



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

Barry Revzin



CODE RECKONS

Science to the CORE

About Me

- ▶ C++ Software Developer at Jump Trading since 2014



jumptrading

About Me



jumptrading

- ▶ C++ Software Developer at Jump Trading since 2014
- ▶ WG21 participant since 2016
 - ▶ C++20: `<=>`, `[...args=args]{}
<code>explicit(bool)</code>, conditionally trivial`
 - ▶ C++23: Deducing this, `if constexpr`
 - ▶ Bunch of ranges papers



About Me



jumptrading

- ▶ C++ Software Developer at Jump Trading since 2014
- ▶ WG21 participant since 2016
 - ▶ C++20: `<=>`, `[...args=args]{}
<code>explicit(bool)</code>, conditionally trivial`
 - ▶ C++23: Deducing this, `if constexpr`
 - ▶ Bunch of ranges papers



<https://brevzin.github.io/>



@BarryRevzin



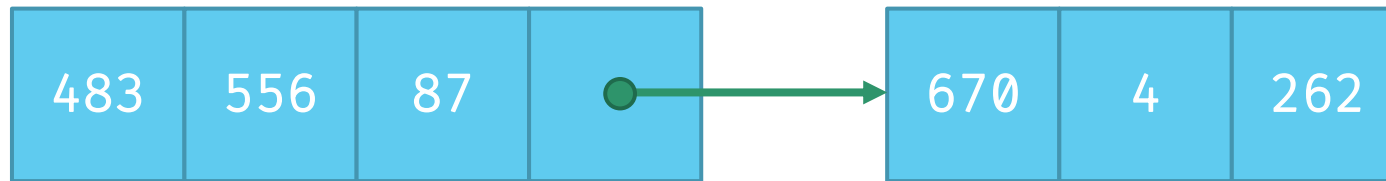
Barry



We have some sequence...

483	556	87	670	4	262
-----	-----	----	-----	---	-----

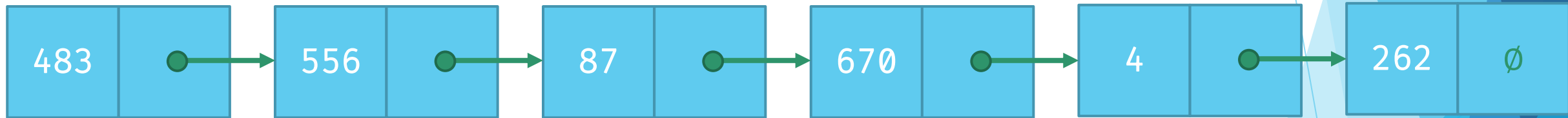
We have some sequence...



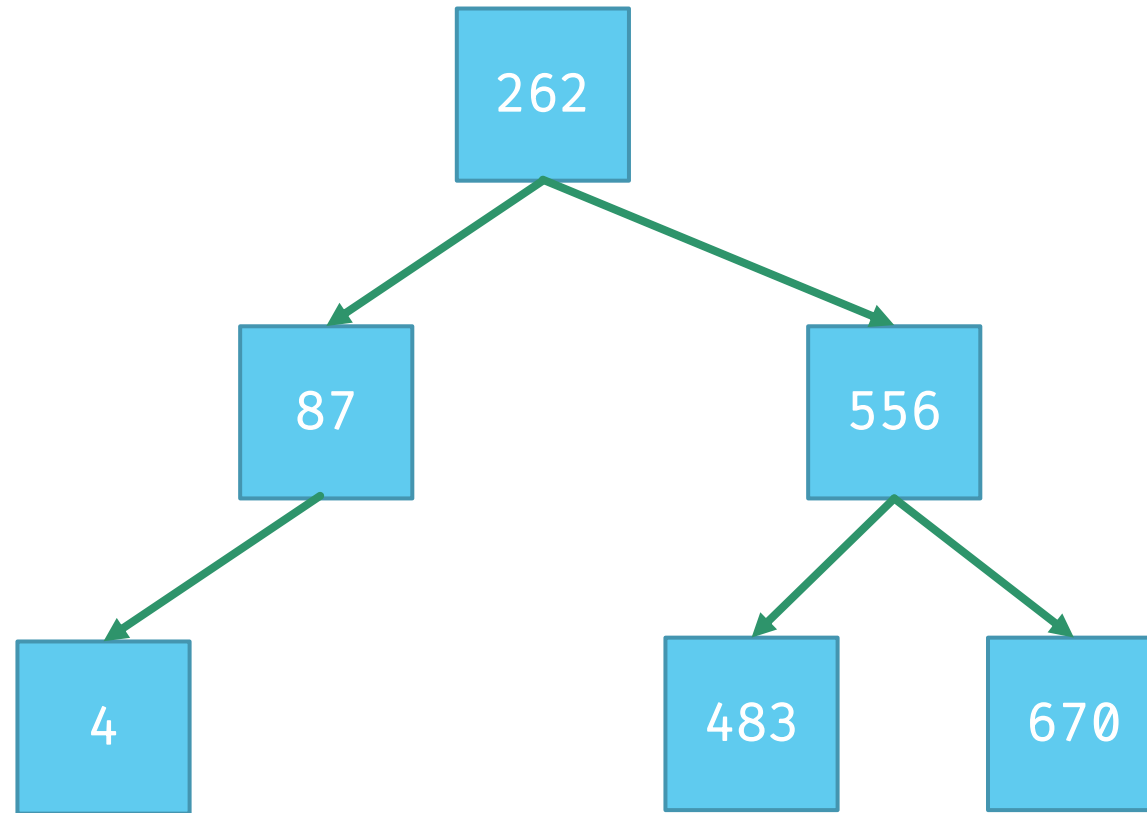
We have some sequence...

