C++ Concepts and Ranges

HOW TO USE THEM?

Mateusz Pusz
November 16, 2018

Note

Some of the following features are still *work in progress* and a subject to change for C++20

Note

Some of the following features are still *work in progress* and a subject to change for C++20

The most of the features described here were voted to the **C++ International Standard Working Draft** in San Diego *less than a week ago*

Agenda

1. Concepts
2. Standard Library Concepts
3. Iterators and Sentinels
   - concepts and customization points
   - new classes
4. Ranges
5. Views and range adaptor objects
6. Algorithms

CONCEPTS

How do you feel about that interface?

```
void* foo(void* t);
```

How do you feel about that interface?

```
void* foo(void* t);
```

```
template<typename T>
auto foo(T&& t);
```

How do you feel about that interface?

```
void* foo(void* t);
```

```
template<typename T>
auto foo(T&& t);
```

```
auto lambda = [](auto&& t){ /* ... */ };
```

How do you feel about that interface?

```
void* foo(void* t);
```

```
template<typename T>
auto foo(T&& t);
```

```
auto lambda = [](auto&& t){ /* ... */ };
```

Unconstrained template parameters are a **`void*`** of C++

Why concepts?

FOO_LIB.H

```
template<typename T>
void foo(T& t);
```

Why concepts?

FOO_LIB.H

```
template<typename T>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
A a;
foo(a);
```

Why concepts?

```
<source>: In instantiation of 'void detail::foo_impl_part_1(T&) [with T = A]':
<source>:22:20:   required from 'void detail::foo_impl(T&) [with T = A]'
<source>:30:20:   required from 'void foo(T&) [with T = A]'
<source>:38:10:   required from here
<source>:16:25: error: 'begin' was not declared in this scope
    foo_impl_part_1(begin(t), end(t));
                    ~~~~~^~~
<source>:16:25: note: suggested alternative:
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/vector:66,
                 from <source>:1:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/range_access.h:105:37: note:   'std::begin'
   template<typename _Tp> const _Tp* begin(const valarray<_Tp>&);
                                     ^~~~~
<source>:16:33: error: 'end' was not declared in this scope
    foo_impl_part_1(begin(t), end(t));
                              ~~~^~~
<source>:16:33: note: suggested alternative:
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/vector:66,
                 from <source>:1:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/range_access.h:107:37: note:   'std::end'
   template<typename _Tp> const _Tp* end(const valarray<_Tp>&);
                                     ^~~
Compiler returned: 1
```

Why concepts?

FOO_LIB.H

```
template<typename T>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
std::list<A> a;
foo(a);
```

Why concepts?

```
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/algorithm:62,
                 from <source>:2:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h: In instantiation of 'void std::__sort(_RandomAccessIterator, _
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:4834:18:   required from 'void std::sort(_RAIter, _RAIter) [wi
<source>:11:13:   required from 'void detail::foo_impl_part_1(T, T) [with T = std::_List_iterator<A>]'
<source>:17:19:   required from 'void detail::foo_impl_part_1(T&) [with T = std::__cxx11::list<A>]'
<source>:23:20:   required from 'void detail::foo_impl(T&) [with T = std::__cxx11::list<A>]'
<source>:31:20:   required from 'void foo(T&) [with T = std::__cxx11::list<A>]'
<source>:39:10:   required from here
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:1969:22: error: no match for 'operator-' (operand types are 'st
     std::__lg(__last - __first) * 2,
               ~~~~~~~^~~~~~~~~
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algobase.h:67,
                 from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/vector:60,
                 from <source>:1:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_iterator.h:389:5: note: candidate: 'template<class _IteratorL, class _
     operator-(const reverse_iterator<_IteratorL>& __x,
     ^~~~~~~~
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_iterator.h:389:5: note:   template argument deduction/substitution fai
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/algorithm:62,
                 from <source>:2:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:1969:22: note:   'std::_List_iterator<A>' is not derived from '
     std::__lg(__last - __first) * 2,
               ~~~~~~~^~~~~~~~~

... 18 lines more ...

Compiler returned: 1
```

Why concepts?

FOO_LIB.H

```
template<typename T>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
std::vector<A> a;
foo(a);
```

Why concepts?

```
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algobase.h:71,
                 from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/vector:60,
                 from <source>:1:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/predefined_ops.h: In instantiation of 'constexpr bool __gnu_cxx::__ops::_I
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:81:17:   required from 'void std::__move_median_to_first(_Itera
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:1921:34:   required from '_RandomAccessIterator std::__unguarde
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:1953:38:   required from 'void std::__introsort_loop(_RandomAcc
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:1968:25:   required from 'void std::__sort(_RandomAccessIterato
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algo.h:4834:18:   required from 'void std::sort(_RAIter, _RAIter) [wit
<source>:10:13:   required from 'void detail::foo_impl_part_1(T, T) [with T = __gnu_cxx::__normal_iterator<A*, std::vector<A> >]'
<source>:16:19:   required from 'void detail::foo_impl_part_1(T&) [with T = std::vector<A>]'
<source>:22:20:   required from 'void detail::foo_impl(T&) [with T = std::vector<A>]'
<source>:30:20:   required from 'void foo(T&) [with T = std::vector<A>]'
<source>:38:10:   required from here
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/predefined_ops.h:43:23: error: no match for 'operator<' (operand types are
       { return *__it1 < *__it2; }
                ~~~~~~~^~~~~~~~
In file included from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_algobase.h:67,
                 from /opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/vector:60,
                 from <source>:1:
/opt/compiler-explorer/gcc-8.1.0/include/c++/8.1.0/bits/stl_iterator.h:889:5: note: candidate: 'template<class _IteratorL, class _
     operator<(const __normal_iterator<_IteratorL, _Container>& __lhs,
     ^~~~~~~~

... 102 lines more ...

Compiler returned: 1
```

Why concepts?

```
template<typename T>
inline constexpr bool is_sortable_v = /* ... */
```

FOO_LIB.H

```
template<typename T>
void foo(T& t) {
  static_assert(is_sortable_v<T>, "Sortable type expected for T");
  // ...
}
```

MAIN.CPP

```
struct A { int data; };
```

```
std::vector<A> a;
foo(a);
```

Why concepts?

```
<source>: In instantiation of 'void foo(T&) [with T = std::vector<A>]':
<source>:53:6:   required from here
<source>:44:19: error: static assertion failed: Sortable type expected for T
   44 |     static_assert(is_sortable_v<T>, "Sortable type expected for T");
      |                   ^~~~~~~~~~~~~~~~
Compiler returned: 1
```

Why concepts?

```
template<bool B>
using Requires = std::enable_if_t<B, bool>;
```

FOO_LIB.H

```
template<typename T,
  Requires<is_sortable_v<T>> = true>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
std::vector<A> a;
foo(a);
```

Why concepts?

```
<source>: In function 'int main()':
<source>:53:6: error: no matching function for call to 'foo(std::vector<A>&)'
   53 | foo(a);
      |      ^
<source>:43:6: note: candidate: 'template<class T, typename std::enable_if<is_sortable_v<T>, bool>::type <anonymous> > void foo(T&
   43 | void foo(T& t)
      |      ^~~
<source>:43:6: note:   template argument deduction/substitution failed:
<source>:42:39: error: no type named 'type' in 'struct std::enable_if<false, bool>'
   42 |          Requires<is_sortable_v<T>> = true>
      |                                       ^~~~
Compiler returned: 1
```

Why concepts?

FOO_LIB.H

```
template<typename T,
  Requires<is_sortable_v<T>> = true>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
std::vector<A> a;
foo(a);
```

Why concepts?

FOO_LIB.H

```
template<typename T>
  requires is_sortable_v<T>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
std::vector<A> a;
foo(a);
```

Why concepts?

FOO_LIB.H

```
template<typename T>
  requires std::ranges::Sortable<T>
void foo(T& t);
```

MAIN.CPP

```
struct A { int data; };
```

```
std::vector<A> a;
foo(a);
```

Why concepts?

```
<source>:39:6: error: cannot call function 'void foo(T&) [with T = std::__cxx11::vector<A>]'
   39 | foo(a);
      |      ^
<source>:29:6: note:   constraints not satisfied
   29 | void foo(T& t)
      |      ^~~
<source>:29:6: note: in the expansion of concept
   'Sortable<T, std::ranges::less<void>, std::ranges::identity>
    template<class I, class R, class P> concept const std::ranges::Sortable<I, R, P>
    [with I = std::__cxx11::vector<A>; R = std::ranges::less<void>; P = std::ranges::identity]'
In file included from /opt/compiler-explorer/libs/cmcstl2/include/stl2/detail/algorithm/partition_point.hpp:28,
                 from /opt/compiler-explorer/libs/cmcstl2/include/stl2/detail/algorithm/lower_bound.hpp:18,
                 from /opt/compiler-explorer/libs/cmcstl2/include/stl2/detail/algorithm/binary_search.hpp:18,
                 from /opt/compiler-explorer/libs/cmcstl2/include/stl2/algorithm.hpp:21,
                 from /opt/compiler-explorer/libs/cmcstl2/include/experimental/ranges/algorithm:12,
                 from <source>:4:
/opt/compiler-explorer/libs/cmcstl2/include/stl2/detail/concepts/algorithm.hpp:60:3: error: template constraint failure
   60 |   IndirectStrictWeakOrder<R, projected<I, P>>;
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

What are concepts?

- **Class** templates, **function** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**

What are concepts?

- **Class** templates, **function** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**
- Constraint specifies **the requirements on template arguments**, which can be used *to select the most appropriate function overloads and template specializations*

What are concepts?

- **Class** templates, **function** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**
- Constraint specifies **the requirements on template arguments**, which can be used *to select the most appropriate function overloads and template specializations*
- *Named sets* of such requirements are called **concepts**

What are concepts?

- **Class** templates, **function** templates, and **non-template functions** (typically members of class templates) may be **associated with a constraint**
- Constraint specifies **the requirements on template arguments**, which can be used *to select the most appropriate function overloads and template specializations*
- *Named sets* of such requirements are called **concepts**
- **Concept**
  - is a *predicate*
  - evaluated *at compile time*
  - a *part of the interface of a template* where it is used as a constraint

Toronto 2017: Consensus

- Concepts TS standardized in 2015
- No consensus on merging to IS as is
- Consensus reached in Toronto 2017 by postponing the merge of
  - introducer syntax
  - terse/natural syntax
- Small changes approved
  - removed **`bool`** from concept syntax
  - removed function concepts
- P0734 **C++ extensions for Concepts** merged with IS

Toronto 2017: Consensus

ACCEPTED FEATURES

Concept definition

```
template<class T>
concept Sortable { /* ... */ }
```

Original template notation

```
template<typename T>
  requires Sortable<T>
void sort(T&);
```

The shorthand notation

```
template<Sortable T>
void sort(T&);
```

Toronto 2017: Consensus

ACCEPTED FEATURES

Concept definition

```
template<class T>
concept Sortable { /* ... */ }
```

Original template notation

```
template<typename T>
  requires Sortable<T>
void sort(T&);
```

The shorthand notation

```
template<Sortable T>
void sort(T&);
```

NOT ACCEPTED FEATURES

The terse/natural notation

```
void sort(Sortable&);
// Not merged to IS
```

The concept introducer notation

```
Sortable{Seq} void sort(Seq&);
// Not merged to IS
```

San Diego 2018: Updated shorthand notation

TYPE PARAMETERS

```
template<typename S>
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>
void foo();
```

San Diego 2018: Updated shorthand notation

TYPE PARAMETERS

```
template<typename S>
void foo();
```

```
template<Concept S>
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>
void foo();
```

```
template<Concept auto S>
void foo();
```

San Diego 2018: Updated shorthand notation

TYPE PARAMETERS

```
template<typename S>
void foo();
```

```
template<Concept S>
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>
void foo();
```

```
template<Concept auto S>
void foo();
```

- **template template** parameters are **not supported** in this short form

San Diego 2018: Updated shorthand notation

TYPE PARAMETERS

```
template<typename S>
void foo();
```

```
template<Concept S>
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>
void foo();
```

```
template<Concept auto S>
void foo();
```

- **template template** parameters are **not supported** in this short form
- **Concept** is restricted to be a concept that takes a **type parameter** or **type parameter pack**
  - **non-type** and **template template** concepts are **not supported** in this short form

San Diego 2018: New terse notation

FULL NOTATION

