COMP6771

Week 7.2 – Generic Programming

Concepts

“ Somewhat more formally, a concept is a description of requriements on one or more types stated in terms of the existence and properties of procedures, type attributes, and type functions defined on the types.

— *Elements of Programming*, by Alexander Stepanov and Paul McJones.

What's in a requirement?

A *constraint* is a syntax requirement. The compiler can check these.

An *axiom* is a semantic requirement. The compiler can't check these, but librarians may assume that they hold true forever. Precondition checks can sometimes check at runtime.

Axioms are usually provided via comments in-tandem with the constraints.

Complexity requirements don't have a fancy name, and can't be checked by the implementation, nor the library author. A sophisticated benchmarking tool might be able to do so.

We say that we *satisfy* the requirements if all the constraints evaluate as true. We *model* the concept if, and only if, we satisfy the requirements and meet the axioms and complexity requirements.

Hello, concepts!

```
 1 template<typename T>
 2 concept type = true;
 3
 4 template<typename T>
 5 requires type<T>
 6 void hello(T)
 7 {
 8         std::cout << "Permissive, base-case\n";
 9 }
10
11 int main()
12 {
13         hello(0);
14         hello(0.0);
15         hello("base-case");
16 }
```

Try it on Compiler Explorer!

Hello, concepts!

```
 1 template<typename T>
 2 concept type = true;
 3
 4 template<type T> // same as using `typename` right now
 5 void hello(T)
 6 {
 7         std::cout << "Permissive, base-case\n";
 8 }
 9
10 int main()
11 {
12         hello(0);
13         hello(0.0);
14         hello("base-case");
15 }
```

Try it on Compiler Explorer!

Hello, concepts!

```
1  template<typename T>
2  concept integral = std::is_integral_v<T>;
3
4  template<integral T>
5  void hello(T)
6  {
7          std::cout << "Integral case\n";
8  }
9
10 int main()
11 {
12         hello(0);
13         //hello(0.0);
14         //hello("base-case");
15 }
```

Try it on Compiler Explorer!

Hello, concepts!

```
 1 template<typename T>
 2 concept floating_point = std::is_floating_point_v<T>;
 3
 4 void hello(floating_point auto f)
 5 {
 6    std::cout << "Floating-point case\n";
 7 }
 8
 9 int main()
10 {
11    // hello(0);
12    hello(0.0);
13    // hello("base case");
14 }
```

Try it on Compiler Explorer!

Hello, concepts!

```
 1 void hello(integral auto)        { std::cout << "Integral case\n";       }
 2 void hello(floating_point auto) { std::cout << "Floating-point case\n"; }
 3 void hello(auto)                 { std::cout << "Base-case\n";           }
 4
 5 int main()
 6 {
 7         hello(0);           // prints "Integral case"
 8         hello(0.0);         // prints "Floating-point case"
 9         hello("base case"); // prints "Base-case"
10 }
```

Try it on Compiler Explorer!

Refining and weakening concepts

A concept C2 *refines* a concept C1, if whenever C2 is modelled, C1 is also modelled.

Refining and weakening concepts

A concept C2 *refines* a concept C1, if whenever C2 is modelled, C1 is also modelled.

A concept C2 *weakens* C1, if its requirements are a proper subset of C1.

Hello, concepts!

```
 1 void hello(integral auto) { std::cout << "Integral case\n"; }
 2
 3 template<typename T>
 4 concept signed_integral = integral<T> and std::is_signed_v<T>;
 5
 6 void hello(signed_integral auto) { std::cout << "Signed integral case\n"; }
 7
 8 int main()
 9 {
10         hello(0);  // prints "Signed integral case"
11         hello(0U); // prints "Integral case"
12 }
```

Try it on Compiler Explorer!

Library concepts

C++20 not only introduced concepts as a language feature, but also three families of highly usable concepts.

Due to many reasons they are available in GCC 10 and recent versions of MSVC,
but not Clang with libc++ (which is what we use in the course).

There's a 1:1 mapping between concepts in range-v3 and what's in the standard library (namespaces aside).

Concepts family	Relevant header	CMakeLists LINK:	Namespace:
C++ Reference: Concepts library	`<concepts/concepts.hpp>`	`concepts::concepts`	`concepts`
C++ Reference: Iterators library	`<range/v3/iterator.hpp>`	`range-v3`	`ranges`
C++ Reference: Ranges library	`<range/v3/range.hpp>`	`range-v3`	`ranges`

Motivation

```
1 struct equal_to {
2   template<typename T, typename U>
3   auto operator()(T const& t, U const& u) const -> bool
4   {
5     return t == u;
6   }
7 };
```

Motivation

```
1 struct equal_to {
2   template<typename T, typename U>
3   auto operator()(T const& t, U const& u) const -> bool
4   {
5     return t == u;
6   }
7 };
```

```
equal_to{}(0, 0)                                // okay, returns true
equal_to{}(std::vector{0}, std::vector{1})      // okay, returns false
equal_to{}(0, 0.0)                              // okay, returns true
equal_to{}(0.0, 0)                              // okay, returns true
equal_to{}(0, std::vector<int>{})               // error: `int == vector` not defined
```

Motivation

```
1 struct equal_to {
2   template<typename T, std::equality_comparable_with<T> U>
3   auto operator()(T const& t, U const& u) const -> bool
4   {
5     return t == u;
6   }
7 };
```

Motivation

```cpp
struct equal_to {
  template<typename T, std::equality_comparable_with<T> U>
  auto operator()(T const& t, U const& u) const -> bool
  {
    return t == u;
  }
};
```

```cpp
equal_to{}(0, 0)                                // okay, returns true
equal_to{}(std::vector{0}, std::vector{1})      // okay, returns false
equal_to{}(0, 0.0)                              // okay, returns true
equal_to{}(0.0, 0)                              // okay, returns true
equal_to{}(0, std::vector<int>{})               // still error, but makes more sense
```

Regularity

```
1 template<typename T>
2 concept movable = std::is_object_v<T>
3                   and std::move_constructible<T>
4                   and std::assignable_from<T&, T>
5                   and std::swappable<T>;
```

Regularity

```
template<typename T>
concept movable = std::is_object_v<T>
              and std::move_constructible<T>
              and std::assignable_from<T&, T>
              and std::swappable<T>;
```

```
template<typename T>
concept copyable = std::copy_constructible<T>
               and std::movable<T>
               and std::assignable_from<T&, T const&>;
```

Regularity

```
1 template<typename T>
2 concept movable = std::is_object_v<T>
3                   and std::move_constructible<T>
4                   and std::assignable_from<T&, T>
5                   and std::swappable<T>;
```

```
1 template<typename T>
2 concept copyable = std::copy_constructible<T>
3                    and std::movable<T>
4                    and std::assignable_from<T&, T const&>;
```

```
1 template<typename T>
2 concept semiregular = std::copyable<T> and std::default_initializable<T>;
```

Regularity

```
1 template<typename T>
2 concept movable = std::is_object_v<T>
3                   and std::move_constructible<T>
4                   and std::assignable_from<T&, T>
5                   and std::swappable<T>;
```

```
1 template<typename T>
2 concept copyable = std::copy_constructible<T>
3                    and std::movable<T>
4                    and std::assignable_from<T&, T const&>;
```

```
1 template<typename T>
2 concept semiregular = std::copyable<T> and std::default_initializable<T>;
```

```
1 template<typename T>
2 concept regular = std::semiregular<T> and std::equality_comparable<T>;
```

Previously on COMP6771...

A range is an ordered sequence of elements with a designated start and rule for finishing

A range is an ordered sequence of elements with a designated start and rule for finishing

```
std::string("Hello, world!")
```

```
std::vector<std::string>{"Hello", "world!"}
```

$$\mathbb{N} \quad \mathbb{Z}^+ \quad \mathbb{Q}^+ \quad \mathbb{R}^+$$

Exercise: how can $\mathbb{C}$ be made into a range?

```
for (auto i = 0; std::cin >> i;) { ... }
```

Iterators and pointers

> *A pointer is an abstraction of a virtual memory address.*
>
> —Sy Brand

Iterators and pointers

> *A pointer is an abstraction of a virtual memory address.*

—Sy Brand

> *Iterators are a family of concepts that abstract different aspects of addresses, ...*

—Elements of Programming, Stepanov & McJones

Iterators and pointers

“ A pointer is an abstraction of a virtual memory address.

—Sy Brand

“ Iterators are a family of concepts that abstract different aspects of addresses, ...

—Elements of Programming, Stepanov & McJones

“ Iterators are a generalization of pointers that allow a C++ program to work with different data structures... in a uniform manner.

—Working Draft, Standard for Programming Language C++

```cpp
template<typename T, std::equality_comparable_with<T> U>
auto find(std::vector<T> const& r, U const& value) -> std::vector<T>::size_type {
   for (auto i = typename std::vector<T>::size_type{0}; i < r.size(); ++i) {
      if (r[i] == value) {
         return i;
      }
   }
   return r.size();
}
```

```cpp
template<typename T, std::equality_comparable_with<T> U>
auto find(std::deque<T> const& r, U const& value) -> std::deque<T>::size_type {
   for (auto i = typename std::deque<T>::size_type{0}; i < r.size(); ++i) {
      if (r[i] == value) {
         return i;
      }
   }
   return r.size();
}
```

x sequence containers

×

y sequence algorithms (e.g. find)

