Iterators and Ranges: Comparing C++ to D to Rust

Barry Revzin

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

Iterators and Ranges

Exploring Library Designs for Traversal

Barry Revzin

About Me

- C++ Software Developer at Jump Trading since 2014

About Me

- C++ Software Developer at Jump Trading since 2014
- WG21 participant since 2016
  - `<=>`, `[...args=args]{}`, `explicit(bool)`, conditionally trivial
  - Deducing this, `if consteval`
  - Bunch of ranges papers

About Me

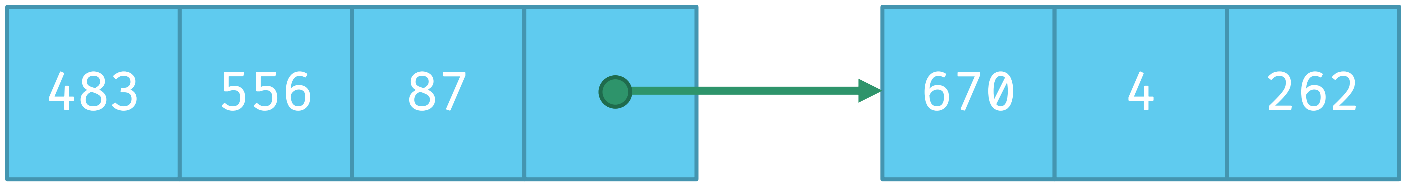
- C++ Software Developer at Jump Trading since 2014
- WG21 participant since 2016
  - `<=>`, `[...args=args]{}`, `explicit(bool)`, conditionally trivial
  - Deducing this, `if consteval`
  - Bunch of ranges papers

https://brevzin.github.io/ @BarryRevzin Barry

We have some sequence…

483	556	87	670	4	262

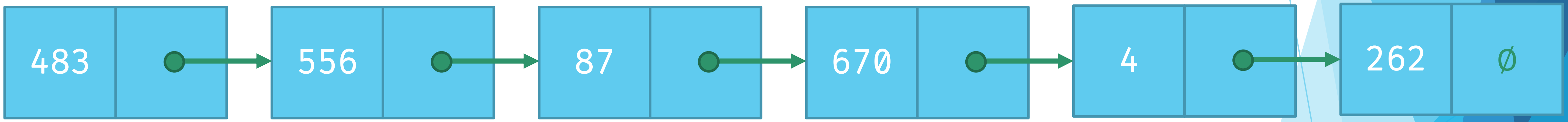
We have some sequence…



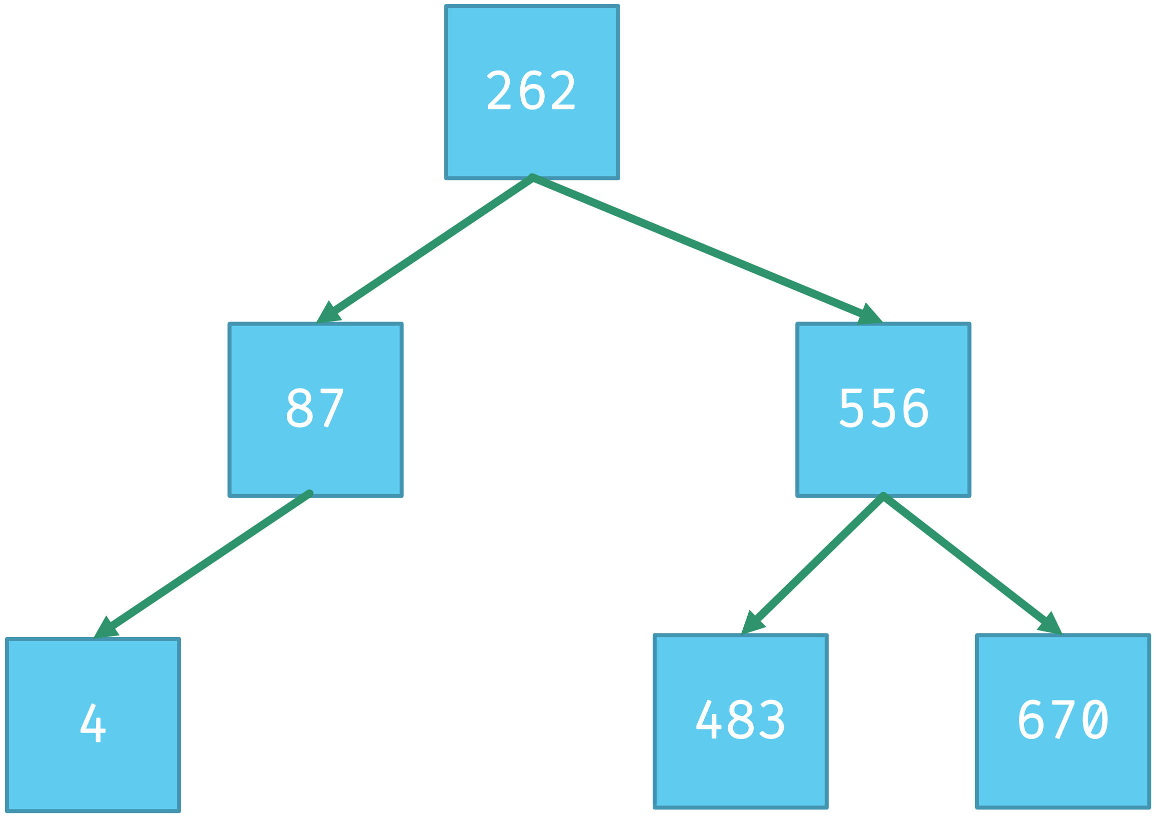
We have some sequence…

483	556	87	∅	670	∅	4	262

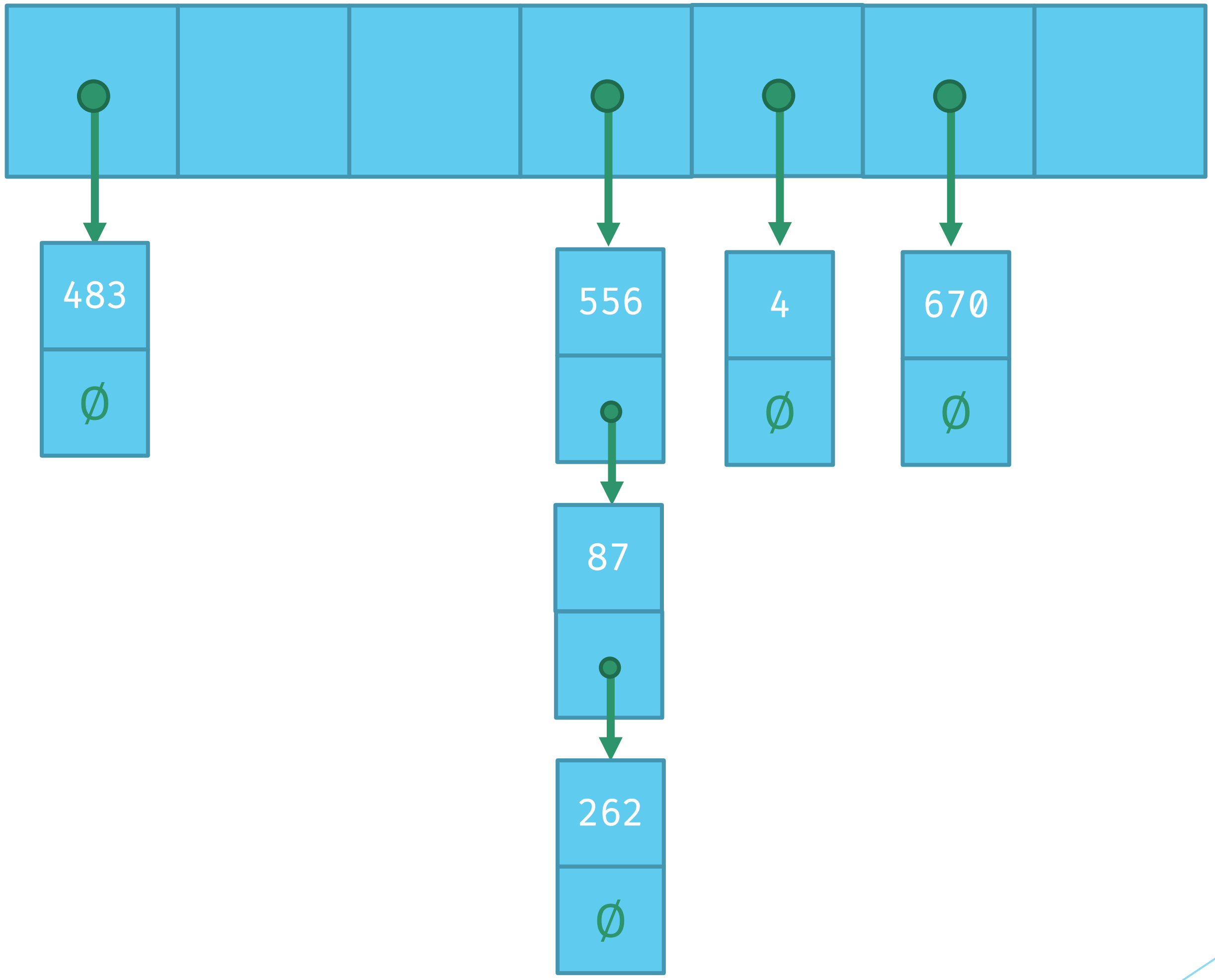
We have some sequence…



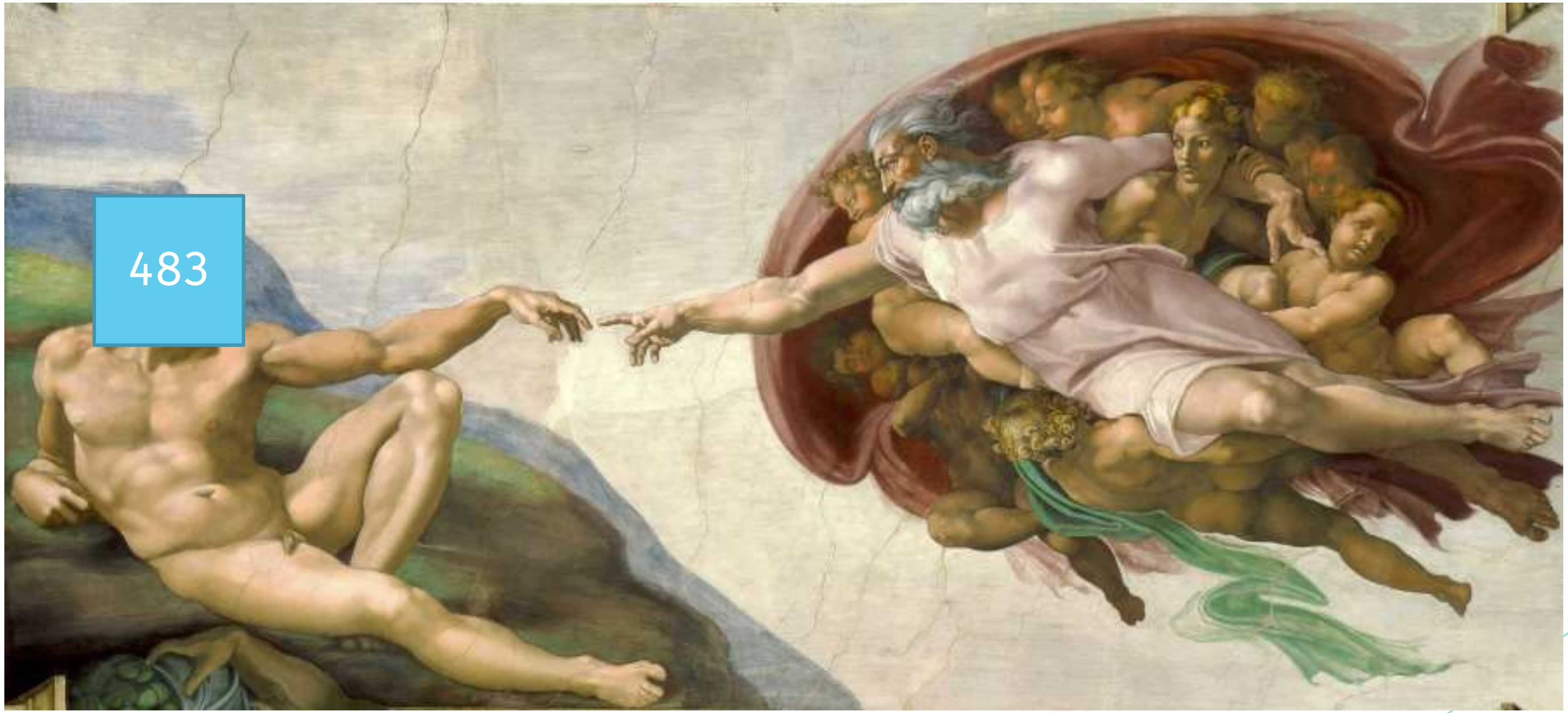
We have some sequence…



We have some sequence…



We have some sequence…



Need some uniform access pattern

Basis Operations

- `read`
- `advance`
- `done?`

Basis Operations

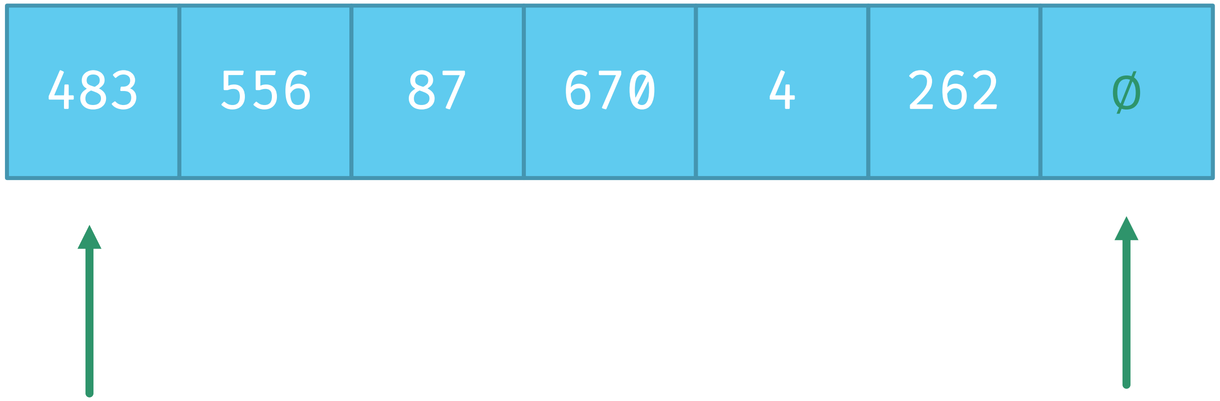
- `read`
- `advance`
- `done?`

Not necessarily three distinct functions!

