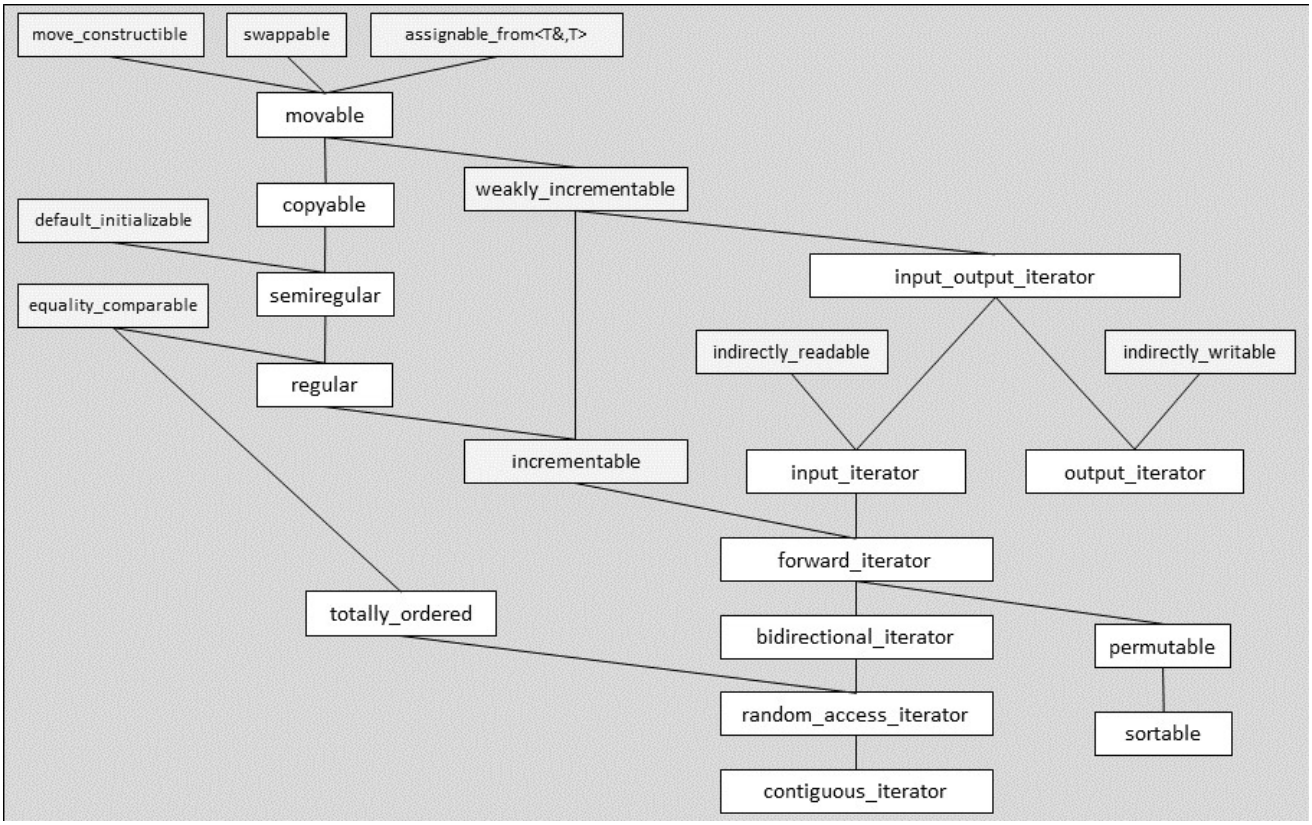

Figure 5.1. Subsumption graph of C++ standard concepts (extract)

For this reason, the description of the concepts lists which key other concepts are subsumed.

5.2 Language-Related Concepts

This section lists the concepts that apply to objects and types in general.

5.2.1 Arithmetic Concepts

`std::integral<T>`

- guarantees that type *T* is an integral type (including `bool` and all character types).
- Requires:
  - The type trait `std::is_integral_v<T>` yields `true`

`std::signed_integral<T>`

- guarantees that type *T* is a signed integral type (including signed character types, which `char` might be).
- Requires:
  - `std::integral<T>` is satisfied
  - The type trait `std::is_signed_v<T>` yields `true`

`std::unsigned_integral<T>`

- guarantees that type *T* is an unsigned integral type (including `bool` and unsigned character types, which `char` might be).
- Requires:
  - `std::integral<T>` is satisfied
  - `std::signed_integral<T>` is not satisfied

`std::floating_point<T>`

- guarantees that type *T* is a floating point type (`float`, `double`, or `long double`).
- The concept was introduced to be able to define the mathematical constants.
- Requires:
  - The type trait `std::is_floating_point_v<T>` yields `true`

5.2.2 Object Concepts

For objects (types that are neither references, functions, or `void`) there is a hierarchy of required basic operations.