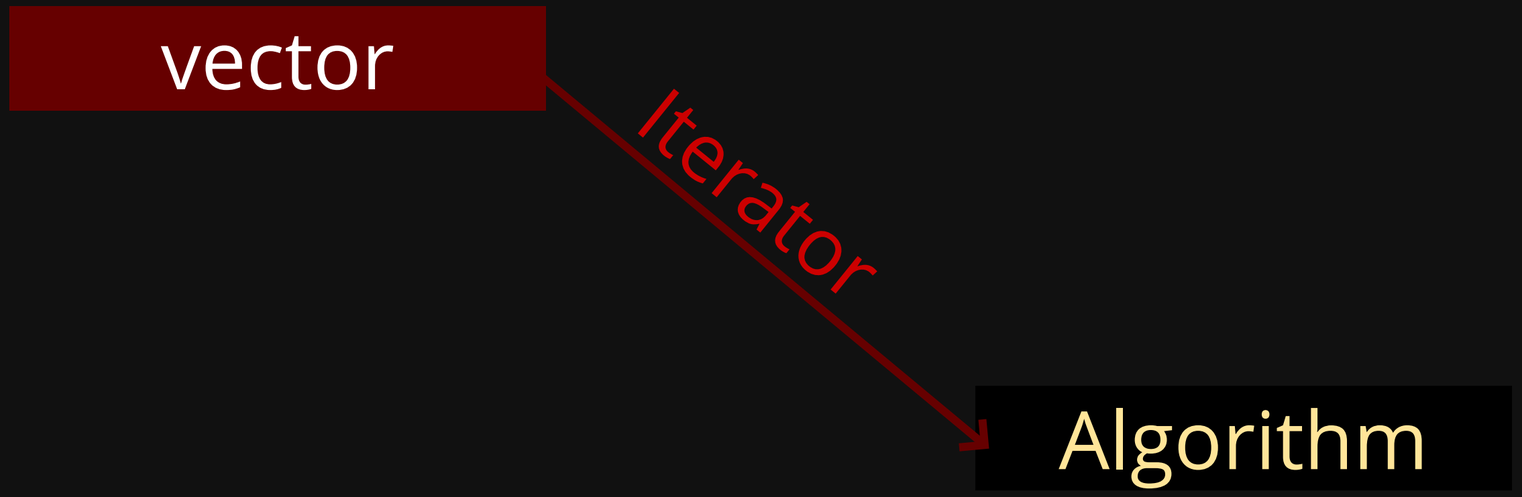
≈

xy algorithm implementations

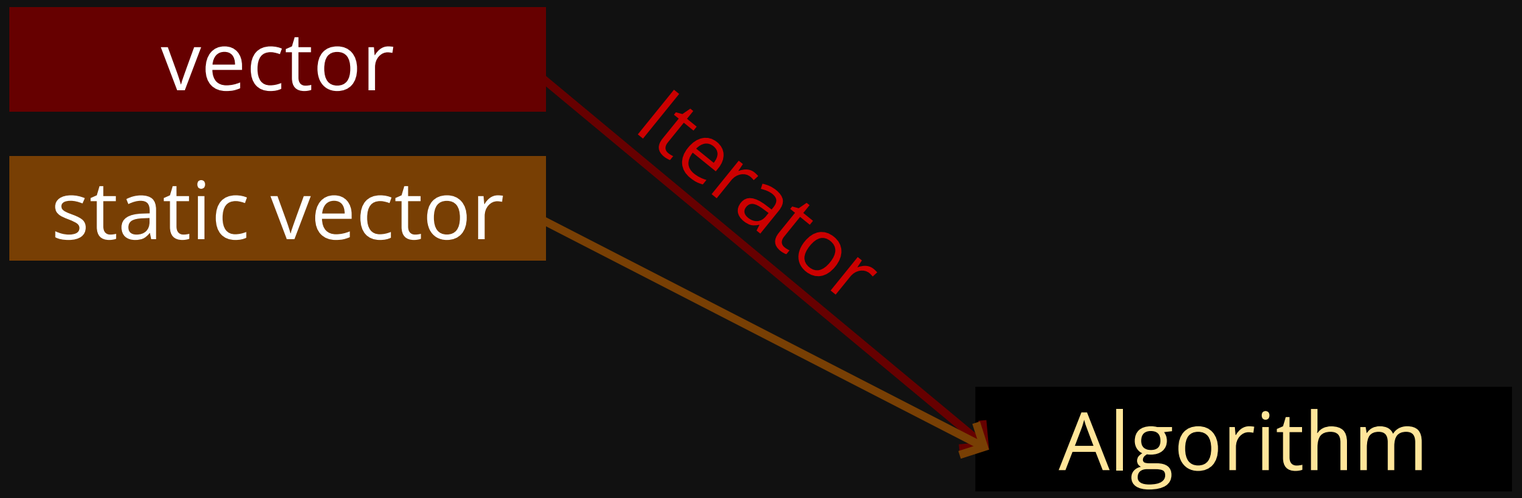
We need an intermediate representation

Algorithm

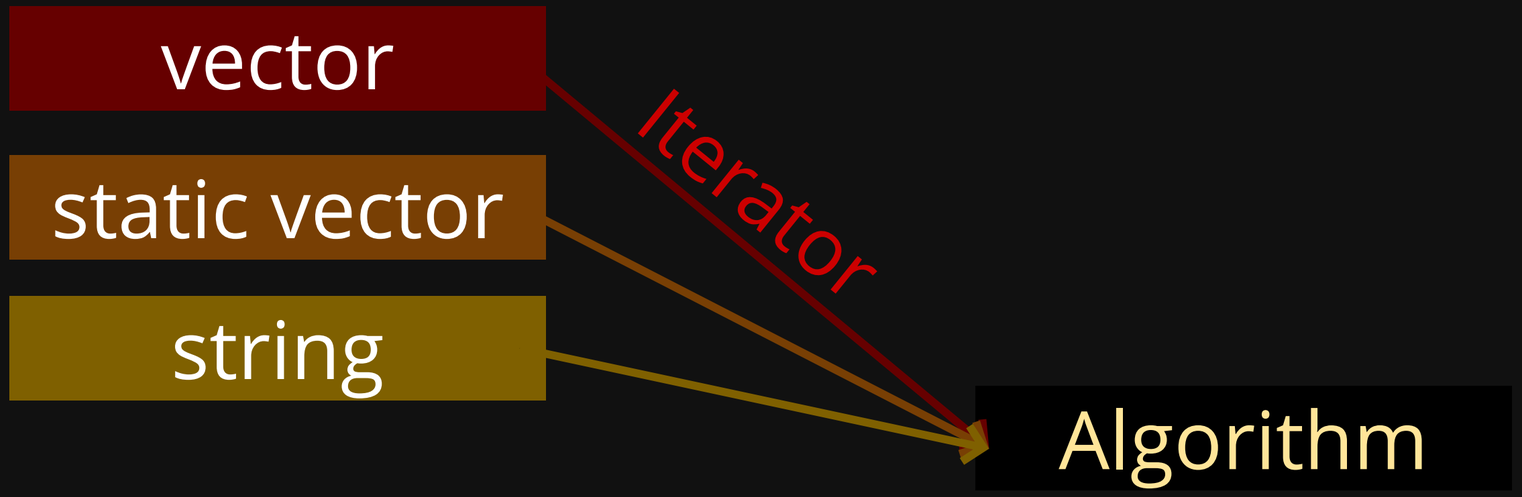
We need an intermediate representation



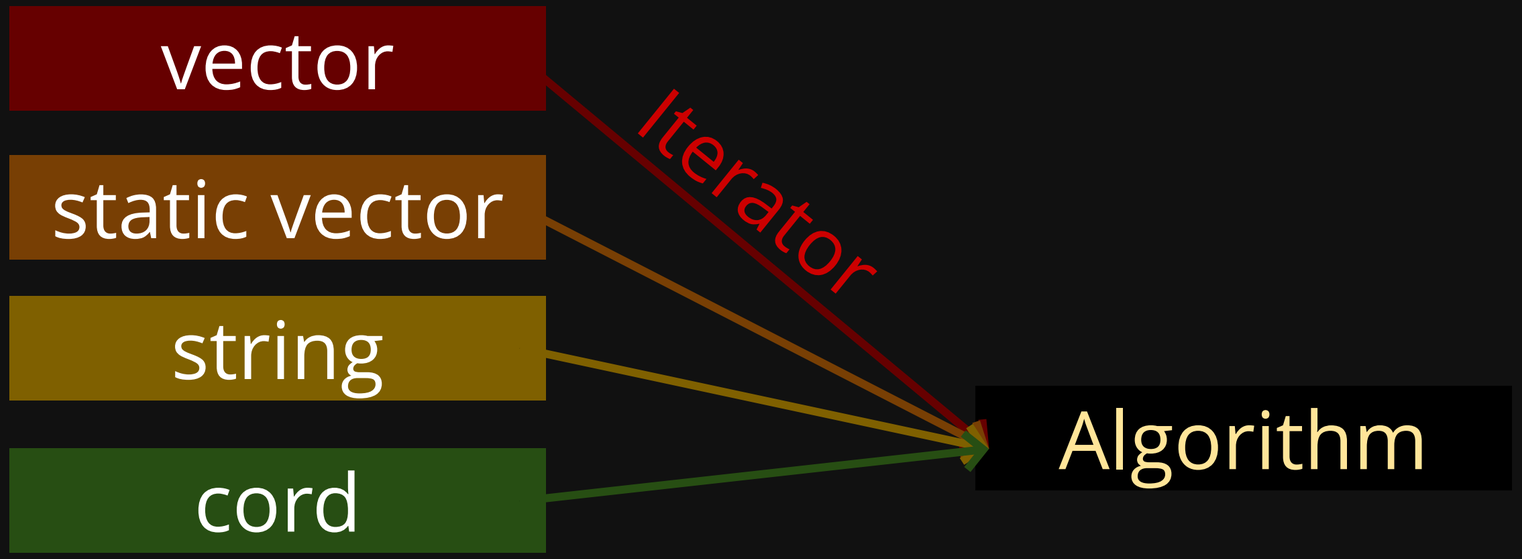
We need an intermediate representation



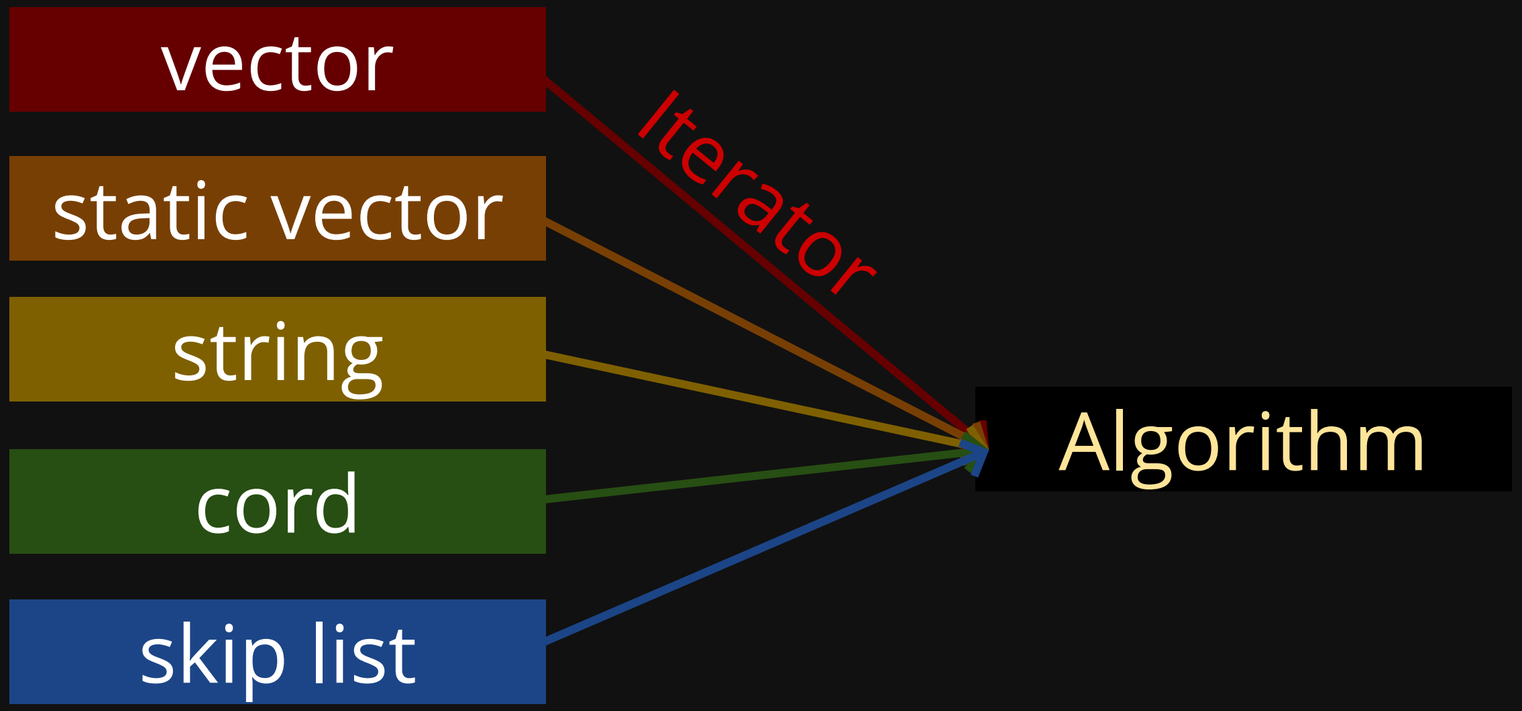
We need an intermediate representation



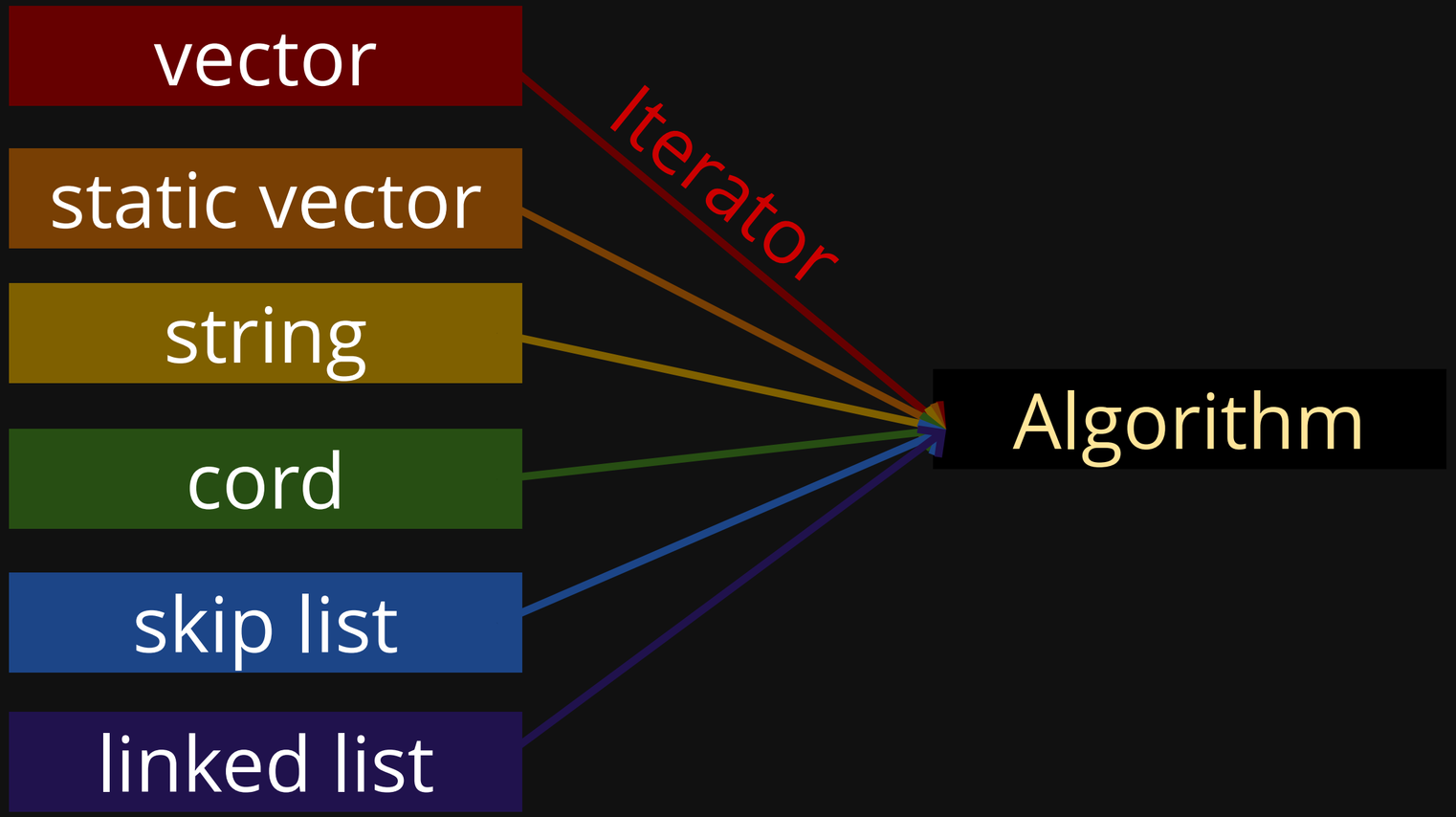
We need an intermediate representation



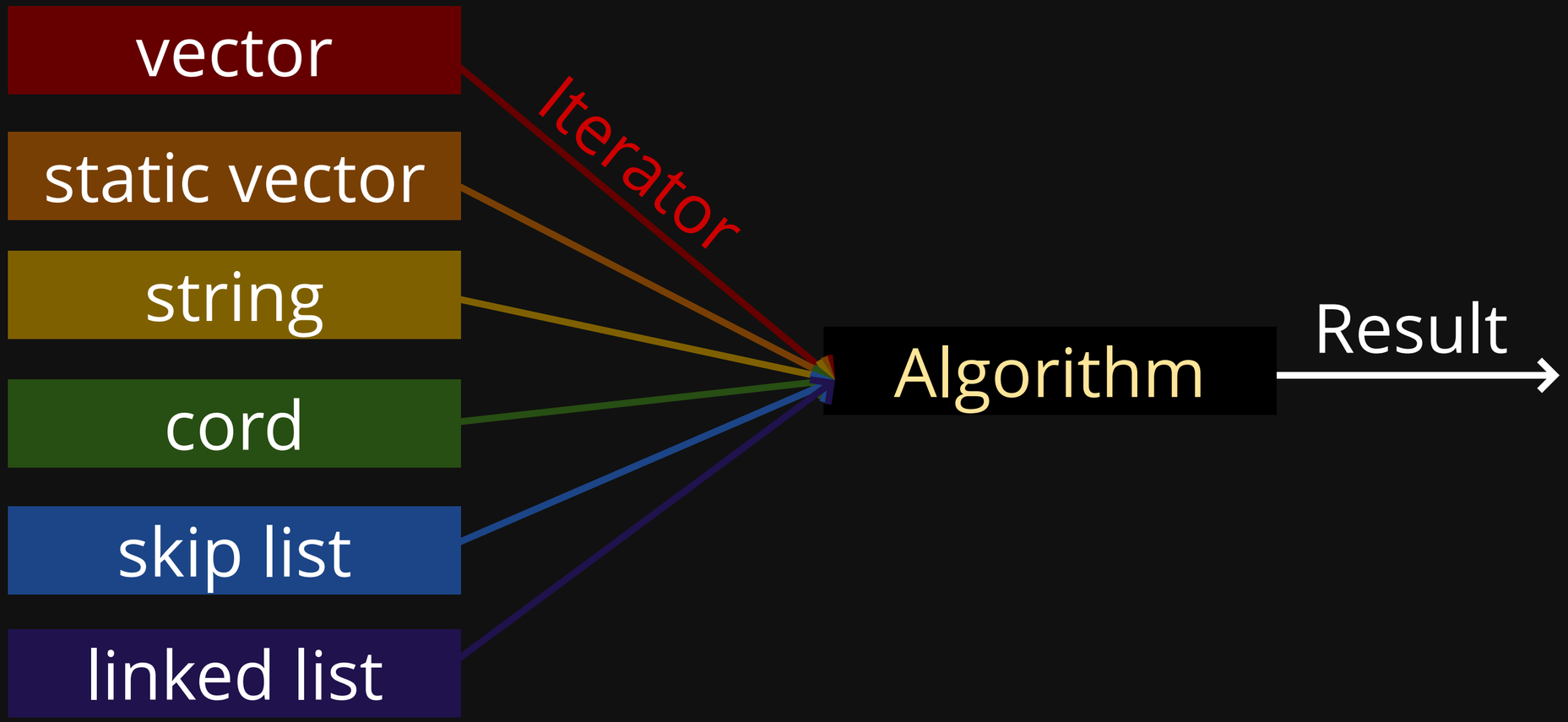
We need an intermediate representation



We need an intermediate representation

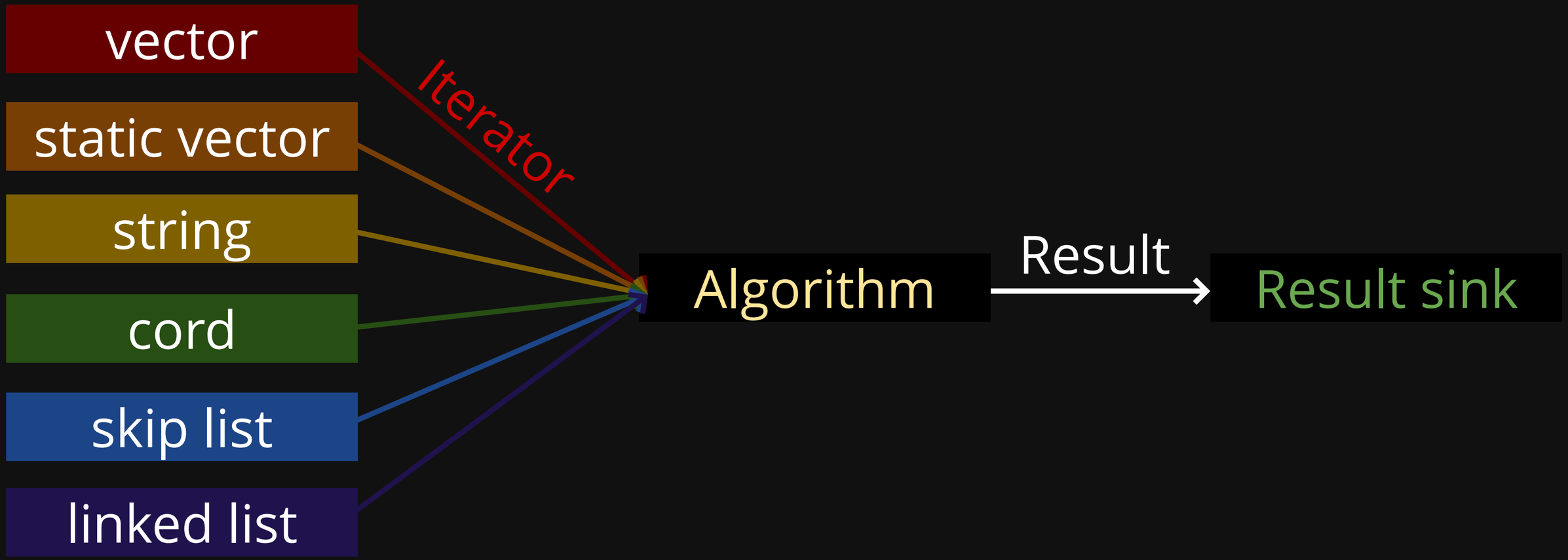


We need an intermediate representation



The result may be a one or more iterators, a scalar value, or some combination of both.

We need an intermediate representation



The result may be a one or more iterators, a scalar value, or some combination of both.

x sequence containers

+

y sequence algorithms (e.g. find)

≈

x + y total implementations

What makes an iterator tick?

Iterators let us write a generic find

```
{
  auto const v = std::deque<int>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  auto result = ranges::find(v.begin(), v.end(), 6);
  if (result != v.end()) {
    std::cout << "Found 6!\n";
  }
  else {
    std::cout << "Didn't find 6.\n";
  }
}
```

Iterator invalidation

- Iterator is an abstract notion of a **pointer**
- What happens when we modify the container?
  - What happens to iterators?
  - What happens to references to elements?
- Using an invalid iterator is undefined

```cpp
auto v = std::vector<int>{1, 2, 3, 4, 5};
// Copy all 2s
for (auto it = v.begin(); it != v.end(); ++it) {
  if (*it == 2) {
    v.push_back(2);
  }
}
// Erase all 2s
for (auto it = v.begin(); it != v.end(); ++it) {
  if (*it == 2) {
    v.erase(it);
  }
}
```

Iterator invalidation - push_back

- Think about the way a vector is stored
- "If the new size() is greater than capacity() then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated."

```
1 auto v = std::vector<int>{1, 2, 3, 4, 5};
2 // Copy all 2s
3 for (auto it = v.begin(); it != v.end(); ++it) {
4   if (*it == 2) {
5     v.push_back(2);
6   }
7 }
```

https://en.cppreference.com/w/cpp/container/vector/push_back

Iterator invalidation - erase

- "Invalidates iterators and references at or after the point of the erase, including the end() iterator."
- For this reason, erase returns a new iterator

```
1 std::vector v{1, 2, 3, 4, 5};
2 // Erase all even numbers
3 for (auto it = v.begin(); it != v.end(); ) {
4   if (*it % 2 == 0) {
5     it = v.erase(it);
6   } else {
7     ++it;
8   }
9 }
```

https://en.cppreference.com/w/cpp/container/vector/erase

Iterator invalidation - general

- Containers generally don't invalidate when you modify values
- But they may invalidate when removing or adding elements
- `std::vector` invalidates everything when adding elements
- `std::unordered_(map/set)` invalidates all iterators when adding elements
- `std::map`/`std::set` doesn't invalidate iterators upon insertion (why?)

Iterator adaptors

Wrapper around a type to grant that type iterator properties.

or

Wrapper around an iterator type to grant additional or different iterator properties.

Example: `std::reverse_iterator`

`std::reverse_iterator` is an iterator adaptor that transforms an existing iterator's `operator++` to mean "move backward" and `operator--` to mean "move forward".

```
std::reverse_iterator<std::vector<int>::iterator>
```

Readable iterators

Operation	Array-like	Node-based	Iterator

Readable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*

Readable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`

Readable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`
Successor	`j = i + n < ranges::distance(v) ? i + n : ranges::distance(v);`	`j = i->successor(n)`	`ranges::next(i, s, n)`

Readable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`
Successor	`j = i + n < ranges::distance(v) ? i + n : ranges::distance(v);`	`j = i->successor(n)`	`ranges::next(i, s, n)`
Advance fwd	`++i`	`i = i->next`	`++i`

Readable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Read element	`v[i]`	`i->value`	`*i`
Successor	`j = i + n < ranges::distance(v) ? i + n : ranges::distance(v);`	`j = i->successor(n)`	`ranges::next(i, s, n)`
Advance fwd	`++i`	`i = i->next`	`++i`
Comparison	`i < ranges::distance(v)`	`i != nullptr`	`i != s`

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

   `std::iter_value_t<I>`

   A type we can create an lvalue from.

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

   `std::iter_value_t<I>`

   A type we can create an lvalue from.

   `std::iter_reference_t<I>`

   The type `I::operator*` returns.

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

`std::iter_value_t<I>`	A type we can create an lvalue from.
`std::iter_reference_t<I>`	The type `I::operator*` returns.
`std::iter_rvalue_reference_t<I>`	The type `ranges::iter_move(i)` returns.

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

`std::iter_value_t<I>`	A type we can create an lvalue from.
`std::iter_reference_t<I>`	The type `I::operator*` returns.
`std::iter_rvalue_reference_t<I>`	The type `ranges::iter_move(i)` returns.