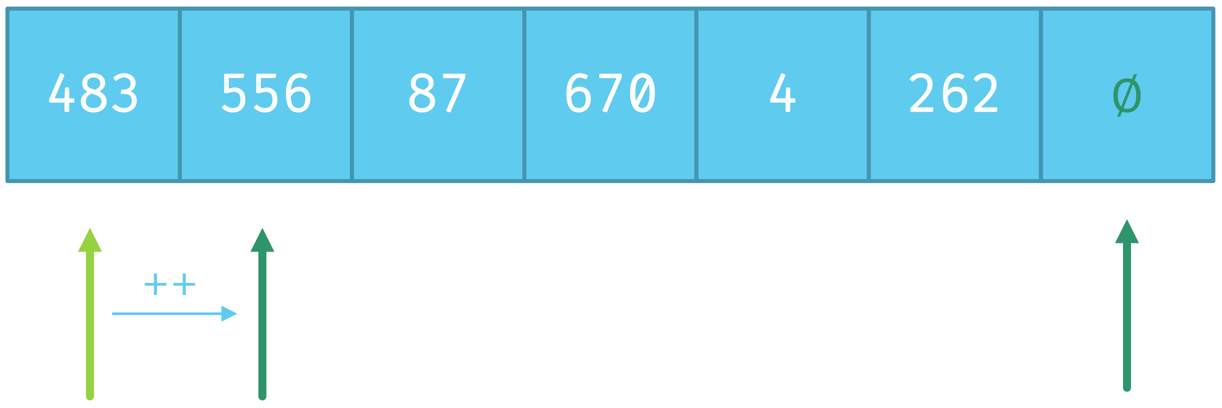
The C++ Iterator Pair Model

From Stepanov with Love

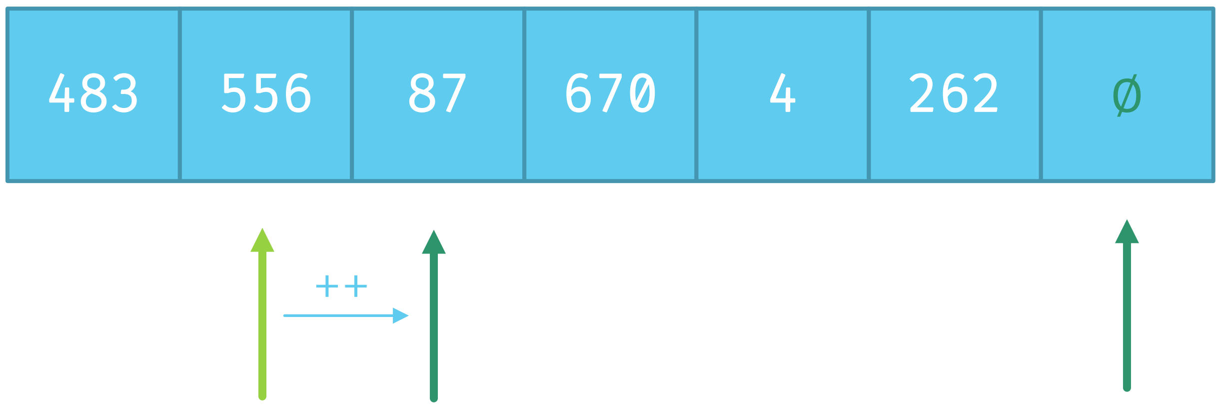
The C++ Iterator Pair Model



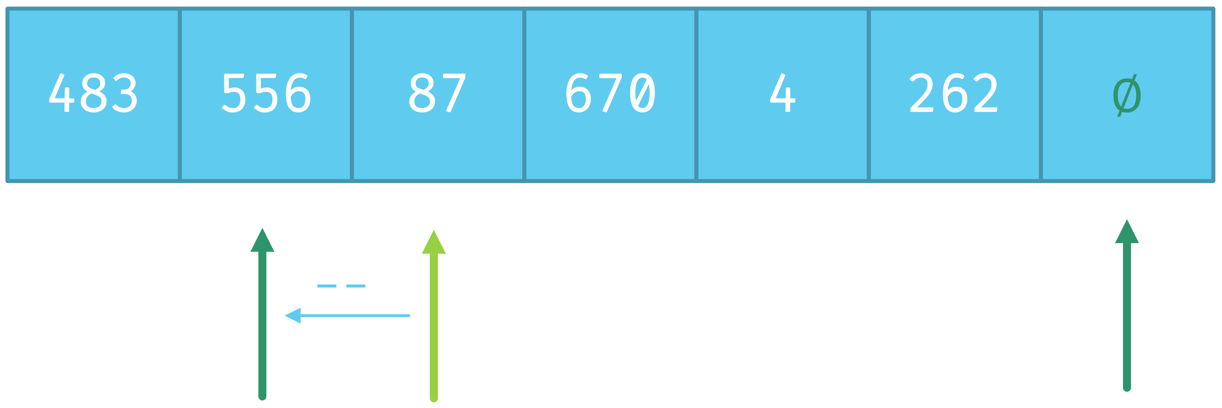
The C++ Iterator Pair Model



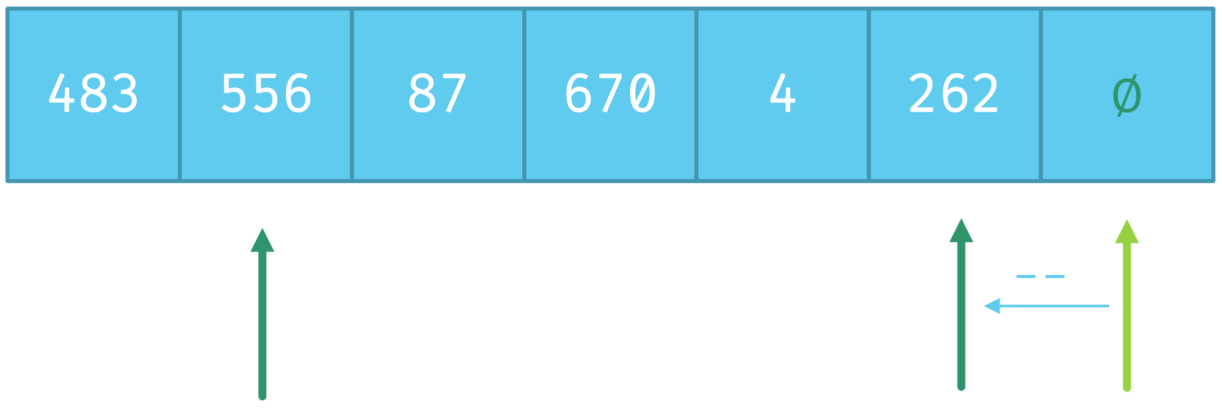
The C++ Iterator Pair Model



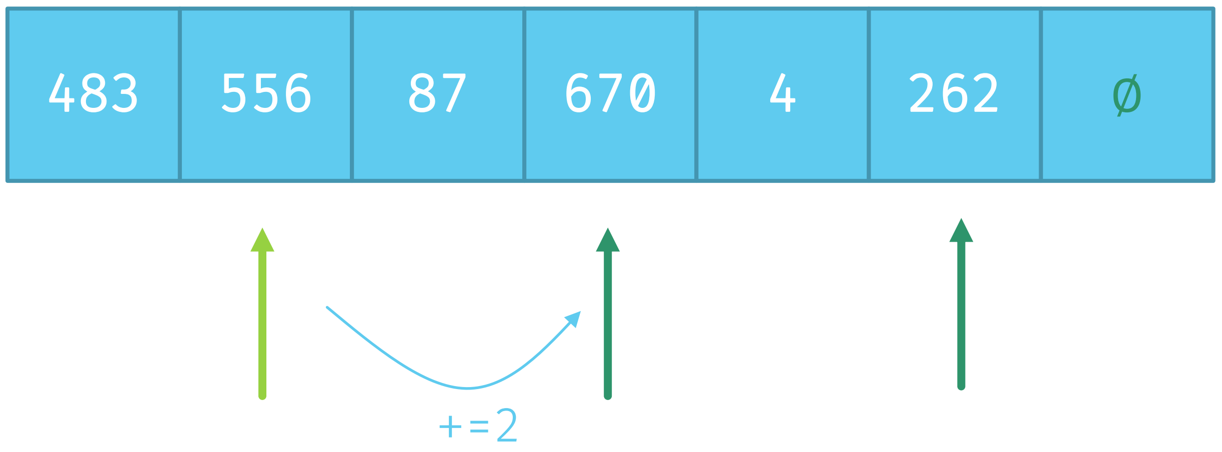
The C++ Iterator Pair Model



The C++ Iterator Pair Model



The C++ Iterator Pair Model



The C++ Iterator Pair Model

	read	advance	done?
	`*it`	`++it`	`it == last`

Basic C++ Range Structure

```
struct Range {
    struct Iterator {
        using reference         = /* ... */;
        using value_type        = /* ... */;
        using iterator_category = /* ... */;
        using iterator_concept  = /* ... */;
        using difference_type   = /* ... */;

        Iterator();
        auto operator*() const -> reference;
        auto operator++() -> Iterator&;
        auto operator++(int) -> Iterator;
        auto operator==(Iterator const&) const -> bool;
    };

    struct Sentinel {
        Sentinel();
        auto operator==(Iterator const&) const -> bool;
    };

    auto begin() -> Iterator;
    auto end()   -> Sentinel;
};
```

consumed

read

advance

done?

Not consumed

Basic C++ Range Structure

```
1 struct Range {
  2 struct Iterator {
        using reference         = /* ... */;
        using value_type        = /* ... */;
        using iterator_category = /* ... */;
        using iterator_concept  = /* ... */;
        using difference_type   = /* ... */;

      1 Iterator();
      2 auto operator*() const -> reference;
      3 auto operator++() -> Iterator&;
      4 auto operator++(int) -> Iterator;
      5 auto operator==(Iterator const&) const -> bool;
    };

  3 struct Sentinel {
      6 Sentinel();
      7 auto operator==(Iterator const&) const -> bool;
    };

  8 auto begin() -> Iterator;
  9 auto end()   -> Sentinel;
};
```

Basic C++ Range Structure

```
template <range R>
void print_all(R&& r) {
    auto it   = ranges::begin(r);
    auto last = ranges::end(r);

    for (; it != last; ++it) {
        fmt::print("{}\n", *it);
    }
}
```

advance

done?

read

Adapting Ranges in C++

`transform` and `filter`

Implementing `transform` in C++

```
template <input_range V, copy_constructible F>
    requires view<V> &&
             regular_invocable<F&, range_reference_t<V>>
class transform_view {
    struct Iterator;
    struct Sentinel;

    V base_;
    F fun_;

public:
    transform_view(V, F);

    auto begin() -> Iterator;
    auto end()   -> Sentinel;
};
```

Implementing map in C++ (transform)

```
template <input_range V, copy_constructible F>
    requires view<V> &&
             regular_invocable<F&, range_reference_t<V>>
class map_view {
    struct Iterator;
    struct Sentinel;

    V base_;
    F fun_;

public:
    map_view(V, F);

    auto begin() -> Iterator;
    auto end()   -> Sentinel;
};
```

Implementing map in C++ (transform)

```
template <input_range V, copy_constructible F>
    requires view<V> &&
             regular_invocable<F&, range_reference_t<V>>
class map_view {
    struct Iterator;
    struct Sentinel;

    V base_;
    F fun_;

public:
    map_view(V, F);

    auto begin() -> Iterator;
    auto end()   -> Sentinel;
    auto end()   -> Iterator requires common_range<V>;
};
```

Implementing `map` in C++ (`transform`)

```cpp
template <input_range V, copy_constructible F>
    requires view<V> &&
             regular_invocable<F&, range_reference_t<V>>
class map_view {
    template <bool IsConst> struct Iterator;
    template <bool IsConst> struct Sentinel;

    V base_;
    F fun_;

public:
    map_view(V, F);

    auto begin() -> Iterator<false>;
    auto end()   -> Sentinel<false>;
    auto end()   -> Iterator<false> requires common_range<V>;

    auto begin() const -> Iterator<true> requires range<V const>;
    auto end()   const -> Sentinel<true> requires range<V const>;
    auto end()   const -> Iterator<true> requires common_range<V const>;
};
```

Implementing `map` in C++ (`transform`)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {

};
```

Implementing map in C++ (transform)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {
    using Parent = maybe_const<IsConst, map_view>;
    using Base = maybe_const<IsConst, V>;
    iterator_t<Base> base_ = iterator_t<Base>();
    Parent* parent_ = nullptr;
};
```

Implementing map in C++ (transform)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {
    using Parent = maybe_const<IsConst, map_view>;
    using Base = maybe_const<IsConst, V>;
    iterator_t<Base> base_ = iterator_t<Base>();
    Parent* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<Base>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<Base>;

    Iterator() = default;
};
```

Implementing map in C++ (transform)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {
    using Parent = maybe_const<IsConst, map_view>;
    using Base = maybe_const<IsConst, V>;
    iterator_t<Base> base_ = iterator_t<Base>();
    Parent* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<Base>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<Base>;

    Iterator() = default;
    Iterator(Parent&, iterator_t<Base>);
    Iterator(Iterator<not IsConst>)
        requires IsConst and convertible_to<iterator_t<V>, iterator_t<Base>>;
};
```

Implementing `map` in C++ (`transform`)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {
    using Parent = maybe_const<IsConst, map_view>;
    using Base = maybe_const<IsConst, V>;
    iterator_t<Base> base_ = iterator_t<Base>();
    Parent* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<Base>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<Base>;