`std::movable<T>`

- guarantees that type *T* is movable and swappable. That is, you can move construct, move assign, and swap with another object of the type.
- Requires:
  - `std::move_constructible<T>` is satisfied
  - `std::assignable_from<T&, T>` is satisfied

  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`copyable`**`<`*T*`>`

- guarantees that type *T* is copyable (which implies movable and swappable).
- Requires:
  - `std::movable<`*T*`>` is satisfied
  - `std::copy_constructible<`*T*`>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`.
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`semiregular`**`<`*T*`>`

- guarantees that type *T* is semiregular (can default initialize, copy, move, and swap).
- Requires:
  - `std::copyable<`*T*`>` is satisfied
  - `std::default_initializable<`*T*`>` is satisfied
  - `std::movable<`*T*`>` is satisfied
  - `std::copy_constructible<`*T*`>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`.
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`regular`**`<`*T*`>`

- guarantees that type *T* is regular. (can default initialize, copy, move, swap, and check for equality).
- Requires:
  - `std::semiregular<`*T*`>` is satisfied
  - `std::equality_comparable<`*T*`>` is satisfied
  - `std::copyable<`*T*`>` is satisfied
  - `std::default_initializable<`*T*`>` is satisfied
  - `std::movable<`*T*`>` is satisfied
  - `std::copy_constructible<`*T*`>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`.
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

5.2.3 Concepts for Relations between Types

`std::`**`same_as`**`<`*T1*`,` *T2*`>`

- guarantees that the types *T1* and *T2* are the same.

- The concept calls the `std::is_same_v` type trait twice to ensure that the order of the parameters does not matter.
- Requires:
  - The type trait `std::is_same_v<`*T1*`,` *T2*`>` yields `true`
  - The order of *T1* and *T2* does not matter.

`std::`**`convertible_to`**`<`*From*`,` *To*`>`

- guarantees that objects of type *From* are both implicitly and explicitly convertible to objects of type *To*.
- Requires:
  - The type trait `std::is_convertible_v<`*From*`,` *To*`>` yields `true`
  - A `static_cast` from *From* to *To* is supported
  - You can return an object of type *From* as a *To*

`std::`**`derived_from`**`<`*D*`,` *B*`>`

- guarantees that type *D* is publicly derived from type *B* (or *D* and *B* are the same) so that any pointer of type *D* can be converted to a pointer of type *B*. In other words: the concept guarantees that a *D* reference/pointer can be used as a *B* reference/pointer.
- Requires:
  - The type trait `std::is_base_of_v<`*B*`,` *D*`>` yields `true`
  - The type trait `std::is_convertible_v` for a `const` pointer of type *D* to *B* yields `true`

`std::`**`constructible_from`**`<`*T*`,` *Args...*`>`

- guarantees that you can initialize an object of type *T* with parameters of types *Args...*
- Requires:
  - `std::destructible<`*T*`>` is satisfied
  - The type trait `std::is_constructible_v<`*T*`,` *Args...*`>` yields true

`std::`**`assignable_from`**`<`*To*`,` *From*`>`

- guarantees that you can move or copy assign a value of type *From* to a value of type *To*.
  The assignment also has to yield the original *To* object.
- Requires:
  - *To* has to be an lvalue reference
  - `std::common_reference_with<`*To*`,` *From*`>` is satisfied for `const` lvalue references of the types
  - Operator = has to be supported and yield the same type as *To*

`std::`**`swappable_with`**`<`*T1*`,` *T2*`>`

- guarantees that values of types *T1* and *T2* can be swapped.
- Requires:
  - `std::common_reference_with<`*T1*`,` *T2*`>` is satisfied
  - Any two objects of types *T1* and *T2* can swap values with each other by using `std::ranges::swap()`.

`std::`**`common_with`**`<`*T1*`,` *T2*`>`

- guarantees that types *T1* and *T2* share a common type to which they can be explicitly converted.
- Requires:
  - The type trait `std::common_type_t<`*T1*`, `*T2*`>` yields a type
  - A `static_cast` is supported to their common type
  - References of both types share a `common_reference` type
  - The order of *T1* and *T2* does not matter.

`std::`**`common_reference_with`**`<`*T1*`, `*T2*`>`

- guarantees that types *T1* and *T2* share a `common_reference` type to which they can be explicitly and implicitly converted.
- Requires:
  - The type trait `std::common_reference_t<`*T1*`, `*T2*`>` yields a type
  - Both types are `std::convertible_to` their common reference type
  - The order of *T1* and *T2* does not matter.

5.2.4 Comparison Concepts

`std::`**`equality_comparable`**`<`*T*`>`

- guarantees that objects of type *T* are comparable with operators `==` and `!=`. The order should not matter.
- Operator `==` for *T* should be symmetric and transitive:
  - `t1==t2` is `true` if and only if `t2==t1`.
  - If `t1==t2` and `t2==t3` are `true`, then `t1==t3` are `true`.

  However, this a semantic constraint that cannot be checked at compile-time.
- Requires:
  - Both `==` and `!=` are supported and yield a value convertible to `bool`.

`std::`**`equality_comparable_with`**`<`*T1*`, `*T2*`>`

- guarantees that objects of types *T1* and *T2* are comparable using operators `==` and `!=`.
- Requires:
  - `==` and `!=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`.