2. These type pairs share a "relationship":

`std::iter_reference_t<I>`	`std::iter_value_t<I>&`

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

`std::iter_value_t<I>`	A type we can create an lvalue from.
`std::iter_reference_t<I>`	The type `I::operator*` returns.
`std::iter_rvalue_reference_t<I>`	The type `ranges::iter_move(i)` returns.

2. These type pairs share a "relationship":

`std::iter_reference_t<I>`	`std::iter_value_t<I>&`
`std::iter_reference_t<I>`	`std::iter_rvalue_reference_t<I>`

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

`std::iter_value_t<I>`	A type we can create an lvalue from.
`std::iter_reference_t<I>`	The type `I::operator*` returns.
`std::iter_rvalue_reference_t<I>`	The type `ranges::iter_move(i)` returns.

2. These type pairs share a "relationship":

`std::iter_reference_t<I>`	`std::iter_value_t<I>&`
`std::iter_reference_t<I>`	`std::iter_rvalue_reference_t<I>`
`std::iter_rvalue_reference_t<I>`	`std::iter_value_t<I> const&`

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

`std::iter_value_t<I>`	A type we can create an lvalue from.
`std::iter_reference_t<I>`	The type `I::operator*` returns.
`std::iter_rvalue_reference_t<I>`	The type `ranges::iter_move(i)` returns.

2. These type pairs share a "relationship":

`std::iter_reference_t<I>`	`std::iter_value_t<I>&`
`std::iter_reference_t<I>`	`std::iter_rvalue_reference_t<I>`
`std::iter_rvalue_reference_t<I>`	`std::iter_value_t<I> const&`

3. Given an object `i` of type `I`, `*i` outputs the same thing when called with the same input.

Indirectly readable

A type `I` models the concept `std::indirectly_readable` if:

1. These types exist

   `std::iter_value_t<I>`

   A type we can create an lvalue from.

   `std::iter_reference_t<I>`

   The type `I::operator*` returns.

   `std::iter_rvalue_reference_t<I>`

   The type `ranges::iter_move(i)` returns.

2. These type pairs share a "relationship":

`std::iter_reference_t<I>`	`std::iter_value_t<I>&`
`std::iter_reference_t<I>`	`std::iter_rvalue_reference_t<I>`
`std::iter_rvalue_reference_t<I>`	`std::iter_value_t<I> const&`

```
// Approximately
std::move(*i)
```

3. Given an object `i` of type `I`, `*i` outputs the same thing when called with the same input.

Generating those iter_ types (preamble)

```
template<typename T>
class linked_list {
public:
  class iterator;
private:
  struct node {
    T value;
    std::unique_ptr<T> next;
    T* prev;
  };

  std::unique_ptr<node> n;
};
```

Generating those iter_ types (preamble)

```
template<typename T>
class linked_list {
public:
  class iterator;
private:
  struct node {
    T value;
    std::unique_ptr<T> next;
    T* prev;
  };

  std::unique_ptr<node> n;
};
```

Generating those iter_ types (preamble)

```
template<typename T>
class linked_list {
public:
  class iterator;
private:
  struct node {
    T value;
    std::unique_ptr<T> next;
    T* prev;
  };

  std::unique_ptr<node> n;
};
```

Generating those iter_ types (preamble)

```
 1 template<typename T>
 2 class linked_list {
 3 public:
 4   class iterator;
 5 private:
 6   struct node {
 7     T value;
 8     std::unique_ptr<T> next;
 9     T* prev;
10   };
11
12   std::unique_ptr<node> n;
13 };
```

Generating those iter_ types (preamble)

```
template<typename T>
class linked_list {
public:
  class iterator;
private:
  struct node {
    T value;
    std::unique_ptr<T> next;
    T* prev;
  };

  std::unique_ptr<node> n;
};
```

Generating those iter_ types (preamble)

```
template<typename T>
class linked_list {
public:
  class iterator;
private:
  struct node {
    T value;
    std::unique_ptr<T> next;
    T* prev;
  };

  std::unique_ptr<node> n;
};
```

Generating those iter_ types

```
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T; // std::iter_value_t<iterator> is value_type

  auto operator*() const noexcept -> value_type const& {
     // iter_reference_t<iterator> is value_type&
     // iter_rvalue_reference<iterator> is value_type&&
     return pointee_->value;
  }
private:
  node* pointee_;

  friend class linked_list<T>;

  explicit iterator(node* pointee)
  : pointee_(pointee) {}
};

static_assert(std::indirectly_readable<linked_list<int>::iterator>);
```

Generating those iter_ types

```
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T; // std::iter_value_t<iterator> is value_type

  auto operator*() const noexcept -> value_type const& {
     // iter_reference_t<iterator> is value_type&
     // iter_rvalue_reference<iterator> is value_type&&
     return pointee_->value;
  }
private:
  node* pointee_;

  friend class linked_list<T>;

  explicit iterator(node* pointee)
  : pointee_(pointee) {}
};

static_assert(std::indirectly_readable<linked_list<int>::iterator>);
```

Generating those iter_ types

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T; // std::iter_value_t<iterator> is value_type

  auto operator*() const noexcept -> value_type const& {
     // iter_reference_t<iterator> is value_type&
     // iter_rvalue_reference<iterator> is value_type&&
     return pointee_->value;
  }
private:
  node* pointee_;

  friend class linked_list<T>;

  explicit iterator(node* pointee)
  : pointee_(pointee) {}
};

static_assert(std::indirectly_readable<linked_list<int>::iterator>);
```

Generating those iter_ types