    Iterator() = default;
    Iterator(Parent&, iterator_t<Base>);
    Iterator(Iterator<not IsConst>) requires IsConst and convertible_to<iterator_t<V>, iterator_t<Base>>;

    auto operator*() const -> reference {
        return invoke(parent_->fun_, *base_);
    }

    auto operator++() -> Iterator& {
        ++base_;
        return *this;
    }
    auto operator++() -> Iterator {
        auto tmp = *this;
        ++*this;
        return tmp;
    }

    auto operator==(Iterator const& rhs) const -> bool {
        return base_ == rhs.base_;
    }
};
```

read

advance

done?

Implementing `map` in C++ (`transform`)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {
    using Parent = maybe_const<IsConst, map_view>;
    using Base = maybe_const<IsConst, V>;
    iterator_t<Base> base_ = iterator_t<Base>();
    Parent* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<Base>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<Base>;

    Iterator() = default;
    Iterator(Parent&, iterator_t<Base>);
    Iterator(Iterator<not IsConst>) requires IsConst and convertible_to<iterator_t<V>, iterator_t<Base>>;

    auto operator*() const -> reference { return invoke(parent_->fun_, *base_); }

    auto operator++() -> Iterator& { ++base_; return *this; }
    auto operator++() -> Iterator  { auto tmp = *this; ++*this; return tmp; }

    auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
};
```

Implementing `map` in C++ (`transform`)

```
template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Iterator<IsConst> {
    using Parent = maybe_const<IsConst, map_view>;
    using Base = maybe_const<IsConst, V>;
    iterator_t<Base> base_ = iterator_t<Base>();
    Parent* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<Base>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<Base>;

    Iterator() = default;
    Iterator(Parent&, iterator_t<Base>);
    Iterator(Iterator<not IsConst>) requires IsConst and convertible_to<iterator_t<V>, iterator_t<Base>>;

    auto operator*() const -> reference { return invoke(parent_->fun_, *base_); }

    auto operator++() -> Iterator& { ++base_; return *this; }
    auto operator++() -> Iterator  { auto tmp = *this; ++*this; return tmp; }

    auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
};

template <input_range V, copy_constructible F>
template <bool IsConst>
class map_view<V, F>::Sentinel<IsConst> {
    using Base = maybe_const<IsConst, V>;
    sentinel_t<Base> end_ = sentinel_t<Base>();

public:
    sentinel() = default;
    sentinel(sentinel_t<Base>);
    sentinel(sentinel<not Const>) requires IsConst and convertible_to<sentinel_t<V>, sentinel_t<Base>>;

    template <bool OtherConst> requires sentinel_for<sentinel_t<Base>, iterator_t<maybe_const<OtherConst, V>>>
    auto operator==(Iterator<OtherConst> const& it) const -> bool { return it.base_ == end_; }
};
```

Implementing `map` in C++ (`transform`)

```
template <input_range V, copy_constructible F> requires view<V> && regular_invocable<F&, range_reference_t<V>>
class map_view {
    template <bool IsConst> struct Iterator;
    template <bool IsConst> struct Sentinel;

    template <bool IsConst>
    class Iterator<IsConst> {
        using Fun = semiregular_box<F>;
        using Base = maybe_const<IsConst, V>;
        iterator_t<Base> base_ = iterator_t<Base>();
        Fun fun = Fun();

    public:
        using iterator_concept = /* ... */;
        using iterator_category = /* ... */;
        using reference = invoke_result_t<F&, range_reference_t<Base>>;
        using value_type = remove_cvref_t<reference>;
        using difference_type = range_difference_t<Base>;

        iterator() = default;
        iterator(Parent&, iterator_t<Base>);
        iterator(iterator<not IsConst>) requires IsConst and convertible_to<iterator_t<V>, iterator_t<Base>>;

        auto operator*() const -> reference { return invoke(parent_->fun_, *base_); }
        auto operator++() -> Iterator& { ++base_; return *this; }
        auto operator++() -> Iterator { auto tmp = *this; ++*this; return tmp; }
        auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
    };

    template <bool IsConst>
    class Sentinel<IsConst> {
        using Base = maybe_const<IsConst, V>;
        sentinel_t<Base> end_ = sentinel_t<Base>();

    public:
        sentinel() = default;
        sentinel(sentinel_t<Base>);
        sentinel(sentinel<not Const>) requires IsConst and convertible_to<sentinel_t<V>, sentinel_t<Base>>;

        template <bool OtherConst> requires sentinel_for<sentinel_t<Base>, iterator_t<maybe_const<OtherConst, V>>>
        auto operator==(Iterator<OtherConst> const& it) const -> bool { return it.base_ == end_; }
    };

    V base_;
    F fun_;

public:
    map_view(V, F);

    auto begin() -> Iterator<false>;
    auto end() -> Sentinel<false>;
    auto end() -> Iterator<false> requires common_range<V>;

    auto begin() const -> Iterator<true> requires range<V const>;
    auto end()   const -> Sentinel<true> requires range<V const>;
    auto end()   const -> Iterator<true> requires common_range<V const>;
};
```

Implementing filter in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
    requires view<V>
class filter_view {
    struct Iterator;
    struct Sentinel;

    V base_;
    F fun_;

public:
    filter_view(V, F);

    auto begin() -> Iterator;
    auto end()   -> Sentinel;
};
```

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
    requires view<V>
class filter_view {
    struct Iterator;
    struct Sentinel;

    V base_;
    F fun_;
    optional<iterator_t<V>> begin_;

public:
    filter_view(V, F);

    auto begin() -> Iterator {
        if (not begin_) {
            begin_ = find_if(base_, fun_);
        }
        return Iterator(*this, *begin_);
    }
    auto end()   -> Sentinel;
};
```

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
    requires view<V>
class filter_view {
    struct Iterator;
    struct Sentinel;

    V base_;
    F fun_;
    optional<iterator_t<V>> begin_;

public:
    filter_view(V, F);

    auto begin() -> Iterator {
        if (not begin_) { begin_ = find_if(base_, fun_); }
        return Iterator(*this, *begin_);
    }
    auto end()   -> Sentinel;
    auto end()   -> Iterator requires common_range<V>;
};
```

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
class filter_view::Iterator {
    iterator_t<V> base_ = iterator_t<V>();
    filter_view* parent_ = nullptr;
};
```

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
class filter_view::Iterator {
    iterator_t<V> base_ = iterator_t<V>();
    filter_view* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = range_reference_t<V>;
    using value_type = range_value_t<V>;
    using difference_type = range_difference_t<V>;

    Iterator() = default;
    Iterator(filter_view&, iterator_t<V>);
};
```

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
class filter_view::Iterator {
    iterator_t<V> base_ = iterator_t<V>();
    filter_view* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = range_reference_t<V>;
    using value_type = range_value_t<V>;
    using difference_type = range_difference_t<V>;

    Iterator() = default;
    Iterator(filter_view&, iterator_t<V>);

    auto operator*() const -> reference {
        return *base_;
    }

    auto operator++() -> Iterator& {
        base_ = find_if(++base_, ranges::end(parent_->base_), parent_->fun_);
        return *this;
    }
    auto operator++(int) -> Iterator {
        auto tmp = *this;
        ++*this;
        return tmp;
    }

    auto operator==(Iterator const& rhs) const -> bool {
        return base_ == rhs.base_;
    }
};
```

read

advance

done?

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
class filter_view::Iterator {
    iterator_t<V> base_ = iterator_t<V>();
    filter_view* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = range_reference_t<V>;
    using value_type = range_value_t<V>;
    using difference_type = range_difference_t<V>;

    Iterator() = default;
    Iterator(filter_view&, iterator_t<V>);

    auto operator*() const -> reference { return *base_; }

    auto operator++() -> Iterator& { base_ = find_if(++base_, ranges::end(parent_->base_), parent_->fun_); return *this; }
    auto operator++(int) -> Iterator { auto tmp = *this; ++*this; return tmp; }

    auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
};
```

Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
class filter_view::Iterator {
    iterator_t<V> base_ = iterator_t<V>();
    filter_view* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = range_reference_t<V>;
    using value_type = range_value_t<V>;
    using difference_type = range_difference_t<V>;

    Iterator() = default;
    Iterator(filter_view&, iterator_t<V>);

    auto operator*() const -> reference { return *base_; }

    auto operator++() -> Iterator& { base_ = find_if(++base_, ranges::end(parent_->base_), parent_->fun_); return *this; }
    auto operator++(int) -> Iterator { auto tmp = *this; ++*this; return tmp; }

    auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
};

template <input_range V, indirect_unary_predicate<iterator_t<V>> F>
class filter_view<V, F>::Sentinel {
    sentinel_t<V> end_ = sentinel_t<V>();
public:
    sentinel() = default;
    sentinel(sentinel_t<V>);
    auto operator==(Iterator const& it) const -> bool {
        return it.base_ == end_;
    }
};
```

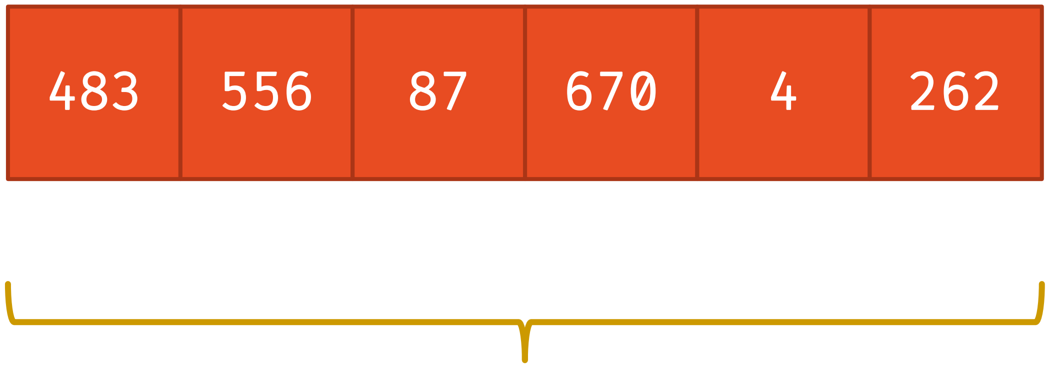
Implementing `filter` in C++

```
template <input_range V, indirect_unary_predicate<iterator_t<V>> F> requires view<V>
class filter_view {
    class Iterator {
        iterator_t<V> base_ = iterator_t<V>();
        filter_view* parent_ = nullptr;
    public:
        using iterator_concept = /* ... */;
        using iterator_category = /* ... */;
        using reference = range_reference_t<V>;
        using value_type = range_value_t<V>;
        using difference_type = range_difference_t<V>;

        Iterator() = default;
        Iterator(filter_view&, iterator_t<V>);

        auto operator*() const -> reference { return *base_; }
        auto operator++() -> Iterator& { base_ = find_if(++base_, ranges::end(parent_->base_), parent_->fun_); return *this; }
        auto operator++(int) -> Iterator { auto tmp = *this; ++*this; return tmp; }
        auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
    };

    class Sentinel {
        sentinel_t<V> end_ = sentinel_t<V>();
    public:
        Sentinel() = default;
        Sentinel(sentinel_t<V>);
        auto operator==(Iterator const& it) const -> bool { return it.base_ == end_; }
    };

    V base_;
    F fun_;
    optional<iterator_t<V>> begin_;

public:
    filter_view(V, F);

    auto begin() -> Iterator {
        if (not begin_) { begin_ = find_if(base_, fun_); }
        return Iterator(*this, *begin_);
    }
    auto end()   -> Sentinel;
    auto end()   -> Iterator requires common_range<V>;
};
```