```
template<typename... T>
  requires Concept<T> && ... && true
void f(T...);
```

San Diego 2018: New terse notation

FULL NOTATION

```
template<typename... T>
  requires Concept<T> && ... && true
void f(T...);
```

SHORTHAND NOTATION

```
template<Concept... T>
void f(T...);
```

San Diego 2018: New terse notation

FULL NOTATION

```
template<typename... T>
  requires Concept<T> && ... && true
void f(T...);
```

SHORTHAND NOTATION

```
template<Concept... T>
void f(T...);
```

TERSE NOTATION

```
void f(Concept auto... T);
```

San Diego 2018: New terse notation

FULL NOTATION

```
template<typename... T>
  requires Concept<T, int> && ... && true
void f(T...);
```

SHORTHAND NOTATION

```
template<Concept<int>... T>
void f(T...);
```

TERSE NOTATION

```
void f(Concept<int> auto... T);
```

San Diego 2018: **auto** parameters produce function templates

CONSISTENCE WITH C++14 LAMBDAS

```
[](auto a, auto& b, const auto& c, auto&& d) { /* ... */ };
```

```
void f1(auto a, auto& b, const auto& c, auto&& d) { /* ... */ }
```

San Diego 2018: **auto** parameters produce function templates

CONSISTENCE WITH C++14 LAMBDAS

```
[](auto a, auto& b, const auto& c, auto&& d) { /* ... */ };
```

```
void f1(auto a, auto& b, const auto& c, auto&& d) { /* ... */ }
```

CONSTRAINING THE AUTO TYPE

```
[](Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ };
```

```
void f2(Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ }
```

San Diego 2018: **auto** parameters produce function templates

CONSISTENCE WITH C++14 LAMBDAS

```
[](auto a, auto& b, const auto& c, auto&& d) { /* ... */ };
```

```
void f1(auto a, auto& b, const auto& c, auto&& d) { /* ... */ }
```

CONSTRAINING THE AUTO TYPE

```
[](Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ };
```

```
void f2(Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ }
```

The appearance of **auto** in a function parameter list tells us that we are dealing with a *function template*

San Diego 2018: Constraining **auto**

RETURN TYPE

```
Concept auto f5();  // deduced and constrained return type; not a template function
```

San Diego 2018: Constraining **auto**

RETURN TYPE

```
Concept auto f5();  // deduced and constrained return type; not a template function
```

VARIABLE TYPE

```
Concept auto x2 = f2();
```

San Diego 2018: Constraining **auto**

RETURN TYPE

```
Concept auto f5();  // deduced and constrained return type; not a template function
```

VARIABLE TYPE

```
Concept auto x2 = f2();
```

NON-TYPE TEMPLATE PARAMETER

```
template<Concept auto N>
void f7();
```

San Diego 2018: Constraining **auto**

RETURN TYPE

```
Concept auto f5();  // deduced and constrained return type; not a template function
```

VARIABLE TYPE

```
Concept auto x2 = f2();
```

NON-TYPE TEMPLATE PARAMETER

```
template<Concept auto N>
void f7();
```

NEW EXPRESSION

```
auto alloc_next() { return new Concept auto(this->next_val()); }
```

San Diego 2018: Constraining **auto**

CONVERSION OPERATOR

```
struct X {
  operator Concept auto() { /* ... */ }
};
```

San Diego 2018: Constraining **auto**

CONVERSION OPERATOR

```
struct X {
  operator Concept auto() { /* ... */ }
};
```

DECLTYPE(AUTO)

```
auto f() -> Concept decltype(auto);
```

```
Concept decltype(auto) x = f();
```

San Diego 2018: Constraining **auto**

CONVERSION OPERATOR

```
struct X {
  operator Concept auto() { /* ... */ }
};
```

DECLTYPE(AUTO)

```
auto f() -> Concept decltype(auto);
```

```
Concept decltype(auto) x = f();
```

Exception: auto in structure bindings cannot be constrained (at least for now)

`requires` clause

```
template<typename T> requires Integral<T>
void f(T t);
```

```
template<typename T>
void f(T t) requires Integral<T>;
```

`requires` clause

```
template<typename T> requires Integral<T>
void f(T t);
```

```
template<typename T>
void f(T t) requires Integral<T>;
```

- The expression must have one of the following forms
  - a *primary expression*
    - concept name (i.e. **`Swappable<T>`**)
    - type trait (i.e. **`std::is_integral_v<T>`**)
    - any parenthesized expression (i.e. **`(std::is_object_v<Args> && ...)`**)

`requires` clause

```
template<typename T> requires Integral<T>
void f(T t);
```

```
template<typename T>
void f(T t) requires Integral<T>;
```

- The expression must have one of the following forms
  - a *primary expression*
    - concept name (i.e. **`Swappable<T>`**)
    - type trait (i.e. **`std::is_integral_v<T>`**)
    - any parenthesized expression (i.e. **`(std::is_object_v<Args> && ...)`**)
  - a *sequence* of primary expressions joined with the ***`operator &&`***
  - a *sequence* of primary expressions joined with the ***`operator ||`***

Different ways of specifying constraints

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;

template<Copyable T>
void f(T t);
```

Different ways of specifying constraints

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;

template<Copyable T>
void f(T t);
```

```
template<typename T>
  requires CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>
void f(T t);
```

Different ways of specifying constraints

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;

template<Copyable T>
void f(T t);
```

```
template<typename T>
  requires CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>
void f(T t);
```

```
template<typename T>
void f(T t) requires CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;
```

Different ways of specifying constraints

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;

template<Copyable T>
void f(T t);
```

```
template<typename T>
  requires CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>
void f(T t);
```

```
template<typename T>
void f(T t) requires CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;
```

```
template<CopyConstructible T>
  requires Movable<T>
void f(T t) requires Assignable<T&, const T&>;
```

`requires` expression

```
template<typename T>
concept Addable = requires (T x) { x + x; };

template<Addable T>
T add(T a, T b) { return a + b; }
```

```
template<typename T>
  requires requires (T x) { x + x; }
T add(T a, T b) { return a + b; }
```

- prvalue expression of type ***`bool`*** *that describes the constraints on some template arguments*
- **`true`** if the constraints are satisfied, and **`false`** otherwise

`requires` expression

```
template<typename T>
concept Addable = requires (T x) { x + x; };

template<Addable T>
T add(T a, T b) { return a + b; }
```

```
template<typename T>
  requires requires (T x) { x + x; }
T add(T a, T b) { return a + b; }
```

- prvalue expression of type ***`bool`*** *that describes the constraints on some template arguments*
- **`true`** if the constraints are satisfied, and **`false`** otherwise

Usage of **`requires requires`** is usually a design error

`requires` expression

SIMPLE

```
template<typename T>
concept Addable = requires (T a, T b) {
  a + b;
};
```

- An *arbitrary expression* statement
- Asserts that the *expression is valid*
- The expression is an *unevaluated operand* (only language correctness is checked)

`requires` expression

TYPE

```
template<typename T>
concept C = requires {
  typename T::inner;
  typename S<T>;
};
```

- Asserts that the *named type is valid*

`requires` expression

COMPOUND

```
{ expression } noexcept(optional) return-type-requirement(optional);
```

- Expression ***E*** must *be valid*

`requires` expression

COMPOUND

```
{ expression } noexcept(optional) return-type-requirement(optional);
```

- Expression ***E*** must *be valid*
- If ***noexcept*** is used the expression *must NOT be potentially throwing*

`requires` expression

COMPOUND

```
{ expression } noexcept(optional) return-type-requirement(optional);
```

- Expression ***E*** must *be valid*
- If ***noexcept*** is used the expression *must NOT be potentially throwing*
- If *return-type-requirement* is present

```
{ E } -> Concept<Args...>;
```

is *equivalent to* (meaning changed in San Diego 2018)

```
E; requires Concept<decltype((E)), Args...>;
```

`requires` expression

NESTED

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

- Used to specify additional constraints *in terms of local parameters*

STANDARD LIBRARY CONCEPTS

Note

If not noted otherwise the following *utilities exist in* the **`std`** namespace

Core language concepts

```
template<class T, class U>          concept Same;
template<class Derived, class Base> concept DerivedFrom;
template<class From, class To>      concept ConvertibleTo;
template<class T, class U>          concept CommonReference;
template<class T, class U>          concept Common;
template<class T>                   concept Integral;
template<class T>                   concept SignedIntegral;
template<class T>                   concept UnsignedIntegral;
template<class LHS, class RHS>      concept Assignable;
template<class T>                   concept Swappable;
template<class T, class U>          concept SwappableWith;
template<class T>                   concept Destructible;
template<class T, class... Args>    concept Constructible;
template<class T>                   concept DefaultConstructible;
template<class T>                   concept MoveConstructible;
template<class T>                   concept CopyConstructible;
```

Core language concepts

```
template<class T, class U>
concept Same = is_same_v<T, U>;
```

Core language concepts

```
template<class T, class U>
concept Same = is_same_v<T, U>;
```

```
template<class Derived, class Base>
concept DerivedFrom =
  is_base_of_v<Base, Derived> &&
  is_convertible_v<const volatile Derived*, const volatile Base*>;
```

Core language concepts

```
template<class T, class U>
concept Same = is_same_v<T, U>;
```

```
template<class Derived, class Base>
concept DerivedFrom =
  is_base_of_v<Base, Derived> &&
  is_convertible_v<const volatile Derived*, const volatile Base*>;
```

```
template<class From, class To>
concept ConvertibleTo =
  is_convertible_v<From, To> &&
  requires(From (&f)()) { static_cast<To>(f()); };
```

Core language concepts

```
template<class T, class U>
concept Same = is_same_v<T, U>;
```

```
template<class Derived, class Base>
concept DerivedFrom =
  is_base_of_v<Base, Derived> &&
  is_convertible_v<const volatile Derived*, const volatile Base*>;
```

```
template<class From, class To>
concept ConvertibleTo =
  is_convertible_v<From, To> &&
  requires(From (&f)()) { static_cast<To>(f()); };
```

```
template<class T, class U>
concept CommonReference =
  Same<common_reference_t<T, U>, common_reference_t<U, T>> &&
  ConvertibleTo<T, common_reference_t<T, U>> &&
  ConvertibleTo<U, common_reference_t<T, U>>;
```

Core language concepts

```
template<class T, class U>
concept Common =
  Same<common_type_t<T, U>, common_type_t<U, T>> &&
  ConvertibleTo<T, common_type_t<T, U>> &&
  ConvertibleTo<U, common_type_t<T, U>> &&
  CommonReference<add_lvalue_reference_t<const T>, add_lvalue_reference_t<const U>> &&
  CommonReference<add_lvalue_reference_t<common_type_t<T, U>>,
                  common_reference_t<add_lvalue_reference_t<const T>,
                                     add_lvalue_reference_t<const U>>>;
```

Core language concepts

```
template<class T, class U>
concept Common =
  Same<common_type_t<T, U>, common_type_t<U, T>> &&
  ConvertibleTo<T, common_type_t<T, U>> &&
  ConvertibleTo<U, common_type_t<T, U>> &&
  CommonReference<add_lvalue_reference_t<const T>, add_lvalue_reference_t<const U>> &&
  CommonReference<add_lvalue_reference_t<common_type_t<T, U>>,
                  common_reference_t<add_lvalue_reference_t<const T>,
                                     add_lvalue_reference_t<const U>>>;
```

```
template<class T>
concept Integral = is_integral_v<T>;
```

Core language concepts

```
template<class T, class U>
concept Common =
  Same<common_type_t<T, U>, common_type_t<U, T>> &&
  ConvertibleTo<T, common_type_t<T, U>> &&
  ConvertibleTo<U, common_type_t<T, U>> &&
  CommonReference<add_lvalue_reference_t<const T>, add_lvalue_reference_t<const U>> &&
  CommonReference<add_lvalue_reference_t<common_type_t<T, U>>,
                  common_reference_t<add_lvalue_reference_t<const T>,
                                     add_lvalue_reference_t<const U>>>;
```

```
template<class T>
concept Integral = is_integral_v<T>;
```

```
template<class T>
concept SignedIntegral = Integral<T> && is_signed_v<T>;
```

Core language concepts