```
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T; // std::iter_value_t<iterator> is value_type

  auto operator*() const noexcept -> value_type const& {
     // iter_reference_t<iterator> is value_type&
     // iter_rvalue_reference<iterator> is value_type&&
     return pointee_->value;
  }
private:
  node* pointee_;

  friend class linked_list<T>;

  explicit iterator(node* pointee)
  : pointee_(pointee) {}
};

static_assert(std::indirectly_readable<linked_list<int>::iterator>);
```

Generating those iter_ types

```
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T; // std::iter_value_t<iterator> is value_type

  auto operator*() const noexcept -> value_type const& {
     // iter_reference_t<iterator> is value_type&
     // iter_rvalue_reference<iterator> is value_type&&
     return pointee_->value;
  }
private:
  node* pointee_;

  friend class linked_list<T>;

  explicit iterator(node* pointee)
  : pointee_(pointee) {}
};

static_assert(std::indirectly_readable<linked_list<int>::iterator>);
```

Weakly incrementable

A type `I` models the concept `std::weakly_incrementable` if:

1. `I` models `std::default_initializable` and `std::movable`

Weakly incrementable

A type `I` models the concept `std::weakly_incrementable` if:

1. `I` models `std::default_initializable` and `std::movable`
2. `std::iter_difference_t<I>` exists and is a signed integer.

Weakly incrementable

A type `I` models the concept `std::weakly_incrementable` if:

1. `I` models `std::default_initializable` and `std::movable`
2. `std::iter_difference_t<I>` exists and is a signed integer.

1. `++i` is valid and returns a reference to itself.

Weakly incrementable

A type `I` models the concept `std::weakly_incrementable` if:

1. `I` models `std::default_initializable` and `std::movable`
2. `std::iter_difference_t<I>` exists and is a signed integer.

1. `++i` is valid and returns a reference to itself.
2. `i++` is valid and has the same domain as `++i`

Weakly incrementable

A type `I` models the concept `std::weakly_incrementable` if:

1. `I` models `std::default_initializable` and `std::movable`
2. `std::iter_difference_t<I>` exists and is a signed integer.

1. `++i` is valid and returns a reference to itself.
2. `i++` is valid and has the same domain as `++i`
3. `++i` and `i++` both advance `i`, with constant time complexity

Weakly incrementable

A type `I` models the concept `std::weakly_incrementable` if:

1. `I` models `std::default_initializable` and `std::movable`
2. `std::iter_difference_t<I>` exists and is a signed integer.

Let `i` be an object of type `I`.

1. `++i` is valid and returns a reference to itself.
2. `i++` is valid and has the same domain as `++i`
3. `++i` and `i++` both advance `i`, with constant time complexity

Generating `std::iter_difference_t<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t; // std::iter_difference_t<iterator> is difference_type

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& {
    pointee_ = pointee_->next.get();
    return *this;
  }

  auto operator++(int) -> void { ++*this; }
private:
  //...
};

static_assert(std::weakly_incrementable<linked_list<int>::iterator>);
```

Generating `std::iter_difference_t<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t; // std::iter_difference_t<iterator> is difference_type

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& {
    pointee_ = pointee_->next.get();
    return *this;
  }

  auto operator++(int) -> void { ++*this; }
private:
  //...
};

static_assert(std::weakly_incrementable<linked_list<int>::iterator>);
```

Generating `std::iter_difference_t<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t; // std::iter_difference_t<iterator> is difference_type

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& {
    pointee_ = pointee_->next.get();
    return *this;
  }

  auto operator++(int) -> void { ++*this; }
private:
  //...
};

static_assert(std::weakly_incrementable<linked_list<int>::iterator>);
```

Generating `std::iter_difference_t<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t; // std::iter_difference_t<iterator> is difference_type

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& {
    pointee_ = pointee_->next.get();
    return *this;
  }

  auto operator++(int) -> void { ++*this; }
private:
  //...
};

static_assert(std::weakly_incrementable<linked_list<int>::iterator>);
```

Iterator basis

`std::input_or_output_iterator` is the root concept for all six iterator categories.

A type `I` models the concept `std::input_or_output_iterator` if:

1. `I` models `std::weakly_incrementable`

Iterator basis

`std::input_or_output_iterator` is the root concept for all six iterator categories.

A type `I` models the concept `std::input_or_output_iterator` if:

1. `I` models `std::weakly_incrementable`
2. `*i` is a valid expression returns a reference to an object.

Input iterators

`std::input_iterator` describes the requirements for an iterator that can be read from.

A type `I` models the concept `std::input_iterator` if:

1. `I` models `std::input_or_output_iterator`

Input iterators

`std::input_iterator` describes the requirements for an iterator that can be read from.

A type `I` models the concept `std::input_iterator` if:

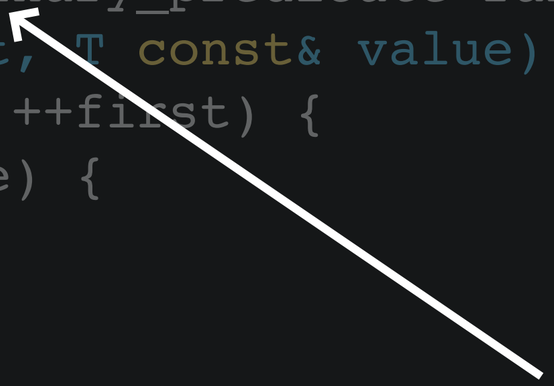
1. `I` models `std::input_or_output_iterator`
2. `I` models `std::indirectly_readable`
3. `I::iterator_category` is a type alias derived from `std::input_iterator_tag`

Input iterators let us write an generic find

```
template<std::input_iterator I, typename T>
requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
auto find(I first, I last, T const& value) -> I {
    for (; first != last; ++first) {
        if (*first == value) {
            return first;
        }
    }

    return last;
}
```

Input iterators let us write an generic find

```
1  template<std::input_iterator I, typename T>
2  requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
3  auto find(I first, I last, T const& value) -> I {
4      for (; first != last; ++first) {
5          if (*first == value) {
6              return first;
7          }
8      }
9
10     return last;
11 }
```

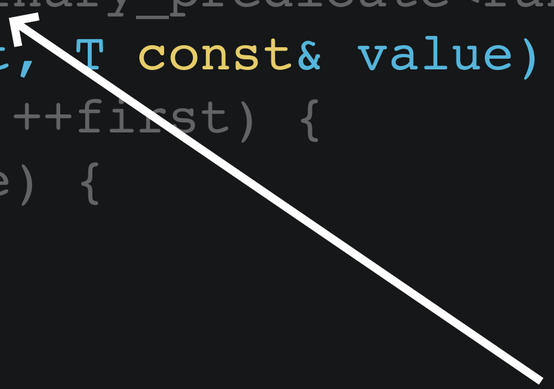


Checks `ranges::equal_to{}(*first, *&value)` is possible.

This is how we check `*first == value` is valid.

Input iterators let us write an generic find

```
1  template<std::input_iterator I, typename T>
2  requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
3  auto find(I first, I last, T const& value) -> I {
4      for (; first != last; ++first) {
5          if (*first == value) {
6              return first;
7          }
8      }
9
10     return last;
11 }
```



Checks `ranges::equal_to{}(*first, *&value)` is possible.

This is how we check `*first == value` is valid.

Input iterators let us write an generic find

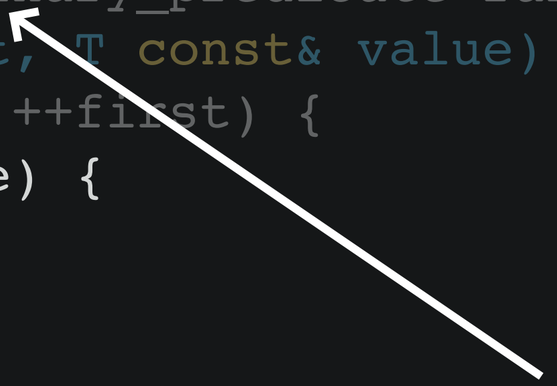
```
1  template<std::input_iterator I, typename T>
2  requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
3  auto find(I first, I last, T const& value) -> I {
4      for (; first != last; ++first) {
5          if (*first == value) {
6              return first;
7          }
8      }
9
10     return last;
11 }
```

Checks `ranges::equal_to{}(*first, *&value)` is possible.

This is how we check `*first == value` is valid.

Input iterators let us write an generic find

```
 1 template<std::input_iterator I, typename T>
 2 requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
 3 auto find(I first, I last, T const& value) -> I {
 4     for (; first != last; ++first) {
 5         if (*first == value) {
 6             return first;
 7         }
 8     }
 9
10     return last;
11 }
```



Checks `ranges::equal_to{}(*first, *&value)` is possible.

This is how we check `*first == value` is valid.

Input iterators let us write an generic find

```
1  template<std::input_iterator I, typename T>
2  requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
3  auto find(I first, I last, T const& value) -> I {
4      for (; first != last; ++first) {
5          if (*first == value) {
6              return first;
7          }
8      }
9
10     return last;
11 }
```

Checks `ranges::equal_to{}(*first, *&value)` is possible.

This is how we check `*first == value` is valid.

Modelling `std::input_iterator<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::input_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> void { ++*this; }
private:
  //...
};

static_assert(std::input_iterator<linked_list<int>::iterator>);
```

tl;dr

`std::weakly_incrementable<I>`

Requires:

- `I::difference_type` is a signed integer
- `++i` and `i++` move `i` to next element in `O(1)` time
- `++i` returns a reference to itself

`std::indirectly_readable<I>`

Requires:

- `I::value_type` exists
- `*i` always outputs the same thing given the same input
- Type of `*i` and `I::value_type` have a type in common

`std::input_or_output_iterator<I>`

Requires:

- `*i` return type can be a reference

`std::input_iterator<I>`

Requires:

- `I::iterator_category` is derived from `std::input_iterator_tag`

Design problem?

```cpp
// real find
template<std::input_iterator I, typename T>
auto find(I first, I last, T const& value) -> I;

// bounded find
template<std::input_iterator I, typename T>
auto find_n(I first, std::iter_difference_t<I> n, T const& value) -> I {
    for (; n > 0; ++first, (void)--n) {
        if (*first == value) {
            return first;
        }
    }
    return first;
}
```

There are >100 algorithms. Do you *really* want to define `copy_n`, `lower_bound_n`, `sort_n`, etc.?

Counted iterators

```
// We leverage this through range-v3 in 20T2
// (it's available as std:: in GCC 10)
template<std::input_iterator I>
counted_iterator<I>::counted_iterator(I first, std::iter_difference_t<I> n);
```

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     ???,
4     value)
```

Counted iterators

```
// We leverage this through range-v3 in 20T2
// (it's available as std:: in GCC 10)
template<std::input_iterator I>
counted_iterator<I>::counted_iterator(I first, std::iter_difference_t<I> n);
```

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     ???,
4     value)
```

Counted iterators

```
// We leverage this through range-v3 in 20T2
// (it's available as std:: in GCC 10)
template<std::input_iterator I>
counted_iterator<I>::counted_iterator(I first, std::iter_difference_t<I> n);
```

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     ???,
4     value)
```

Counted iterators

```
// We leverage this through range-v3 in 20T2
// (it's available as std:: in GCC 10)
template<std::input_iterator I>
counted_iterator<I>::counted_iterator(I first, std::iter_difference_t<I> n);
```

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     std::counted_iterator<int>(), // well, that's a bit weird
4     value)
```

It's weird because we are giving meaning to an arbitrary value, and it doesn't really express intentions to the reader.

It's also limiting, because we can't express any additional information. What if we wanted to also stop on the first even `int`?

Counted iterators

```
// We leverage this through range-v3 in 20T2
// (it's available as std:: in GCC 10)
template<std::input_iterator I>
counted_iterator<I>::counted_iterator(I first, std::iter_difference_t<I> n);
```

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     std::counted_iterator<int>(), // well, that's a bit weird
4     value)
```

It's weird because we are giving meaning to an arbitrary value, and it doesn't really express intentions to the reader.

It's also limiting, because we can't express any additional information. What if we wanted to also stop on the first even `int`?

Counted iterators

```
// We leverage this through range-v3 in 20T2
// (it's available as std:: in GCC 10)
template<std::input_iterator I>
counted_iterator<I>::counted_iterator(I first, std::iter_difference_t<I> n);
```

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     std::counted_iterator<int>(), // well, that's a bit weird
4     value)
```

It's weird because we are giving meaning to an arbitrary value, and it doesn't really express intentions to the reader.

It's also limiting, because we can't express any additional information. What if we wanted to also stop on the first even `int`?

Introducing sentinels

A sentinel is a type that denotes the end of a range. It might be an iterator of the same type (like with containers), or it might be a completely different type.

Types `S` and `I` model the concept `std::sentinel_for<S, I>` if:

1. `I` models `std::input_or_output_iterator`

Introducing sentinels

A sentinel is a type that denotes the end of a range. It might be an iterator of the same type (like with containers), or it might be a completely different type.

Types `S` and `I` model the concept `std::sentinel_for<S, I>` if:

1. `I` models `std::input_or_output_iterator`
2. `S` models `std::semiregular`

Introducing sentinels

A sentinel is a type that denotes the end of a range. It might be an iterator of the same type (like with containers), or it might be a completely different type.

Types `S` and `I` model the concept `std::sentinel_for<S, I>` if:

1. `I` models `std::input_or_output_iterator`
2. `S` models `std::semiregular`

1. `i == s` is well-defined (i.e. it returns `bool` and we have the other three).

Introducing sentinels

A sentinel is a type that denotes the end of a range. It might be an iterator of the same type (like with containers), or it might be a completely different type.

Types `S` and `I` model the concept `std::sentinel_for<S, I>` if:

1. `I` models `std::input_or_output_iterator`
2. `S` models `std::semiregular`

1. `i == s` is well-defined (i.e. it returns `bool` and we have the other three).
2. If `i != s` is true, then `i` is dereferenceable.

Introducing sentinels

A sentinel is a type that denotes the end of a range. It might be an iterator of the same type (like with containers), or it might be a completely different type.

Types `S` and `I` model the concept `std::sentinel_for<S, I>` if:

1. `I` models `std::input_or_output_iterator`
2. `S` models `std::semiregular`

Let `i` be an object of type `I` and `s` be an object of type `S`.