`std::`**`totally_ordered`**`<`*T*`>`

- guarantees that objects of type *T* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` so that two values are always either, equal, or less or greater.
- The concept does ***not*** claim that type `T` has a total order for all values. In fact, the expression `std::totally_ordered<double>` yields `true` although floating-point values do not have a total order:

```
std::totally_ordered<double>                    // true
std::totally_ordered<std::pair<double, double>>  // true
```

```
std::totally_ordered<std::complex<int>>          // false
```

This concepts is therefore provided to check the formal requirements to be able to sort element. It is used by `std::ranges::less`, which is the default sorting criterion for sorting algorithms. That way, sorting algorithms do not fail to compile if types do not have a total order for all values. Supporting all 6 basic comparison operators is enough. The values to sort should have a total or weak order. However, this a semantic constraint that cannot be checked at compile-time.

- Requires:
  - `std::equality_comparable<`*T*`>` is satisfied
  - All comparisons with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`.

`std::`**`totally_ordered_with`**`<`*T1*`, `*T2*`>`

- guarantees that objects of types *T1* and *T2* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` so that two values are always either equal or less or greater.
- Requires:
  - `==`, `!=`, `<`, `<=`, `>`, and `>=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`.

`std::`**`three_way_comparable`**`<`*T*`>`
`std::`**`three_way_comparable`**`<`*T*`, `*Cat*`>`

- guarantees that objects of type *T* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and operator `<=>` (and have at least the comparison category type *Cat*). If no *Cat* is passed `std::partial_ordering` is required.
- This concept is defined in header file `<compare>`
- Note that this concept does not imply and subsume `std::equality_comparable` because the latter requires that operator `==` yields true only for two objects that are equal. With a weak or partial order this might not be the case.
- Note that this concept does not imply and subsume std::totally_ordered because the latter requires that the comparison category is `std::strong_ordering` or `std::weak_ordering`.
- Requires:
  - All comparisons with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`.
  - Any comparison with operators `<=>` yields a comparison category (which is at least *Cat*)

`std::`**`three_way_comparable_with`**`<`*T1*`, `*T2*`>`
`std::`**`three_way_comparable_with`**`<`*T1*`, `*T2*`, `*Cat*`>`

- guarantees that any two objects of types *T1* and *T2* are comparable with operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and operator `<=>` (and have at least the comparison category type *Cat*). If no *Cat* is passed `std::partial_ordering` is required.
- This concept is defined in header file `<compare>`
- Note that this concept does not imply and subsume `std::equality_comparable_with` because the latter requires that operator `==` yields true only for two objects that are equal. With a weak or partial order this might not be the case.

- Note that this concept does now imply and subsume std::totally_ordered_with because the latter requires that the comparison category is `std::strong_ordering` or `std::weak_ordering`.
- Requires:
  - `std::three_way_comparable` is satisfied for values and common references of *T1* and *T2* (and *Cat*)
  - All comparisons with operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`.
  - Any comparison with operators `<=>` yields a comparison category (which is at least *Cat*)
  - The order of *T1* and *T2* does not matter.

5.3 Concepts for Iterators and Ranges

This section lists all basic concepts for iterators and ranges, which are useful to use in algorithms and similar functions.

Note that the concepts for ranges are provided in namespace `std::ranges` instead of `std`. They are declared in header file `<ranges>`.

Concepts for iterators are declared in header file `<iterator>`.

5.3.1 Concepts for Ranges and Views

Several concepts are defined to constrain ranges. They correspond with the concepts for iterators and deal with the fact that we have new iterator categories since C++20.

`std::ranges::`**`range`**`<`*Rg*`>`

- guarantees that *Rg* is a valid range.
- This means that objects of type *Rg* support to iterate over the elements by using `std::ranges::begin()` and `std::ranges::end()`.

  This is the case if the range either is an array, or provides `begin()` and `end()` members or can be used with free-standing `begin()` and `end()` functions.
- In addition, for `std::ranges::begin()` and `std::ranges::end()` the following constraints apply:
  - They have to operate in (amortized) constant time.
  - They do not modify the range.
  - `begin()` yields the same position when called multiple times (unless the range does not provide at least forward iterators).

  That all means that we can iterate over all elements with good performance (even multiple times unless we have pure input iterators).
- Requires:
  - For an objects `rg` of types *Rg* `std::ranges::begin(rg)` is supported and `std::ranges::end(rg)` is supported

`std::ranges::`**`output_range`**`<`*Rg*`, `*T*`>`

- guarantees that *Rg* is a range that provides at least output iterators (iterators you can use to write) that accept values of type *T*.

- Requires:
  - `std::range<Rg>` is satisfied
  - `std::output_iterator` is satisfied for the iterator type and `T`

`std::ranges::`**`input_range`**`<Rg>`

- guarantees that *Rg* is a range that provides at least input iterators (iterators you can use to read).
- Requires:
  - `std::range<Rg>` is satisfied
  - `std::input_iterator` is satisfied for the iterator type

`std::ranges::`**`forward_range`**`<Rg>`

- guarantees that *Rg* is a range that provides at least forward iterators (iterators you can use to read and write and to iterate over multiple times).
- Note that the `iterator_category` member of the iterators may not match. For iterators that yield prvalues it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::input_range<Rg>` is satisfied
  - `std::forward_iterator` is satisfied for the iterator type

`std::ranges::`**`bidirectional_range`**`<Rg>`

- guarantees that *Rg* is a range that provides at least bidirectional iterators (iterators you can use to read and write and to iterator over also backwards).
- Note that the `iterator_category` member of the iterators may not match. For iterators that yield prvalues it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::forward_range<Rg>` is satisfied
  - `std::bidirectional_iterator` is satisfied for the iterator type

`std::ranges::`**`random_access_range`**`<Rg>`

- guarantees that *Rg* is a range that provides random-access iterators (iterators you can use to read and write, jump back and forth, and compute the distance).
- Note that the `iterator_category` member of the iterators may not match. For iterators that yield prvalues it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::bidirectional_range<Rg>` is satisfied
  - `std::random_access_iterator` is satisfied for the iterator type

`std::ranges::`**`contiguous_range`**`<Rg>`

- guarantees that *Rg* is a range that provides random access iterators with the additional constraint that the elements are stored in contiguous memory.