483	556	87	∅	670	∅	4	262
-----	-----	----	---	-----	---	---	-----

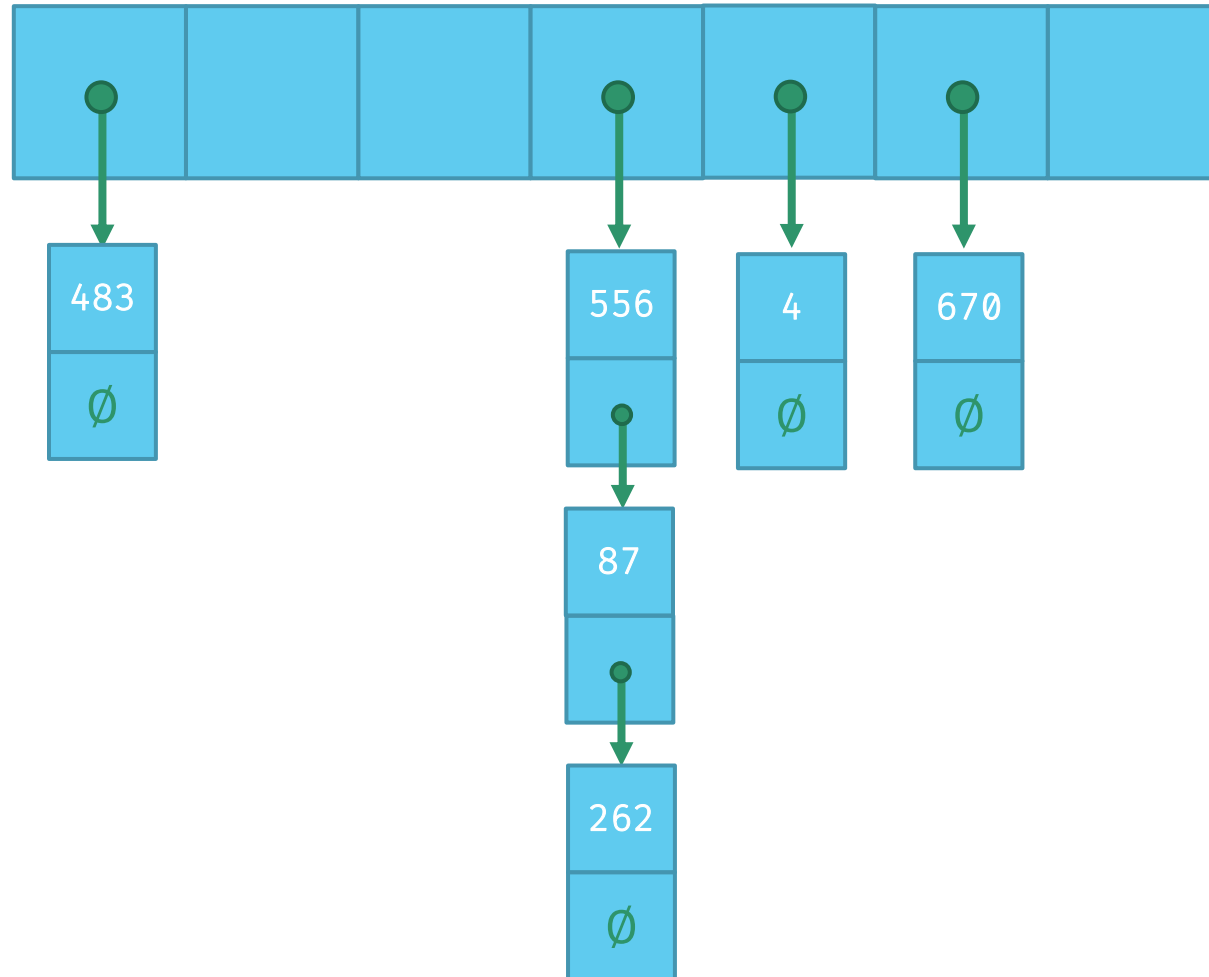
We have some sequence...



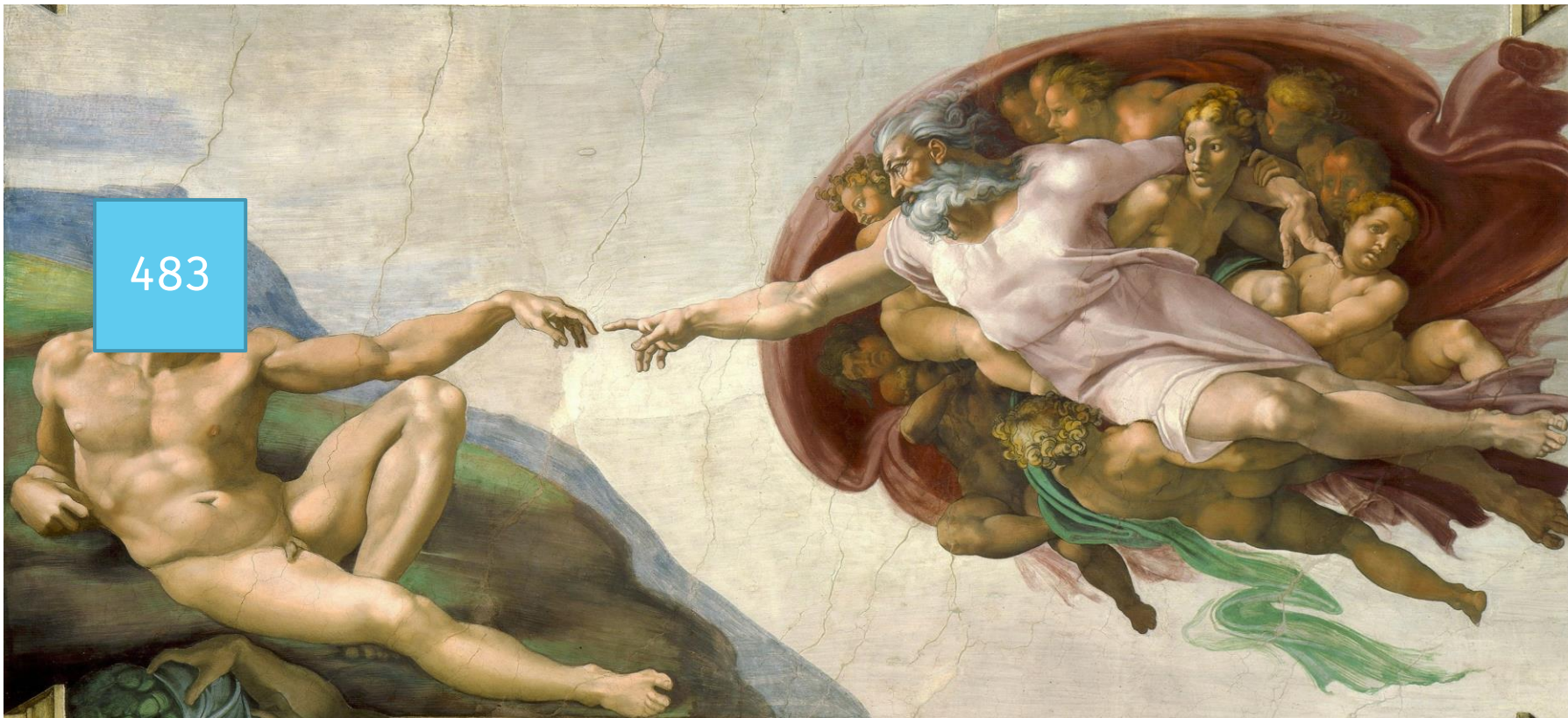
We have some sequence...