1. `i == s` is well-defined (i.e. it returns `bool` and we have the other three).
2. If `i != s` is true, then `i` is dereferenceable.

std::default_sentinel

The default sentinel is type-based a way of deferring the comparison rule to the iterator when there's no meaningful definition for an end *value*.

```
1 std::ranges::find(
2     std::counted_iterator{first, 10},
3     std::default_sentinel,
4     value)
```

```
1 template<std::input_iterator I>
2 auto counted_iterator<I>::operator==(std::default_sentinel) const -> bool {
3     return n_ == 0;
4 }
```

std::unreachable_sentinel

The unreachable sentinel is a way of saying "there is no end to this range".

```
struct unreachable_sentinel_t {
    template<std::weakly_incrementable I>
    friend constexpr bool operator==(unreachable_sentinel_t, I const&) noexcept {
        return false;
    }
};
```

Completing our implementation of find

```
 1 template<std::input_iterator I, std::sentinel_for<I> S, typename T>
 2 requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
 3 auto find(I first, S last, T const& value) -> I {
 4     for (; first != last; ++first) {
 5         if (*first == value) {
 6             return first;
 7         }
 8     }
 9
10     return first;
11 }
```

Completing our implementation of find

```
 1 template<std::input_iterator I, std::sentinel_for<I> S, typename T>
 2 requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
 3 auto find(I first, S last, T const& value) -> I {
 4     for (; first != last; ++first) {
 5         if (*first == value) {
 6             return first;
 7         }
 8     }
 9
10     return first;
11 }
```

Completing our implementation of find

```
 1 template<std::input_iterator I, std::sentinel_for<I> S, typename T>
 2 requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
 3 auto find(I first, S last, T const& value) -> I {
 4     for (; first != last; ++first) {
 5         if (*first == value) {
 6             return first;
 7         }
 8     }
 9
10     return first;
11 }
```

Completing our implementation of find

```cpp
template<std::input_iterator I, std::sentinel_for<I> S, typename T>
requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
auto find(I first, S last, T const& value) -> I {
    for (; first != last; ++first) {
        if (*first == value) {
            return first;
        }
    }

    return first;
}
```

Completing our implementation of find

```
 1 template<std::input_iterator I, std::sentinel_for<I> S, typename T>
 2 requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
 3 auto find(I first, S last, T const& value) -> I {
 4     for (; first != last; ++first) {
 5         if (*first == value) {
 6             return first;
 7         }
 8     }
 9
10     return first;
11 }
```

Completing our implementation of find

```
template<std::input_iterator I, std::sentinel_for<I> S, typename T>
requires std::indirect_binary_predicate<ranges::equal_to, I, T const*>
auto find(I first, S last, T const& value) -> I {
    for (; first != last; ++first) {
        if (*first == value) {
            return first;
        }
    }

    return first;
}
```

Relationship between iterators and ranges

Let's say there's an object `r` of type `R`

std::ranges::begin

```
template<typename R>
std::input_or_output_iterator auto std::ranges::begin(R&& r);
```

Returns an object that models `std::input_or_output_iterator`.

`std::ranges::begin`

```
template<typename R>
std::input_or_output_iterator auto std::ranges::begin(R&& r);
```

Returns an object that models `std::input_or_output_iterator`.

Works on lvalues and sometimes on rvalues.

std::ranges::begin

```
template<typename R>
std::input_or_output_iterator auto std::ranges::begin(R&& r);
```

- Returns an object that models `std::input_or_output_iterator`.
- Works on lvalues and sometimes on rvalues.
- Works on types that use `r.begin()` (e.g. `std::vector`)

std::ranges::begin

```
template<typename R>
std::input_or_output_iterator auto std::ranges::begin(R&& r);
```

- Returns an object that models `std::input_or_output_iterator`.
- Works on lvalues and sometimes on rvalues.
- Works on types that use `r.begin()` (e.g. `std::vector`)
- Works on types that use `begin(r)` without needing to do this insanity in every scope:

```
using std::begin;
auto i = begin(r);
```

std::ranges::begin

```
template<typename R>
std::input_or_output_iterator auto std::ranges::begin(R&& r);
```

Returns an object that models `std::input_or_output_iterator`.

Works on lvalues and sometimes on rvalues.

Works on types that use `r.begin()` (e.g. `std::vector`)

Works on types that use `begin(r)` without needing to do this insanity in every scope:

```
using std::begin;
auto i = begin(r);
```

Example usage:

```
auto i = std::ranges::begin(r);
```

std::ranges::begin

```
template<typename R>
std::input_or_output_iterator auto std::ranges::begin(R&& r);
```

- Returns an object that models `std::input_or_output_iterator`.
- Works on lvalues and sometimes on rvalues.
- Works on types that use `r.begin()` (e.g. `std::vector`)
- Works on types that use `begin(r)` without needing to do this insanity in every scope:

```
using std::begin;
auto i = begin(r);
```

Example usage:

```
auto i = std::ranges::begin(r);
```

`std::iterator_t<R>` is defined as the deduced return type for `std::ranges::begin(r)`.

std::ranges::end

```
template<typename R>
std::sentinel_for<std::ranges::iterator_t<R>> auto
std::ranges::end(R&& r);
```

Returns an object that models `std::sentinel_for<std::iterator_t<R>>`.

std::ranges::end

```
template<typename R>
std::sentinel_for<std::ranges::iterator_t<R>> auto
std::ranges::end(R&& r);
```

Returns an object that models `std::sentinel_for<std::iterator_t<R>>`.

Works on lvalues and sometimes on rvalues.

std::ranges::end

```
template<typename R>
std::sentinel_for<std::ranges::iterator_t<R>> auto
std::ranges::end(R&& r);
```

- Returns an object that models `std::sentinel_for<std::iterator_t<R>>`.
- Works on lvalues and sometimes on rvalues.
- Works on types that use `r.end()` (e.g. `std::vector`)

std::ranges::end

```
template<typename R>
std::sentinel_for<std::ranges::iterator_t<R>> auto
std::ranges::end(R&& r);
```

Returns an object that models `std::sentinel_for<std::iterator_t<R>>`.

Works on lvalues and sometimes on rvalues.

Works on types that use `r.end()` (e.g. `std::vector`)

Works on types that use `end(r)` without needing to do this insanity in every scope:

```
using std::end;
auto i = end(r);
```

std::ranges::end

```
template<typename R>
std::sentinel_for<std::ranges::iterator_t<R>> auto
std::ranges::end(R&& r);
```

Returns an object that models `std::sentinel_for<std::iterator_t<R>>`.

Works on lvalues and sometimes on rvalues.

Works on types that use `r.end()` (e.g. `std::vector`)

Works on types that use `end(r)` without needing to do this insanity in every scope:

```
using std::end;
auto i = end(r);
```

Example usage:

```
auto s = std::ranges::end(r);
```

std::ranges::end

```
template<typename R>
std::sentinel_for<std::ranges::iterator_t<R>> auto
std::ranges::end(R&& r);
```

Returns an object that models `std::sentinel_for<std::iterator_t<R>>`.

Works on lvalues and sometimes on rvalues.

Works on types that use `r.end()` (e.g. `std::vector`)

Works on types that use `end(r)` without needing to do this insanity in every scope:

```
using std::end;
auto i = end(r);
```

Example usage:

```
auto s = std::ranges::end(r);
```

`std::sentinel_t<R>` is defined as the deduced return type for `std::ranges::end(r)`.

std::ranges::range

`R` models the concept `range` when:

- `R` is a valid type parameter for both `std::ranges::begin` and `std::ranges::end`, where `std::ranges::end`
- `ranges::begin(r)` returns an iterator in amortised `O(1)` time.
- `ranges::end(r)` returns a sentinel in amortised `O(1)` time.
- [`ranges::begin(r)`, `ranges::end(r)`) denotes a valid range (i.e. there's a finite number of iterations between the two).

```
template<typename T>
concept range = requires(R& r) {
   std::ranges::begin(r); // returns an iterator
   std::ranges::end(r);   // returns a sentinel for that iterator
};
```

Note: `std::ranges::begin(r)` is not required to return the same result on each call.

`std::ranges::input_range`

Refines `range` to make sure `ranges::begin(r)` returns an input iterator.

```
template<typename T>
concept input_range =
   std::ranges::range<T> and
   std::input_iterator<std::ranges::iterator_t<T>>;
```

A range-based find

```
1 template<ranges::input_range R, typename T>
2 requires std::indirect_binary_predicate<ranges::equal_to,
3                                         ranges::iterator_t<R>>, const T*>
4 auto find(R&& r, const T& value) -> ranges::borrowed_iterator_t<R> {
5    return comp6771::find(ranges::begin(r), ranges::end(r), value)
6 }
```

The range-based `find` simply defers to the iterator-based `find`. We prefer this algorithm when we need both begin and end, as opposed to arbitrary iterators, so that we don't mix iterators up and create invalid ranges (it's also more readable).

A range-based find

```
1 template<ranges::input_range R, typename T>
2 requires std::indirect_binary_predicate<ranges::equal_to,
3                                         ranges::iterator_t<R>>, const T*>
4 auto find(R&& r, const T& value) -> ranges::borrowed_iterator_t<R> {
5    return comp6771::find(ranges::begin(r), ranges::end(r), value)
6 }
```

The range-based `find` simply defers to the iterator-based `find`. We prefer this algorithm when we need both begin and end, as opposed to arbitrary iterators, so that we don't mix iterators up and create invalid ranges (it's also more readable).

`ranges::borrowed_iterator_t<R>` is the same as `ranges::iterator_t<R>` if `R` is an lvalue reference or a reference to a borrowed range (not discussed), and a non-iterator otherwise.

A range-based find

```
1 template<ranges::input_range R, typename T>
2 requires std::indirect_binary_predicate<ranges::equal_to,
3                                         ranges::iterator_t<R>>, const T*>
4 auto find(R&& r, const T& value) -> ranges::borrowed_iterator_t<R> {
5    return comp6771::find(ranges::begin(r), ranges::end(r), value)
6 }
```

The range-based `find` simply defers to the iterator-based `find`. We prefer this algorithm when we need both begin and end, as opposed to arbitrary iterators, so that we don't mix iterators up and create invalid ranges (it's also more readable).

`ranges::borrowed_iterator_t<R>` is the same as `ranges::iterator_t<R>` if `R` is an lvalue reference or a reference to a borrowed range (not discussed), and a non-iterator otherwise.

```
auto v = std::vector<int>{0, 1, 2, 3};
ranges::find(v, 2); // returns an iterator
ranges::find(views::iota(0, 100), 2); // returns an iterator

ranges::find(std::vector<int>{0, 1, 2, 3}, 2); // returns ranges::dangling which is more useful
                                               // than void (better compile-time diagnostic info)
```

Find last

```
template<std::input_iterator I, std::sentinel_for<I> S, class Val>
requires std::indirect_relation<ranges::equal_to, I, Val const*>
auto find_last(I first, S last, Val const& value) -> I {
        auto cache = std::optional<I>();
        for (; first != last; ++first) {
                if (*first == value) {
                        cache = first;
                }
        }

        return cache.value_or(std::move(first));
}
```

Find last

```
1  template<std::input_iterator I, std::sentinel_for<I> S, class Val>
2  requires std::indirect_relation<ranges::equal_to, I, Val const*>
3  auto find_last(I first, S last, Val const& value) -> I {
4          auto cache = std::optional<I>();
5          for (; first != last; ++first) {
6                  if (*first == value) {
7                          cache = first;
8                  }
9          }
10
11         return cache.value_or(std::move(first));
12 }
```

`I` isn't guaranteed to model `std::copyable`

Even if it were, `I` is only guaranteed to work for a single-pass

What if we are reading input from a network socket?

Incrementable

Refines `std::weakly_incrementable` so you can copy and iterate over the same sequence of values multiple times.

Incrementable

Refines `std::weakly_incrementable` so you can copy and iterate over the same sequence of values multiple times.

A type `I` models the concept `std::incrementable` if:

1. `I` models `std::weakly_incrementable`

Incrementable

Refines `std::weakly_incrementable` so you can copy and iterate over the same sequence of values multiple times.

A type `I` models the concept `std::incrementable` if:

1. `I` models `std::weakly_incrementable`
2. `I` models `std::regular`

Incrementable

Refines `std::weakly_incrementable` so you can copy and iterate over the same sequence of values multiple times.

A type `I` models the concept `std::incrementable` if:

1. `I` models `std::weakly_incrementable`
2. `I` models `std::regular`
3. `i++` is equivalent to

```
auto I::operator++(int) -> I {
    auto temp = i;
    ++i;
    return temp;
}
```

Modelling `std::incrementable<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::input_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> iterator {
     auto temp = *this;
     ++*this;
     return temp;
  }
private:
  //...
};

static_assert(std::incrementable<linked_list<int>::iterator>);
```

Forward iterators

Refines input iterators so you can copy and iterate over the same sequence of values multiple times.

Forward iterators

Refines input iterators so you can copy and iterate over the same sequence of values multiple times.

A type `I` models the concept `std::forward_iterator` if:

1. `I` models `std::input_iterator`

Forward iterators

Refines input iterators so you can copy and iterate over the same sequence of values multiple times.

A type `I` models the concept `std::forward_iterator` if:

1. `I` models `std::input_iterator`
2. `I` models `std::incrementable`

Forward iterators

Refines input iterators so you can copy and iterate over the same sequence of values multiple times.

A type `I` models the concept `std::forward_iterator` if:

1. `I` models `std::input_iterator`
2. `I` models `std::incrementable`
3. `I::iterator_category` is derived from `std::forward_iterator_tag`
4. `I` can be its own sentinel

Modelling `std::forward_iterator<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::forward_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type& { ... }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> iterator { ... }

  auto operator==(iterator, iterator) const -> bool = default;
private:
  //...
};

static_assert(std::forward_iterator<linked_list<int>::iterator>);
```

Find last

```
 1 template<std::forward_iterator I, std::sentinel_for<I> S, class Val>
 2 requires std::indirect_relation<ranges::equal_to, I, Val const*>
 3 auto find_last(I first, S last, Val const& value) -> I {
 4         auto cache = std::optional<I>();
 5         for (; first != last; ++first) {
 6                 if (*first == value) {
 7                         cache = first;
 8                 }
 9         }
10
11         return cache.value_or(std::move(first));
12 }
```

Can we optimise this algorithm further?

Forward range

Refines `input_range` to make sure `ranges::begin(r)` returns a forward iterator.