```
template<class T, class U>
concept Common =
  Same<common_type_t<T, U>, common_type_t<U, T>> &&
  ConvertibleTo<T, common_type_t<T, U>> &&
  ConvertibleTo<U, common_type_t<T, U>> &&
  CommonReference<add_lvalue_reference_t<const T>, add_lvalue_reference_t<const U>> &&
  CommonReference<add_lvalue_reference_t<common_type_t<T, U>>,
                  common_reference_t<add_lvalue_reference_t<const T>,
                                     add_lvalue_reference_t<const U>>>;
```

```
template<class T>
concept Integral = is_integral_v<T>;
```

```
template<class T>
concept SignedIntegral = Integral<T> && is_signed_v<T>;
```

```
template<class T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

Core language concepts

```
template<class LHS, class RHS>
concept Assignable = is_lvalue_reference_v<LHS> &&
  CommonReference<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&> &&
  requires(LHS lhs, RHS&& rhs) {
    { lhs = std::forward<RHS>(rhs) } -> Same<LHS>;
  };
```

Core language concepts

```
template<class LHS, class RHS>
concept Assignable = is_lvalue_reference_v<LHS> &&
  CommonReference<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&> &&
  requires(LHS lhs, RHS&& rhs) {
    { lhs = std::forward<RHS>(rhs) } -> Same<LHS>;
  };
```

```
template<class T>
concept Swappable = requires(T& a, T& b) { ranges::swap(a, b); };
```

Core language concepts

```
template<class LHS, class RHS>
concept Assignable = is_lvalue_reference_v<LHS> &&
  CommonReference<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&> &&
  requires(LHS lhs, RHS&& rhs) {
    { lhs = std::forward<RHS>(rhs) } -> Same<LHS>;
  };
```

```
template<class T>
concept Swappable = requires(T& a, T& b) { ranges::swap(a, b); };
```

```
template<class T, class U>
concept SwappableWith =
  CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  requires(T&& t, U&& u) {
    ranges::swap(std::forward<T>(t), std::forward<T>(t));
    ranges::swap(std::forward<U>(u), std::forward<U>(u));
    ranges::swap(std::forward<T>(t), std::forward<U>(u));
    ranges::swap(std::forward<U>(u), std::forward<T>(t));
  };
```

Why we need a new kind of customization point?

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // how to swap for all possible types?

}
```

Why we need a new kind of customization point?

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // how to swap for all possible types?

}
```

```
struct X {};

void swap(X&, X&)
{
  std::cout << "my swap\n";
  // ...
}
```

Why we need a new kind of customization point?

```cpp
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // how to swap for all possible types?

}
```

```cpp
struct X {};

void swap(X&, X&)
{
  std::cout << "my swap\n";
  // ...
}
```

```cpp
int main()
{
  wrapper<std::string> s1, s2;
  swap(s1, s2);

  wrapper<X> x1, x2;
  swap(x1, x2);

  wrapper<int> i1, i2;
  swap(i1, i2);
}
```

Why we need a new kind of customization point?

```cpp
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // how to swap for all possible types?
  std::swap(lhs.data_, rhs.data_);
}
```

```cpp
struct X {};

void swap(X&, X&)
{
  std::cout << "my swap\n";
  // ...
}
```

```cpp
int main()
{
  wrapper<std::string> s1, s2;
  swap(s1, s2);              // OK

  wrapper<X> x1, x2;
  swap(x1, x2);              // Error

  wrapper<int> i1, i2;
  swap(i1, i2);              // OK
}
```

Why we need a new kind of customization point?

```cpp
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // how to swap for all possible types?
  swap(lhs.data_, rhs.data_);
}
```

```cpp
struct X {};

void swap(X&, X&)
{
  std::cout << "my swap\n";
  // ...
}
```

```cpp
int main()
{
  wrapper<std::string> s1, s2;
  swap(s1, s2);              // OK

  wrapper<X> x1, x2;
  swap(x1, x2);              // OK

  wrapper<int> i1, i2;
  swap(i1, i2);              // Error
}
```

Why we need a new kind of customization point?

```cpp
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  using std::swap;
  swap(lhs.data_, rhs.data_);
}
```

```cpp
struct X {};

void swap(X&, X&)
{
  std::cout << "my swap\n";
  // ...
}
```

```cpp
int main()
{
  wrapper<std::string> s1, s2;
  swap(s1, s2);              // OK

  wrapper<X> x1, x2;
  swap(x1, x2);              // OK

  wrapper<int> i1, i2;
  swap(i1, i2);              // OK
}
```

Why we need a new kind of customization point?

```
namespace corp {

  class StrangeContainer {
    std::string t_;
  public:
    StrangeContainer(std::string t): t_{std::move(t)} {}
    char* First()             { return &t_.front(); }
    const char* First() const { return &t_.front(); }
    char* Last()              { return &t_.back(); }
    const char* Last() const  { return &t_.back(); }
  };

}
```

```
namespace corp {

  inline char* begin(StrangeContainer& c)
  { return c.First(); }
  inline const char* begin(const StrangeContainer& c)
  { return c.First(); }
  inline char* end(StrangeContainer& c)
  { return c.Last() + 1; }
  inline const char* end(const StrangeContainer& c)
  { return c.Last() + 1; }

}
```

Why we need a new kind of customization point?

```
namespace corp {

  class StrangeContainer {
    std::string t_;
  public:
    StrangeContainer(std::string t): t_{std::move(t)} {}
    char* First()             { return &t_.front(); }
    const char* First() const { return &t_.front(); }
    char* Last()              { return &t_.back(); }
    const char* Last() const  { return &t_.back(); }
  };

}
```

```
namespace corp {

  inline char* begin(StrangeContainer& c)
  { return c.First(); }
  inline const char* begin(const StrangeContainer& c)
  { return c.First(); }
  inline char* end(StrangeContainer& c)
  { return c.Last() + 1; }
  inline const char* end(const StrangeContainer& c)
  { return c.Last() + 1; }

}
```

```
template<typename Container>
void count_t(const Container& txt)
{
  using std::begin;
  using std::end;
  std::cout << "Count: " << std::count(begin(txt), end(txt), 't') << '\n';
}
```

Why we need a new kind of customization point?

```cpp
namespace corp {

  class StrangeContainer {
    std::string t_;
  public:
    StrangeContainer(std::string t): t_{std::move(t)} {}
    char* First()             { return &t_.front(); }
    const char* First() const { return &t_.front(); }
    char* Last()              { return &t_.back(); }
    const char* Last() const  { return &t_.back(); }
  };

}
```

```cpp
namespace corp {

  inline char* begin(StrangeContainer& c)
  { return c.First(); }
  inline const char* begin(const StrangeContainer& c)
  { return c.First(); }
  inline char* end(StrangeContainer& c)
  { return c.Last() + 1; }
  inline const char* end(const StrangeContainer& c)
  { return c.Last() + 1; }

}
```

```cpp
template<typename Container>
void count_t(const Container& txt)
{
  using std::begin;
  using std::end;
  std::cout << "Count: " << std::count(begin(txt), end(txt), 't') << '\n';
}
```

```cpp
count_t(corp::StrangeContainer{"I hate this class"});
count_t("I love standard library interface");
```

Customization point objects

```
template<MyOpConstraints T>
constexpr void my_op(T&& t);
```

Customization point objects

```
template<MyOpConstraints T>
constexpr void my_op(T&& t);
```

```
namespace std {
  namespace __my_op {
    struct fn {









    };
  }

  inline constexpr __my_op::fn my_op{};
}
```

Customization point objects

```
template<MyOpConstraints T>
constexpr void my_op(T&& t);
```

```
namespace std {
  namespace __my_op {
    struct fn {
      template<MyOpConstraints T>
      constexpr void operator()(T&& t) const
      {
        // default implementation
      }









    };
  }

  inline constexpr __my_op::fn my_op{};
}
```

Customization point objects

```
template<MyOpConstraints T>
constexpr void my_op(T&& t);
```

```
namespace std {
  namespace __my_op {



    template<typename>
    inline constexpr bool has_customization = false;

    template<typename T>
      requires requires(T&& t) {
        my_op(std::forward<T>(t));
      }
    inline constexpr bool has_customization<T> = true;
  }
}
```

```
namespace std {
  namespace __my_op {
    struct fn {
      template<MyOpConstraints T>
      constexpr void operator()(T&& t) const
      {
        // default implementation
      }









    };
  }

  inline constexpr __my_op::fn my_op{};
}
```

Customization point objects

```
template<MyOpConstraints T>
constexpr void my_op(T&& t);
```

```
namespace std {
  namespace __my_op {



    template<typename>
    inline constexpr bool has_customization = false;

    template<typename T>
      requires requires(T&& t) {
        my_op(std::forward<T>(t));
      }
    inline constexpr bool has_customization<T> = true;
  }
}
```

```
namespace std {
  namespace __my_op {
    struct fn {
      template<MyOpConstraints T>
      constexpr void operator()(T&& t) const
      {
        // default implementation
      }

      template<MyOpConstraints T>
        requires has_customization<T>
      constexpr void operator()(T&& t) const
      {
        my_op(std::forward<T>(t));  // uses ADL
      }
    };
  }

  inline constexpr __my_op::fn my_op{};
}
```

Customization point objects

```
template<MyOpConstraints T>
constexpr void my_op(T&& t);
```

```
namespace std {
  namespace __my_op {
    // non-ADL block
    void my_op(); // undefined

    template<typename>
    inline constexpr bool has_customization = false;

    template<typename T>
      requires requires(T&& t) {
        my_op(std::forward<T>(t));
      }
    inline constexpr bool has_customization<T> = true;
  }
}
```

```
namespace std {
  namespace __my_op {
    struct fn {
      template<MyOpConstraints T>
      constexpr void operator()(T&& t) const
      {
        // default implementation
      }

      template<MyOpConstraints T>
        requires has_customization<T>
      constexpr void operator()(T&& t) const
      {
        my_op(std::forward<T>(t));  // uses ADL
      }
    };
  }

  inline constexpr __my_op::fn my_op{};
}
```

Customization point object: `ranges::swap(E1, E2)`

- **`(void)swap(E1, E2)`** if **`E1`** or **`E2`** has *class type* and that expression is valid, with *overload resolution performed in a context that includes the declarations*

```
template<class T>
void swap(T&, T&) = delete;

template<class T, size_t N>
void swap(T(&)[N], T(&)[N]) = delete;
```

and *does not include a declaration* of **`ranges::swap`**

Customization point object: `ranges::swap(E1, E2)`

- **`(void)swap(E1, E2)`** if **E1** or **E2** has *class type* and that expression is valid, with *overload resolution performed in a context that includes the declarations*

```
template<class T>
void swap(T&, T&) = delete;

template<class T, size_t N>
void swap(T(&)[N], T(&)[N]) = delete;
```

  and *does not include a declaration* of **`ranges::swap`**

- **`(void)ranges::swap_ranges(E1, E2)`** if **E1** and **E2** are lvalues of *array types with equal extent* and **`ranges::swap(*E1, *E2)`** is a valid expression

Customization point object: `ranges::swap(E1, E2)`

- **`(void)swap(E1, E2)`** if **E1** or **E2** has *class type* and that expression is valid, with *overload resolution performed in a context that includes the declarations*

```
template<class T>
void swap(T&, T&) = delete;

template<class T, size_t N>
void swap(T(&)[N], T(&)[N]) = delete;
```

  and *does not include a declaration* of **`ranges::swap`**

- **`(void)ranges::swap_ranges(E1, E2)`** if **E1** and **E2** are lvalues of *array types with equal extent* and **`ranges::swap(*E1, *E2)`** is a valid expression
- If **E1** and **E2** are lvalues of *the same type **T*** that models **`MoveConstructible<T>`** and **`Assignable<T&, T>`**, **exchanges the denoted values**

Customization point object: `ranges::swap(E1, E2)`

EXAMPLE

```
namespace N {
  struct A { int m; };
  struct Proxy { A* a; };

  void swap(A& x, Proxy p) { std::ranges::swap(x.m, p.a->m); }
  void swap(Proxy p, A& x) { swap(x, p); } // satisfy symmetry requirement
}
```