We have some sequence...



We have some sequence...



Need some **uniform** access pattern

Basis Operations

- ▶ read
- ▶ advance
- ▶ done?

Basis Operations

- ▶ read
- ▶ advance
- ▶ done?

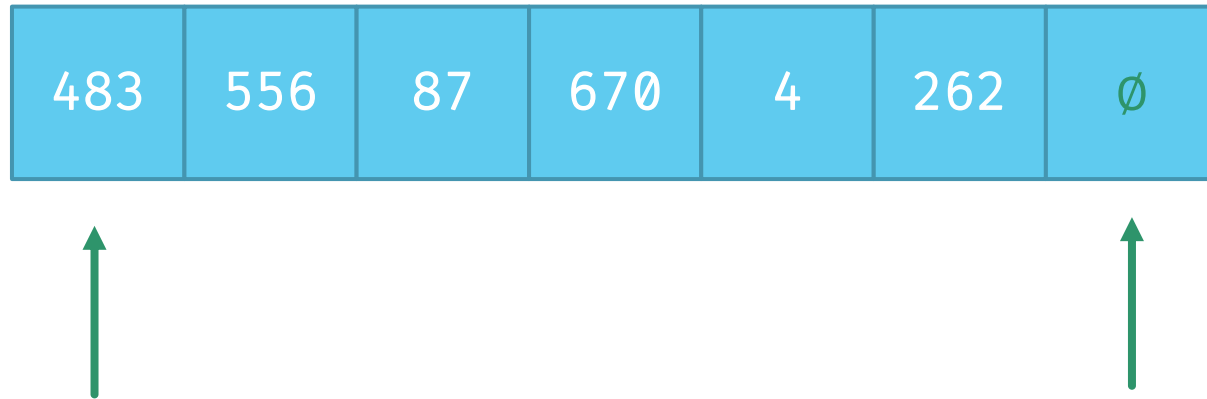
Not necessarily three **distinct** functions!