`ranges::begin(r)` now returns the same value given the same input.

```
template<typename T>
concept forward_range =
    std::ranges::input_range<T> and
    std::forward_iterator<std::ranges::iterator_t<T>>;
```

tl;dr

`std::input_iterator<I>`

(See previous tl;dr)

`std::incrementable<I>`

Requires:

`std::weakly_incrementable<I>`

`std::regular<I>`

```
1 auto I::operator++(int) -> I {
2     auto temp = *this;
3     ++*this;
4     return temp;
5 }
```

`std::forward_iterator<I>`

Requires:

`I::iterator_category` is derived from `std::forward_iterator_tag`

`auto operator==(I, I) -> bool;`

`auto operator!=(I, I) -> bool;` (before C++20)

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

1. `I` models `std::forward_iterator`

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

1. `I` models `std::forward_iterator`
2. `I::iterator_category` is derived from `std::bidirectional_iterator_tag`

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

1. `I` models `std::forward_iterator`
2. `I::iterator_category` is derived from `std::bidirectional_iterator_tag`
3. `--i` is valid and returns a reference to itself.

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

1. `I` models `std::forward_iterator`
2. `I::iterator_category` is derived from `std::bidirectional_iterator_tag`
3. `--i` is valid and returns a reference to itself.
4. `i--` is valid and has the same domain as `--i`

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

1. `I` models `std::forward_iterator`
2. `I::iterator_category` is derived from `std::bidirectional_iterator_tag`
3. `--i` is valid and returns a reference to itself.
4. `i--` is valid and has the same domain as `--i`
5. `--i` and `i--` both decline `i` in constant time complexity

Bidirectional iterators

Refines forward iterators so you can iterate in reverse order.

A type `I` models the concept `std::bidirectional_iterator` if:

1. `I` models `std::forward_iterator`
2. `I::iterator_category` is derived from `std::bidirectional_iterator_tag`
3. `--i` is valid and returns a reference to itself.
4. `i--` is valid and has the same domain as `--i`
5. `--i` and `i--` both decline `i` in constant time complexity
6. `i--` is equivalent to

```
auto I::operator--(int) -> I {
    auto temp = i;
    --i;
    return temp;
}
```

Modelling `std::bidirectional_iterator<I>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type const& { ... }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> iterator { ... }

  auto operator--() -> iterator& {
     pointee_ = pointee_->prev;
     return *this;
  }

  auto operator--(int) -> iterator {
     auto temp = *this;
     --*this;
     return temp;
  }

  auto operator==(iterator) const -> bool = default;
private:
  //...
};

static_assert(std::bidirectional_iterator<linked_list<int>::iterator>);
```

Find last

```
template<std::bidirectional_iterator I, class Val>
requires std::indirect_relation<ranges::equal_to, I, Val const*>
auto find_last(I first, I last, Val const& value) -> I {
        while (first != last) {
                --last;
                if (*last == value) {
                        return last;
                }
        }

        return first;
}
```

Find last

```
template<std::bidirectional_iterator I, class Val>
requires std::indirect_relation<ranges::equal_to, I, Val const*>
auto find_last(I first, I last, Val const& value) -> I {
        while (first != last) {
                --last;
                if (*last == value) {
                        return last;
                }
        }

        return first;
}
```

Find last

```
1  template<std::bidirectional_iterator I, class Val>
2  requires std::indirect_relation<ranges::equal_to, I, Val const*>
3  auto find_last(I first, I last, Val const& value) -> I {
4          while (first != last) {
5                  --last;
6                  if (*last == value) {
7                          return last;
8                  }
9          }
10
11         return first;
12 }
```

We keep both implementations

```
template<std::forward_iterator I, std::sentinel_for<I> S, class Val>
requires std::indirect_relation<ranges::equal_to, I, Val const*>
auto find_last(I first, S last, Val const& value) -> I {
        auto cache = std::optional<I>();
        for (; first != last; ++first) {
                if (*first == value) {
                        cache = first;
                }
        }

        return cache.value_or(std::move(first));
}

template<std::bidirectional_iterator I, class Val>
requires std::indirect_relation<ranges::equal_to, I, Val const*>
auto find_last(I first, I last, Val const& value) -> I {
        while (first != last--) {
                if (*last == value) {
                        return last;
                }
        }

        return first;
}
```

tl;dr

`std::forward_iterator<I>`

(See previous tl;dr)

`std::bidirectional_iterator<I>`

Requires:

`I::iterator_category` is derived from `std::bidirectional_iterator_tag`

`--i` and `i--` move to the previous element, with `O(1)` time complexity

`--i` returns a reference to itself

```
1 auto I::operator--(int) -> I {
2     auto temp = *this;
3     --*this;
4     return temp;
5 }
```

A new example type

Our `linked_list` has been a good friend up until now.

It's sadly run its course for the lecture content and we can't use it any more.

Let's now define something similar to `ranges::views::iota`

simple_iota_view

```
template<std::integral I>
class simple_iota_view {
    class iterator;
public:
    simple_iota_view() = default;

    simple_iota_view(I first, I last)
    : start_(std::move(first))
    , stop_(std::move(last)) {}

    auto begin() const -> iterator { return iterator(*this, start_); }
    auto end() const -> iterator { return iterator(*this, stop_); }
private:
    I start_ = I();
    I stop_ = I();
};
```

simple_iota_view

```
template<std::integral I>
class simple_iota_view {
    class iterator;
public:
    simple_iota_view() = default;

    simple_iota_view(I first, I last)
    : start_(std::move(first))
    , stop_(std::move(last)) {}

    auto begin() const -> iterator { return iterator(*this, start_); }
    auto end() const -> iterator { return iterator(*this, stop_); }
private:
    I start_ = I();
    I stop_ = I();
};
```

simple_iota_view

```
template<std::integral I>
class simple_iota_view {
    class iterator;
public:
    simple_iota_view() = default;

    simple_iota_view(I first, I last)
    : start_(std::move(first))
    , stop_(std::move(last)) {}

    auto begin() const -> iterator { return iterator(*this, start_); }
    auto end() const -> iterator { return iterator(*this, stop_); }
private:
    I start_ = I();
    I stop_ = I();
};
```

simple_iota_view

```
template<std::integral I>
class simple_iota_view {
    class iterator;
public:
    simple_iota_view() = default;

    simple_iota_view(I first, I last)
    : start_(std::move(first))
    , stop_(std::move(last)) {}

    auto begin() const -> iterator { return iterator(*this, start_); }
    auto end() const -> iterator { return iterator(*this, stop_); }
private:
    I start_ = I();
    I stop_ = I();
};
```

simple_iota_view

```
template<std::integral I>
class simple_iota_view {
    class iterator;
public:
    simple_iota_view() = default;

    simple_iota_view(I first, I last)
    : start_(std::move(first))
    , stop_(std::move(last)) {}

    auto begin() const -> iterator { return iterator(*this, start_); }
    auto end() const -> iterator { return iterator(*this, stop_); }
private:
    I start_ = I();
    I stop_ = I();
};
```

simple_iota_view::iterator

```
1  template<std::integral I>
2  class simple_iota_view<I>::iterator {
3  public:
4     using value_type = I;
5     using difference_type = std::iter_difference_t<I>;
6     using iterator_category = iterator_category_helper_t<I>;
7
8     iterator() = default;
9
10    explicit iterator(simple_iota_view const& base, I const& value)
11    : base_(std::addressof(base))
12    , current_(value) {}
13
14    auto operator*() const -> I { return current_; }
15
16    auto operator++() -> iterator& { ++current_; return *this; }
17    auto operator++(int) -> iterator { ... }
18
19    auto operator--() -> iterator& { --current_; return *this; }
20    auto operator--(int) -> iterator { ... }
21
22    auto operator==(iterator) const -> bool = default;
23 private:
24    simple_iota_view const* base_ = nullptr;
25    I current_ = I();
26 };
```

simple_iota_view::iterator

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
    using value_type = I;
    using difference_type = std::iter_difference_t<I>;
    using iterator_category = iterator_category_helper_t<I>;

    iterator() = default;

    explicit iterator(simple_iota_view const& base, I const& value)
    : base_(std::addressof(base))
    , current_(value) {}

    auto operator*() const -> I { return current_; }

    auto operator++() -> iterator& { ++current_; return *this; }
    auto operator++(int) -> iterator { ... }

    auto operator--() -> iterator& { --current_; return *this; }
    auto operator--(int) -> iterator { ... }

    auto operator==(iterator) const -> bool = default;
private:
    simple_iota_view const* base_ = nullptr;
    I current_ = I();
};
```

simple_iota_view::iterator

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
    using value_type = I;
    using difference_type = std::iter_difference_t<I>;
    using iterator_category = iterator_category_helper_t<I>;

    iterator() = default;

    explicit iterator(simple_iota_view const& base, I const& value)
    : base_(std::addressof(base))
    , current_(value) {}

    auto operator*() const -> I { return current_; }

    auto operator++() -> iterator& { ++current_; return *this; }
    auto operator++(int) -> iterator { ... }

    auto operator--() -> iterator& { --current_; return *this; }
    auto operator--(int) -> iterator { ... }

    auto operator==(iterator) const -> bool = default;
private:
    simple_iota_view const* base_ = nullptr;
    I current_ = I();
};
```

simple_iota_view::iterator

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
    using value_type = I;
    using difference_type = std::iter_difference_t<I>;
    using iterator_category = iterator_category_helper_t<I>;

    iterator() = default;

    explicit iterator(simple_iota_view const& base, I const& value)
    : base_(std::addressof(base))
    , current_(value) {}

    auto operator*() const -> I { return current_; }

    auto operator++() -> iterator& { ++current_; return *this; }
    auto operator++(int) -> iterator { ... }

    auto operator--() -> iterator& { --current_; return *this; }
    auto operator--(int) -> iterator { ... }

    auto operator==(iterator) const -> bool = default;
private:
    simple_iota_view const* base_ = nullptr;
    I current_ = I();
};
```

Check if two ranges meet some predicate, element-wise

```cpp
template<std::input_iterator I1, std::sentinel_for<I1> S1,
         std::input_iterator I2, std::sentinel_for<I2> S2,
         std::indirect_binary_predicate<Pred, I1, I2> Pred = ranges::equal_to>
bool comp6771::equal(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {})
{
    auto result = ranges::mismatch(std::move(first1), last1,
                                   std::move(first2), last2,
                                   std::ref(pred));
    return (result.in1 == last1) and (result.in2 == last2);

}
```

A good starting point, but `ranges::mismatch` isn't designed to short-circuit.

What might we be able to do to optimise this?

An optimisation

If ranges don't have the same distance, then complexity should be constant!

Checking the distance of a range is currently a linear operation...

```
template<std::input_iterator I1, std::sentinel_for<I1> S1,
         std::input_iterator I2, std::sentinel_for<I2> S2,
         std::indirect_binary_predicate<Pred, I1, I2> Pred = ranges::equal_to>
bool comp6771::equal(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {})
{
   auto result = ranges::mismatch(std::move(first1), last1,
                                  std::move(first2), last2,
                                  std::ref(pred));
   return (result.in1 == last1) and (result.in2 == last2);

}
```

Sized sentinels

Refines `std::sentinel` so that we can compute the distance between two elements in constant time.

Sized sentinels

Refines `std::sentinel` so that we can compute the distance between two elements in constant time.

Given an iterator type `I`, A type `S` models the concept `std::sized_sentinel_for<I>` if:

Sized sentinels

Refines `std::sentinel` so that we can compute the distance between two elements in constant time.

Given an iterator type `I`, A type `S` models the concept `std::sized_sentinel_for<I>` if:

1. `S` models `std::sentinel_for<I>`

Sized sentinels

Refines `std::sentinel` so that we can compute the distance between two elements in constant time.

Given an iterator type `I`, A type `S` models the concept `std::sized_sentinel_for<I>` if:

1. `S` models `std::sentinel_for<I>`
2. `ranges::disable_sized_sentinel_for<S, I>` is `false`

Used to opt *out* of being a sized sentinel for `I` if `s - i` doesn't evaluate in `O(1)` time (i.e. this is `false` by default).

Sized sentinels

Refines `std::sentinel` so that we can compute the distance between two elements in constant time.

Given an iterator type `I`, A type `S` models the concept `std::sized_sentinel_for<I>` if:

1. `S` models `std::sentinel_for<I>`
2. `ranges::disable_sized_sentinel_for<S, I>` is `false`
3. `s - i` returns the number of elements between the iterator and sentinel, as an object of type `std::iter_difference_t<I>`, in constant time
4. `i - s` is equivalent to `-(s - i)`

Used to opt *out* of being a sized sentinel for `I` if `s - i` doesn't evaluate in `O(1)` time (i.e. this is `false` by default).