Customization point object: `ranges::swap(E1, E2)`

EXAMPLE

```
namespace N {
  struct A { int m; };
  struct Proxy { A* a; };

  void swap(A& x, Proxy p) { std::ranges::swap(x.m, p.a->m); }
  void swap(Proxy p, A& x) { swap(x, p); } // satisfy symmetry requirement
}
```

```
template<class T, std::SwappableWith<T> U>
void value_swap(T&& t, U&& u)
{
  std::ranges::swap(std::forward<T>(t), std::forward<U>(u));
}
```

Core language concepts

```
template<class T>
concept Destructible = is_nothrow_destructible_v<T>;
```

Core language concepts

```
template<class T>
concept Destructible = is_nothrow_destructible_v<T>;
```

```
template<class T, class... Args>
concept Constructible = Destructible<T> && is_constructible_v<T, Args...>;
```

Core language concepts

```
template<class T>
concept Destructible = is_nothrow_destructible_v<T>;
```

```
template<class T, class... Args>
concept Constructible = Destructible<T> && is_constructible_v<T, Args...>;
```

```
template<class T>
concept DefaultConstructible = Constructible<T>;
```

Core language concepts

```
template<class T>
concept Destructible = is_nothrow_destructible_v<T>;
```

```
template<class T, class... Args>
concept Constructible = Destructible<T> && is_constructible_v<T, Args...>;
```

```
template<class T>
concept DefaultConstructible = Constructible<T>;
```

```
template<class T>
concept MoveConstructible = Constructible<T, T> && ConvertibleTo<T, T>;
```

Core language concepts

```
template<class T>
concept Destructible = is_nothrow_destructible_v<T>;
```

```
template<class T, class... Args>
concept Constructible = Destructible<T> && is_constructible_v<T, Args...>;
```

```
template<class T>
concept DefaultConstructible = Constructible<T>;
```

```
template<class T>
concept MoveConstructible = Constructible<T, T> && ConvertibleTo<T, T>;
```

```
template<class T>
concept CopyConstructible = MoveConstructible<T> &&
  Constructible<T, T&> && ConvertibleTo<T&, T> &&
  Constructible<T, const T&> && ConvertibleTo<const T&, T> &&
  Constructible<T, const T> && ConvertibleTo<const T, T>;
```

Comparison concepts

```
template<class B>          concept Boolean;
template<class T>          concept EqualityComparable;
template<class T, class U> concept EqualityComparableWith;
template<class T>          concept StrictTotallyOrdered;
template<class T, class U> concept StrictTotallyOrderedWith;
```

Comparison concepts

```
template<class B>          concept Boolean;
template<class T>          concept EqualityComparable;
template<class T, class U> concept EqualityComparableWith;
template<class T>          concept StrictTotallyOrdered;
template<class T, class U> concept StrictTotallyOrderedWith;
```

- **`Boolean`** is satisfied by a type that is *convertible to* ***`bool`*** and provides *typical boolean operations* (**op!**, **op&&**, **op||**, **op==**, **op!=**) with *results convertible to* ***`bool`***

Comparison concepts

```
template<class B>          concept Boolean;
template<class T>          concept EqualityComparable;
template<class T, class U> concept EqualityComparableWith;
template<class T>          concept StrictTotallyOrdered;
template<class T, class U> concept StrictTotallyOrderedWith;
```

- **`Boolean`** is satisfied by a type that is *convertible to* ***`bool`*** and provides *typical boolean operations* (**`op!`**, **`op&&`**, **`op||`**, **`op==`**, **`op!=`**) with *results convertible to* ***`bool`***
- **`EqualityComparable<T>`** requires
  - **`{ op==(T, T) } -> Boolean`**, **`{ op!=(T, T) } -> Boolean`**

Comparison concepts

```
template<class B>          concept Boolean;
template<class T>          concept EqualityComparable;
template<class T, class U> concept EqualityComparableWith;
template<class T>          concept StrictTotallyOrdered;
template<class T, class U> concept StrictTotallyOrderedWith;
```

- **Boolean** is satisfied by a type that is *convertible to* ***bool*** and provides *typical boolean operations* (**op!**, **op&&**, **op||**, **op==**, **op!=**) with *results convertible to* ***bool***
- **EqualityComparable<T>** requires
  - **{ op==(T, T) } -> Boolean**, **{ op!=(T, T) } -> Boolean**
- **StrictTotallyOrdered<T>** requires
  - **EqualityComparable<T>**
  - **{ op<(T,T) } -> Boolean**, **{ op>(T,T) } -> Boolean**
  - **{ op<=(T,T) } -> Boolean**, **{ op>=(T,T) } -> Boolean**

Object concepts

```
template<class T> concept Movable;
template<class T> concept Copyable;
template<class T> concept Semiregular;
template<class T> concept Regular;
```

Object concepts

```
template<class T>
concept Movable = is_object_v<T> && MoveConstructible<T> && Assignable<T&, T> && Swappable<T>;
```

Object concepts

```
template<class T>
concept Movable = is_object_v<T> && MoveConstructible<T> && Assignable<T&, T> && Swappable<T>;
```

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;
```

Object concepts

```
template<class T>
concept Movable = is_object_v<T> && MoveConstructible<T> && Assignable<T&, T> && Swappable<T>;
```

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;
```

```
template<class T>
concept Semiregular = Copyable<T> && DefaultConstructible<T>;
```

Object concepts

```
template<class T>
concept Movable = is_object_v<T> && MoveConstructible<T> && Assignable<T&, T> && Swappable<T>;
```

```
template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;
```

```
template<class T>
concept Semiregular = Copyable<T> && DefaultConstructible<T>;
```

```
template<class T>
concept Regular = Semiregular<T> && EqualityComparable<T>;
```

Beware of ogres!

Beware of ogres!

```
class A {
  // ...
public:
  A() = default;
  ~A() noexcept(false);
  // ...
};

class B {
  A a;
public:
  B() = default;
};

static_assert(DefaultConstructible<B>); // Error
```

Beware of ogres!

```
class A {
  // ...
public:
  A() = default;
  ~A() noexcept(false);
  // ...
};

class B {
  A a;
public:
  B() = default;
};

static_assert(DefaultConstructible<B>); // Error
```

```
struct C {
  // ...
  C();
  C(const C&);
  void operator=(const C&);
  ~C();
};

bool operator==(const C& lhs, const C& rhs);
bool operator!=(const C& lhs, const C& rhs);

static_assert(Regular<C>); // Error
```

Callable concepts

```
template<class F, class... Args>    concept Invocable;
template<class F, class... Args>    concept RegularInvocable;
template<class F, class... Args>    concept Predicate;
template<class R, class T, class U> concept Relation;
template<class R, class T, class U> concept StrictWeakOrder;
```

Equality preserving expressions

- Given *equal inputs*, the expression results in *equal outputs*

Equality preserving expressions

- Given *equal inputs*, the expression results in *equal outputs*
- *Not all input values must be valid* for a given expression
  - e.g. for integers **`a`** and **`b`**, the expression **`a / b`** is not well-defined when **`b`** is **`0`**

Equality preserving expressions

- Given *equal inputs*, the expression results in *equal outputs*
- *Not all input values must be valid* for a given expression
  - e.g. for integers **`a`** and **`b`**, the expression **`a / b`** is not well-defined when **`b`** is **`0`**
- *Equality preserving* expressions are further required to be **stable**
  - two evaluations of such an expression with the *same input objects must have equal outputs*

Equality preserving expressions

- Given *equal inputs*, the expression results in *equal outputs*
- *Not all input values must be valid* for a given expression
  - e.g. for integers **`a`** and **`b`**, the expression **`a / b`** is not well-defined when **`b`** is **`0`**
- *Equality preserving* expressions are further required to be **stable**
  - two evaluations of such an expression with the *same input objects must have equal outputs*
- Expressions declared in a ***`requires`*** *expression* are **required to be equality preserving**

Equality preserving expressions

- Given *equal inputs*, the expression results in *equal outputs*
- *Not all input values must be valid* for a given expression
  - e.g. for integers **`a`** and **`b`**, the expression **`a / b`** is not well-defined when **`b`** is **`0`**
- *Equality preserving* expressions are further required to be **stable**
  - two evaluations of such an expression with the *same input objects must have equal outputs*
- Expressions declared in a ***`requires`*** *expression* are **required to be equality preserving**
  - except noted otherwise

Callable concepts

```
template<class F, class... Args>
concept Invocable = requires(F&& f, Args&&... args) {
  invoke(std::forward<F>(f), std::forward<Args>(args)...);
};
```

- The **invoke** function call expression is *not required to be equality-preserving*

Callable concepts

```
template<class F, class... Args>
concept Invocable = requires(F&& f, Args&&... args) {
  invoke(std::forward<F>(f), std::forward<Args>(args)...);
};
```

- The **invoke** function call expression is *not required to be equality-preserving*

```
template<class F, class... Args>
concept RegularInvocable = Invocable<F, Args...>;
```

- The **invoke** function call expression *shall be equality-preserving* and shall not modify the function object or the arguments
- The distinction between **Invocable** and **RegularInvocable** is purely **semantic**

Callable concepts

```
template<class F, class... Args>
concept Predicate =
  RegularInvocable<F, Args...> &&
  Boolean<invoke_result_t<F, Args...>>;
```

Callable concepts

```
template<class F, class... Args>
concept Predicate =
  RegularInvocable<F, Args...> &&
  Boolean<invoke_result_t<F, Args...>>;
```

```
template<class R, class T, class U>
concept Relation =
  Predicate<R, T, T> && Predicate<R, U, U> &&
  Predicate<R, T, U> && Predicate<R, U, T>;
```

Callable concepts

```
template<class F, class... Args>
concept Predicate =
  RegularInvocable<F, Args...> &&
  Boolean<invoke_result_t<F, Args...>>;
```

```
template<class R, class T, class U>
concept Relation =
  Predicate<R, T, T> && Predicate<R, U, U> &&
  Predicate<R, T, U> && Predicate<R, U, T>;
```

```
template<class R, class T, class U>
concept StrictWeakOrder = Relation<R, T, U>;
```

- A **`Relation`** satisfies **`StrictWeakOrder`** only if it imposes a *strict weak ordering* on its arguments

ITERATORS AND SENTINELS

CONCEPTS AND CUSTOMIZATION POINTS

Note

If not noted otherwise the following *utilities exists in* the **`std::ranges`** namespace

Incrementable traits and `iter_difference_t`

```
template<class>
struct incrementable_traits {};
```

```
template<class T>
  requires is_object_v<T>
struct incrementable_traits<T*> {
  using difference_type = ptrdiff_t;
};
```

```
template<class I>
struct incrementable_traits<const I>
    : incrementable_traits<I> {};
```

Incrementable traits and `iter_difference_t`

```
template<class>
struct incrementable_traits {};
```

```
template<class T>
  requires is_object_v<T>
struct incrementable_traits<T*> {
  using difference_type = ptrdiff_t;
};
```

```
template<class I>
struct incrementable_traits<const I>
    : incrementable_traits<I> {};
```

```
template<class T>
  requires requires { typename T::difference_type; }
struct incrementable_traits<T> {
  using difference_type = typename T::difference_type;
};
```

```
template<class T>
  requires (!requires { typename T::difference_type; }) &&
           requires(const T& a, const T& b) {
               { a - b } -> Integral;
           }
struct incrementable_traits<T> {
  using difference_type =
    make_signed_t<decltype(declval<T>() - declval<T>())>;
};
```

Incrementable traits and `iter_difference_t`

```
template<class>
struct incrementable_traits {};
```

```
template<class T>
  requires is_object_v<T>
struct incrementable_traits<T*> {
  using difference_type = ptrdiff_t;
};
```

```
template<class I>
struct incrementable_traits<const I>
    : incrementable_traits<I> {};
```

```
template<class T>
  requires requires { typename T::difference_type; }
struct incrementable_traits<T> {
  using difference_type = typename T::difference_type;
};
```

```
template<class T>
  requires (!requires { typename T::difference_type; }) &&
           requires(const T& a, const T& b) {
               { a - b } -> Integral;
           }
struct incrementable_traits<T> {
  using difference_type =
    make_signed_t<decltype(declval<T>() - declval<T>())>;
};
```

```
template<class T>
using iter_difference_t<T> = iterator_traits<I>::difference_type; /* or */ incrementable_traits<I>::difference_type;
```