- Note that the iterator_category member of the iterators does not match. If defined, it is only std::random_access_iterator_tag or if the values are prvalues it is even only std::input_iterator_tag.
- Requires:
  - std::random_access_range<*Rg*> is satisfied
  - std::contiguous_iterator is satisfied for the iterator type
  - Calling std::ranges::data() yields a raw pointer to the first element

std::ranges::**sized_range**<*Rg*>

- guarantees that *Rg* is a range where the number of elements can be computed in constant time (either by calling the member size() or by computing the difference between the begin and the end).
- If this concept is satisfied, std::ranges::size() is fast and well defined for objects of type *Rg*.
- Note that the performance aspect of this concept is a semantic constraint, which cannot be checked at compile time. To signal that a types does not satisfy this concept although it provides size(), you can define that std::disable_sized_range<*Rg*> yields true.[1]
- Requires:
  - std::range<*Rg*> is satisfied
  - Calling std::ranges::size() is supported

std::ranges::**common_range**<*Rg*>

- guarantees that *Rg* is a range where the begin iterator and the sentinel (end iterator) have the same type.
- The guarantee is always given by:
  - All standard containers (vector, list, etc.)
  - empty_view
  - single_view
  - common_view

  The guarantee is not given by
  - take views
  - const drop views
  - iota views with no end value or an end value of different type

  For other views, it depends on the type of the underlying ranges.
- Requires:
  - std::range<*Rg*> is satisfied
  - std::ranges::iterator_t<*Rg*> and std::ranges::sentinel_t<*Rg*> have the same type.

std::ranges::**borrowed_range**<*Rg*>

- guarantees that *Rg* is an lvalue or a *borrowed range*. So, the concept specifies that *Rg* ***can be used*** like a borrowed range.

[1] The real meaning of disable_sized_range is "ignore size() for sized_range."

- That is, the iterators are not tied to the lifetime of the range. That means that iterators are more safe to use, because they cannot dangle when the range they were created from gets destroyed. However, they can still dangle if the iterators of the range refer to an underlying range and the underlying range is no longer there.
- The guarantee is given if *Rg* is an lvalue (such as an object with a name) or if the variable template `std::ranges::enable_borrowed_range<`*Rg*`>` is `true`, which is the case for the following views: `subrange`, `ref_view`, `string_view`, `span`, `iota_view`, and `empty_view`.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - *Rg* is an lvalue or `enable_borrowed_range<`*Rg*`>` yields `true`

`std::ranges::`**`view`**`<`*Rg*`>`

- guarantees that *Rg* is a view (a range that is cheap to copy or move, assign, and destroy).
- A view has the following requirements:[2]
  - It has to be a range (support to iterate over the elements).
  - It has to be movable.
  - Move construction and if available copy construction have to have constant complexity.
  - Move assignment and if available copy assignment have to be cheap (constant complexity or not worse that destruction plus creation).

  All but the last requirements are checked by corresponding concepts. The last requirement is a semantic constraint and has to be guaranteed by the implementer of a type by deriving publicly from `std::ranges::view_interface` or from `std::ranges::view_base` or by specializing `std::ranges::enable_view<`*Rg*`>` to yield `true`.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::movable<`*Rg*`>` is satisfied
  - The variable template `std::ranges::enable_view<`*Rg*`>` is `true`

`std::ranges::`**`viewable_range`**`<`*Rg*`>`

- guarantees that *Rg* is a range that can be safely converted to a view with the `std::views::all()` adaptor.
- The concept is satisfied if *Rg* either is already a view or an lvalue of a range or a movable rvalue or a range but no initializer list.[3]
- Requires:
  - `std::range<`*Rg*`>` is satisfied

[2] Initially C++20 required that views are also default constructible and that destruction has to be cheap. However, these requirements were removed with `http://wg21.link/p2325r3` and `http://wg21.link/p2415r2`.

[3] For lvalues of move-only view types there is an issue that `all()` is ill-formed although `viewable_range` is satisfied.

5.3.2 Concepts for Pointers-Like Objects

This section lists all standard concepts for objects for which you can use operator * to deal with a value they point to. This usually applies to raw pointers, smart pointers, and iterators. Therefore, these concepts are usually used as base constraints for concepts that deal with iterators and algorithms.

Note that types support of operator -> is not required for these concepts.

These concepts are declared in header `<iterator>`.

Basic Concepts for Indirect Support

`std::`**`indirectly_writable`**`<`*P*`, `*Val*`>`

- guarantees that *P* is a pointer-like object supporting operator * to assign a *Val*.
- satisfied by non-const raw pointers, smart pointers, and iterators provided *Val* can be assigned to where *P* refers to.