An optimisation

```cpp
template<std::input_iterator I1, std::sentinel_for<I1> S1,
         std::input_iterator I2, std::sentinel_for<I2> S2,
         std::indirect_binary_predicate<Pred, I1, I2> Pred = ranges::equal_to>
bool comp6771::equal(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {})
{
   if constexpr (std::sized_sentinel_for<S1, I1> and std::sized_sentinel_for<S2, I2>) {
      if (last1 - first1 != last2 - first2) {
         return false;
      }
   }

   auto result = ranges::mismatch(std::move(first1), last1,
                                  std::move(first2), last2,
                                  std::ref(pred));
   return (result.in1 == last1) and (result.in2 == last2);

}
```

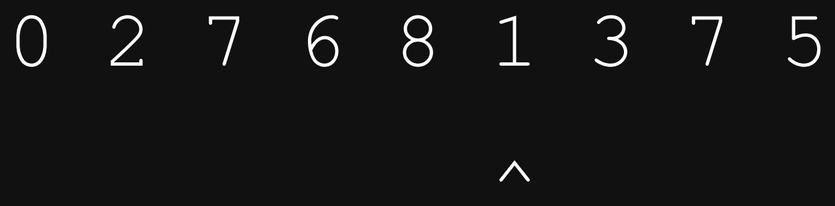
Modelling `std::sized_sentinel_for<S, I>`

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
   using value_type = I;
   using difference_type = std::iter_difference_t<I>;
   using iterator_category = iterator_category_helper_t<I>;

   iterator() = default;

   explicit iterator(simple_iota_view const& base, I const& value);

   auto operator*() const -> I;
   auto operator++() -> iterator&;
   auto operator++(int) -> iterator;
   auto operator--() -> iterator&;
   auto operator--(int) -> iterator;

   auto operator==(iterator) const -> bool = default;

   friend auto operator-(iterator const x, iterator const y) -> difference_type {
     assert(x.base_ == y.base_);
     return x.current_ - y.current_;
   }
private: // ...
};
static_assert(std::sized_sentinel_for<simple_iota_view<int>::iterator, simple_iota_view<int>::iterator>);
```

Partition point

0 2 7 6 8 1 3 7 5

Partition point



Partition point

```
1  template<std::forward_iterator I, std::sentinel_for<I> S,
2           std::indirect_unary_predicate<I> Pred>
3  auto partition_point(I first, S last, Pred pred) -> I {
4     auto end = ranges::next(first, last);
5     while (first != end) {
6        auto middle = ranges::next(first, ranges::distance(first, end) / 2);
7        if (std::invoke(pred, *middle)) {
8           first = std::move(middle);
9           ++first;
10          continue;
11       }
12       end = std::move(middle);
13    }
14
15    return first;
16 }
```

What is the complexity of this `partition_point`?

Partition point

```
1  template<std::forward_iterator I, std::sentinel_for<I> S,
2           std::indirect_unary_predicate<I> Pred>
3  auto partition_point(I first, S last, Pred pred) -> I {
4     auto end = ranges::next(first, last);
5     while (first != end) {
6        auto middle = ranges::next(first, ranges::distance(first, end) / 2);
7        if (std::invoke(pred, *middle)) {
8           first = std::move(middle);
9           ++first;
10          continue;
11       }
12       end = std::move(middle);
13    }
14
15    return first;
16 }
```

What is the complexity of this `partition_point`?

O(*log*(`last - first`)) applications of `pred`

Partition point complexity

O(*log*(`last - first`)) applications of `pred`

But there's currently *O*(*n*) steps through the range.

Wouldn't it be great for the applications and steps to be the same?

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`
2. `I::iterator_category` is derived from `std::random_access_iterator_tag`

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`
2. `I::iterator_category` is derived from `std::random_access_iterator_tag`
3. `I` models `std::totally_ordered`

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`
2. `I::iterator_category` is derived from `std::random_access_iterator_tag`
3. `I` models `std::totally_ordered`
4. `I` is a sized sentinel for itself

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`
2. `I::iterator_category` is derived from `std::random_access_iterator_tag`
3. `I` models `std::totally_ordered`
4. `I` is a sized sentinel for itself

Let `i` and `j` be objects of type `I` and `n` be an object of type `iter_difference_t<I>`.

1. `i += n` and `i -= n` are valid and return references to the same object.
2. `i ±= n` advances/declines `i` by `n` elements in constant time.

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`
2. `I::iterator_category` is derived from `std::random_access_iterator_tag`
3. `I` models `std::totally_ordered`
4. `I` is a sized sentinel for itself

Let `i` and `j` be objects of type `I` and `n` be an object of type `iter_difference_t<I>`.

1. `i += n` and `i -= n` are valid and return references to the same object.
2. `i ±= n` advances/declines `i` by `n` elements in constant time.
3. `j ± n` advances/declines a copy of `j` by `n` elements in constant time.
4. `n + j` is the same as `j + n`.

Random-access iterators

Refines bidirectional iterators so you can make arbitrary steps in constant time

A type `I` models the concept `std::random_access_iterator` if:

1. `I` models `std::bidirectional_iterator`
2. `I::iterator_category` is derived from `std::random_access_iterator_tag`
3. `I` models `std::totally_ordered`
4. `I` is a sized sentinel for itself

Let `i` and `j` be objects of type `I` and `n` be an object of type `iter_difference_t<I>`.

1. `i += n` and `i -= n` are valid and return references to the same object.
2. `i ±= n` advances/declines `i` by `n` elements in constant time.
3. `j ± n` advances/declines a copy of `j` by `n` elements in constant time.
4. `n + j` is the same as `j + n`.
5. `j[n]` is the same as `*(j + n)`.

Modelling `std::random_access_iterator<I>`

```
1  template<std::integral I>
2  class simple_iota_view<I>::iterator {
3  public:
4     using value_type = I;
5     using difference_type = std::iter_difference_t<I>;
6     using iterator_category = std::random_access_iterator_tag;
7
8     // stuff from previous slides
9
10    auto operator<=>(iterator) const -> std::strong_ordering = default;
11
12    auto operator+=(difference_type const n) -> iterator& {
13       current_ += n;
14       return *this;
15    }
16
17    auto operator-=(difference_type const n) -> iterator& { return *this += -n; }
18    auto operator[](difference_type const n) const -> value_type { return *(*this + n); }
19
20    friend auto operator+(iterator j, difference_type const n) -> iterator { return j += n; }
21    friend auto operator+(difference_type const n, iterator j) -> iterator { return j += n; }
22    friend auto operator-(iterator j, difference_type const n) -> iterator { return j -= n; }
23    friend auto operator-(iterator const x, iterator const y) -> difference_type { ... }
24 };
25
26 static_assert(std::random_access_iterator<simple_iota_view<int>::iterator>);
```

Modelling `std::random_access_iterator<I>`

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
   using value_type = I;
   using difference_type = std::iter_difference_t<I>;
   using iterator_category = std::random_access_iterator_tag;

   // stuff from previous slides

   auto operator<=>(iterator) const -> std::strong_ordering = default;

   auto operator+=(difference_type const n) -> iterator& {
      current_ += n;
      return *this;
   }

   auto operator-=(difference_type const n) -> iterator& { return *this += -n; }
   auto operator[](difference_type const n) const -> value_type { return *(*this + n); }

   friend auto operator+(iterator j, difference_type const n) -> iterator { return j += n; }
   friend auto operator+(difference_type const n, iterator j) -> iterator { return j += n; }
   friend auto operator-(iterator j, difference_type const n) -> iterator { return j -= n; }
   friend auto operator-(iterator const x, iterator const y) -> difference_type { ... }
};

static_assert(std::random_access_iterator<simple_iota_view<int>::iterator>);
```

Modelling `std::random_access_iterator<I>`

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
   using value_type = I;
   using difference_type = std::iter_difference_t<I>;
   using iterator_category = std::random_access_iterator_tag;

   // stuff from previous slides

   auto operator<=>(iterator) const -> std::strong_ordering = default;

   auto operator+=(difference_type const n) -> iterator& {
      current_ += n;
      return *this;
   }

   auto operator-=(difference_type const n) -> iterator& { return *this += -n; }
   auto operator[](difference_type const n) const -> value_type { return *(*this + n); }

   friend auto operator+(iterator j, difference_type const n) -> iterator { return j += n; }
   friend auto operator+(difference_type const n, iterator j) -> iterator { return j += n; }
   friend auto operator-(iterator j, difference_type const n) -> iterator { return j -= n; }
   friend auto operator-(iterator const x, iterator const y) -> difference_type { ... }
};

static_assert(std::random_access_iterator<simple_iota_view<int>::iterator>);
```

Modelling `std::random_access_iterator<I>`

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
   using value_type = I;
   using difference_type = std::iter_difference_t<I>;
   using iterator_category = std::random_access_iterator_tag;

   // stuff from previous slides

   auto operator<=>(iterator) const -> std::strong_ordering = default;

   auto operator+=(difference_type const n) -> iterator& {
      current_ += n;
      return *this;
   }

   auto operator-=(difference_type const n) -> iterator& { return *this += -n; }
   auto operator[](difference_type const n) const -> value_type { return *(*this + n); }

   friend auto operator+(iterator j, difference_type const n) -> iterator { return j += n; }
   friend auto operator+(difference_type const n, iterator j) -> iterator { return j += n; }
   friend auto operator-(iterator j, difference_type const n) -> iterator { return j -= n; }
   friend auto operator-(iterator const x, iterator const y) -> difference_type { ... }
};

static_assert(std::random_access_iterator<simple_iota_view<int>::iterator>);
```

Modelling `std::random_access_iterator<I>`

```
1  template<std::integral I>
2  class simple_iota_view<I>::iterator {
3  public:
4     using value_type = I;
5     using difference_type = std::iter_difference_t<I>;
6     using iterator_category = std::random_access_iterator_tag;
7
8     // stuff from previous slides
9
10    auto operator<=>(iterator) const -> std::strong_ordering = default;
11
12    auto operator+=(difference_type const n) -> iterator& {
13       current_ += n;
14       return *this;
15    }
16
17    auto operator-=(difference_type const n) -> iterator& { return *this += -n; }
18    auto operator[](difference_type const n) const -> value_type { return *(*this + n); }
19
20    friend auto operator+(iterator j, difference_type const n) -> iterator { return j += n; }
21    friend auto operator+(difference_type const n, iterator j) -> iterator { return j += n; }
22    friend auto operator-(iterator j, difference_type const n) -> iterator { return j -= n; }
23    friend auto operator-(iterator const x, iterator const y) -> difference_type { ... }
24 };
25
26 static_assert(std::random_access_iterator<simple_iota_view<int>::iterator>);
```

Modelling `std::random_access_iterator<I>`

```cpp
template<std::integral I>
class simple_iota_view<I>::iterator {
public:
   using value_type = I;
   using difference_type = std::iter_difference_t<I>;
   using iterator_category = std::random_access_iterator_tag;

   // stuff from previous slides

   auto operator<=>(iterator) const -> std::strong_ordering = default;

   auto operator+=(difference_type const n) -> iterator& {
      current_ += n;
      return *this;
   }

   auto operator-=(difference_type const n) -> iterator& { return *this += -n; }
   auto operator[](difference_type const n) const -> value_type { return *(*this + n); }

   friend auto operator+(iterator j, difference_type const n) -> iterator { return j += n; }
   friend auto operator+(difference_type const n, iterator j) -> iterator { return j += n; }
   friend auto operator-(iterator j, difference_type const n) -> iterator { return j -= n; }
   friend auto operator-(iterator const x, iterator const y) -> difference_type { ... }
};

static_assert(std::random_access_iterator<simple_iota_view<int>::iterator>);
```

tl;dr

`std::bidirectional_iterator<I>`

(See previous tl;dr)

`std::sized_sentinel_for<I, I>`

Requires:

`std::sentinel_for<I, I>`

`std::disable_sentinel_for<I, I> == false`

`s - i` returns the number of elements between `i` and `s` in `O(1)` time

`i - s` is equivalent to `s - i`

`std::random_access_iterator<I>`

Requires:

`std::totally_ordered<I>` (i.e. has all traditional comparison operators)

`I::iterator_category` is derived from `std::random_access_iterator_tag`

`i += n` and `i -= n` advance/decline `i` by `n` elements in `O(1)` time

`j + n`, `n + j` and `j - n` advance/decline a copy of `j` by `n` elements in `O(1)` time

`j[n]` is equivalent to `*(j + n)`

What happens if we want to write to a range?

```
template<std::input_iterator I, std::sentinel_for<I> S,
         ??? O>
auto copy(I first, S last, O result) -> copy_result<I, O> {
   for (; first != last; ++first, (void)++result) {
      *result = *first;
   }
   return {std::move(first), std::move(result)};
}
```

What do `O` and `I` need to model?

```
template<typename T,
         ??? O,
         std::sentinel_for<O> S>
auto fill(O first, S last, T const& value) -> O {
   for (first != last) {
      *first++ = value;
   }
   return first;
}
```

What does `O` need to model?

Writable iterators

Operation	Array-like	Node-based	Iterator
Iteration type	`gsl_lite::index`	`node*`	*`unspecified`*
Write to element	`v[i] = x`	`i->value = x`	`*i = x`
Successor	`j = i + n < ranges::distance(v) ? i + n : ranges::distance(v);`	`j = i->successor(n)`	`ranges::next(i, s, n)`
Advance fwd	`++i`	`i = i->next`	`++i`
Comparison	`i < ranges::distance(v)`	`i != nullptr`	`i != s`

Indirectly writable

The concept `indirectly_writable` determines if we can write a value to whatever the iterator is referencing.

Let `o` be a possibly-constant object of type `O` and `val` be an object of type `T`. A type `O` models the concept `std::indirectly_writable<T>` if:

1. `*o = val` is possible regardless of whether `O` is `const`-qualified
2. These expressions are possible for `O&` and `O&&`:
   - `*o = std::move(val)`
   - `*std::move(o) = std::move(val)`
3. All forms of `*o = val` result in `*o` returning a reference.

Modelling `std::indirectly_writable<O, T>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type& { return pointee_->data; }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> iterator { ... }
  auto operator--() -> iterator& { ... }
  auto operator--(int) -> iterator { ... }

  auto operator==(iterator) const -> bool = default;
private:
  //...
};

static_assert(std::indirectly_writable<linked_list<int>::iterator, int>);
```

Applying this to copy

```
1 template<std::input_iterator I, std::sentinel_for<I> S,
2          std::weakly_incrementable O>
3 requires std::indirectly_copyable<I, O>
4 auto copy(I first, S last, O result) -> copy_result<I, O> {
5    for (; first != last; ++first, (void)++result) {
6       *result = *first;
7    }
8    return {std::move(first), std::move(result)};
9 }
```

Applying this to copy

```
template<std::input_iterator I, std::sentinel_for<I> S,
         std::weakly_incrementable O>
requires std::indirectly_copyable<I, O>
auto copy(I first, S last, O result) -> copy_result<I, O> {
   for (; first != last; ++first, (void)++result) {
      *result = *first;
   }
   return {std::move(first), std::move(result)};
}
```

Where `std::indirectly_copyable<I, O>` means "we can copy the value we've read from `I` into `O`". In other words:

```
template<typename In, typename Out>
concept indirectly_copyable =
   std::indirectly_readable<In> and
   std::indirectly_writable<Out, std::iter_reference_t<In>>;
```

Output iterators

The concept `output_iterator` refines the base iterator concept by requiring the iterator models `indirectly_writable`.

Let `o` be a possibly-constant object of type `O` and `val` be an object of type `T`. A type `O` models the concept `std::output_iterator<T>` if:

1. `O` models `std::input_or_output_iterator`
2. `O` models `std::indirectly_writable<T>`
3. - `*(o++) = val` is valid, if `T` is an lvalue, and is equivalent to

```
*o = val;
++o;
```

   - `*(o++) = std::move(val)` is valid, if `T` is an rvalue, and is equivalent to

```
*o = std::move(val);
++o;
```

Modelling `std::output_iterator<O, T>`

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  // ...
private:
  //...
};

static_assert(std::bidirectional_iterator<linked_list<int>::iterator>);
static_assert(std::output_iterator<linked_list<int>::iterator, int>);
```

Iterators that model both `std::bidirectional_iterator<I>` and `std::indirectly_writable<O, T>` model `std::output_iterator<O, T>` by default.