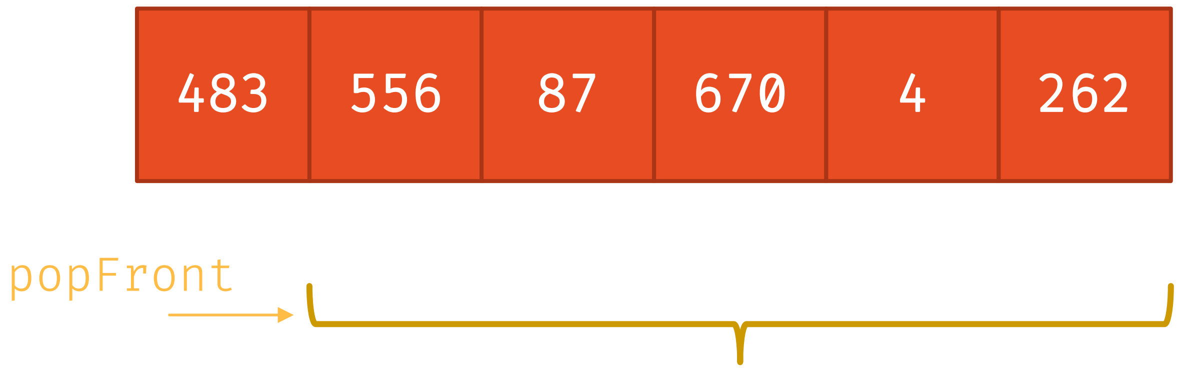
The D Ranges Model

Iterators Must Die

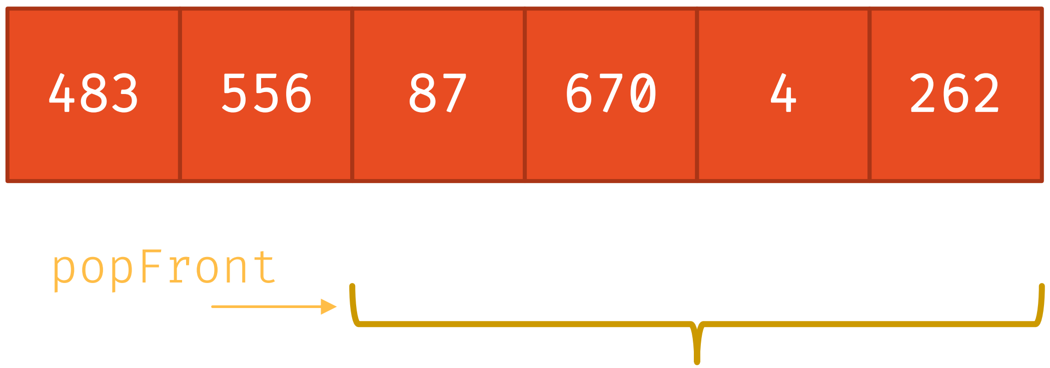
The D Ranges Model



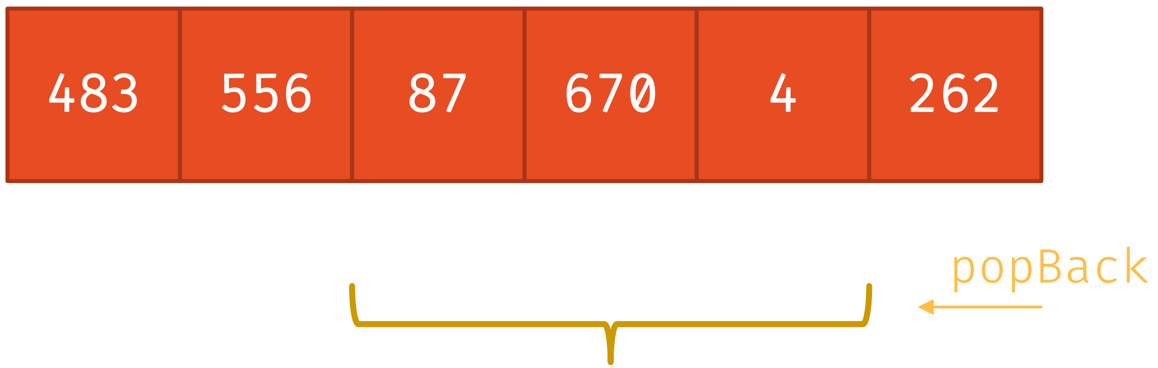
The D Ranges Model



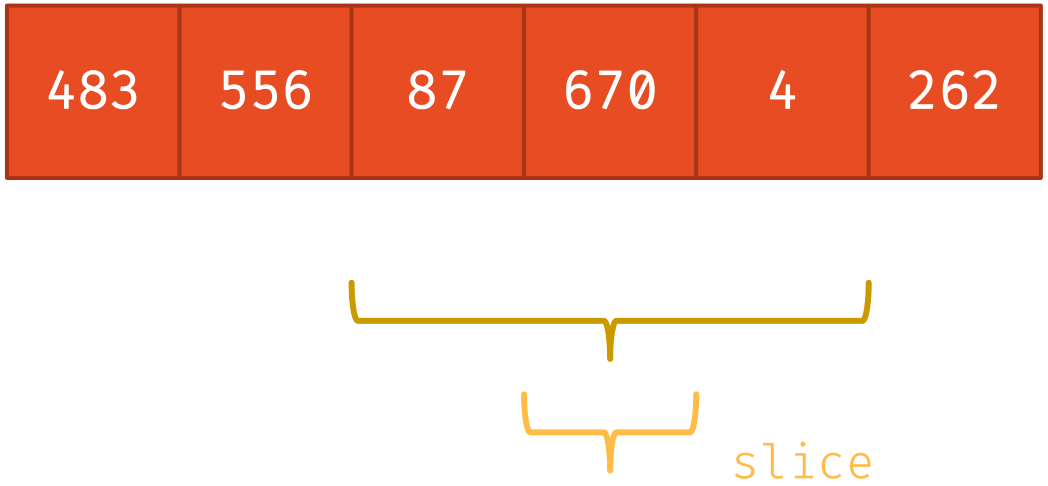
The D Ranges Model



The D Ranges Model



The D Ranges Model



The D Ranges Model

	read	advance	done?
	`*it`	`++it`	`it == last`
	`r.front()`	`r.popFront()`	`r.empty()`

Basic D Range Structure

```
struct Range {
    using reference       = /* ... */;
    using value_type      = /* ... */;
    using range_category  = /* ... */;
    using difference_type = /* ... */;

    auto front() -> reference;
    void popFront();
    auto empty() -> bool;
};
```

consumed

read

advance

done?

Basic D Range Structure

```
template <input_iterator I, sentinel_for<I> S>
class drange {
    I first;
    S last;

public:
    using reference       = iter_reference_t<I>;
    using value_type      = iter_value_t<I>;
    using range_category  = /* ... */;
    using difference_type = iter_difference_t<I>;

    auto front() -> reference { return *first; }
    void popFront() { ++first; }
    auto empty() -> bool { return first == last; }
};
```

Basic D Range Structure

advance

```
template <drange R>
void print_all(R r) {
    for (; not r.empty(); r.popFront()) {
        fmt::print("{}\n", r.front());
    }
}
```

done?

read

Adapting Ranges in D

`map` and `filter`

Implementing map in D

```
template <input_drange R, copy_constructible F>
    requires regular_invocable<F&, range_reference_t<R>>
class map_range {
    R base_;
    F fun_;

public:
    map_range(R, F);
};
```

Implementing map in D

```
template <input_drange R, copy_constructible F>
    requires regular_invocable<F&, range_reference_t<R>>
class map_range {
    R base_;
    F fun_;

public:
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<R>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<R>;

    map_range(R, F);
};
```

Implementing map in D

```
template <input_drange R, copy_constructible F>
    requires regular_invocable<F&, range_reference_t<R>>
class map_range {
    R base_;
    F fun_;

public:
    using iterator_category = /* ... */;
    using reference = invoke_result_t<F&, range_reference_t<R>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = range_difference_t<R>;

    map_range(R, F);

    auto front() -> reference {
        return invoke(fun_, base_.front());
    }

    void popFront() {
        base_.popFront();
    }

    auto empty() -> bool {
        return base_.empty();
    }
};
```

read

advance

done?

Implementing `filter` in D

```
template <input_drange R, indirect_unary_predicate<R> P>
class filter_range {
    R base_;
    P pred_;

public:
    using iterator_category = /* ... */;
    using reference = range_reference_t<R>;
    using value_type = range_value_t<R>;
    using difference_type = range_difference_t<R>;

    filter_range(R, P);
};
```

Implementing filter in D

```
template <input_drange R, indirect_unary_predicate<R> P>
class filter_range {
    R base_;
    P pred_;
    bool primed_ = false;

    void prime() {
        if (not primed_) {
            while (not base_.empty() and not invoke(pred_, base_.front())) {
                base_.popFront();
            }
            primed_ true;
        }
    }

public:
    using iterator_category = /* ... */;
    using reference = range_reference_t<R>;
    using value_type = range_value_t<R>;
    using difference_type = range_difference_t<R>;

    filter_range(R, P);
};
```

Implementing `filter` in D

```
template <input_drange R, indirect_unary_predicate<R> P>
class filter_range {
    R base_;
    P pred_;
    bool primed_ = false;

    void prime() {
        if (not primed_) {
            base_ = find_if(base_, pred_);
            primed_ true;
        }
    }

public:
    using iterator_category = /* ... */;
    using reference = range_reference_t<R>;
    using value_type = range_value_t<R>;
    using difference_type = range_difference_t<R>;

    filter_range(R, P);
};
```

Implementing `filter` in D

```
template <input_drange R, indirect_unary_predicate<R> P>
class filter_range {
    R base_;
    P pred_;
    bool primed_ = false;

    void prime() {
        if (not primed_) {
            base_ = find_if(std::move(base_), pred_);
            primed_ true;
        }
    }

public:
    using iterator_category = /* ... */;
    using reference = range_reference_t<R>;
    using value_type = range_value_t<R>;
    using difference_type = range_difference_t<R>;

    filter_range(R, P);

    auto front() -> reference {
        prime();
        return base_.front();
    }

    void popFront() {
        prime();
        base_.popFront();
        base_ = find_if(base_, pred_);
    }

    auto empty() -> bool {
        prime();
        return base_.empty();
    }
};
```

Implementing `filter` in D

```
  template <input_drange R, indirect_unary_predicate<R> P>
1 class filter_range {
      R base_;
      P pred_;
      bool primed_ = false;

    1 void prime() {
          if (not primed_) {
              base_ = find_if(std::move(base_), pred_);
              primed_ true;
          }
      }

  public:
      using iterator_category = /* ... */;
      using reference = range_reference_t<R>;
      using value_type = range_value_t<R>;
      using difference_type = range_difference_t<R>;

    2 filter_range(R, P);

    3 auto front() -> reference { prime(); return base_.front(); }
    4 void popFront()           { prime(); base_.popFront(); base_ = find_if(base_, pred_); }
    5 auto empty() -> bool      { prime(); return base_.empty();   }
  };
```

Ensure 1st element is correct

The C# IEnumerator Model

	read	advance	done?
	*it	++it	it == last
	r.front()	r.popFront()	r.empty()
	e.Current()	e.MoveNext()	

advance && done?

```
template <ienumerator E>
void print_all(E e) {
    while (e.MoveNext()) {
        fmt::print("{}\n", e.Current());
    }
}
```

read

Reading Languages

- `read` is a distinct, idempotent function
- But this has one interesting downside…

```
auto some_operation(int) -> int;

void impl() {
    std::vector<int> v = {1, 2, 3, 4, 5, 6};
    auto r = v
           | map(some_operation)
           | filter([](int i){ return i % 2 == 0; });
    for (int i : r) {
        fmt::print("{}\n", i);
    }
}
```

How many times is `some_operation` invoked??

Reading Languages

```
// map
auto map_view<V, F>::Iterator::operator*() const -> reference {
    return invoke(f_, *base_);
}

// filter
auto filter_view<V, P>::Iterator::operator*() const -> reference {
    return *base_;
}

auto filter_view<V, P>::Iterator::operator++() -> Iterator& {
    for (++base_; base_ != ranges::end(parent_->base_); ++base_) {
        if (invoke(pred_, *base_)) {
            break;
        }
    }
    return *this;
}
```

Elements that satisfy the predicate are transformed twice!

Reading Languages

```
// map
auto map_range<R, F>::front() -> reference {
    return invoke(f_, base_.front());
}

// filter
auto filter_range<R, P>::front() -> reference {
    prime();
    return base_.front();
}

void filter_range<R, P>::popFront() {
    prime();
    for (base_.popFront(); not base_.empty(); base_.popFront()) {
        if (invoke(pred_, base_.front()) {
            return;
        }
    }
}
```

Reading Languages

```
// map
auto map_enumerator<E, F>::Current() -> reference {
    return invoke(f_, base_.Current());
}

// filter
auto filter_enumerator<E, P>::Current() -> reference {
    return base_.Current();
}

auto filter_enumerator<E, P>::MoveNext() -> bool {
    while (base_.MoveNext()) {
        if (invoke(pred_, base_.Current())) {
            return true;
        }
    }
    return false;
}
```

C++ Iterators vs D Ranges

- D model is *much* simpler
  - `map` was (5 types, 18 functions) in C++ vs (1 type, 4 functions) in D
  - `filter` was (3 types, 13 functions) in C++ vs (1 type, 5 functions) in D
- So why didn't we copy it?

C++ Iterators vs D Ranges: `find_if`

- Consider: I want the first element that satisfies a predicate

```
// C++
template <input_iterator I, sentinel_for<I> S, indirect_unary_predicate<I> Pred>
auto find_if(I first, S last, Pred pred) -> I {
    for (; first != last; ++first) {
        if (invoke(pred, *first)) {
            break;
        }
    }
    return first;
}

// D
template <input_drange R, indirect_unary_predicate<R> Pred>
auto find_if(R range, Pred pred) -> R {
    for (; not range.empty(); range.popFront()) {
        if (invoke(pred, range.front())) {
            break;
        }
    }
    return range;
}
```

C++ Iterators vs D Ranges: `until`

- Consider: I want the range *until* the first element that satisfies a predicate

```
// C++
template <forward_iterator I, sentinel_for<I> S, indirect_unary_predicate<I> Pred>
auto until(I first, S last, Pred pred) -> subrange<I> {
    I it = find_if(first, last, pred);
    return {first, it};
}


// D
template <forward_drange R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) -> R {
    R r = find_if(range, pred);
    // ????
}
```

C++ Iterators vs D Ranges: `until`

- Consider: I want the range *until* the first element that satisfies a predicate

```
// C++
template <forward_iterator I, sentinel_for<I> S, indirect_unary_predicate<I> Pred>
auto until(I first, S last, Pred pred) -> subrange<I> {
    I it = find_if(first, last, pred);
    return {first, it};
}

// D
template <forward_range R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) {
    struct until_range {
        R base;
        Pred pred;

        // aliases ...

        auto front() -> range_reference_t<R> { return base.front(); }
        void popFront() { base.popFront(); }
        auto empty() -> bool { return base.empty() or invoke(pred, base.front()); }
    };

    return {range, pred};
}
```

C++ Iterators vs D Ranges: `until`

- Consider: I want the range *until* the first element that satisfies a predicate

```
// C++
template <forward_iterator I, sentinel_for<I> S, indirect_unary_predicate<I> Pred>
auto until(I first, S last, Pred pred) -> subrange<I> {
    I it = find_if(first, last, pred);
    return {first, it};
}

// D
template <forward_range R, indirect_unary_predicate<R> Pred>
struct take_while_range {
    R base;
    Pred pred;

    // aliases ...

    auto front() -> range_reference_t<R> { return base.front(); }
    void popFront() { base.popFront(); }
    auto empty() -> bool { return base.empty() or not invoke(pred, base.front()); }
};

template <forward_range R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) {
    return take_while_range{range, not_fn(pred)};
}
```

Implementing `take` in D

```
template <forward_range R>
struct take_range {
    R base;
    int n;

    // aliases ...

    auto front() -> range_reference_t<R> { return base.front(); }
    void popFront() { base.popFront(); --n; }
    auto empty() -> bool { return n == 0 or base.empty(); }
};
```

Implementing `until` with `take` in D

```
template <forward_range R>
struct take_range {
    R base;
    int n;

    auto front() -> range_reference_t<R> { return base.front(); }
    void popFront() { base.popFront(); --n; }
    auto empty() -> bool { return n == 0 or base.empty(); }
};

template <forward_range R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) -> take_range<R> {
    R orig = range;
    int n = 0;
    for (; not range.empty(); range.popFront(), ++n) {
        if (invoke(pred, range.front())) {
            break;
        }
    }

    return take_range<R>{orig, n};
}
```

Implementing `until` with `take_exactly`

```
template <forward_range R>
struct take_exactly_range {
    R base;
    int n;

    auto front() -> range_reference_t<R> { return base.front(); }
    void popFront() { base.popFront(); --n; }
    auto empty() -> bool { return n == 0; }
};

template <forward_range R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) -> take_exactly_range<R> {
    R orig = range;
    int n = 0;
    for (; not range.empty(); range.popFront(), ++n) {
        if (invoke(pred, range.front())) {
            break;
        }
    }

    return take_exactly_range<R>{orig, n};
}
```