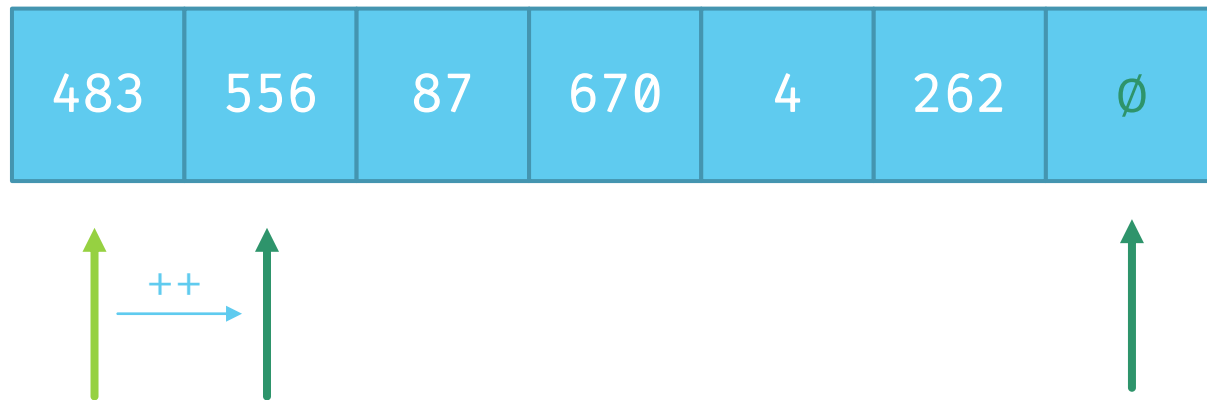
The C++ Iterator Pair Model

From Stepanov with Love

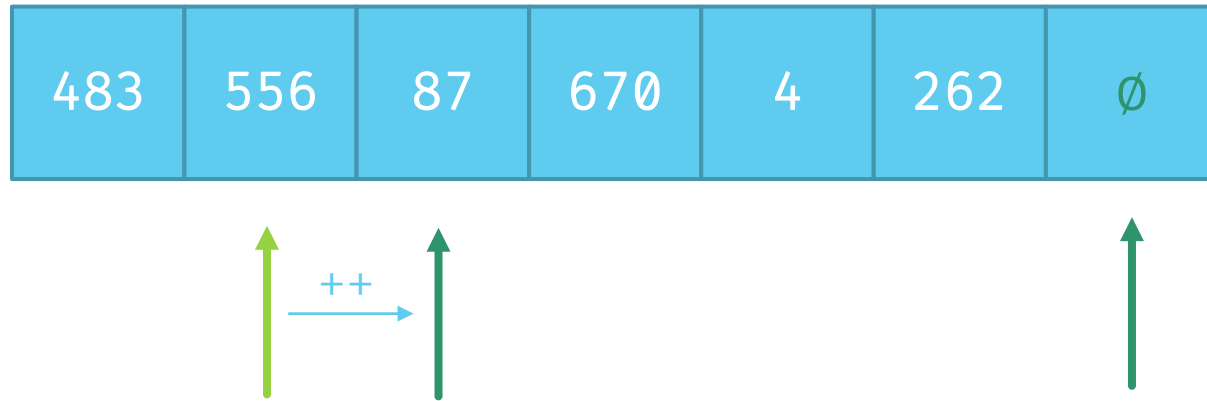
The C++ Iterator Pair Model



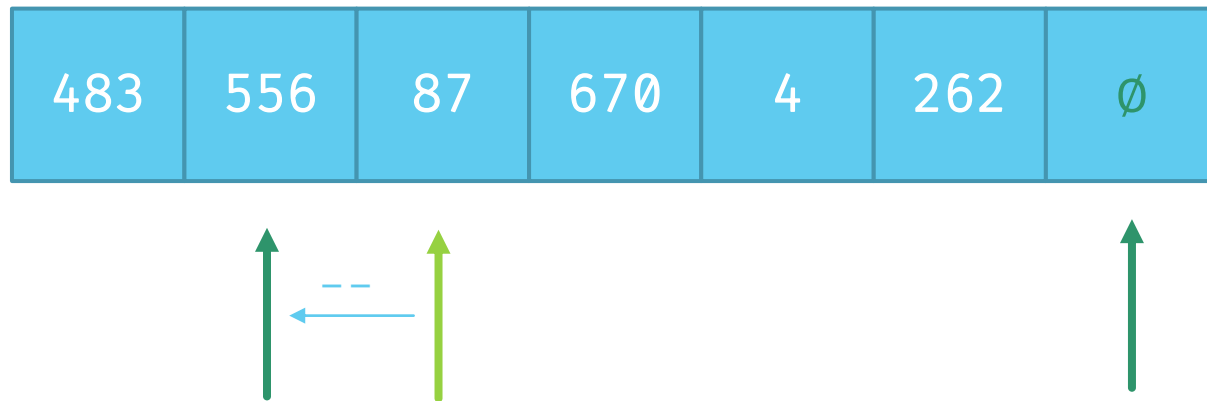
The C++ Iterator Pair Model



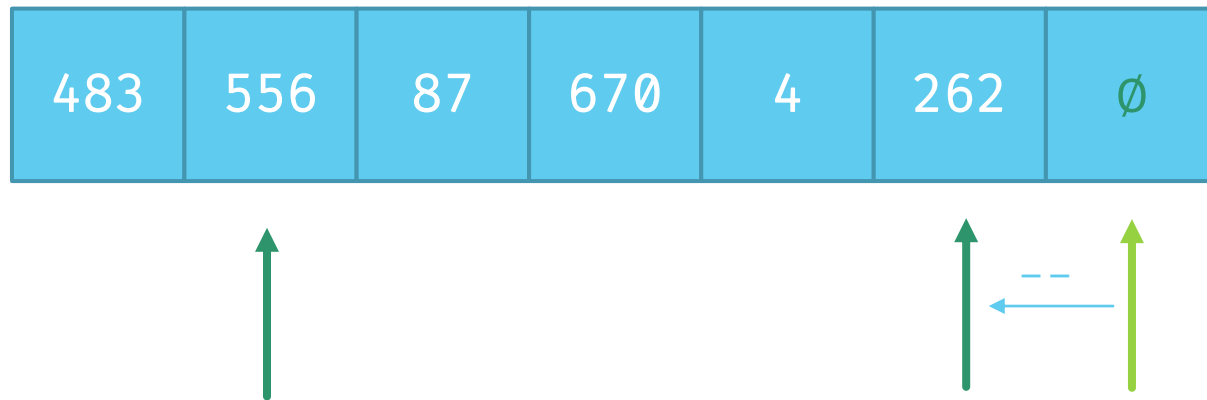
The C++ Iterator Pair Model



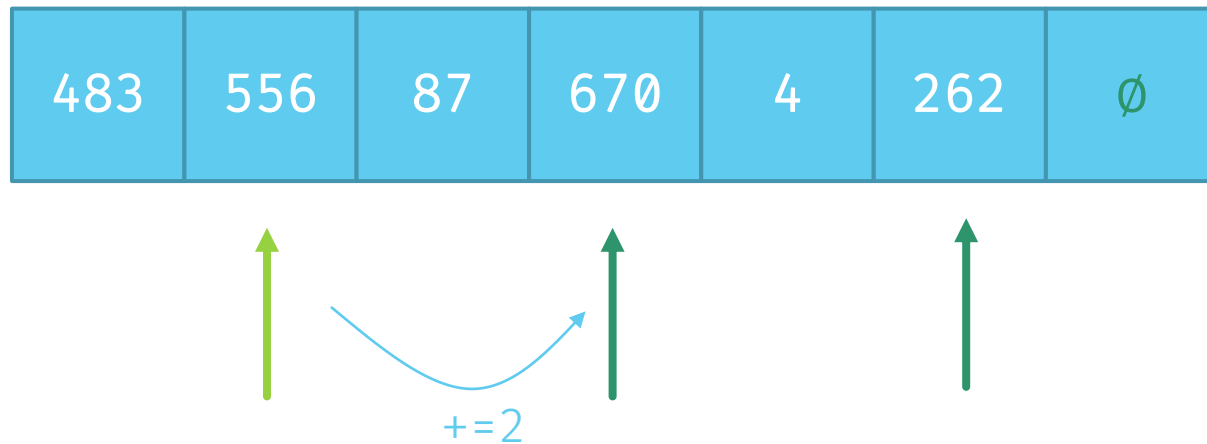
The C++ Iterator Pair Model



The C++ Iterator Pair Model



The C++ Iterator Pair Model



The C++ Iterator Pair Model



	read	advance	done?
	<code>*it</code>	<code>++it</code>	<code>it == last</code>



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)

```
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++(int) -> Iterator {
        auto tmp = *this;
        ++*this;
        return tmp;
    }

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

Diagram illustrating the implementation of the `map_view` iterator:

- read**: Points to the `operator*` function, which returns the value at the current position.
- advance**: Points to the `operator++` functions, which increment the iterator.
- done?**: Points to the `operator==` function, which checks if the iterator has reached the end.

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++(int) -> 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++(int) -> 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)



```
1 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;

3       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++(int) -> Iterator { auto tmp = *this; ++*this; return tmp; }
           auto operator==(Iterator const& rhs) const -> bool { return base_ == rhs.base_; }
       };

5       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>>;

11          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:
12          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_;
```

read

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

advance

```
    auto operator++(int) -> Iterator {
        auto tmp = *this;
        ++*this;
        return tmp;
```

done?