Incrementable traits and `iter_difference_t`

```
template<class>
struct incrementable_traits {};
```

```
template<class T>
  requires is_object_v<T>
struct incrementable_traits<T*> {
  using difference_type = ptrdiff_t;
};
```

```
template<class I>
struct incrementable_traits<const I>
    : incrementable_traits<I> {};
```

```
template<class T>
  requires requires { typename T::difference_type; }
struct incrementable_traits<T> {
  using difference_type = typename T::difference_type;
};
```

```
template<class T>
  requires (!requires { typename T::difference_type; }) &&
           requires(const T& a, const T& b) {
               { a - b } -> Integral;
           }
struct incrementable_traits<T> {
  using difference_type =
    make_signed_t<decltype(declval<T>() - declval<T>())>;
};
```

```
template<class T>
using iter_difference_t<T> = iterator_traits<I>::difference_type; /* or */ incrementable_traits<I>::difference_type;
```

- Users may specialize **`incrementable_traits`** on program-defined types

Readable traits and `iter_value_t`

```
template<class>
struct cond-value-type {}; // exposition only

template<class T>
  requires is_object_v<T>
struct cond-value-type {
  using value_type = remove_cv_t<T>;
};
```

Readable traits and `iter_value_t`

```
template<class>
struct cond-value-type {}; // exposition only

template<class T>
  requires is_object_v<T>
struct cond-value-type {
  using value_type = remove_cv_t<T>;
};
```

```
template<class>
struct readable_traits {};
```

```
template<class T>
struct readable_traits<T*>
    : cond-value-type<T> {};
```

Readable traits and `iter_value_t`

```
template<class>
struct cond-value-type {}; // exposition only

template<class T>
  requires is_object_v<T>
struct cond-value-type {
  using value_type = remove_cv_t<T>;
};
```

```
template<class>
struct readable_traits {};
```

```
template<class T>
struct readable_traits<T*>
    : cond-value-type<T> {};
```

```
template<class I>
  requires is_array_v<I>
struct readable_traits<I> {
  using value_type = remove_cv_t<remove_extent_t<I>>;
};
```

```
template<class I>
struct readable_traits<const I> : readable_traits<I> {};
```

```
template<class T>
  requires requires { typename T::value_type; }
struct readable_traits<T>
    : cond-value-type<typename T::value_type> {};
```

```
template<class T>
  requires requires { typename T::element_type; }
struct readable_traits<T>
    : cond-value-type<typename T::element_type> {};
```

Readable traits and `iter_value_t`

```
template<class>
struct cond-value-type {}; // exposition only

template<class T>
  requires is_object_v<T>
struct cond-value-type {
  using value_type = remove_cv_t<T>;
};
```

```
template<class>
struct readable_traits {};
```

```
template<class T>
struct readable_traits<T*>
    : cond-value-type<T> {};
```

```
template<class I>
  requires is_array_v<I>
struct readable_traits<I> {
  using value_type = remove_cv_t<remove_extent_t<I>>;
};
```

```
template<class I>
struct readable_traits<const I> : readable_traits<I> {};
```

```
template<class T>
  requires requires { typename T::value_type; }
struct readable_traits<T>
    : cond-value-type<typename T::value_type> {};
```

```
template<class T>
  requires requires { typename T::element_type; }
struct readable_traits<T>
    : cond-value-type<typename T::element_type> {};
```

```
template<class T>
using iter_value_t = iterator_traits<I>::value_type; /* or */ readable_traits<I>::value_type;
```

iterator_traits<T*> and iterator_concept

```
template<class T>
  requires is_object_v<T>
struct iterator_traits<T*> {
  using difference_type = ptrdiff_t;
  using value_type = remove_cv_t<T>;
  using pointer = T*;
  using reference = T&;
  using iterator_category = random_access_iterator_tag;
  using iterator_concept = contiguous_iterator_tag;
};
```

`iterator_traits<T*>` and `iterator_concept`

```
template<class T>
  requires is_object_v<T>
struct iterator_traits<T*> {
  using difference_type = ptrdiff_t;
  using value_type = remove_cv_t<T>;
  using pointer = T*;
  using reference = T&;
  using iterator_category = random_access_iterator_tag;
  using iterator_concept = contiguous_iterator_tag;
};
```

- Program-defined specializations of **`iterator_traits`** may have a member type **`iterator_concept`** that is *used to opt in or out* of conformance to the *iterator concepts*

Customization point object: `ranges::iter_move(E)`

- `iter_move(E)`, if that expression is well-formed when evaluated in a context that does not include **`ranges::iter_move`** but does include the lookup set produced by *argument-dependent lookup*

Customization point object: `ranges::iter_move(E)`

- `iter_move(E)`, if that expression is well-formed when evaluated in a context that does not include **`ranges::iter_move`** but does include the lookup set produced by *argument-dependent lookup*
- If the expression ***`*E`*** *is well-formed*
  - if **`*E`** is an lvalue, `std::move(*E);`
  - otherwise, `*E`

Customization point object: `ranges::iter_move(E)`

- `iter_move(E)`, if that expression is well-formed when evaluated in a context that does not include **`ranges::iter_move`** but does include the lookup set produced by *argument-dependent lookup*
- If the expression ***E** is well-formed*
  - if **`*E`** is an lvalue, `std::move(*E);`
  - otherwise, `*E`

EXAMPLE

```
template<Iterator I, Sentinel<I> S>
  requires (!Same<I, S>)
class common_iterator {
  // ...
  friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
      noexcept(noexcept(ranges::iter_move(declval<const I&>()))) requires InputIterator<I>
  {
    return ranges::iter_move(get<I>(i.v_));
  }
};
```

Iterator concepts: Readable

```
template<class In>
concept Readable =
  requires {
    typename iter_value_t<In>;
    typename iter_reference_t<In>;
    typename iter_rvalue_reference_t<In>;
  } &&
  CommonReference<iter_reference_t<In>&&, iter_value_t<In>&> &&
  CommonReference<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
  CommonReference<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;
```

- The **Readable** concept is satisfied by types that are *readable by applying* ***operator**** including pointers, smart pointers, and iterators

Iterator concepts: `Writable`

```
template<class Out, class T>
concept Writable =
  requires(Out&& o, T&& t) {
    *o = std::forward<T>(t);
    *std::forward<Out>(o) = std::forward<T>(t);
    const_cast<const iter_reference_t<Out>&&>(*o) = std::forward<T>(t);
    const_cast<const iter_reference_t<Out>&&>(*std::forward<Out>(o)) = std::forward<T>(t);
  };
```

- The **`Writable`** concept specifies the requirements for *writing a value into an iterator's referenced object*
- Concept requirements are *not required to be equality-preserving*

Iterator concepts: `WeaklyIncrementable`

```
template<class I>
concept WeaklyIncrementable =
  Semiregular<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
  };
```

- The **`WeaklyIncrementable`** concept specifies the requirements on types that can be *incremented with the pre- and post-increment* operators

Iterator concepts: `WeaklyIncrementable`

```
template<class I>
concept WeaklyIncrementable =
  Semiregular<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
  };
```

- The **`WeaklyIncrementable`** concept specifies the requirements on types that can be *incremented with the pre- and post-increment* operators
- For **`WeaklyIncrementable`** types, ***`a`*** *equals* ***`b`*** *does not imply that* ***`++a`*** *equals* ***`++b`***

Iterator concepts: `WeaklyIncrementable`

```
template<class I>
concept WeaklyIncrementable =
  Semiregular<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
  };
```

- The **`WeaklyIncrementable`** concept specifies the requirements on types that can be *incremented with the pre- and post-increment* operators
- For **`WeaklyIncrementable`** types, ***a*** *equals* ***b*** *does not imply that* ***++a*** *equals* ***++b***
- Algorithms on weakly incrementable types *should never attempt to pass through the same incrementable value twice*
  - they should be *single pass algorithms*

Iterator concepts: `Incrementable`

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

- 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`***

Iterator concepts: `Incrementable`

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

- 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`***
- The requirement that **`a`** equals **`b`** implies **`++a`** equals **`++b`** allows the use of *multi-pass one-directional algorithms* with types that satisfy **`Incrementable`**

Iterator concepts: `Iterator`

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

- *Every iterator satisfies the **`Iterator`** requirements*

Iterator concepts: **Sentinel**

```
template<class S, class I>
concept Sentinel =
  Semiregular<S> &&
  Iterator<I> &&
  weakly-equality-comparable-with<S, I>;
```

- The **`Sentinel`** concept *specifies the relationship* between an **`Iterator`** type and a **`Semiregular`** type whose values denote a range

Iterator concepts: `SizedSentinel`

```
template<class S, class I>
concept SizedSentinel =
  Sentinel<S, I> &&
  !disable_sized_sentinel<remove_cv_t<S>, remove_cv_t<I>> &&
  requires(const I& i, const S& s) {
    { s - i } -> Same<iter_difference_t<I>>;
    { i - s } -> Same<iter_difference_t<I>>;
  };
```

- The **`SizedSentinel`** concept specifies requirements on an **`Iterator`** and a **`Sentinel`** that *allow the use of the* ***`operator-`*** to compute the distance between them *in constant time*

Iterator concepts: `SizedSentinel`

```
template<class S, class I>
concept SizedSentinel =
  Sentinel<S, I> &&
  !disable_sized_sentinel<remove_cv_t<S>, remove_cv_t<I>> &&
  requires(const I& i, const S& s) {
    { s - i } -> Same<iter_difference_t<I>>;
    { i - s } -> Same<iter_difference_t<I>>;
  };
```

- The **`SizedSentinel`** concept specifies requirements on an **`Iterator`** and a **`Sentinel`** that *allow the use of the* ***`operator-`*** to compute the distance between them *in constant time*
- **`disable_sized_sentinel`** provides a mechanism to enable use of sentinels and iterators with the library that *meet the syntactic requirements but do not in fact satisfy* ***`SizedSentinel`***

```
template<class Iterator1, class Iterator2>
  requires !SizedSentinel<Iterator1, Iterator2>
inline constexpr bool disable_sized_sentinel<reverse_iterator<Iterator1>, reverse_iterator<Iterator2>> = true;
```

Iterator concepts: `InputIterator`

```
template<class I>
concept InputIterator =
  Iterator<I> &&
  Readable<I> &&
  requires { typename ITER_CONCEPT(I); } &&
  DerivedFrom<ITER_CONCEPT(I), input_iterator_tag>;
```

- The **`InputIterator`** concept defines requirements for a type whose *referenced values can be read* and which *can be both pre- and post-incremented*

Iterator concepts: `InputIterator`

```
template<class I>
concept InputIterator =
  Iterator<I> &&
  Readable<I> &&
  requires { typename ITER_CONCEPT(I); } &&
  DerivedFrom<ITER_CONCEPT(I), input_iterator_tag>;
```

- The **`InputIterator`** concept defines requirements for a type whose *referenced values can be read* and which *can be both pre- and post-incremented*

Unlike the C++17 input iterator requirements, the **`InputIterator`** concept *does not require equality comparison* since it is typically compared to a sentinel

Iterator concepts: `OutputIterator`

```
template<class I, class T>
concept OutputIterator =
  Iterator<I> &&
  Writable<I, T> &&
  requires(I i, T&& t) {
    *i++ = std::forward<T>(t); // not required to be equality-preserving
  };
```

- The **`OutputIterator`** concept defines requirements for a type that can be used to *write values* and which *can be both pre- and post-incremented*

Iterator concepts: `OutputIterator`

```
template<class I, class T>
concept OutputIterator =
  Iterator<I> &&
  Writable<I, T> &&
  requires(I i, T&& t) {
    *i++ = std::forward<T>(t); // not required to be equality-preserving
  };
```

- The **`OutputIterator`** concept defines requirements for a type that can be used to *write values* and which *can be both pre- and post-incremented*

Algorithms on output iterators *should never attempt to pass through the same iterator twice*. They should be single pass algorithms.