Implementing `until` with `take_exactly`

```
template <forward_range R>
struct take_exactly_range {
    R base;
    int n;

    auto front() -> range_reference_t<R> { return base.front(); }
    void popFront() { base.popFront(); --n; }
    auto empty() -> bool { return n == 0; }
};

template <forward_range R, indirect_unary_predicate<R> Pred>
auto position(R range, Pred pred) -> int {
    int n = 0;
    for (; not range.empty(); range.popFront(), ++n) {
        if (invoke(pred, range.front())) { break; }
    }
    return n;
}

template <forward_range R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) -> take_exactly_range<R> {
    return take_exactly_range<R>{range, position(range, pred)};
}
```

C++ Iterators vs D Ranges: `until`

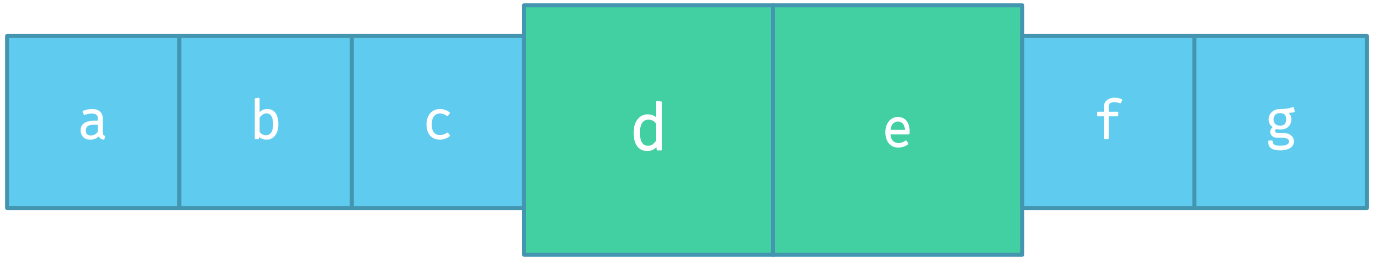
- Consider: I want the range *until* the first element that satisfies a predicate

```
// C++
template <forward_iterator I, sentinel_for<I> S, indirect_unary_predicate<I> Pred>
auto until(I first, S last, Pred pred) -> subrange<I> {
    return subrange<I>{first, find_if(first, last, pred)};
}

// D
template <forward_range R, indirect_unary_predicate<R> Pred>
auto position(R range, Pred pred) -> int {
    int n = 0;
    for (; not range.empty(); range.popFront(), ++n) {
        if (invoke(pred, range.front())) { break; }
    }
    return n;
}

template <forward_range R, indirect_unary_predicate<R> Pred>
auto until(R range, Pred pred) -> take_exactly_range<R> {
    return take_exactly_range<R>{range, position(range, pred)};
}
```

C++ Iterators vs D Ranges: splitting

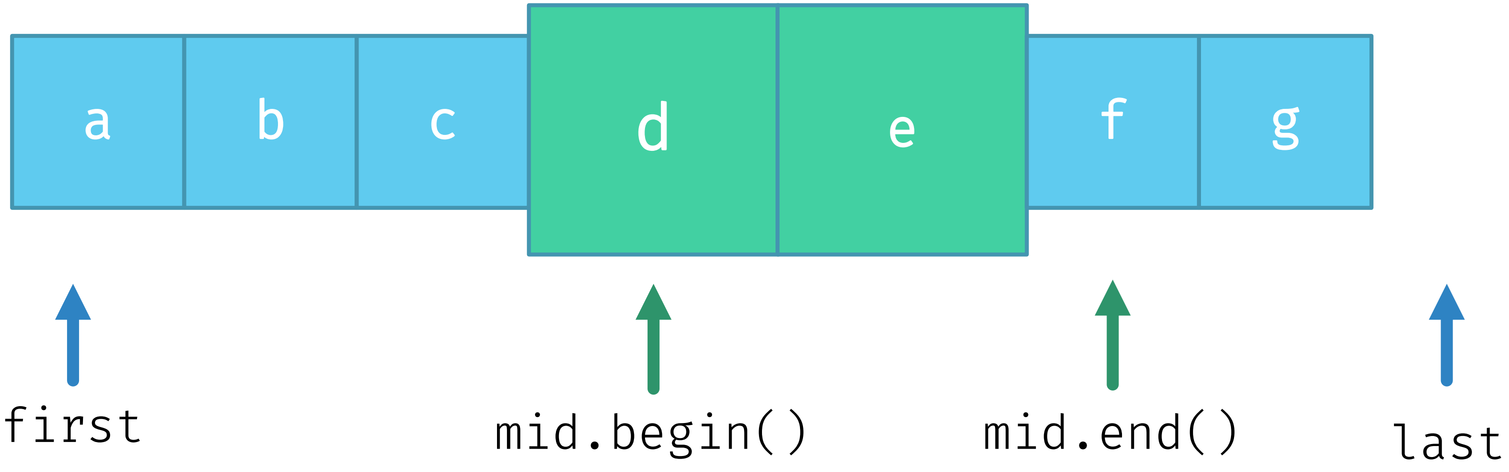
a	b	c	d	e	f	g

C++ Iterators vs D Ranges: splitting



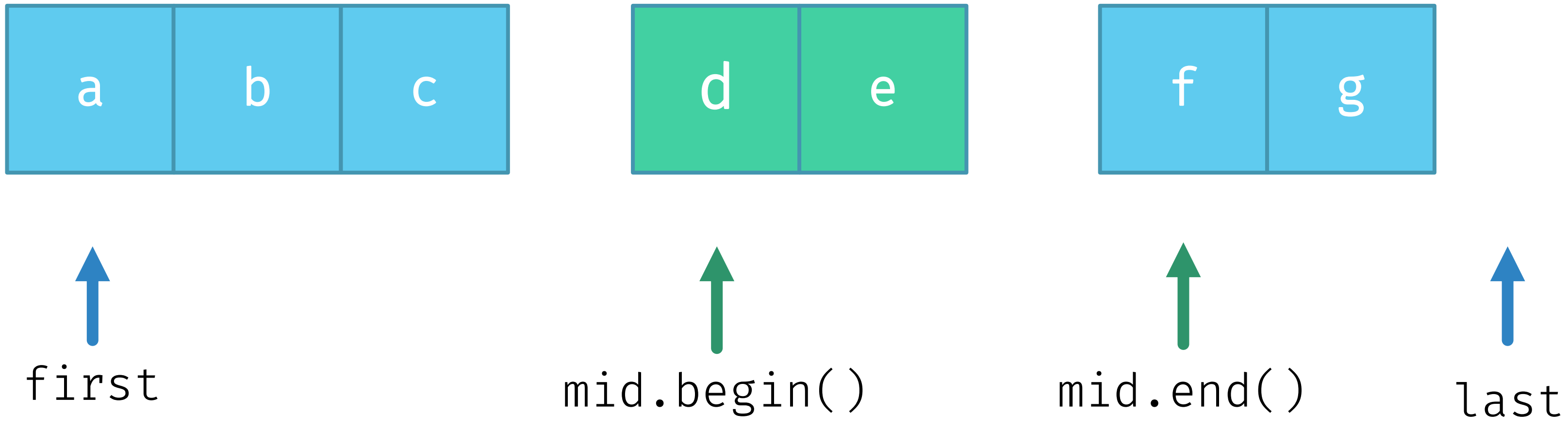

```
auto mid = ranges::search(first, last, first2, last2);
```

C++ Iterators vs D Ranges: splitting



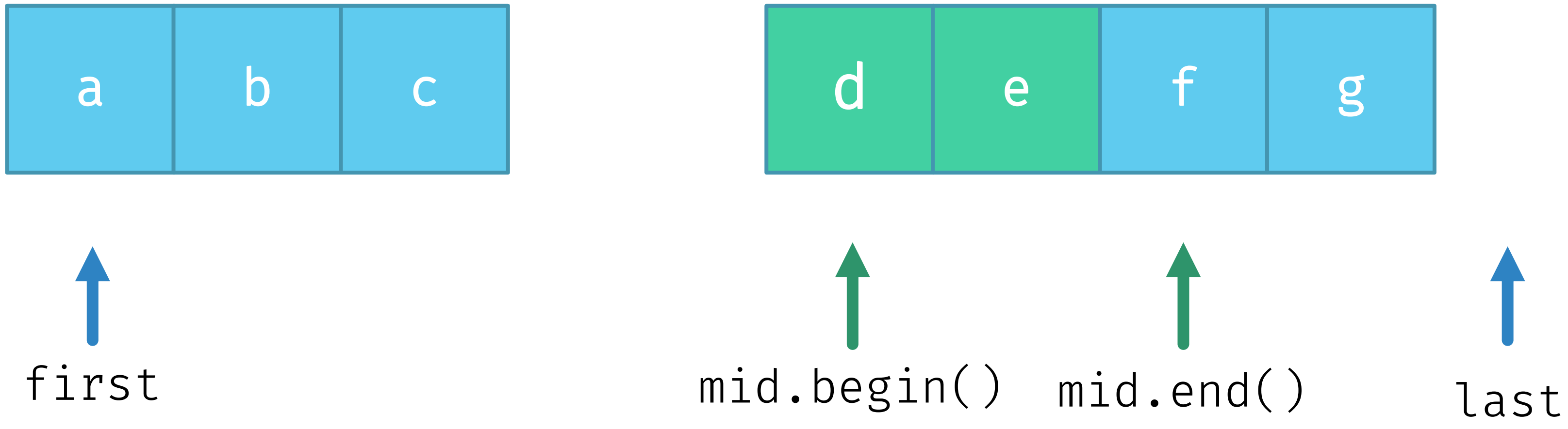

```
auto mid = ranges::search(first, last, first2, last2);
```

C++ Iterators vs D Ranges: splitting



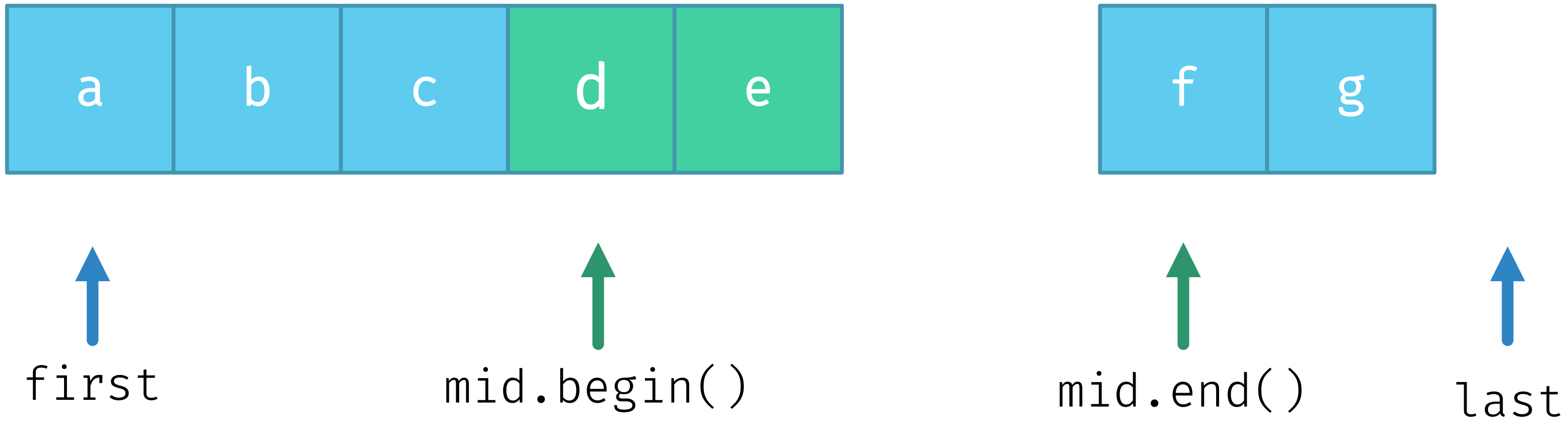

```
template <forward_iterator I,  sentinel_for<I> S,
          forward_iterator I2, sentinel_for<I2> S2>
auto find_split(I first, S last, I2 first2, S2 last2) {
    auto mid  = ranges::search(first, last, first2, last2);
    auto pre  = ranges::subrange(first, mid.begin());
    auto post = ranges::subrange(mid.end(), last);
    return tuple(pre, mid, post);
}
```

C++ Iterators vs D Ranges: splitting




```cpp
template <forward_iterator I,  sentinel_for<I> S,
          forward_iterator I2, sentinel_for<I2> S2>
auto find_split_before(I first, S last, I2 first2, S2 last2) {
    auto mid  = ranges::search(first, last, first2, last2);
    auto pre  = ranges::subrange(first, mid.begin());
    auto post = ranges::subrange(mid.begin(), last);
    return tuple(pre, post);
}
```

C++ Iterators vs D Ranges: splitting

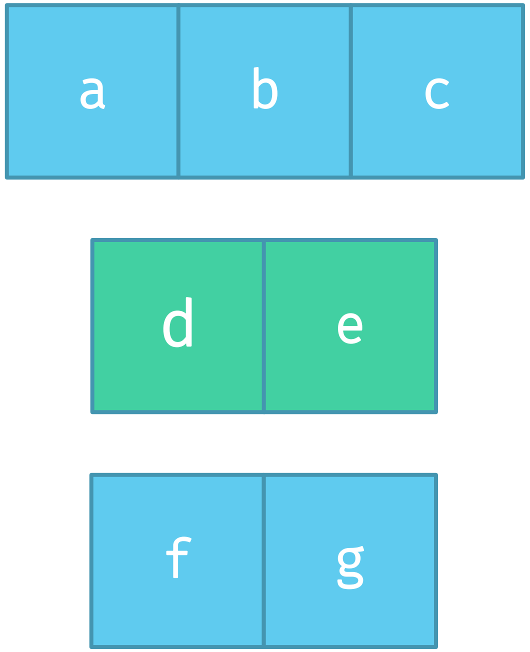



```
template <forward_iterator I,  sentinel_for<I> S,
          forward_iterator I2, sentinel_for<I2> S2>
auto find_split_after(I first, S last, I2 first2, S2 last2) {
    auto mid  = ranges::search(first, last, first2, last2);
    auto pre  = ranges::subrange(first, mid.end());
    auto post = ranges::subrange(mid.end(), last);
    return tuple(pre, post);
}
```