```
    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_; }
};
```

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++



```
1 template <input_range V, indirect_unary_predicate<iterator_t<V>> F> requires view<V>
   class filter_view {
2   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>;

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

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

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

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

   public:
       10 filter_view(V, F);

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

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



The D Ranges Model

Iterators Must Go

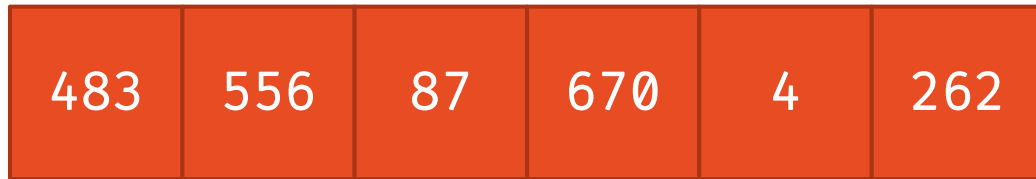
The D Ranges Model



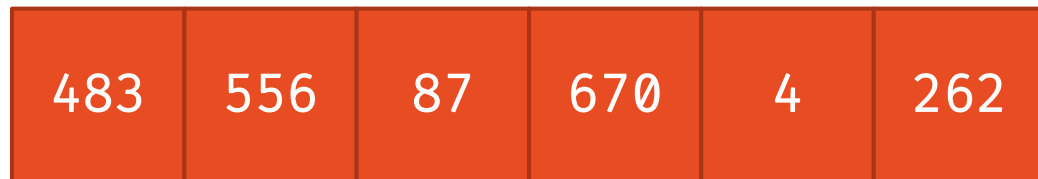
483	556	87	670	4	262
-----	-----	----	-----	---	-----



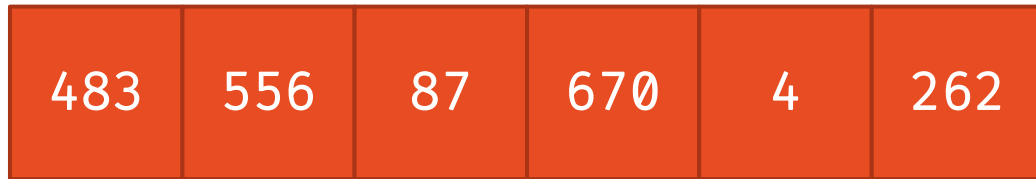
The D Ranges Model



The D Ranges Model



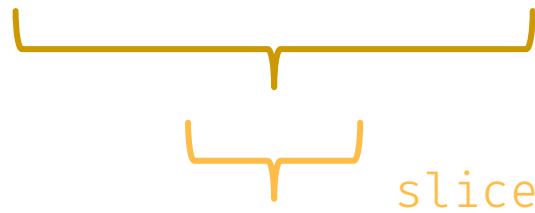
The D Ranges Model



The D Ranges Model





483	556	87	670	4	262
-----	-----	----	-----	---	-----



The D Ranges Model



	read	advance	done?
	<code>*it</code>	<code>++it</code>	<code>it == last</code>
	<code>r.front()</code>	<code>r.popFront()</code>	<code>r.empty()</code>



Basic D Range Structure

```
1 struct Range {  
    using reference      = /* ... */;  
    using value_type     = /* ... */;  
    using range_category = /* ... */;  
    using difference_type = /* ... */;  
  
    1 auto front() -> reference;  
    2 void popFront();  
    3 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>;
```

1 map_range(R, F);

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

Diagram: A red box labeled "read" has an arrow pointing to base_.front(). A red line connects the box to the base_.front() expression.

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

Diagram: A red box labeled "advance" has an arrow pointing to base_.popFront(). A red line connects the box to the base_.popFront() expression.

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

Diagram: A red box labeled "done?" has an arrow pointing to base_.empty(). A red line connects the box to the base_.empty() expression.

};

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(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

```
1 template <input_drange R, indirect_unary_predicate<R> P>
   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()	

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

advance && done?

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;
}
```

A diagram consisting of two purple arrows. One arrow originates from the `invoke` parameter in the `Current()` method of `map_enumerator` and points to the `invoke` parameter in the `Current()` method of `filter_enumerator`. The second arrow originates from the `invoke` parameter in the `MoveNext()` method of `filter_enumerator` and points to the `invoke` parameter in the `Current()` method of `map_enumerator`. This illustrates how the `filter_enumerator` class relies on the `map_enumerator` class's `invoke` method for both its `Current()` and `MoveNext()` operations.

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 until_range{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