Iterator concepts: `ForwardIterator`

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

- The **`ForwardIterator`** concept refines **`InputIterator`**, *adding equality comparison and the multipass guarantee*
- *Pointers and references* obtained from a forward iterator into a range **`[i, s)`** shall *remain valid* while **`[i, s)`** continues to denote a range

Iterator concepts: `ForwardIterator`

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

- The **`ForwardIterator`** concept refines **`InputIterator`**, *adding equality comparison and the multipass guarantee*
- *Pointers and references* obtained from a forward iterator into a range **`[i, s)`** shall *remain valid* while **`[i, s)`** continues to denote a range

It is allowed to use *multi-pass one-directional algorithms* with forward iterators

Iterator concepts: `BidirectionalIterator`

```
template<class I>
concept BidirectionalIterator =
  ForwardIterator<I> &&
  DerivedFrom<ITER_CONCEPT(I), bidirectional_iterator_tag> &&
  requires(I i) {
    { --i } -> Same<I>&;
    { i-- } -> Same<I>;
  };
```

- The **`BidirectionalIterator`** concept refines **`ForwardIterator`**, and *adds the ability to move an iterator backward* as well as forward

Iterator concepts: `RandomAccessIterator`

```
template<class I>
concept RandomAccessIterator =
  BidirectionalIterator<I> &&
  DerivedFrom<ITER_CONCEPT(I), random_access_iterator_tag> &&
  StrictTotallyOrdered<I> &&
  SizedSentinel<I, I> &&
  requires(I i, const I j, const iter_difference_t<I> n) {
    { i += n } -> Same<I>&;
    { j + n } -> Same<I>;
    { n + j } -> Same<I>;
    { i -= n } -> Same<I>&;
    { j - n } -> Same<I>;
    { j[n] } -> Same<iter_reference_t<I>>;
  };
```

- The **`RandomAccessIterator`** concept refines **`BidirectionalIterator`** and *adds support for constant-time advancement with **`+=`**, **`+`**, **`-=`**, and **`-`***, and the computation of *distance in constant time with **`-`***
- Random access iterators also *support array notation* via subscripting

Iterator concepts: `ContiguousIterator`

```
template<class I>
concept ContiguousIterator =
  RandomAccessIterator<I> &&
  DerivedFrom<ITER_CONCEPT(I), contiguous_iterator_tag> &&
  is_lvalue_reference_v<iter_reference_t<I>> &&
  Same<iter_value_t<I>, remove_cvref_t<iter_reference_t<I>>>;
```

- The **`ContiguousIterator`** concept refines **`RandomAccessIterator`** and provides a *guarantee that* the denoted *elements are stored contiguously in memory*

ITERATORS AND SENTINELS

NEW CLASSES

Default sentinel

```
struct default_sentinel_t {};
inline constexpr default_sentinel_t default_sentinel{};
```

- An *empty type* used to **denote the end of a range**

Default sentinel

```
struct default_sentinel_t {};
inline constexpr default_sentinel_t default_sentinel{};
```

- An *empty type* used to **denote the end of a range**
- It is intended to be *used together with iterator* types *that know the bound of their range* (e.g. **`counted_iterator`**)

Counted iterators

```
template<Iterator I>
class counted_iterator;
```

- An iterator *adaptor with the same behavior* as the underlying iterator
- Keeps track of its *distance from its starting position*

Counted iterators

```
template<Iterator I>
class counted_iterator;
```

- An iterator *adaptor with the same behavior* as the underlying iterator
- Keeps track of its *distance from its starting position*
- Can be used together with **`default_sentinel`** to operate on a range of **`N`** elements starting at a given position *without needing to know the end position* a priori

Counted iterators

```
template<Iterator I>
class counted_iterator;
```

- An iterator *adaptor with the same behavior* as the underlying iterator
- Keeps track of its *distance from its starting position*
- Can be used together with **`default_sentinel`** 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> l;
vector<string> v;
ranges::copy(counted_iterator(l.begin(), 10), default_sentinel, back_inserter(v));
```

Unreachable sentinel

```
struct unreachable_t;
inline constexpr unreachable_t unreachable{};
```

- Class **`unreachable_t`** can be used with any **`WeaklyIncrementable`** type to denote the *"upper bound" of an open interval*

Unreachable sentinel

```
struct unreachable_t;
inline constexpr unreachable_t unreachable{};
```

- Class **`unreachable_t`** can be used with any **`WeaklyIncrementable`** type to denote the *"upper bound" of an open interval*
- Comparing anything for *equality* with an object of type **`unreachable`** *always returns* ***`false`***

Unreachable sentinel

```
struct unreachable_t;
inline constexpr unreachable_t unreachable{};
```

- Class **`unreachable_t`** can be used with any **`WeaklyIncrementable`** type to denote the *"upper bound" of an open interval*
- Comparing anything for *equality* with an object of type **`unreachable`** *always returns* ***`false`***

EXAMPLE

```
char* p;
char* nl = find(p, unreachable, '\n');
```

- Provided a newline character really exists in the buffer, the use of **`unreachable_sentinel`** potentially makes the call to **`find`** *more efficient* since the *loop test against the sentinel does not require a conditional branch*

move_sentinel

```
template<Semiregular S>
class move_sentinel;
```

- A sentinel adaptor *useful for denoting ranges together with* ***move_iterator***
- When an input iterator type **I** and sentinel type **S** model **Sentinel<S, I>**, **move_sentinel<S>** and **move_iterator<I>** model **Sentinel<move_sentinel<S>, move_iterator<I>>** as well

move_sentinel

```
template<Semiregular S>
class move_sentinel;
```

- A sentinel adaptor *useful for denoting ranges together with* ***move_iterator***
- When an input iterator type **I** and sentinel type **S** model **Sentinel<S, I>**, **move_sentinel<S>** and **move_iterator<I>** model **Sentinel<move_sentinel<S>, move_iterator<I>>** as well

EXAMPLE

```
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, IndirectUnaryPredicate<I> Pred>
  requires IndirectlyMovable<I, O>
void move_if(I first, S last, O out, Pred pred)
{
  std::ranges::copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}
```

Common Iterators

```
template<Iterator I, Sentinel<I> S>
  requires !Same<I, S>
class common_iterator;
```

- An iterator/sentinel *adaptor* to *represent a non-common range* of elements as a common range
- *Holds* either an iterator or a sentinel, and *implements the equality comparison* operators

Common Iterators

```
template<Iterator I, Sentinel<I> S>
  requires !Same<I, S>
class common_iterator;
```

- An iterator/sentinel *adaptor* to *represent a non-common range* of elements as a common range
- *Holds* either an iterator or a sentinel, and *implements the equality comparison* operators
- Useful for interfacing with *legacy code that expects the begin and end of a range to have the same type*

Common Iterators

```
template<Iterator I, Sentinel<I> S>
  requires !Same<I, S>
class common_iterator;
```

- An iterator/sentinel *adaptor* to *represent a non-common range* of elements as a common range
- *Holds* either an iterator or a sentinel, and *implements the equality comparison* operators
- Useful for interfacing with *legacy code that expects the begin and end of a range to have the same type*

EXAMPLE

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

```
list<int> s;
using CI = common_iterator<counted_iterator<list<int>::iterator>, default_sentinel_t>;
fun(CI(counted_iterator(s.begin(), 10)), CI(default_sentinel));
```

RANGES

Range

A **range** is an **iterator** and a **sentinel** that designate the *beginning and end* of the computation

Range

A **range** is an **iterator** and a **sentinel** that designate the *beginning and end* of the computation

A **counted range** is an **iterator** and a **count** that designate the *beginning and the number of elements* to which the computation is to be applied

Range

A **range** is an **iterator** and a **sentinel** that designate the *beginning and end* of the computation

A **counted range** is an **iterator** and a **count** that designate the *beginning and the number of elements* to which the computation is to be applied

- An iterator and a sentinel denoting a range *are comparable*

ranges::advance

```
template<Iterator I>
constexpr void advance(I& i, iter_difference_t<I> n);
```

```
template<Iterator I, Sentinel<I> S>
constexpr void advance(I& i, S bound);
```

```
template<Iterator I, Sentinel<I> S>
constexpr iter_difference_t<I> advance(I& i, iter_difference_t<I> n, S bound);
```

ranges::distance

```
template<Iterator I, Sentinel<I> S>
constexpr iter_difference_t<I> distance(I first, S last);
```

```
template<Range R>
constexpr range_difference_t<R> distance(R&& r);
```

ranges::next

```
template<Iterator I>
constexpr I next(I x);
```

```
template<Iterator I>
constexpr I next(I x, iter_difference_t<I> n);
```

```
template<Iterator I, Sentinel<I> S>
constexpr I next(I x, S bound);
```

```
template<Iterator I, Sentinel<I> S>
constexpr I next(I x, iter_difference_t<I> n, S bound);
```

ranges::prev

```
template<BidirectionalIterator I>
constexpr I prev(I x);
```

```
template<BidirectionalIterator I>
constexpr I prev(I x, iter_difference_t<I> n);
```

```
template<BidirectionalIterator I>
constexpr I prev(I x, iter_difference_t<I> n, I bound);
```

Customization point objects

```
namespace ranges {
  inline namespace unspecified {
    // range access
    inline constexpr unspecified begin = unspecified;
    inline constexpr unspecified end = unspecified;
    inline constexpr unspecified cbegin = unspecified;
    inline constexpr unspecified cend = unspecified;
    inline constexpr unspecified rbegin = unspecified;
    inline constexpr unspecified rend = unspecified;
    inline constexpr unspecified crbegin = unspecified;
    inline constexpr unspecified crend = unspecified;

    // range primitives
    inline constexpr unspecified size = unspecified;
    inline constexpr unspecified empty = unspecified;
    inline constexpr unspecified data = unspecified;
    inline constexpr unspecified cdata = unspecified;
  }
}
```

Customization point object: `ranges::begin`

- `(E) + 0` if *`E` is an lvalue of array type*

Customization point object: `ranges::begin`

- **`(E) + 0`** if ***E*** *is an lvalue of array type*
- **`DECAY_COPY((E).begin())`** if ***E*** *is an lvalue* and return type of the expression models **`Iterator`**

Customization point object: `ranges::begin`

- **`(E) + 0`** if ***E*** *is an lvalue of array type*
- **`DECAY_COPY((E).begin())`** if ***E*** *is an lvalue* and return type of the expression models **`Iterator`**
- **`DECAY_COPY(begin(E))`** if it *is a valid expression* and its return type models **`Iterator`** with overload resolution performed in a context that includes the declarations

```
template<class T> void begin(T&&) = delete;
template<class T> void begin(initializer_list<T>&&) = delete;
```

and does not include a declaration of **`ranges::begin`**

Customization point object: `ranges::begin`

- **`(E) + 0`** if ***E*** *is an lvalue of array type*
- **`DECAY_COPY((E).begin())`** if ***E*** *is an lvalue* and return type of the expression models **`Iterator`**
- **`DECAY_COPY(begin(E))`** if it *is a valid expression* and its return type models **`Iterator`** with overload resolution performed in a context that includes the declarations

```
template<class T> void begin(T&&) = delete;
template<class T> void begin(initializer_list<T>&&) = delete;
```

and does not include a declaration of **`ranges::begin`**

It means that by default those customization point object *will not work for* ***E*** *being an rvalue*. Users can *opt-in* by explicitly providing non-member function overload taking an rvalue of their type.

Concept Range

```
template<class T>
concept range-impl = // exposition only
  requires(T&& t) {
    ranges::begin(std::forward<T>(t)); // sometimes equality-preserving
    ranges::end(std::forward<T>(t));
  };
```

```
template<class T>
concept Range = range-impl<T&>;
```

- The **Range** concept requires that ***begin*** *and* ***end*** *return an iterator and a sentinel*
- Ranges are an *abstraction of containers* that allow a C++ program to operate on elements of data structures uniformly

Concept SizedRange

```
template<class T>
concept SizedRange =
  Range<T> &&
  !disable_sized_range<remove_cvref_t<T>> &&
  requires(T& t) { ranges::size(t); };
```

- The **SizedRange** concept adds the requirement that the *number of elements* in the range can be *determined in constant time* using the **size** function
- **disable_sized_range** enables use of range types with the library that satisfy but do not in fact model **SizedRange**

Common range refinements

```
template<class R, class T>
concept OutputRange = Range<R> && OutputIterator<iterator_t<R>, T>;

template<class T>
concept InputRange = Range<T> && InputIterator<iterator_t<T>>;

template<class T>
concept ForwardRange = InputRange<T> && ForwardIterator<iterator_t<T>>;

template<class T>
concept BidirectionalRange = ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;

template<class T>
concept RandomAccessRange = BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```