C++ Iterators vs D Ranges

- `findSplit`

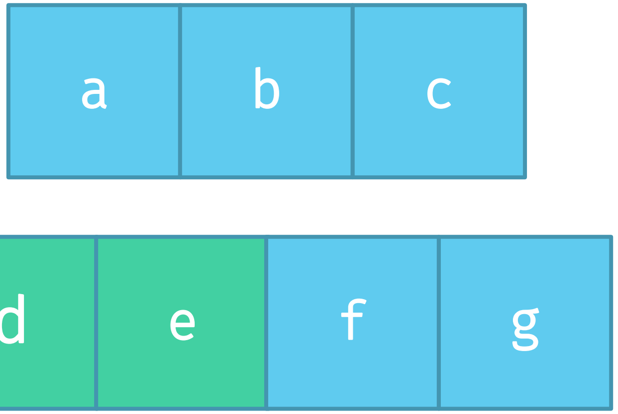
```d
auto findSplit(alias pred = "a == b", R1, R2)(R1 haystack, R2 needle)
if (isForwardRange!R1 && isForwardRange!R2)
{
    static struct Result(S1, S2) if (isForwardRange!S1 &&
                                     isForwardRange!S2)
    {
        this(S1 pre, S1 separator, S2 post)
        {
            asTuple = typeof(asTuple)(pre, separator, post);
        }
        void opAssign(typeof(asTuple) rhs)
        {
            asTuple = rhs;
        }
        Tuple!(S1, S1, S2) asTuple;
        static if (hasConstEmptyMember!(typeof(asTuple[1])))
        {
            bool opCast(T : bool)() const
            {
                return !asTuple[1].empty;
            }
        }
        else
        {
            bool opCast(T : bool)()
            {
                return !asTuple[1].empty;
            }
        }
        alias asTuple this;
    }

    static if (isSomeString!R1 && isSomeString!R2
            || (isRandomAccessRange!R1 && hasLength!R1 && hasSlicing!R1 && hasLength!R2))
    {
        auto balance = find!pred(haystack, needle);
        immutable pos1 = haystack.length - balance.length;
        immutable pos2 = balance.empty ? pos1 : pos1 + needle.length;
        return Result!(typeof(haystack[0 .. pos1]),
                       typeof(haystack[pos2 .. haystack.length]))(haystack[0 .. pos1],
                                                                  haystack[pos1 .. pos2],
                                                                  haystack[pos2 .. haystack.length]);
    }
    else
    {
        import std.range : takeExactly;
        auto original = haystack.save;
        auto h = haystack.save;
        auto n = needle.save;
        size_t pos1, pos2;
        while (!n.empty && !h.empty)
        {
            if (binaryFun!pred(h.front, n.front))
            {
                h.popFront();
                n.popFront();
                ++pos2;
            }
            else
            {
                haystack.popFront();
                n = needle.save;
                h = haystack.save;
                pos2 = ++pos1;
            }
        }
        if (!n.empty) // incomplete match at the end of haystack
        {
            pos1 = pos2;
        }
        return Result!(typeof(takeExactly(original, pos1)),
                       typeof(h))(takeExactly(original, pos1),
                                  takeExactly(haystack, pos2 - pos1),
                                  h);
    }
}
```



C++ Iterators vs D Ranges

- `findSplitBefore`

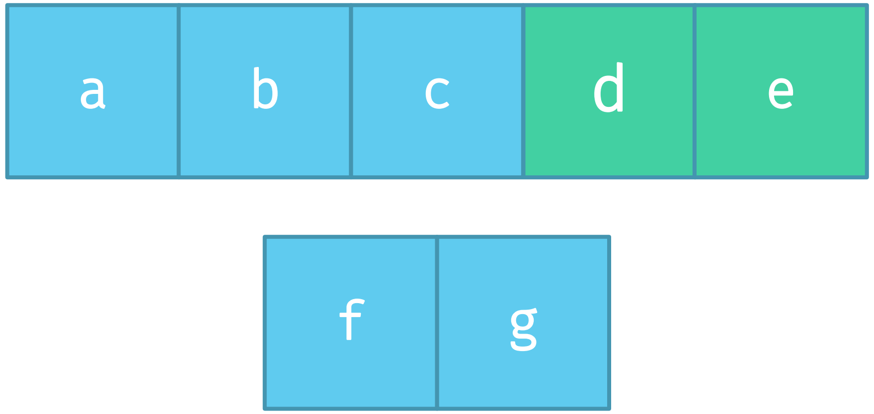
```
auto findSplitBefore(alias pred = "a == b", R1, R2)(R1 haystack, R2 needle)
if (isForwardRange!R1 && isForwardRange!R2)
{
    static struct Result(S1, S2) if (isForwardRange!S1 &&
                                     isForwardRange!S2)
    {
        this(S1 pre, S2 post)
        {
            asTuple = typeof(asTuple)(pre, post);
        }
        void opAssign(typeof(asTuple) rhs)
        {
            asTuple = rhs;
        }
        Tuple!(S1, S2) asTuple;
        static if (hasConstEmptyMember!(typeof(asTuple[1])))
        {
            bool opCast(T : bool)() const
            {
                return !asTuple[1].empty;
            }
        }
        else
        {
            bool opCast(T : bool)()
            {
                return !asTuple[1].empty;
            }
        }
        alias asTuple this;
    }

    static if (isSomeString!R1 && isSomeString!R2
            || (isRandomAccessRange!R1 && hasLength!R1 && hasSlicing!R1 && hasLength!R2))
    {
        auto balance = find!pred(haystack, needle);
        immutable pos = haystack.length - balance.length;
        return Result!(typeof(haystack[0 .. pos]),
                       typeof(haystack[pos .. haystack.length]))(haystack[0 .. pos],
                                                                 haystack[pos .. haystack.length]);
    }
    else
    {
        import std.range : takeExactly;
        auto original = haystack.save;
        auto h = haystack.save;
        auto n = needle.save;
        size_t pos1, pos2;
        while (!n.empty && !h.empty)
        {
            if (binaryFun!pred(h.front, n.front))
            {
                h.popFront();
                n.popFront();
                ++pos2;
            }
            else
            {
                haystack.popFront();
                n = needle.save;
                h = haystack.save;
                pos2 = ++pos1;
            }
        }
        if (!n.empty) // incomplete match at the end of haystack
        {
            pos1 = pos2;
            haystack = h;
        }
        return Result!(typeof(takeExactly(original, pos1)),
                       typeof(haystack))(takeExactly(original, pos1),
                                         haystack);
    }
}
```



C++ Iterators vs D Ranges

- `findSplitAfter`

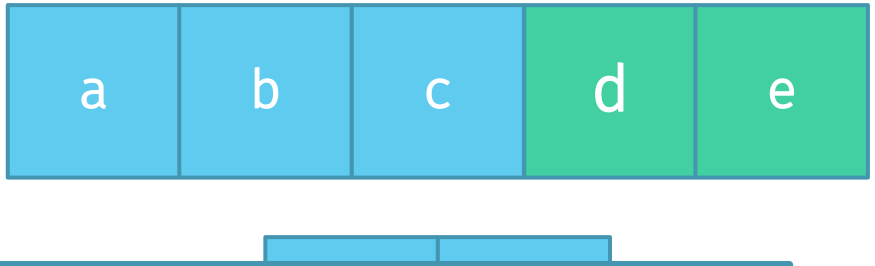
```d
auto findSplitAfter(alias pred = "a == b", R1, R2)(R1 haystack, R2 needle)
if (isForwardRange!R1 && isForwardRange!R2)
{
    static struct Result(S1, S2) if (isForwardRange!S1 &&
                                     isForwardRange!S2)
    {
        this(S1 pre, S2 post)
        {
            asTuple = typeof(asTuple)(pre, post);
        }
        void opAssign(typeof(asTuple) rhs)
        {
            asTuple = rhs;
        }
        Tuple!(S1, S2) asTuple;
        static if (hasConstEmptyMember!(typeof(asTuple[1])))
        {
            bool opCast(T : bool)() const
            {
                return !asTuple[0].empty;
            }
        }
        else
        {
            bool opCast(T : bool)()
            {
                return !asTuple[0].empty;
            }
        }
        alias asTuple this;
    }

    static if (isSomeString!R1 && isSomeString!R2
            || (isRandomAccessRange!R1 && hasLength!R1 && hasSlicing!R1 && hasLength!R2))
    {
        auto balance = find!pred(haystack, needle);
        immutable pos = balance.empty ? 0 : haystack.length - balance.length + needle.length;
        return Result!(typeof(haystack[0 .. pos]),
                       typeof(haystack[pos .. haystack.length]))(haystack[0 .. pos],
                                                                 haystack[pos .. haystack.length]);
    }
    else
    {
        import std.range : takeExactly;
        auto original = haystack.save;
        auto h = haystack.save;
        auto n = needle.save;
        size_t pos1, pos2;
        while (!n.empty)
        {
            if (h.empty)
            {
                // Failed search
                return Result!(typeof(takeExactly(original, 0)),
                               typeof(original))(takeExactly(original, 0),
                                                 original);
            }
            if (binaryFun!pred(h.front, n.front))
            {
                h.popFront();
                n.popFront();
                ++pos2;
            }
            else
            {
                haystack.popFront();
                n = needle.save;
                h = haystack.save;
                pos2 = ++pos1;
            }
        }
        return Result!(typeof(takeExactly(original, pos2)),
                       typeof(h))(takeExactly(original, pos2),
                                  h);
    }
}
```



C++ Iterators vs D Ranges

- `findSplitAfter`

```d
auto findSplitAfter(alias pred = "a == b", R1, R2)(R1 haystack, R2 needle)
if (isForwardRange!R1 && isForwardRange!R2)
{
    static struct Result(S1, S2) if (isForwardRange!S1 &&
                                     isForwardRange!S2)
    {
        this(S1 pre, S2 post)
        {
            asTuple = typeof(asTuple)(pre, post);
        }
        void opAssign(typeof(asTuple) rhs)
        {
            asTuple = rhs;
        }
        Tuple!(S1, S2) asTuple;
        static if (hasConstEmptyMember!(typeof(asTuple[1])))
        {
            bool opCast(T : bool)() const
            {
                return !asTuple[0].empty;
            }
        }
        else
        {
            bool opCast(T : bool)()
            {
                return !asTuple[0].empty;
            }
        }
        alias asTuple this;
    }
```



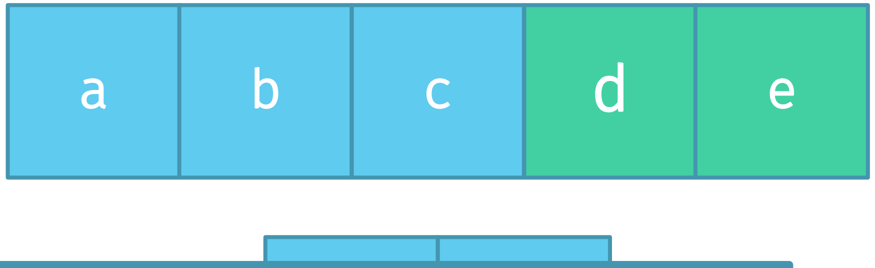

```d
static if (isSomeString!R1 && isSomeString!R2
        || (isRandomAccessRange!R1 && hasLength!R1 && hasSlicing!R1 && hasLength!R2))
{
    auto balance = find!pred(haystack, needle);
    immutable pos = balance.empty ? 0 : haystack.length - balance.length + needle.length;
    return Result!(typeof(haystack[0 .. pos]),
                   typeof(haystack[pos .. haystack.length]))(haystack[0 .. pos],
                                                             haystack[pos .. haystack.length]);
}
```

```d
            {
                h.popFront();
                n.popFront();
                ++pos2;
            }
            else
            {
                haystack.popFront();
                n = needle.save;
                h = haystack.save;
                pos2 = ++pos1;
            }
        }
        return Result!(typeof(takeExactly(original, pos2)),
                       typeof(h))(takeExactly(original, pos2),
                                  h);
    }
}
```

C++ Iterators vs D Ranges

- `findSplitAfter`

```d
auto findSplitAfter(alias pred = "a == b", R1, R2)(R1 haystack, R2 needle)
if (isForwardRange!R1 && isForwardRange!R2)
{
    static struct Result(S1, S2) if (isForwardRange!S1 &&
                                     isForwardRange!S2)
    {
        this(S1 pre, S2 post)
        {
            asTuple = typeof(asTuple)(pre, post);
        }
        void opAssign(typeof(asTuple) rhs)
        {
            asTuple = rhs;
        }
        Tuple!(S1, S2) asTuple;
        static if (hasConstEmptyMember!(typeof(asTuple[1])))
        {
            bool opCast(T : bool)() const
            {
                return !asTuple[0].empty;
            }
        }
        else
        {
            bool opCast(T : bool)()
            {
                return !asTuple[0].empty;
            }
        }
        alias asTuple this;
    }
```




```d
static if (isSomeString!R1 && isSomeString!R2
        || (isRandomAccessRange!R1 && hasLength!R1 && hasSlicing!R1 && hasLength!R2))
{
    auto balance = find!pred(haystack, needle);
    immutable pos = balance.empty ? 0 : haystack.length - balance.length + needle.length;
    return Result!(/* ... */)(haystack[0 .. pos], haystack[pos .. haystack.length]);
}
```