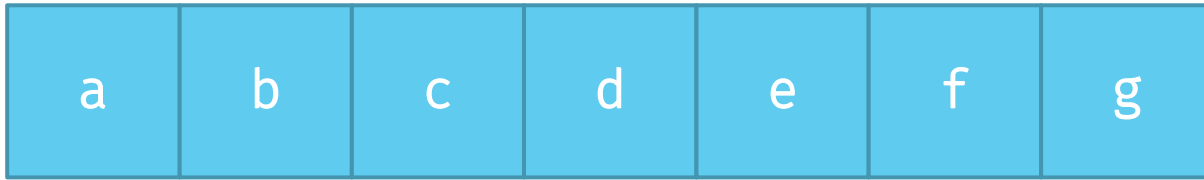
- Consider: I want the range *until* the first element that satisfies a predicate

```
// 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



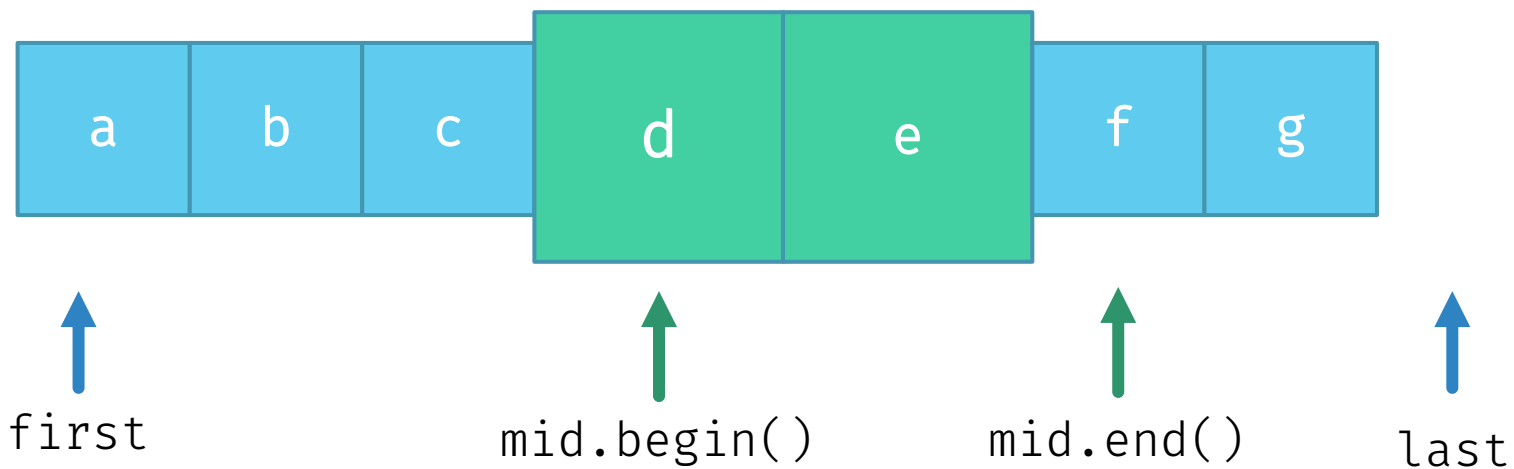
C++ Iterators vs D Ranges: splitting



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

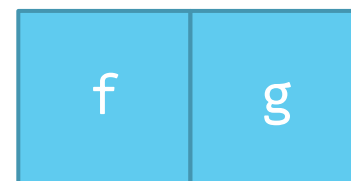
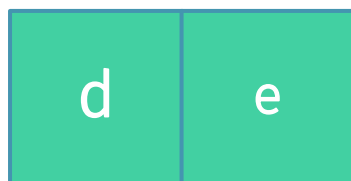
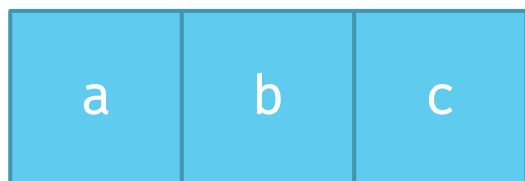


C++ Iterators vs D Ranges: splitting



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

C++ Iterators vs D Ranges: splitting



↑
first

↑
mid.begin()

↑
mid.end()

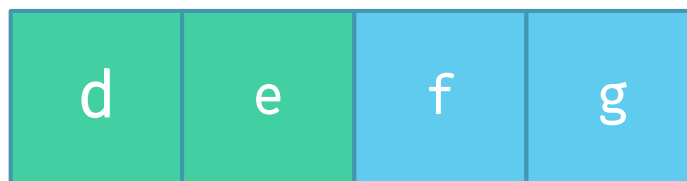
↑
last

```
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



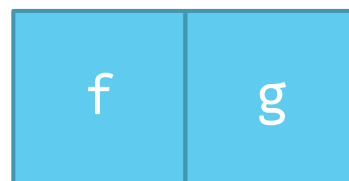
↑
first



↑ ↑ ↑
mid.begin() mid.end() last

```
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



↑
first

↑
mid.begin()

↑
mid.end()

↑
last

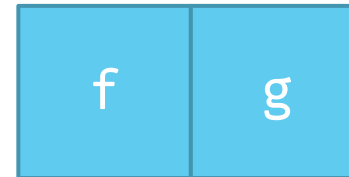
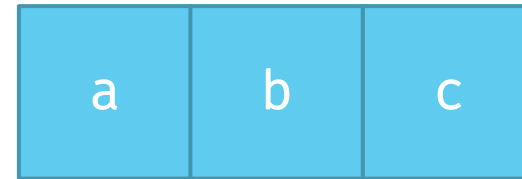
```
template <forward_iterator I, sentinel_for<I> S,  
         forward_iterator I2, sentinel_for<I2> S2>  
auto find_split_after(I first, S last, I2 first2, S2 last2) {  
    auto mid = ranges::search(first, last, first2, last2);  
    auto pre = ranges::subrange(first, mid.end());  
    auto post =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

C++ Iterators vs D Ranges

► findSplit

```
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]),
                       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(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);
    }
}
```

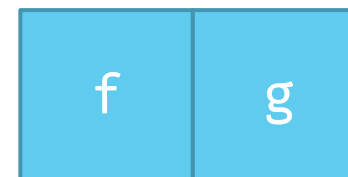
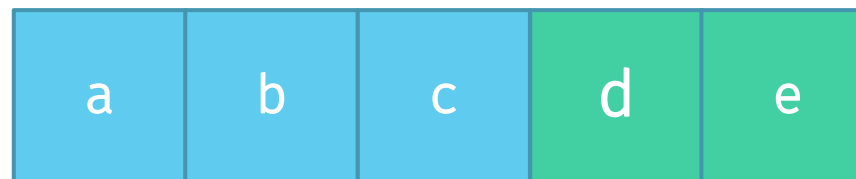


C++ Iterators vs D Ranges

► findSplitAfter

```
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 = typeid(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;
    }

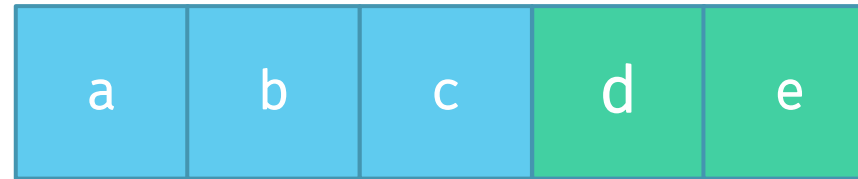
    static if (isSomeString!R1 && isSomeString!R2
        || (isRandomAccessRange!R1 && hasLength!R1 && hasSlicing!R1 && hasLength!R2))
    {
        auto balance = find!pred(haystack, needle);
        immutable pos = balance.empty ? 0 : haystack.length - balance.length + needle.length;
        return Result!(typeof(haystack[0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