Common range refinements

```
template<class R, class T>
concept OutputRange = Range<R> && OutputIterator<iterator_t<R>, T>;

template<class T>
concept InputRange = Range<T> && InputIterator<iterator_t<T>>;

template<class T>
concept ForwardRange = InputRange<T> && ForwardIterator<iterator_t<T>>;

template<class T>
concept BidirectionalRange = ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;

template<class T>
concept RandomAccessRange = BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```

```
template<class T>
concept ContiguousRange = RandomAccessRange<T> && ContiguousIterator<iterator_t<T>> &&
  requires(T& t) {
    { ranges::data(t) } -> Same<add_pointer_t<range_reference_t<T>>>;
  };
```

Concept CommonRange

```
template<class T>
concept CommonRange =
  Range<T> &&
  Same<iterator_t<T>, sentinel_t<T>>;
```

- Range type for which **`ranges::begin`** and **`ranges::end`** *return objects of the same type*

VIEWS AND RANGE ADAPTOR OBJECTS

Concept `View`

```
template<class T>
concept View = Range<T> && Semiregular<T> && enable_view<T>;
```

- The `View` concept specifies requirements on a **Range** type with **constant-time copy and assign operations**
- *The cost* of these operations *is not proportional to the number of elements* in the **`View`**
- Since the difference between **Range** and **`View`** is largely semantic, the two are *differentiated with* the help of the ***`enable_view`*** *trait*

Customization point: `enable_view`

```
template<class T>
inline constexpr bool enable_view;
```

Customization point: `enable_view`

```
template<class T>
inline constexpr bool enable_view;
```

- `true` if *`DerivedFrom<T, view_base>`* where

```
struct view_base {};
```

Customization point: `enable_view`

```
template<class T>
inline constexpr bool enable_view;
```

- `true` if *`DerivedFrom<T, view_base>`* where

```
struct view_base {};
```

- `false` if both **`T`** and **`const T`** model **`Range`**, and **`range_reference_t<T>`** is *NOT the same type* as **`range_reference_t<const T>`**
  - deep const-ness implies element ownership
  - shallow const-ness implies reference semantics

Customization point: `enable_view`

```
template<class T>
inline constexpr bool enable_view;
```

- `true` if *`DerivedFrom<T, view_base>`* where

```
struct view_base {};
```

- `false` if both `T` and `const T` model **`Range`**, and **`range_reference_t<T>`** is *NOT the same type* as **`range_reference_t<const T>`**
  - deep const-ness implies element ownership
  - shallow const-ness implies reference semantics
- `true` *otherwise*

Customization point: `enable_view`

```
template<class T>
inline constexpr bool enable_view;
```

- Partial specializations with value `false` provided for
  - `initializer_list`
  - `set`, `multiset`
  - `unordered_set`, `unordered_multiset`

Customization point: `enable_view`

```
template<class T>
inline constexpr bool enable_view;
```

- Partial specializations with value `false` provided for
  - **`initializer_list`**
  - **`set`**, **`multiset`**
  - **`unordered_set`**, **`unordered_multiset`**

Users may specialize **`enable_view`** for their types and overwrite the defaults to specify if their type properly models **`View`** semantics

Exposition only concept `forwarding-range`

```
template<class T>
concept forwarding-range = // exposition only
  Range<T> &&
  range-impl<T>;
```

- Range for which the *validity of iterators is not tied to the lifetime* of an object

Exposition only concept `forwarding-range`

```
template<class T>
concept forwarding-range = // exposition only
  Range<T> &&
  range-impl<T>;
```

- Range for which the *validity of iterators is not tied to the lifetime* of an object
- A function can accept arguments of such a type by value and return iterators obtained from it *without danger of dangling*
  - e.g. passing an rvalue of **`std::span`**, **`std::string_view`**, or **`subrange`**

Concept `ViewableRange`

```
template<class T>
concept ViewableRange =
  Range<T> &&
  (forwarding-range<T> || View<decay_t<T>>);
```

- Range type that *can be converted to a* ***`View`*** safely

view_interface

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface : public view_base {

public:









};
```

- **D** is a type of a derived class (CRTP)

view_interface

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface : public view_base {
  constexpr const D& derived() const noexcept { return static_cast<const D&>(*this); }
public:
  constexpr bool empty() const requires ForwardRange<const D>;
  constexpr explicit operator bool() const requires requires { ranges::empty(derived()); }









};
```

- **D** is a type of a derived class (CRTP)

view_interface

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface : public view_base {
  constexpr const D& derived() const noexcept { return static_cast<const D&>(*this); }
public:
  constexpr bool empty() const requires ForwardRange<const D>;
  constexpr explicit operator bool() const requires requires { ranges::empty(derived()); }

  constexpr auto data() requires ContiguousIterator<iterator_t<D>>;
  constexpr auto data() const requires Range<const D> && ContiguousIterator<iterator_t<const D>>;

  constexpr auto size() const requires ForwardRange<const D> && SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;









};
```

- **D** is a type of a derived class (CRTP)

view_interface

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface : public view_base {
  constexpr const D& derived() const noexcept { return static_cast<const D&>(*this); }
public:
  constexpr bool empty() const requires ForwardRange<const D>;
  constexpr explicit operator bool() const requires requires { ranges::empty(derived()); }

  constexpr auto data() requires ContiguousIterator<iterator_t<D>>;
  constexpr auto data() const requires Range<const D> && ContiguousIterator<iterator_t<const D>>;

  constexpr auto size() const requires ForwardRange<const D> && SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;

  constexpr decltype(auto) front() requires ForwardRange<D>;
  constexpr decltype(auto) front() const requires ForwardRange<const D>;
  constexpr decltype(auto) back() requires BidirectionalRange<D> && CommonRange<D>;
  constexpr decltype(auto) back() const requires BidirectionalRange<const D> && CommonRange<const D>;




};
```

- **D** is a type of a derived class (CRTP)

view_interface

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface : public view_base {
  constexpr const D& derived() const noexcept { return static_cast<const D&>(*this); }
public:
  constexpr bool empty() const requires ForwardRange<const D>;
  constexpr explicit operator bool() const requires requires { ranges::empty(derived()); }

  constexpr auto data() requires ContiguousIterator<iterator_t<D>>;
  constexpr auto data() const requires Range<const D> && ContiguousIterator<iterator_t<const D>>;

  constexpr auto size() const requires ForwardRange<const D> && SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;

  constexpr decltype(auto) front() requires ForwardRange<D>;
  constexpr decltype(auto) front() const requires ForwardRange<const D>;
  constexpr decltype(auto) back() requires BidirectionalRange<D> && CommonRange<D>;
  constexpr decltype(auto) back() const requires BidirectionalRange<const D> && CommonRange<const D>;

  template<RandomAccessRange R = D>
  constexpr decltype(auto) operator[](range_difference_t<R> n);
  template<RandomAccessRange R = const D>
  constexpr decltype(auto) operator[](range_difference_t<R> n) const;
};
```

- **D** is a type of a derived class (CRTP)

subrange

```
template<Iterator I, Sentinel<I> S = I,
         subrange_kind K = SizedSentinel<S, I> ?
                           subrange_kind::sized :
                           subrange_kind::unsized>
  requires (K == subrange_kind::sized || !SizedSentinel<S, I>)
class subrange : public view_interface<subrange<I, S, K>> { /* ... */ };
```

- Bundles together an iterator and a sentinel into a single object that *models the **`View`** concept*
- Models the ***`SizedRange`** concept* when the **`subrange_kind`** template parameter is

  `subrange_kind::sized`

Views

`COMMON_VIEW`

- Takes a **`View`** which *has different types for its iterator and sentinel* and turns it into an equivalent **`View`** where *the iterator and sentinel have the same type*
- Useful for *calling legacy algorithms* that expect a range's iterator and sentinel types to be the same

Views

COMMON_VIEW

- Takes a **`View`** which *has different types for its iterator and sentinel* and turns it into an equivalent **`View`** where *the iterator and sentinel have the same type*
- Useful for *calling legacy algorithms* that expect a range's iterator and sentinel types to be the same

EXAMPLE

```
template<class ForwardIterator>
size_t count(ForwardIterator first, ForwardIterator last);
```

```
template<ForwardRange R>
void my_algo(R&& r) {
  auto&& common = common_view{r};
  auto cnt = count(common.begin(), common.end());
  // ...
}
```

Views

`ALL_VIEW`

- Returns a **`View`** that *includes all elements* of its **`Range`** argument

Views

`ALL_VIEW`

- Returns a **`View`** that *includes all elements* of its **`Range`** argument

`EMPTY_VIEW`

- Produces a **`View`** of *no elements* of a particular type

Views

ALL_VIEW

- Returns a **View** that *includes all elements* of its **Range** argument

EMPTY_VIEW

- Produces a **View** of *no elements* of a particular type

SINGLE_VIEW

- Produces a **View** that contains *exactly one element* of a specified value

```
single_view s{4};
for(int i : s)
  std::cout << i; // prints 4
```

Views

REVERSE_VIEW

- Takes a *bidirectional **View*** and produces another **View** that iterates *the same elements in reverse order*

```
vector<int> is{0, 1, 2, 3, 4};
reverse_view rv{is};
for(int i : rv)
  std::cout << i << ' '; // prints: 4 3 2 1 0
```

Views

REVERSE_VIEW

- Takes a *bidirectional* ***View*** and produces another **View** that iterates *the same elements in reverse order*

```
vector<int> is{0, 1, 2, 3, 4};
reverse_view rv{is};
for(int i : rv)
  std::cout << i << ' '; // prints: 4 3 2 1 0
```

TRANSFORM_VIEW

- Creates a **View** of an underlying sequence *after applying a transformation function* to each element

```
vector<int> is{0, 1, 2, 3, 4};
transform_view squares{is, [](int i) { return i * i; }};
for(int i : squares)
  std::cout << i << ' '; // prints: 0 1 4 9 16
```

Views

IOTA_VIEW

- Generates a *sequence of elements by repeatedly incrementing* an initial value

```
for(int i : iota_view{1, 10})
  std::cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9
```

Views

IOTA_VIEW

- Generates a *sequence of elements by repeatedly incrementing* an initial value

```
for(int i : iota_view{1, 10})
  std::cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9
```

FILTER_VIEW

- Creates a **View** of an underlying sequence *without the elements that fail* to satisfy a predicate

```
vector<int> is{0, 1, 2, 3, 4, 5, 6};
filter_view evens{is, [](int i) { return 0 == i % 2; }};
for(int i : evens)
  std::cout << i << ' '; // prints: 0 2 4 6
```

Views

TAKE_VIEW

- Produces a **View** of *at most* ***N*** *elements* from another **View**

```
vector<int> is{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
take_view few{is, 5};
for(int i : few)
  std::cout << i << ' '; // prints: 0 1 2 3 4
```

Views

TAKE_VIEW

- Produces a **View** of *at most **N** elements* from another **View**

```
vector<int> is{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
take_view few{is, 5};
for(int i : few)
  std::cout << i << ' '; // prints: 0 1 2 3 4
```

TAKE_WHILE_VIEW

- Produces a **View** of *first elements* from another **View** *that match predicate*

```
vector<int> is{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
take_while_view square_less_20{is, [](int i) { return i * i < 20; }};
for(int i : square_less_20)
  std::cout << i << ' '; // prints: 0 1 2 3 4
```

Views

DROP_VIEW

- Produces a **View** of *all but the **N** first elements* from another **View**

```
vector<int> is{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
drop_view few{is, 5};
for(int i : few)
  std::cout << i << ' '; // prints: 5 6 7 8 9
```

Views

DROP_VIEW

- Produces a **View** of *all but the **N** first elements* from another **View**

```
vector<int> is{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
drop_view few{is, 5};
for(int i : few)
  std::cout << i << ' '; // prints: 5 6 7 8 9
```

DROP_WHILE_VIEW

- Produces a **View** of *all but the first elements* from another **View** *that match predicate*

```
vector<int> is{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
drop_while_view square_greater_20{is, [](int i) { return i * i < 20; }};
for(int i : square_greater_20)
  std::cout << i << ' '; // prints: 5 6 7 8 9
```

Views

JOIN_VIEW

- *Flattens a **View** of ranges into a **View***

```
vector<string> ss{"hello", " ", "world", "!"};
join_view greeting{ss};
for(char ch : greeting)
  std::cout << ch; // prints: hello world!
```