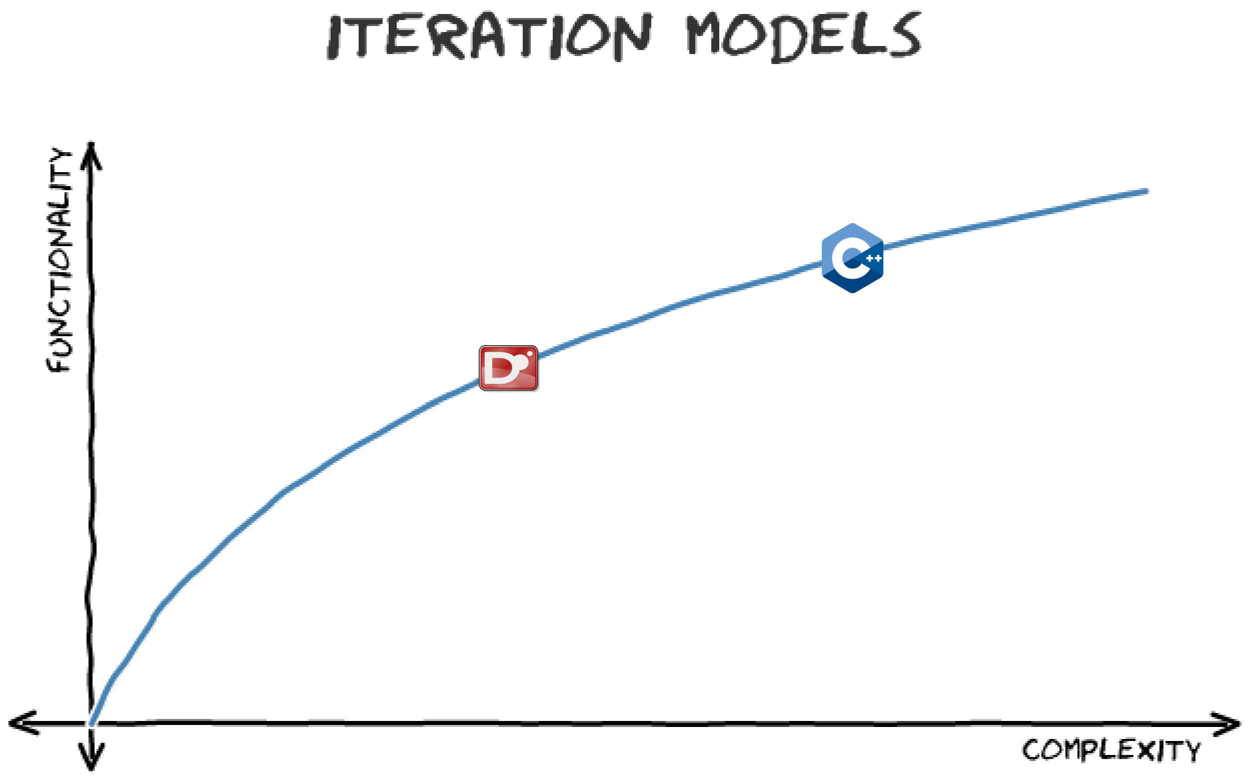
```d
            {
                h.popFront();
                n.popFront();
                ++pos2;
            }
```

```cpp
ranges::subrange(first, mid.end());
```

```cpp
ranges::subrange(mid.end(), last);
```

```d
            }
        }
        return Result!(typeof(takeExactly(original, pos2)),
                       typeof(h))(takeExactly(original, pos2),
                                  h);
    }
}
```

C++ Iterators vs D Ranges



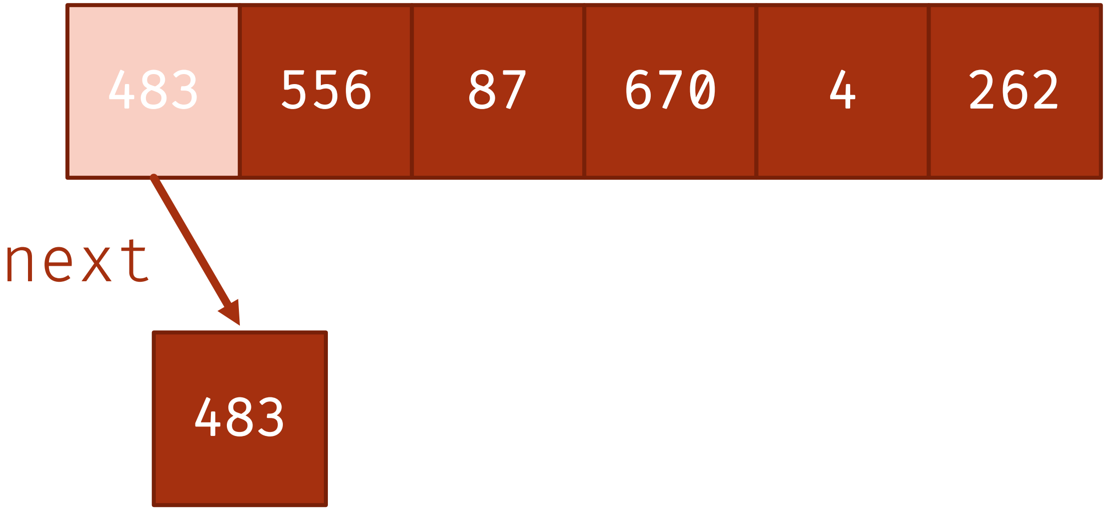
The Rust Iterator Model

Have you thought about just rewriting everything in Rust?

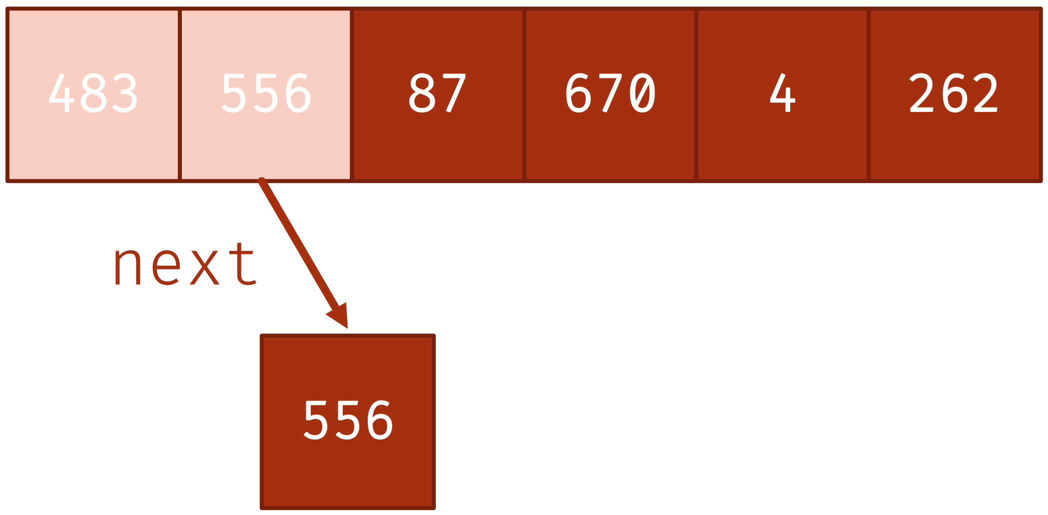
The Rust Iterator Model

483	556	87	670	4	262

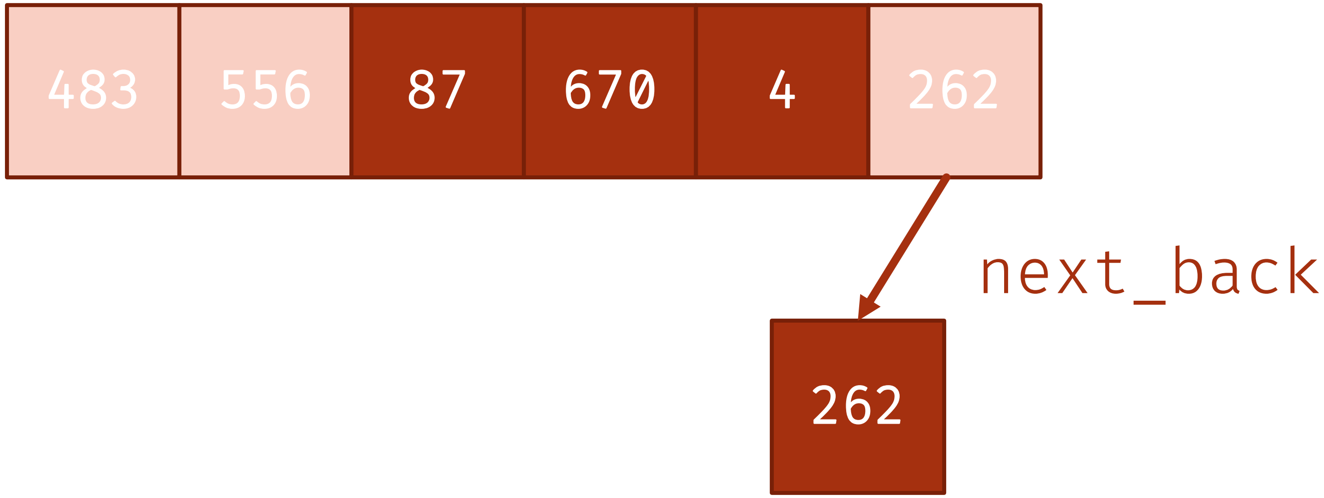
The Rust Iterator Model



The Rust Iterator Model



The Rust Iterator Model



The Rust Iterator Model

	read	advance	done?
	*it	++it	it == last
	r.front()	r.popFront()	r.empty()
	e.Current()	e.MoveNext()	
	it.next()		

Basic Rust Iterator Structure

consumed

```
1 struct Iterator {
      using reference = /* ... */;
      using value_type = /* ... */;
      using difference_type = /* ... */;

  1   auto next() -> Optional<reference>;
  };
```

NOT `std::optional`
We need to support `optional<T&>` here

`advance, read, && done?`

Basic Rust Iterator Structure

```
template <forward_iterator I, sentinel_for<I> S>
class rust_iterator {
    I first;
    S last;

public:
    using reference = iter_reference_t<I>;
    using value_type = iter_value_t<I>;
    using difference_type = iter_difference_t<I>;

    auto next() -> Optional<reference> {
        if (first != last) {
            return *first++;
        }
        return nullopt;
    }
};
```

Basic Rust Iterator Structure

```
template <input_iterator I, sentinel_for<I> S>
class rust_iterator {
    I first;
    S last;
    bool advance = false;

public:
    using reference = iter_reference_t<I>;
    using value_type = iter_value_t<I>;
    using difference_type = iter_difference_t<I>;

    auto next() -> Optional<reference> {
        if (advance) { ++first; }
        advance = true;

        if (first != last) {
            return *first;
        }
        return nullopt;
    }
};
```

Basic Rust Iterator Structure

```
template <rust_iterator I>
void print_all(I it) {
    while (auto val = it.next()) {
        fmt::print("{}\n", *val);
    }
}
```

advance, read, && done?

Adapting Iterators in Rust

`map` and `filter`

Implementing `map` in Rust

```
template <rust_iterator I, typename F>
    requires regular_invocable<F&, iter_reference_t<I>>
class map_iterator {
    I base_;
    F fun_;

public:
    using reference = invoke_result_t<F&, iter_reference_t<I>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = iter_difference_t<I>;

    map_iterator(I, F);

    auto next() -> Optional<reference> {
        if (auto val = base_.next()) {
            return invoke(fun_, *val);
        }
        return nullopt;
    }
};
```

Implementing `map` in Rust

```
template <rust_iterator I, typename F>
    requires regular_invocable<F&, iter_reference_t<I>>
class map_iterator {
    I base_;
    F fun_;

public:
    using reference = invoke_result_t<F&, iter_reference_t<I>>;
    using value_type = remove_cvref_t<reference>;
    using difference_type = iter_difference_t<I>;

    map_iterator(I, F);

    auto next() -> Optional<reference> {
        return base_.next().map(fun_);
    }
};
```

1 class map_iterator

1 map_iterator(I, F);

2 auto next()

everything

Implementing `filter` in Rust

```
template <rust_iterator I, indirect_unary_invocable<I> P>
1 class filter_iterator {
    I base_;
    P pred_;

public:
    using reference = iter_reference_t<I>;
    using value_type = iter_value_t<I>;
    using difference_type = iter_difference_t<I>;

  1 filter_iterator(I, F);

  2 auto next() -> Optional<reference> {
        while (auto val = base_.next()) {
            if (invoke(pred_, *val)) {
                return val;
            }
        }
        return nullopt;
    }
};
```

For a complete implementation, see https://github.com/tcbrindle/libflow

The Rust Iterator Model

	read	advance	done?
	`*it`	`++it`	`it == last`
	`r.front()`	`r.popFront()`	`r.empty()`
	`e.Current()`	`e.MoveNext()`	
	`it.next()`		

The Rust/Python Iterator Model

	read	advance	done?
	*it	++it	it == last
	r.front()	r.popFront()	r.empty()
	e.Current()	e.MoveNext()	
	it.next()		
	it.__next__()		

The Rust/Python/Java/… Iterator Model

	read	advance	done?
	*it	++it	it == last
	r.front()	r.popFront()	r.empty()
	e.Current()	e.MoveNext()	
	it.next()		
	it.__next__()		
	it.next()		it.hasNext()

Reading Languages

Iterator Languages

Implementing `filter` in Python

```
template <py_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;

public:
    auto __next__() -> reference {
        for (;;) {
            reference val = base_.__next__();
            if (invoke(pred_, val)) {
                return val;
            }
        }
    }
};
```

Implementing `filter` in Java

```
template <java_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;

public:
    auto next() -> reference {
        for (;;) {
            reference val = base_.next();
            if (invoke(pred_, val)) {
                return val;
            }
        }
    }

    auto hasNext() -> bool {
        // ???
    }
};
```

Implementing `filter` in Java

```
template <java_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;
    Optional<reference> next_elem_;

    auto set_next() -> bool {
        while (base_.hasNext()) {
            next_elem_ = base_.next();
            if (invoke(pred_, *next_elem_)) {
                return true;
            }
        }
        return false;
    }

public:
    auto next() -> reference {
        // ...
    }

    auto hasNext() -> bool {
        return next_elem_ or set_next();
    }
};
```

Implementing `filter` in Java

```
template <java_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;
    Optional<reference> next_elem_;

    auto set_next() -> bool {
        while (base_.hasNext()) {
            next_elem_ = base_.next();
            if (invoke(pred_, *next_elem_)) {
                return true;
            }
        }
        return false;
    }

public:
    auto next() -> reference {
        if (not next_elem_ and not set_next()) {
            throw NoSuchElement();
        }
        reference v = *next_elem_;
        next_elem_.reset();
        return v;
    }

    auto hasNext() -> bool {
        return next_elem_ or set_next();
    }
};
```

Implementing `filter` in Java

```
template <java_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;
    Optional<reference> next_elem_;

    auto set_next() -> bool {
        while (base_.hasNext()) {
            next_elem_ = base_.next();
            if (invoke(pred_, *next_elem_)) {
                return true;
            }
        }
        return false;
    }

public:
    auto next() -> reference {
        if (not hasNext()) {
            throw NoSuchElement();
        }
        reference v = *next_elem_;
        next_elem_.reset();
        return v;
    }

    auto hasNext() -> bool {
        return next_elem_ or set_next();
    }
};
```

Implementing `filter` in Java

```
template <java_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;
    Optional<reference> next_elem_;

    auto set_next() -> bool {
        while (base_.hasNext()) {
            next_elem_ = base_.next();
            if (invoke(pred_, *next_elem_)) {
                return true;
            }
        }
        return false;
    }

public:
    auto next() -> reference {
        if (not hasNext()) {
            throw NoSuchElement();
        }

        return *next_elem_.take();
    }

    auto hasNext() -> bool {
        return next_elem_ or set_next();
    }
};
```

Implementing peek in Rust

```
template <rust_iterator I>
class peek_iterator {
    I base_;
    Optional<iter_reference_t<I>> next_elem_;

public:
    auto next() -> Optional<reference> {
        if (next_elem_) {
            return next_elem_.take();
        }
        return base_.next();
    }

    auto peek() -> Optional<reference> {
        if (not next_elem_) {
            next_elem_ = base_.next();
        }
        return next_elem_;
    }
};
```

Iterator Languages

- Let’s go back to this example:

```
auto some_operation(int) -> int;

void impl() {
    std::vector<int> v = {1, 2, 3, 4, 5, 6};
    auto r = v
           | map(some_operation)
           | filter([](int i){ return i % 2 == 0; });
    for (int i : r) {
        fmt::print("{}\n", i);
    }
}
```

How many times is `some_operation` invoked??