```
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 = tuple(pre, post);
        }
        void opAssign(tuple 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]);
}
```

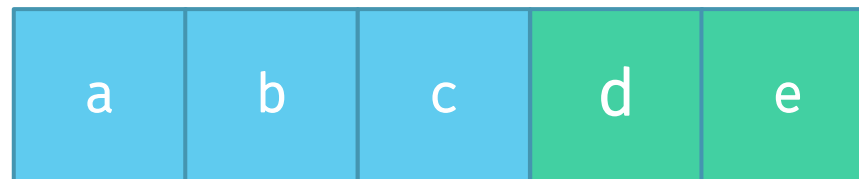
```
if (BinaryFun!pred(h, n, pos1))
{
    h.popFront();
    n.popFront();
    ++pos2;
}
else
{
    haystack.popFront();
    n = needle.save;
    h = haystack.save;
    pos2 = ++pos1;
}
return Result!(typeof(takeExactly(original, pos2)),
               typeof(h))(takeExactly(original, pos2),
        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



C++ Iterators vs D Ranges

► findSplitAfter

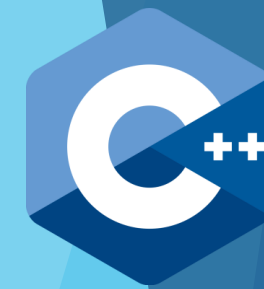
```
auto findSplitAfter(alias pred = "a == b", R1, R2)(R1 haystack, R2 needle)
if (isForwardRange!R1 && isForwardRange!R2)
{
    static struct Result(S1, S2) if (isForwardRange!S1 &&
                                     isForwardRange!S2)
    {
        this(S1 pre, S2 post)
        {
            asTuple = tuple(pre, post);
        }
        void opAssign(tuple 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!(/* ... */)(haystack[0 .. pos], haystack[pos .. haystack.length]);
}
```

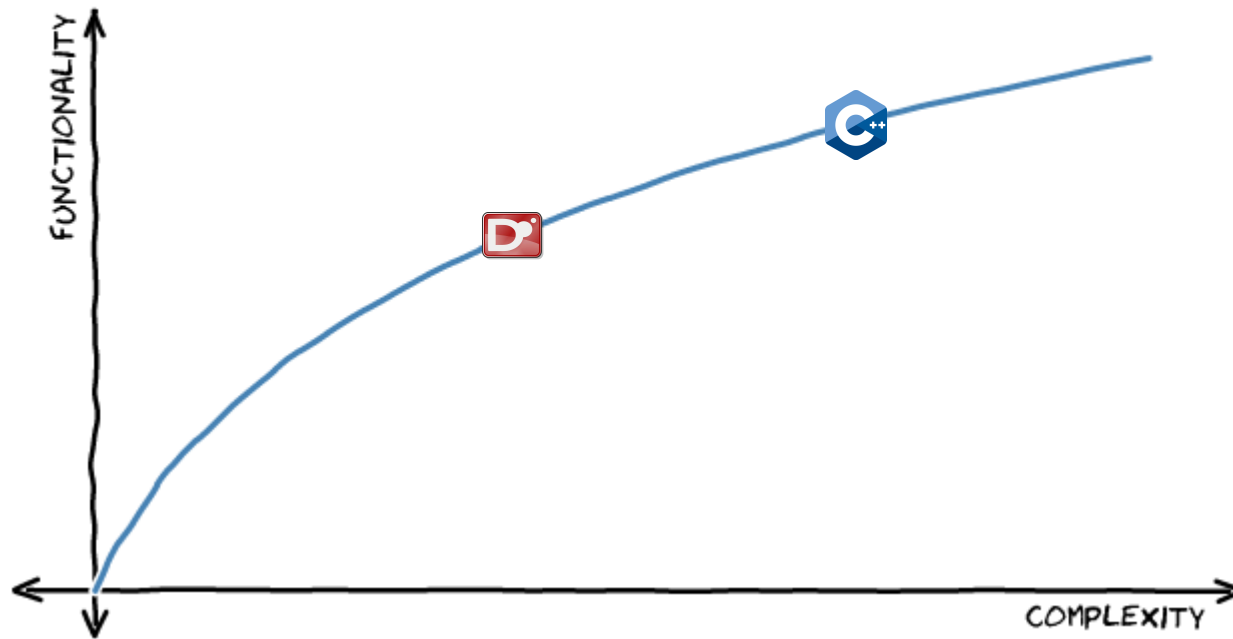
`ranges::subrange(first, mid.end());`

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



C++ Iterators vs D Ranges

ITERATION MODELS





The Rust Iterator Model

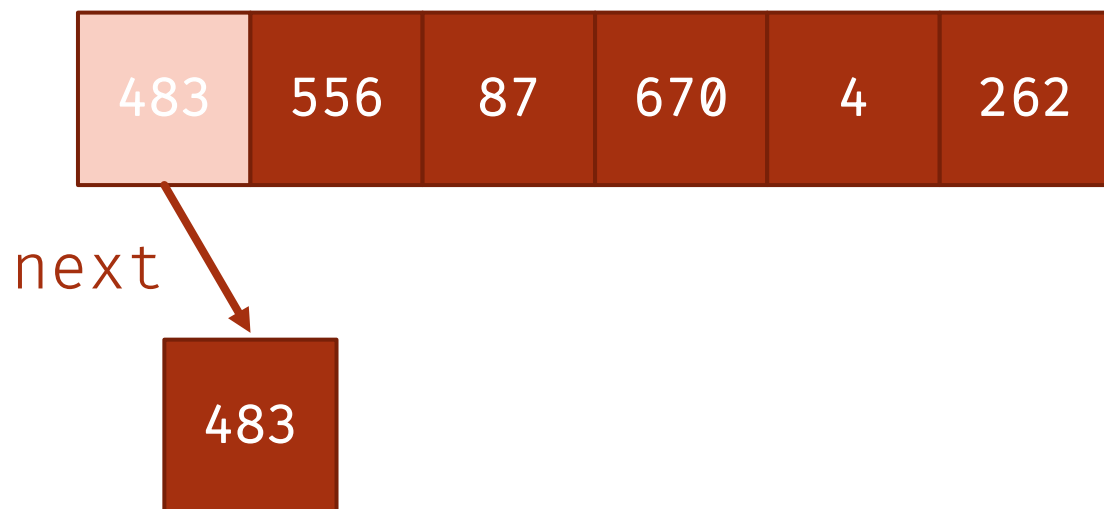
Have you thought about just rewriting everything in Rust?