`std::`**`indirectly_readable`**`<`*P*`>`

- guarantees that *P* is a pointer-like object supporting operator * for read access.
- satisfied by raw pointers, smart pointers, and iterators.
- Requires:
  - The resulting value has the same reference type for both `const` and non-`const` objects (which rules out `std::optional<>`). That ensures that the constness of *P* does not propagate to where it points to (which is usually not the case when the operator returns a reference to a member).
  - `std::iter_value_t<`*P*`>` must be valid. The type does ***not*** have to support operator ->.

Concepts for Indirectly Readable Objects

For pointer-like concept that are indirectly readable you can check additional constraints:

`std::`**`indirectly_movable`**`<`*InP*`,`*OutP*`>`

- guarantees that values of *InP* can be move assigned directly to values of *OutP*.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP, outPos) {
  *outPos = std::move(*inPos);
}
```

- Requires:
  - `std::indirectly_readable<`*InP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for rvalue references of the values of *InP* to (the values of) *OutP*.

`std::`**`indirectly_movable_storable`**`<`*InP*`,`*OutP*`>`

- guarantees that values of *InP* can be move assigned indirectly to values of *OutP* even when using a (temporary) object of the type to where *OutP* points to.

- With this concept, the following code is valid:

```
void foo(InP inPos, OutP, outPos) {
  OutP::value_type tmp = std::move(*inPos);
  *outPos = std::move(tmp);
}
```

- Requires:
  - `std::indirectly_movable<`*InP*`, `*OutP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for the *InP* value to the objects *OutP* refers to.
  - `std::movable` is satisfied for the values *InP* refers to.
  - Rvalues *InP* refers to are copy/move constructible and assignable.

`std::`**`indirectly_copyable`**`<`*InP*`, `*OutP*`>`

- guarantees that values of *InP* can be assigned directly to values of *OutP*.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP outPos) {
  *outPos = *inPos;
}
```

- Requires:
  - `std::indirectly_readable<`*InP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for references of the values of *InP* to (the values of) *OutP*.

`std::`**`indirectly_copyable_storable`**`<`*InP*`, `*OutP*`>`

- guarantees that values of *InP* can be assigned indirectly to values of *OutP* even when using a (temporary) object of the type to where *OutP* points to.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP outPos) {
  OutP::value_type tmp = *inPos;
  *outPos = tmp;
}
```

- Requires:
  - `std::indirectly_copyable<`*InP*`, `*OutP*`>` is satisfied.
  - `std::indirectly_writable` is satisfied for `const` lvalue and rvalue references of the *InP* value to the objects *OutP* refers to.
  - `std::copyable` is satisfied for the values *InP* refers to.
  - The values *InP* refers to are copy/move constructible and assignable.

`std::`**`indirectly_swappable`**`<`*P*`>`
`std::`**`indirectly_swappable`**`<`*P1*`, `*P2*`>`

- guarantees that values of *P* or *P1* and *P2* can be swapped (using `std::ranges::iter_swap()`).
- Requires:

  - `std::indirectly_readable<`*P1*`>` (and `std::indirectly_readable<`*P2*`>`) are satisfied.
  - For any two objects of types *P1* and *P2* `std::ranges::iter_swap()` is supported.

`std::`**`indirectly_comparable`**`<`*P1*`, `*P2*`, `*Comp*`>`
`std::`**`indirectly_comparable`**`<`*P1*`, `*P2*`, `*Comp*`, `*Proj1*`>`
`std::`**`indirectly_comparable`**`<`*P1*`, `*P2*`, `*Comp*`, `*Proj1*`, `*Proj2*`>`

- guarantees that you can compare the elements (optionally transformed with *Proj1* and *proj2*) to where *P1* and *P2* refer
- Requires:
  - `std::indirect_binary_predicate` for *Comp*
  - `std::projected<`*P1*`, `*Proj1*`>` and `std::projected<`*P2*`, `*Proj2*`>` are satisfied with `std::identity` as default projections

5.3.3 Concepts for Iterators

This section lists the concepts to require different types of iterators. They deal with the fact that we have new iterator categories since C++20 and correspond with the concepts for ranges.

These concepts are provided in header `<iterator>`.

`std::`**`input_output_iterator`**`<`*Pos*`>`

- guarantees that *Pos* supports the basic interface of all iterators: `operator++` and `operator*`, where `operator*` has to refer to a value.

  The concept does not require that the iterator is copyable (thus, this is less than the basic requirements for iterators used by algorithms).
- Requires:
  - `std::weakly_incrementable<`*pos*`>` is satisfied
  - Operator `*` yields a reference

`std::`**`output_iterator`**`<`*Pos*`, `*T*`>`

- guarantees that *Pos* is an output iterator (an iterator where you can assign values to the elements) to which values you can assign values of type*T*.
- Iterators of type *Pos* can be used to assign a value `val` of type *T* with:

```
*i++ = val;
```

- These iterators are only good for single-pass iterations.
- Requires:
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_writable<`*Pos*`, `*I*`>` is satisfied

`std::`**`input_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is an input iterator (an iterator where you can read values from the elements).
- These iterators are only good for single-pass iterations if not also `std::forward_iterator` is satisfied.

- Requires:
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::input_iterator_tag`

`std::`**`forward_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is a forward iterator (a reading iterator with which you can iterate forward multiple times over elements).
- Note that the `iterator_category` member of the iterator may not match. For iterators that yield prvalues it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::input_iterator<`*Pos*`>` is satisfied
  - `std::incrementable<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::forward_iterator_tag`

`std::`**`bidirectional_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is a bidirectional iterator (a reading iterator with which you can iterate forward and backward multiple times over elements).
- Note that the `iterator_category` member of the iterator may not match. For iterators that yield prvalues it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::forward_iterator<`*Pos*`>` is satisfied
  - Support to iterate backward with operator `--`
  - *Pos* has an iterator category derived from `std::bidirectional_iterator_tag`

`std::`**`random_access_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is a random-access iterator (a reading iterator with which you can jump back and forth over the elements).
- Note that the `iterator_category` member of the iterator may not match. For iterators that yield prvalues it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::bidirectional_iterator<`*Pos*`>` is satisfied
  - `std::totally_ordered<`*Pos*`>` is satisfied
  - `std::sized_sentinel_for<`*Pos*`, `*Pos*`>` is satisfied
  - Support of `+`, `+=`, `-`, `-=`, `[]`
  - *Pos* has an iterator category derived from `std::random_access_iterator_tag`

`std::`**`contiguous_iterator`**`<`*Pos*`>`

- guarantees that *Pos* is an iterator iterating over elements in contiguous memory.

- Note that the `iterator_category` member of the iterator does not match. If defined, it is only `std::random_access_iterator_tag` or if the values are prvalues it is even only `std::input_iterator_tag`.
- Requires:
  - `std::random_access_iterator<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::contiguous_iterator_tag`
  - `to_address()` for an element is a raw pointer to the element

`std::`**`sentinel_for`**`<`*S*`, `*Pos*`>`

- guarantees that *S* can be used as sentinel (end iterator of maybe different type) for *Pos*.
- Requires:
  - `std::semiregular<`*S*`>` is satisfied
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - Can compare *Pos* and *S* with operators == and `!=`

`std::`**`sized_sentinel_for`**`<`*S*`, `*Pos*`>`

- guarantees that *S* can be used as sentinel (end iterator of maybe different type) for *Pos* and you can compute the distance between them in constant time.
- To signal that you can compute the distance but that does not have constant complexity, you can define that `std::disable_sized_sentinel_for<`*Rg*`>` yields `true`.
- Requires:
  - `std::sentinel_for<`*S*`, `*Pos*`>` is satisfied
  - Calling operator - for a *Pos* and *S* yields a value of the difference type of the iterator
  - `std::disable_sized_sentinel_for<`*S*`, `*Pos*`>` is not defined to yield `true`

5.3.4 Iterator Concepts for Algorithms

`std::`**`permutable`**`<`*Pos*`>`

- guarantees that you can iterate forward by using operator ++ and reorder elements by moving and swapping them
- Requires:
  - `std::forward_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_movable_storable<`*Pos*`>` is satisfied
  - `std::indirectly_swappable<`*Pos*`>` is satisfied

`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`, `*Proj1*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`, `*Proj1*`, `*Proj2*`>`

- guarantees that you can merge the elements of two sorted sequences to where *Pos1* and *Pos2* refer by copying them into a sequence to where *ToPos* refers. The order is defined by operator < or *Comp* (applied to the values optionally transformed with projections *Proj1* and *Proj2*).
- Requires:
  - `std::input_iterator` is satisfied for both *Pos1* and *Pos2*
  - `std::weakly_incrementable<`*ToPos*`>` is satisfied
  - `std::indirectly_copyable<`*PosN*`,` *ToPos*`>` is satisfied for both *Pos1* and *Pos2*
  - `std::indirect_strict_weak_order` of *Comp* as well as `std::projected<`*Pos1*`,` *Proj1*`>`, and `std::projected<`*Pos2*`,` *Proj2*`>` are satisfied (with < as default comparison and `std::identity` as default projections), which implies
    - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
    - `std::copy_constructible<`*Comp*`>` is satisfied
    - `std::strict_weak_order<`*Comp*`>` is satisfied for *Comp*`&` and the (projected) value/reference types

`std::`**`sortable`**`<`*Pos*`>`
`std::`**`sortable`**`<`*Pos*`,` *Comp*`>`
`std::`**`sortable`**`<`*Pos*`,` *Comp*`,` *Proj*`>`

- guarantees that you can sort the elements iterator *Pos* refers to with operator < or *Comp* (after optionally applying the projection *Proj* to the values).
- Requires:
  - `std::permutable<`*Pos*`>` is satisfied
  - `std::indirect_strict_weak_order` of *Comp* and the (projected) values is satisfied (with < as default comparison and `std::identity` as default projection), which implies
    - `std::indirectly_readable<`*Pos*`>` is satisfied
    - `std::copy_constructible<`*Comp*`>` is satisfied
    - `std::strict_weak_order<`*Comp*`>` is satisfied for *Comp*`&` and the (projected) value/reference types

5.4 Concepts for Callables

This section lists all concepts for callables

- Functions
- Function objects
- Lambdas
- Pointer to members (with the type of the object as additional parameter)

5.4.1 Basic Concepts for Callables

`std::`**`invocable`**`<`*Op*`, `*ArgTypes...*`>`

- guarantees that *Op* is callable for arguments of types *ArgTypes...*.
- *Op* can be a function, function object, lambda, or pointer-to-member.
- To document that neither the operation nor the passed arguments are modified you can use `std::regular_invocable` instead. However, note that there is only a semantic difference between these two concepts, which cannot be checked at compile time. So the concept differences only document the intention.
- For example:

```
struct S {
  int member;
  int mfunc(int);
  void operator()(int i) const;
};

void testCallable()
{
  std::invocable<decltype(testCallable)>       // satisfied
  std::invocable<decltype([](int){}), char>    // satisfied (char converts to int)
  std::invocable<decltype(&S::mfunc), S, int> // satisfied (member-pointer, object, args)
  std::invocable<decltype(&S::member), S>;     // satisfied (member-pointer and object)
  std::invocable<S, int>;                      // satisfied due to operator()
}
```

  Note that as a type constraint, you only have to specify the parameter types:

```
void callWithIntAndString(std::invocable<int, std::string> auto op);
```

  For a complete example, see the use of a lambda as a non-type template parameter.
- Requires:
  - `std::invoke(op, args)` is valid for an `op` of type *Op* and `args` of type *ArgTypes...*

`std::`**`regular_invocable`**`<`*Op*`, `*ArgTypes...*`>`

- guarantees that *Op* is callable for arguments of types *ArgTypes...* and that the call does neither change the state of the passed operation nor of the passed arguments.
- *Op* can be a function, function object, lambda, or pointer-to-member.
- Note that the difference to the concept `std::invocable` is purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::invoke(op, args)` is valid for an `op` of type *Op* and `args` of type *ArgTypes...*

`std::`**`predicate`**`<`*Op*`, `*ArgTypes...*`>`

- guarantees that the callable (function, function object, lambda) *Op* is a predicate when called for the arguments of types *ArgTypes...*.
- This implies that the call does neither change the state of the passed operation nor of the passed arguments.
- Requires:
  - `std::regular_invocable<`*Op*`>` is satisfied
  - All calls of *Op* with *ArgTypes...* yield a value usable as Boolean value and convertible to `bool`

`std::`**`relation`**`<`*Pred*`, `*T1*`, `*T2*`>`

- guarantees that any two objects of types *T1* and *T2* have a binary relationship in that they can be passed as arguments to the binary predicate *Pred*
- This implies that the call does neither change the state of the passed operation nor of the passed arguments.
- Note that the differences to the concepts `std::equivalence_relation` and `std::strict_weak_order` are purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`equivalence_relation`**`<`*Pred*`, `*T1*`, `*T2*`>`

- guarantees that any two objects of types *T1* and *T2* have an equivalence relationship when compared with *Pred*.

  That is, they can be passed as arguments to the binary predicate *Pred* and the relationship is reflexive, symmetric, and transitive.
- This implies that the call does neither change the state of the passed operation nor of the passed arguments.
- Note that the differences to the concepts `std::relation` and `std::strict_weak_order` are purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`strict_weak_order`**`<`*Pred*`, `*T1*`, `*T2*`>`

- guarantees that any two objects of types *T1* and *T2* have a strict weak ordering relationship when compared with *Pred*.

  That is, they can be passed as arguments to the binary predicate *Pred* and the relationship is irreflexive and transitive.
- This implies that the call does neither change the state of the passed operation nor of the passed arguments.
- Note that the differences to the concepts `std::relation` and `std::equivalence_relation` are purely semantic and cannot be checked at compile time. So the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`uniform_random_bit_generator`**`<`*Op*`>`

- guarantees that *Op* can be used as a random number generator returning unsigned integral values of (ideally) equal probability
- This concept is defined in header file `<random>`
- Requires:
  - `std::invocable<`*Op*`&>` is satisfied
  - `std::unsigned_integral` is satisfied for the result of the call
  - Expressions *Op*`::min()` and *Op*`::max()` are supported and yield the same type as the generator calls
  - *Op*`::min()` is less than *Op*`::max()`

5.4.2 Concepts for Callables Used by Iterators

`std::`**`indirectly_unary_invocable`**`<`*Op*`, `*Pos*`>`

- guarantees that *Op* can be called with the values *Pos* refers to.
- Note that the difference to the concept `indirectly_regular_unary_invocable` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Op*`>` is satisfied
  - `std::invocable` is satisfied for *Op*`&` and the value and (common) reference type of *Pos*
  - The results of calling *Op* with both a value and a reference have a common reference type.

`std::`**`indirectly_regular_unary_invocable`**`<`*Op*`, `*Pos*`>`

- guarantees that *Op* can be called with the values *Pos* refers to and the call does not change the state of *Op*
- Note that the difference to the concept `std::indirectly_unary_invocable` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Op*`>` is satisfied
  - `std::regular_invocable` is satisfied for *Op*`&` and the value and (common) reference type of *Pos*
  - The result of calling *Op* with both a value and a reference have a common reference type.

`std::`**`indirect_unary_predicate`**`<`*Pred*`, `*Pos*`>`

- guarantees that the unary predicate *Pred* can be called with the values *Pos* refers to.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::predicate` is satisfied for *Pred*`&` and the value and (common) reference type of *Pos*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

`std::`**`indirect_binary_predicate`**`<`*Pred*`, `*Pos1*`, `*Pos2*`>`

- guarantees that the binary predicate *Pred* can be called with the values *Pos1* and *Pos2* refer to.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::predicate` is satisfied for *Pred*&, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

`std::`**`indirect_equivalence_relation`**`<`*Pred*`, `*Pos1*`>`
`std::`**`indirect_equivalence_relation`**`<`*Pred*`, `*Pos1*`, `*Pos2*`>`

- guarantees that the binary predicate *Pred* can be called to check whether two values of *Pos1* and *Pos1*/*Pos2* are equivalent
- Note that the difference to the concept `std::indirectly_strict_weak_order` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::equivalence_relation` is satisfied for *Pred*&, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

`std::`**`indirect_strict_weak_order`**`<`*Pred*`, `*Pos1*`>`
`std::`**`indirect_strict_weak_order`**`<`*Pred*`, `*Pos1*`, `*Pos2*`>`

- guarantees that the binary predicate *Pred* can be called to check whether two values of *Pos1* and *Pos1*/*Pos2* have a strict weak order
- Note that the difference to the concept `std::indirectly_equivalence_relation` is purely semantic and cannot be checked at compile time. So the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::strict_weak_order` is satisfied for *Pred*&, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value usable as Boolean value and convertible to `bool`

5.5 Auxiliary Concepts

This section describes some standardized concepts that are mainly specified to implement other concepts. You should usually not use them in application code.

5.5.1 Concepts for Specific Type Attributes

`std::`**`default_initializable`**`<`*T*`>`

- guarantees that the type *T* supports default construction
- Requires:
  - `std::constructible_from<`*T*`>` is satisfied
  - `std::destructible<`*T*`>` is satisfied
  - `T{};` is valid
  - `T x;` is valid

`std::`**`move_constructible`**`<`*T*`>`

- guarantees that objects of type *T* can be initialized with rvalues of their type.
- That means the following operation is valid (although it might copy instead of move):