Exactly 6.

Iterator Languages

```
auto map_iterator<I, F>::next() -> Optional<reference> {
    return base_.next().map(fun_);
}

auto filter_iterator<I, F>::next() -> Optional<eference> {
    while (auto val = base_.next()) {
        if (invoke(pred_, *val)) {
            return val;
        }
    }
    return nullopt;
}
```

Iterator Languages

```
auto map_iterator<I, F>::next() -> reference {
    return invoke(fun_, base_.next());
}

auto filter_iterator<I, F>::next() -> reference {
    for (;;) {
        reference val = base_.next();
        if (invoke(pred_, val)) {
            return val;
        }
    }
}
```

Iterator Languages

```
auto map_iterator<I, F>::next() -> reference {
    return invoke(fun_, base_.next());
}

template <java_iterator I, indirect_unary_predicate<I> P>
class filter_iterator {
    I base_;
    P pred_;
    Optional<reference> next_elem_;

    auto set_next() -> bool {
        while (base_.hasNext()) {
            next_elem_ = base_.next();
            if (invoke(pred_, *next_elem_)) {
                return true;
            }
        }
        return false;
    }

public:
    auto next() -> reference {
        if (not hasNext()) {
            throw NoSuchElement();
        }
        return *next_elem_.take();
    }

    auto hasNext() -> bool {
        return next_elem_ or set_next();
    }
};
```

Iterator Languages

- Do the iterator languages offer free performance?
- Consider this example:

```
auto some_operation(int) -> int;

void impl() {
    std::vector<int> v = {1, 2, 3, 4, 5, 6};
    auto r = v
        | map(some_operation)
        | drop(2);
    for (int i : r) {
        fmt::print("{}\n", i);
    }
}
```

How many times is `some_operation` invoked??

Exactly 6.

But C++/D/C# do it in 4

Iterator Languages

- Do the iterator languages offer free performance?
- Consider this example:

```
auto some_operation(int) -> int;

void impl() {
    std::vector<int> v = {1, 2, 3, 4, 5, 6};
    auto r = v
           | drop(2)
           | map(some_operation);
    for (int i : r) {
        fmt::print("{}\n", i);
    }
}
```

How many times is `some_operation` invoked??

Exactly 4.

But C++/D/C# do it in 4

Iterator Languages

- Do the iterator languages offer free performance?
- Do the iterator languages offer equivalent functionality?
- Consider `find_if`…

Iterator Languages: find_if

```
template <input_iterator I, sentinel_for<I> S,
          indirect_unary_predicate<I> Pred>
auto find_if(I first, S last, Pred pred) -> I {
    for (; first != last; ++first) {
        if (invoke(pred, *first)) {
            return first;
        }
    }
    return first;
}
```

Iterator Languages: find_if

```
template <input_iterator I, sentinel_for<I> S,
          indirect_unary_predicate<I> Pred>
auto find_if(I first, S last, Pred pred) -> I {
    for (; first != last; ++first) {
        if (invoke(pred, *first)) {
            return first;
        }
    }
    return first;
}

auto it = find_if(v.begin(), v.end(), is_bad);
if (it != v.end()) {
    v.erase(it);
}
```

Iterator Languages: `find_if`

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto find_if(I it, Pred pred) -> ???? {
    // ...
}

auto r = find_if(v.iter(), is_bad);
```

Iterator Languages: find_if

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto find_if(I it, Pred pred) -> I {
    // ...
}

auto r = find_if(v.iter(), is_bad);
```

Iterator Languages: `find_if`

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto find_if(I it, Pred pred) -> Optional<iter_reference_t<I>> {
    // ...
}

auto r = find_if(v.iter(), is_bad);
```

Iterator Languages: `find_if`

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto find_if(I it, Pred pred) -> Optional<iter_reference_t<I>> {
    while (auto val = iter.next()) {
        if (invoke(pred, *val)) {
            return val;
        }
    }
    return nullopt;
}

auto r = find_if(v.iter(), is_bad);
```

Iterator Languages: `find_if`

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto find_if(I it, Pred pred) -> Optional<iter_reference_t<I>> {
    while (auto val = iter.next()) {
        if (invoke(pred, *val)) {
            return val;
        }
    }
    return nullopt;
}

auto r = find_if(v.iter(), is_bad);
if (r) {
    // ???
}
```

Iterator Languages: `find_if`

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto find_if(I it, Pred pred) -> Optional<iter_reference_t<I>> {
    while (auto val = iter.next()) {
        if (invoke(pred, *val)) {
            return val;
        }
    }
    return nullopt;
}

auto r = find_if(v.iter(), is_bad);
if (r) {
    v.erase_at_index(&*r - v.data());
}
```

Iterator Languages: position

```
template <rust_iterator I, indirect_unary_predicate<I> Pred>
auto position(I iter, Pred pred) -> Optional<size_t> {
    size_t n = 0;
    while (auto val = iter.next()) {
        if (invoke(pred, *val)) {
            return n;
        }
        ++n;
    }
    return nullopt;
}

auto pos = position(v.iter(), is_bad);
if (pos) {
    v.erase_at_index(*pos);
}
```

Hope v is random access?

Iterator Languages: functional gaps

- No container/iterator cohesion
- What about algorithms?
- Consider `group_by`…

Iterator Languages: `group_by`

- There are two approaches to group_by.
  1. Binary

     ```
     >>> groupBy (<=) [1,2,2,3,1,2,0,4,5,2]
     [[1,2,2,3],[1,2],[0,4,5],[2]]
     ```

  2. Unary

     ```
     >>> groupOn (\x -> x `div` 2) [1,2,2,3,1,2,0,4,5,2]
     [[1],[2,2,3],[1],[2],[0],[4,5],[2]]
     ```

Iterator Languages: `group_by`

```
template <rust_iterator I, indirect_binary_predicate<I> P>
class group_by_inner {
    I base_;
    P pred_;

public:
    auto next() -> Optional<iter_reference_t<I>> {
        auto val = base_.next();
        if (not val) {
            return nullopt;
        }

        if (invoke(pred_, /* ??? */, *val)) {
            return val;
        } else {
            // ...
        }
    }
};
```

Iterator Languages: `group_by`

```
template <rust_iterator I, indirect_binary_predicate<I> P>
class group_by_inner {
    I base_;
    P pred_;
  1 Optional<iter_reference_t<I>> prev_;

public:
    auto next() -> Optional<iter_reference_t<I>> {
      2 auto val = base_.next();
        if (not val) {
            return nullopt;
        }

        if (invoke(pred_, *prev_, *val)) {
            return val;
        } else {
            // ...
        }
    }
};
```

Iterator Languages: `getlines`

```
class getlines {
    std::istream* stream_;
    std::string value_;

public:
    auto next() -> Optional<std::string&> {
        if (std::getline(*stream_, value_)) {
            return value_;
        }
        return nullopt;
    }
};
```

Always same reference

Iterator Languages: `group_by`

```
template <rust_iterator I, indirect_binary_predicate<I> P>
class group_by_inner {
    I base_;
    P pred_;
  1 Optional<iter_reference_t<I>> prev_;

public:
    auto next() -> Optional<iter_reference_t<I>> {
      2 auto val = base_.next();
        if (not val) {
            return nullopt;
        }

        if (invoke(pred_, *prev_, *val)) {
            return val;
        } else {
            // ...
        }
    }
};
```

Iterator Languages: `group_by`

```cpp
template <rust_iterator I, indirect_binary_predicate<I> P>
class group_by_inner {
    I base_;
    P pred_;
    Optional<iter_value_t<I>> prev_;

public:
    auto next() -> Optional<iter_reference_t<I>> {
        auto val = base_.next();
        if (not val) {
            return nullopt;
        }

        if (invoke(pred_, *prev_, *val)) {
            return val;
        } else {
            // ...
        }
    }
};
```

Iterator Languages: `group_by`

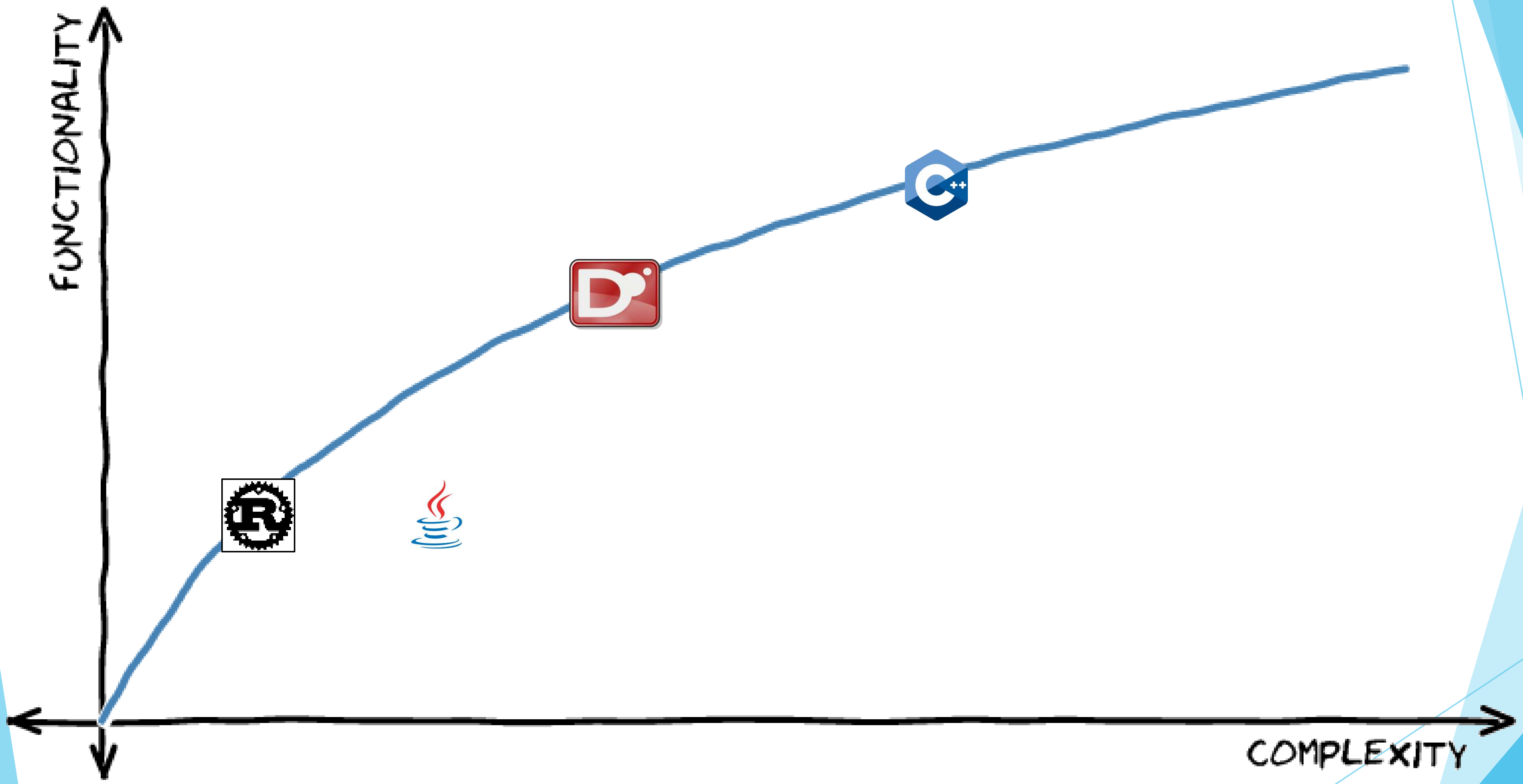
```
template <rust_iterator I, indirect_binary_predicate<I> P>
class group_by_inner {
    I base_;
    P pred_;
    Optional<iter_value_t<I>> prev_;

public:
    auto next() -> Optional<iter_reference_t<I>> {
        auto val = base_.next();
        if (not val) {
            return nullopt;
        }

        if (invoke(pred_, *prev_, *val)) {
            prev_ = *val;
            return val;
        } else {
            // ...
        }
    }
};
```

Copy *every* element

Iterator languages: functional gaps

- No container/iterator cohesion
- What about algorithms?
  - No binary `group_by`
  - No `adjacent_find` or `adjacent_difference`
  - No `sort`
  - No `slide`
  - No `search` or `mismatch` or `find_end`
  - No `lower_bound`, `upper_bound`, `equal_range`, `binary_search`
  - No `next_permutation`, `prev_permutation`
  - No `stable_partition` (`partition` returns index)
  - No `min_element`, `max_element`, `minmax_element`

```
vector<string> names = {"Bob", "Steve", "Jane"};
vector<int> ages = {37, 27, 31};

// after this:
//   ages = [27, 31, 37 ]
//   names = [Steve, Jane, Bob]
ranges::sort(views::zip(ages, names));
```


FUNCTIONALITY
COMPLEXITY

Questions?