Views

SPLIT_VIEW

- Takes a **View** and a delimiter, and *splits the **View** into subranges* on the delimiter
- The delimiter can be a single element or a **View** of elements

```
string str{"the quick brown fox"};
split_view sentence{str, ' '};
for(auto word : sentence) {
  for(char ch : word)
    std::cout << ch;
  std::cout << "*";
} // prints: the*quick*brown*fox*
```

Range adaptor objects

RANGE ADAPTOR CLOSURE OBJECT

- A *unary function object* that accepts a **`ViewableRange`** argument and returns a **`View`**

```
adaptor(range)
range | adaptor
```

Range adaptor objects

RANGE ADAPTOR CLOSURE OBJECT

- A *unary function object* that accepts a **`ViewableRange`** argument and returns a **`View`**

```
adaptor(range)
range | adaptor
```

RANGE ADAPTOR OBJECT

- An object that *accepts a **`ViewableRange`** as its first argument* and returns a **`View`**
- If a range adaptor object accepts *more than one arguments*, then the following expressions are equivalent

```
adaptor(range, args...)
adaptor(args...)(range)
range | adaptor(args...)
```

Range adaptor objects

- *Declared in* namespace **`std::ranges::view`**
- Can be *chained to create pipelines* of range transformations that *evaluate lazily* as the resulting view is iterated

Range adaptor objects

- *Declared in* namespace **`std::ranges::view`**
- Can be *chained to create pipelines* of range transformations that *evaluate lazily* as the resulting view is iterated
- The *bitwise or operator is overloaded* for the purpose of creating adaptor chain pipelines

```
vector<int> ints{0, 1, 2, 3, 4, 5};

auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i) { return i * i; };

for(int i : ints | view::filter(even) | view::transform(square))
  std::cout << i << ' '; // prints: 0 4 16
```

Views vs Range Adaptor Objects

VIEW

```
template<View Rng>
  requires BidirectionalRange<Rng>
class reverse_view : public view_interface<reverse_view<Rng>> { /* ... */ };
```

RANGE ADAPTOR OBJECT

```
namespace view {
  struct __reverse_fn : detail::__pipeable<__reverse_fn> {
    template<BidirectionalRange Rng>
      requires ViewableRange<Rng>
    constexpr auto operator()(Rng&& rng) const {
      return reverse_view{std::forward<Rng>(rng)};
    }
  };

  inline constexpr __reverse_fn reverse {};
}
```

ALGORITHMS

Function objects

```
namespace ranges {

  struct equal_to {
    using is_transparent = unspecified;

    template<class T, class U>
      requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
    constexpr bool operator()(T&& t, U&& u) const;
  };

}
```

- Similar definition for **`not_equal_to`**

Function objects

```
namespace ranges {

  struct equal_to {
    using is_transparent = unspecified;

    template<class T, class U>
      requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
    constexpr bool operator()(T&& t, U&& u) const;
  };

}
```

- Similar definition for **`not_equal_to`**

Ranges provide so-called diamond operators only

Function objects

```
namespace ranges {

  struct less {
    using is_transparent = unspecified;

    template<class T, class U>
      requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(U, <, T)
    constexpr bool operator()(T&& t, U&& u) const;
  };

}
```

- Similar definitions for **`greater`**, **`greater_equal`**, and **`less_equal`**

Projection

Projection is a *transformation* that an algorithm applies *before inspecting* the values of elements

Projection

Projection is a *transformation* that an algorithm applies *before inspecting* the values of elements

EXAMPLE

- To sort the pairs in increasing order of their first members

```
std::pair<int, std::string_view> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::ranges::less{}, [](auto const& p) { return p.first; });
```

Algorithms Lookup

- The *algorithms* defined in the **`std::ranges`** namespace *are not found by argument-dependent name lookup*

Algorithms Lookup

- The *algorithms* defined in the **`std::ranges`** namespace *are not found by argument-dependent name lookup*
- When found by unqualified name lookup for the postfix-expression in a function call, they *inhibit argument-dependent name lookup*

```
void foo()
{
  using namespace std::ranges;
  std::vector<int> vec{1, 2, 3};
  find(begin(vec), end(vec), 2);    // invokes std::ranges::find, not std::find
}
```

Algorithms Lookup

- The *algorithms* defined in the **`std::ranges`** namespace *are not found by argument-dependent name lookup*
- When found by unqualified name lookup for the postfix-expression in a function call, they *inhibit argument-dependent name lookup*

```
void foo()
{
  using namespace std::ranges;
  std::vector<int> vec{1, 2, 3};
  find(begin(vec), end(vec), 2);    // invokes std::ranges::find, not std::find
}
```

- Overloads of *algorithms that take **`Range`** arguments* behave as if they are implemented by *calling **`ranges::begin`** and **`ranges::end`** on the **`Range`** and dispatching* to the overload in namespace ranges that takes separate iterator and sentinel arguments

Algorithm: find_if

```
namespace std {

  template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);









}
```

Algorithm: find_if

```
namespace std {

  template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

  namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if(I first, S last, Pred pred, Proj proj = Proj{});









  }

}
```

Algorithm: find_if

```
namespace std {

  template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

  namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if(I first, S last, Pred pred, Proj proj = Proj{});

    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R> find_if(R&& r, Pred pred, Proj proj = Proj{});

  }

}
```

Algorithm: find_if

```
namespace std {

  template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

  namespace ranges {

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if(I first, S last, Pred pred, Proj proj = Proj{});

    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R> find_if(R&& r, Pred pred, Proj proj = Proj{});

  }

}
```

Safety included

```
namespace std {
  class dangling {
  public:
    constexpr dangling() noexcept = default;
    template<class... Args>
    constexpr dangling(Args&&...) noexcept {}
  };

  namespace ranges {
    template<Range R>
    using safe_iterator_t = conditional_t<forwarding-range<R>,
                                          iterator_t<R>,
                                          dangling>;

    template<Range R>
    using safe_subrange_t = conditional_t<forwarding-range<R>,
                                          subrange<iterator_t<R>>,
                                          dangling>;
  }
}
```

Safety included

```
vector<int> f();
```

Safety included

```
vector<int> f();
```

RVALUE

```
auto result1 = ranges::find(f(), 42);
static_assert(Same<decltype(result1), dangling>);
```

Safety included

```
vector<int> f();
```

RVALUE

```
auto result1 = ranges::find(f(), 42);
static_assert(Same<decltype(result1), dangling>);
```

LVALUE

```
auto vec = f();
auto result2 = ranges::find(vec, 42);
static_assert(Same<decltype(result2), vector<int>::iterator>);
```

Safety included

```
vector<int> f();
```

RVALUE

```
auto result1 = ranges::find(f(), 42);
static_assert(Same<decltype(result1), dangling>);
```

LVALUE

```
auto vec = f();
auto result2 = ranges::find(vec, 42);
static_assert(Same<decltype(result2), vector<int>::iterator>);
```

FORWARDING RANGE

```
auto result3 = ranges::find(subrange{vec}, 42);
static_assert(Same<decltype(result3), vector<int>::iterator>);
```

SUMMARY

Take aways

Take aways

1. **Constrain** your template parameters with concepts

Take aways

1. **Constrain** your template parameters with concepts
2. **Learn and use** C++ Library and Ranges Concepts
   - Do not reinvent the wheel!

Take aways

1. **Constrain** your template parameters with concepts
2. **Learn and use** C++ Library and Ranges Concepts
   - Do not reinvent the wheel!
3. **Limit** `requires requires` usage

Take aways

1. Constrain your template parameters with concepts
2. Learn and use C++ Library and Ranges Concepts
   - Do not reinvent the wheel!
3. Limit `requires requires` usage
4. If needed, define customization points for your types to make them work properly with standard facilities

Take aways

1. **Constrain** your template parameters with concepts
2. **Learn and use** C++ Library and Ranges Concepts
   - Do not reinvent the wheel!
3. **Limit** `requires requires` usage
4. If needed, **define customization points** for your types to make them work properly with standard facilities
5. Prefer to **use algorithms from `std::ranges`** rather than namespace **`std`**

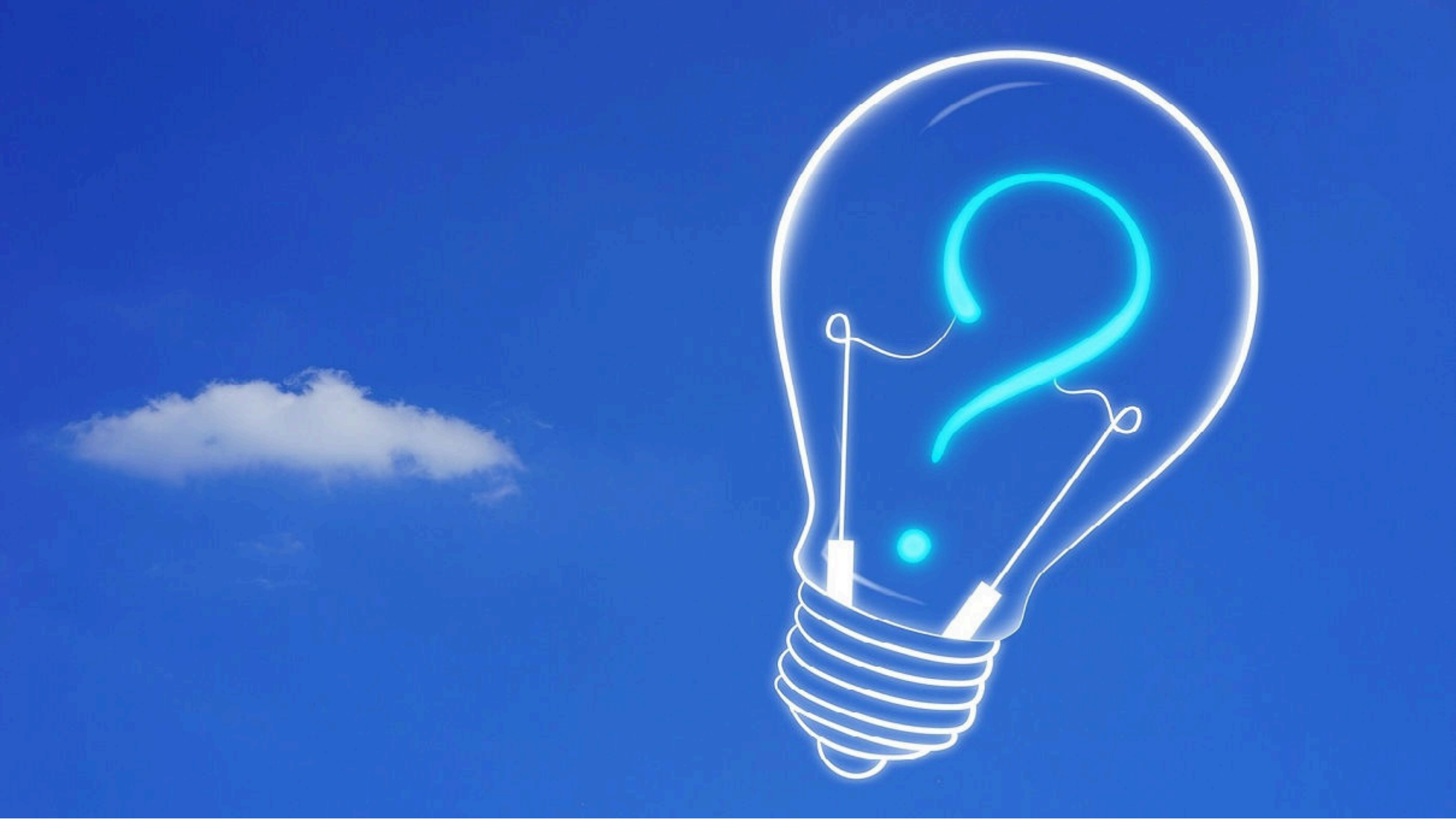
Implementations

RANGE-V3

- https://github.com/ericniebler/range-v3
- C++11 implementation for clang & gcc
- Ranges TS and much more
  - not 100% compatible with IS working draft
- Namespace **`::ranges`**

CMCSTL2

- https://github.com/CaseyCarter/cmcstl2
- Concepts-based implementation for recent gcc releases (**`-fconcepts`**)
- High fidelity to the IS working draft + proposed resolutions
- Namespace **`::std::experimental::ranges`**




CAUTION
Programming
is addictive
(and too much fun)