  ```
  T t2{std::move(t1)}   // for any t1 of type T
  ```

  Afterwards, `t2` shall have the former value of `t1`, which is a semantic constraint that cannot be checked at compile-time.
- Requires:
  - `std::constructible_from<`*T*`, `*T*`>` is satisfied.
  - `std::convertible<`*T*`, `*T*`>` is satisfied.
  - `std::destructible<`*T*`>` is satisfied.

`std::`**`copy_constructible`**`<`*T*`>`

- guarantees that objects of type *T* can be initialized with lvalues of their type.
- That means the following operation is valid:

  ```
  T t2{t1}   // for any t1 of type T
  ```

  Afterwards, `t2` shall be equal to `t1`, which is a semantic constraint that cannot be checked at compile-time.
- Requires:
  - `std::move_constructible<`*T*`>` is satisfied.
  - `std::constructible_from` and `std::convertible_to` are satisfied for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*.
  - `std::destructible<`*T*`>` is satisfied.

`std::`**`destructible`**`<`*T*`>`

- guarantees that objects of type *T* are destructible without throwing an exception.
  Note that even implemented destructors automatically guarantee not to throw unless you explicitly mark them with `noexcept(false)`.
- Requires:
  - The type trait `std::is_nothrow_destructible_v<`*T*`>` yields `true`.

`std::`**`swappable`**`<T>`

- guarantees that you can swap the values of two objects of type *T*.
- Requires:
  - `std::ranges::swap()` can be called for two objects of type *T*

5.5.2 Concepts for Incrementable Types

`std::`**`weakly_incrementable`**`<T>`

- guarantees that type *T* supports increment operators.
- Note that this concept does ***not*** require that two increments of the same value yield equal results. Therefore, this is a requirement for single-pass iterations only.
- In contrast to `std::incrementable` the concept is also satisfied if
  - the type is not default constructible, not copyable, or not equality comparable
  - the postfix increment might return `void` (or any other type)
  - two increments of the same value might give different results
- Note that the differences to concept `std::incrementable` are purely semantic differences so that for types for which increments yield different results might still technically satisfy the concept `incrementable`. To implement different behavior for this semantic difference, you should use iterator concepts instead.
- Requires:
  - `std::default_initializable<T>` is satisfied
  - `std::movable<T>` is satisfied
  - `std::iter_difference_t<T>` is a valid signed integral type

`std::`**`incrementable`**`<T>`

- guarantees that type *T* is an incrementable type, so that you can iterate multiple times over the same sequence of values.
- In contrast to `std::weakly_incrementable` the concept
  - requires that two increments of the same value yield the same results (as it is the case for forward iterators).
  - requires that type *T* is default constructible, copyable, and equality comparable.
  - requires postfix increment to return a copy of the iterator (return type *T*).
- Note that the differences to concept `std::weakly_incrementable` are purely semantic differences so that for types for which increments yield different results might still technically satisfy the concept `incrementable`. To implement different behavior for this semantic difference, you should use iterator concepts instead.
- Requires:
  - `std::weakly_incrementable<T>` is satisfied
  - `std::regular<T>` is satisfied so that the type is default constructible, copyable, and equality comparable.