There isn't a concept for these kinds of iterators, but they're known as *mutable* iterators in generic programming theory.

Applying this to fill

```
1 template<typename T,
2           std::output_iterator<T const&> O,
3           std::sentinel_for<O> S>
4 auto fill(O first, S last, T const& value) -> O {
5     while (first != last)
6         *first++ = value;
7     }
8     return first;
9 }
```

Non-mutable output iterators are write-once-then-advance.

Iterators and `const`

Mutable iterators are always `indirectly_writable`, even when `const`-qualified.

```
void write_access(linked_list<int>::iterator const i) {
    *i = 42; // okay, changes what *i refers to
    i = {}; // error: can't change i itself
}
```

This is the same problem as `T* const`: we have a constant iterator, not an iterator pointing-to-`const`.

const_iterators

A `const_iterator` is the iterator-equivalent of `T const*`.

```
void no_write_access(linked_list<int>::const_iterator i) {
    i = {}; // okay

    *i = 42; // error: can't indirectly write to i
}
```

How do we make `const_iterator`s?

Attempt 1: Two types

```cpp
template<typename T>
class linked_list<T>::iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type& {
     return pointee_->data;
  }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> iterator { ... }
  auto operator--() -> iterator& { ... }
  auto operator--(int) -> iterator { ... }

  auto operator==(iterator) const -> bool = default;
private:
  //...
};

static_assert(
   std::indirectly_writable<linked_list<int>::iterator, int>);
```

```cpp
template<typename T>
class linked_list<T>::const_iterator {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator() = default;

  auto operator*() const noexcept -> value_type const& {
     return pointee_->data;
  }

  auto operator++() -> iterator& { ... }
  auto operator++(int) -> iterator { ... }
  auto operator--() -> iterator& { ... }
  auto operator--(int) -> iterator { ... }

  auto operator==(iterator) const -> bool = default;
private:
  //...
};

static_assert(
   not std::indirectly_writable<linked_list<int>::const_iterator,
                                int>);
```

Why might this be a bad approach?

Attempt 2: Parameterise on constness

```
template<typename T>
class linked_list {
  template<bool is_const>
  class iterator_impl;
public:
  using iterator = iterator_impl<false>;
  using const_iterator = iterator_impl<true>;

  // ...
};
```

We define one template class `iterator_impl` and *parameterise* it based on constness.

Then we add metadata to `iterator_impl` to help it understand what `is_const` means.

Attempt 2: Parameterise on constness

```cpp
template<typename T>
template<bool is_const>
class linked_list<T>::iterator_impl {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator_impl() = default;

  auto operator*() const noexcept -> value_type const&
  requires is_const {
     return pointee_->data;
  }

  auto operator*() const noexcept -> value_type&
  requires (not is_const) {
     return pointee_->data;
  }

  auto operator++() -> iterator_impl& { ... }
  auto operator++(int) -> iterator_impl { ... }
  auto operator--() -> iterator_impl& { ... }
  auto operator--(int) -> iterator_impl { ... }
  auto operator==(iterator_impl) const -> bool = default;
private:
  maybe_const_t<is_const, T>* pointee_;

  iterator_impl(maybe_const_t<is_const, T>* ptr) { ... }
  friend class linked_list<T>;
};
```

Attempt 2: Parameterise on constness

```cpp
template<typename T>
template<bool is_const>
class linked_list<T>::iterator_impl {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator_impl() = default;

  auto operator*() const noexcept -> value_type const&
  requires is_const {
     return pointee_->data;
  }

  auto operator*() const noexcept -> value_type&
  requires (not is_const) {
     return pointee_->data;
  }

  auto operator++() -> iterator_impl& { ... }
  auto operator++(int) -> iterator_impl { ... }
  auto operator--() -> iterator_impl& { ... }
  auto operator--(int) -> iterator_impl { ... }
  auto operator==(iterator_impl) const -> bool = default;
private:
  maybe_const_t<is_const, T>* pointee_;

  iterator_impl(maybe_const_t<is_const, T>* ptr) { ... }
  friend class linked_list<T>;
};
```

```cpp
template<bool is_const, typename T>
struct maybe_const {
  using type = T;
};

template<typename T>
struct maybe_const<true, T> {
  using type = T const;
};

template<bool is_const, typename T>
using maybe_const_t =
   typename maybe_const<is_const, T>;
```

This is partial template specialisation, which we cover in Week 8. For now, you can think of it like a switch on constness, based on some compile-time property.

Attempt 2: Parameterise on constness

```cpp
template<typename T>
template<bool is_const>
class linked_list<T>::iterator_impl {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator_impl() = default;

  auto operator*() const noexcept -> value_type const&
  requires is_const {
     return pointee_->data;
  }

  auto operator*() const noexcept -> value_type&
  requires (not is_const) {
     return pointee_->data;
  }

  auto operator++() -> iterator_impl& { ... }
  auto operator++(int) -> iterator_impl { ... }
  auto operator--() -> iterator_impl& { ... }
  auto operator--(int) -> iterator_impl { ... }
  auto operator==(iterator_impl) const -> bool = default;
private:
  maybe_const_t<is_const, T>* pointee_;

  iterator_impl(maybe_const_t<is_const, T>* ptr) { ... }
  friend class linked_list<T>;
};
```

```cpp
template<bool is_const, typename T>
struct maybe_const {
  using type = T;
};

template<typename T>
struct maybe_const<true, T> {
  using type = T const;
};

template<bool is_const, typename T>
using maybe_const_t =
   typename maybe_const<is_const, T>;
```

This is partial template specialisation, which we cover in Week 8. For now, you can think of it like a switch on constness, based on some compile-time property.

Attempt 2: Parameterise on constness

```cpp
template<typename T>
template<bool is_const>
class linked_list<T>::iterator_impl {
public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  iterator_impl() = default;

  auto operator*() const noexcept -> value_type const&
  requires is_const {
     return pointee_->data;
  }

  auto operator*() const noexcept -> value_type&
  requires (not is_const) {
     return pointee_->data;
  }

  auto operator++() -> iterator_impl& { ... }
  auto operator++(int) -> iterator_impl { ... }
  auto operator--() -> iterator_impl& { ... }
  auto operator--(int) -> iterator_impl { ... }
  auto operator==(iterator_impl) const -> bool = default;
private:
  maybe_const_t<is_const, T>* pointee_;

  iterator_impl(maybe_const_t<is_const, T>* ptr) { ... }
  friend class linked_list<T>;
};
```

```cpp
template<bool is_const, typename T>
struct maybe_const {
  using type = T;
};

template<typename T>
struct maybe_const<true, T> {
  using type = T const;
};

template<bool is_const, typename T>
using maybe_const_t =
   typename maybe_const<is_const, T>;
```

This is partial template specialisation, which we cover in Week 8. For now, you can think of it like a switch on constness, based on some compile-time property.

Getting the right kind of iterator

Either way, you'll need to have two overloads for `begin` and for `end`. We don't want `linked_list<int> const` to return mutable iterators!

```
template<typename T>
class linked_list {
  template<bool is_const>
  class iterator_impl;
public:
  using iterator = iterator_impl<false>;
  using const_iterator = iterator_impl<true>;

  auto begin() -> iterator { return iterator(head_); }
  auto end() -> iterator { return iterator(tail_); }

  auto begin() const -> const_iterator { return iterator(head_); }
  auto end() const -> const_iterator { return iterator(tail_); }
};
```

How might you improve this code?

Getting the right kind of iterator

Either way, you'll need to have two overloads for `begin` and for `end`. We don't want `linked_list<int> const` to return mutable iterators!

```
template<typename T>
class linked_list {
  template<bool is_const>
  class iterator_impl;
public:
  using iterator = iterator_impl<false>;
  using const_iterator = iterator_impl<true>;

  auto begin() -> iterator { return iterator(head_); }
  auto end() -> iterator { return iterator(tail_); }

  auto begin() const -> const_iterator { return iterator(head_); }
  auto end() const -> const_iterator { return iterator(tail_); }
};
```

How might you improve this code?

Getting the right kind of iterator DRY style

```
template<typename T>
class linked_list {
public:
  using iterator = iterator_impl<false>;
  using const_iterator = iterator_impl<true>;

  auto begin() -> iterator { begin_impl(*this); }
  auto end() -> iterator { return end_impl(*this); }

  auto begin() const -> const_iterator { return begin_impl(*this); }
  auto end() const -> const_iterator { return end_impl(*this); }
private:
  template<typename T>
  static auto begin_impl(T& t) -> decltype(t.begin()) {
    return iterator_impl<std::is_const_v<T>>(head_);
  }

  template<typename T>
  static auto end_impl(T& t) -> decltype(t.end()) {
    return iterator_impl<std::is_const_v<T>>(tail_);
  }
};
```

Getting the right kind of iterator DRY style

```cpp
template<typename T>
class linked_list {
public:
  using iterator = iterator_impl<false>;
  using const_iterator = iterator_impl<true>;

  auto begin() -> iterator { begin_impl(*this); }
  auto end() -> iterator { return end_impl(*this); }

  auto begin() const -> const_iterator { return begin_impl(*this); }
  auto end() const -> const_iterator { return end_impl(*this); }
private:
  template<typename T>
  static auto begin_impl(T& t) -> decltype(t.begin()) {
    return iterator_impl<std::is_const_v<T>>(head_);
  }

  template<typename T>
  static auto end_impl(T& t) -> decltype(t.end()) {
    return iterator_impl<std::is_const_v<T>>(tail_);
  }
};
```

Getting the right kind of iterator DRY style

```
template<typename T>
class linked_list {
public:
  using iterator = iterator_impl<false>;
  using const_iterator = iterator_impl<true>;

  auto begin() -> iterator { begin_impl(*this); }
  auto end() -> iterator { return end_impl(*this); }

  auto begin() const -> const_iterator { return begin_impl(*this); }
  auto end() const -> const_iterator { return end_impl(*this); }
private:
  template<typename T>
  static auto begin_impl(T& t) -> decltype(t.begin()) {
    return iterator_impl<std::is_const_v<T>>(head_);
  }

  template<typename T>
  static auto end_impl(T& t) -> decltype(t.end()) {
    return iterator_impl<std::is_const_v<T>>(tail_);
  }
};
```

When to have differently-qualified overloads

Is the iterator `indirectly_writable`?	Then you should use
Yes	`begin()` `end()`
No	`begin() const` `end() const`
No, but the range does housekeeping (e.g. keeps a cache)	`begin()` `end()`

What about `cbegin` and `cend`?

Standard containers ship with `cbegin` and `cend` as member functions.

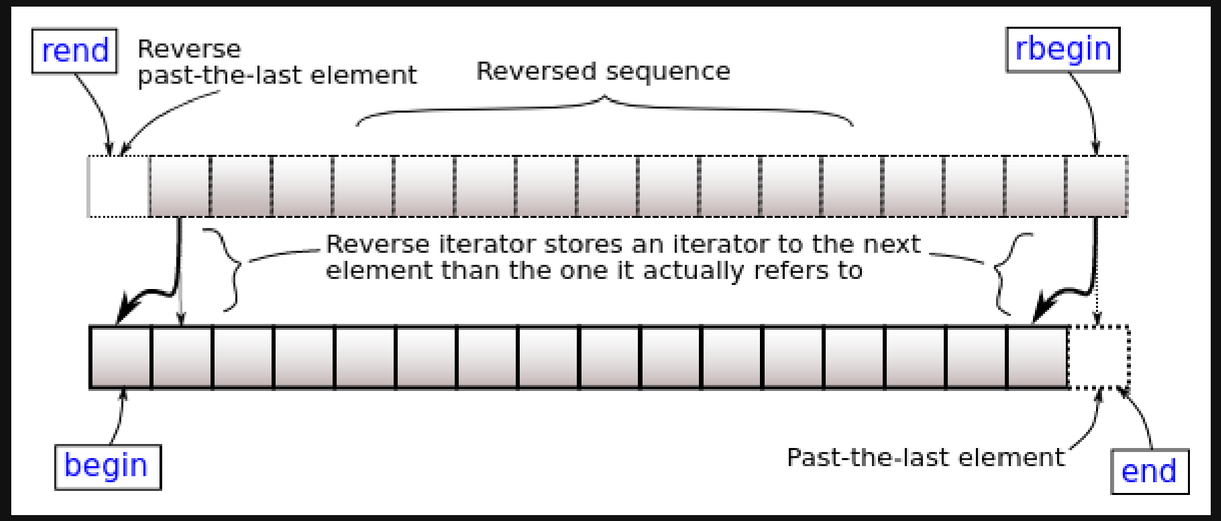
While they *might* be handy, they're not *necessary*.

`ranges::cbegin(r)` and `ranges::cend(r)` will Do The Right Thing™ if you have a const-qualified begin member function. (Try it on our `linked_list` and see if it works!)

Not all ranges will have `cbegin`/`cend` member functions, so the only reliable way to get an iterator to a constant range is to use one of the above.

`rbegin` and `rend`

`rbegin` and `rend` idiomatically return `reverse_iterator<iterator>`s.



Unlike `cbegin`/`cend`, you'll probably want to define `rbegin`/`rend` as members for backwards-compatibility reasons.

`crbegin`/`crend` are in the same category as `cbegin`/`cend`.

Video Resources

pacific++

Generic Programming, by Sean Parent

cppcon the c++ conference

- *Concepts in 60: Everything You Need to Know and Nothing You Don't*, by Andrew Sutton
- *Concepts: A Day in the Life*, by Saar Raz
- *An Overview of Standard Ranges*, by Tristan Brindle
- *Algorithm Intuition*, by Conor Hoekstra
- *Range Algorithms, Views and Actions: A Comprehensive Guide*, by Dvir Yitzchacki
- *From STL to Ranges: Using Ranges Effectively*, by Jeff Garland
- *What a View! Building Your own (Lazy) Range Adaptors*, by Christopher Di Bella

Written Resources

From Mathematics to Generic Programming, by Alexander Stepanov and Daniel Rose

Elements of Programming, by Alexander Stepanov and Paul McJones