The Rust Iterator Model

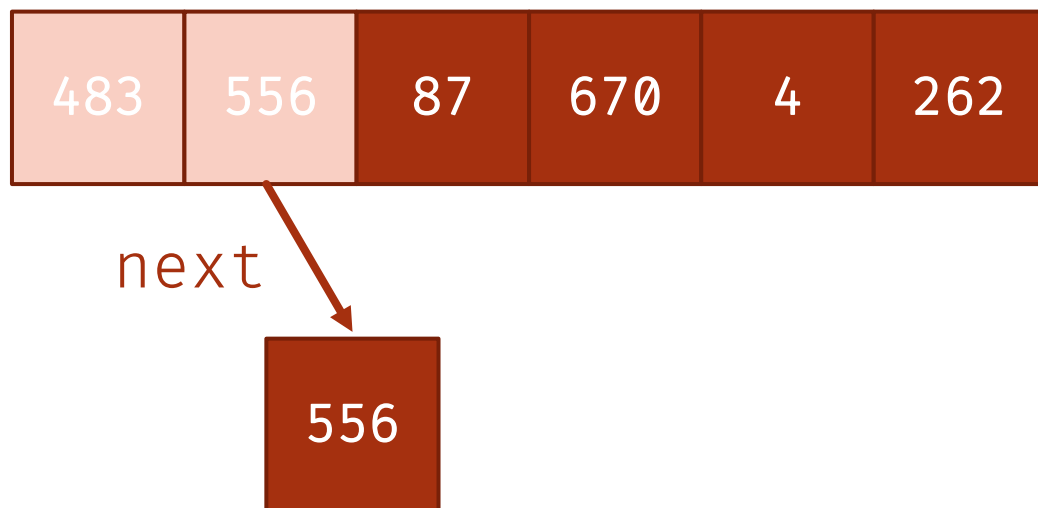


483	556	87	670	4	262
-----	-----	----	-----	---	-----

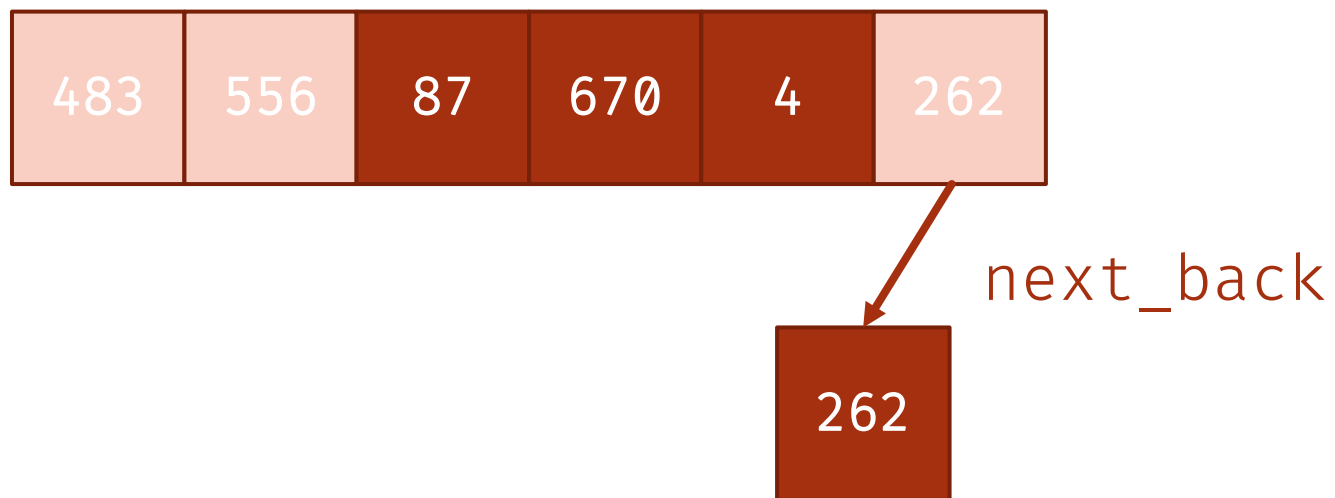
The Rust Iterator Model







The Rust Iterator Model



The Rust Iterator Model



The Rust Iterator Model

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



Basic Rust Iterator Structure

```
1 struct Iterator {  
    using reference = /* ... */;  
    using value_type = /* ... */;  
    using difference_type = /* ... */;  
  
    1 auto next() -> Optional<reference>;  
};
```

consumed

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>>>
1 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>;

    1 map_iterator(I, F);

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

Diagram illustrating the implementation of the `map` function in Rust. A box labeled "everything" points to the `next()` method call in the `map_iterator` class, indicating that the entire expression `base_.next().map(fun_)` is handled by the `next()` method.



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?
	<code>*it</code>	<code>++it</code>	<code>it == last</code>
	<code>r.front()</code>	<code>r.popFront()</code>	<code>r.empty()</code>
	<code>e.Current()</code>	<code>e.MoveNext()</code>	
	<code>it.next()</code>		









The Rust/Python Iterator Model

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



The Rust/Python/Java/... Iterator Model

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

↑ 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 NoSuchElementException();
        }
        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 NoSuchElementException();
        }
        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 NoSuchElementException();
        }

        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);  
    }  
}
```

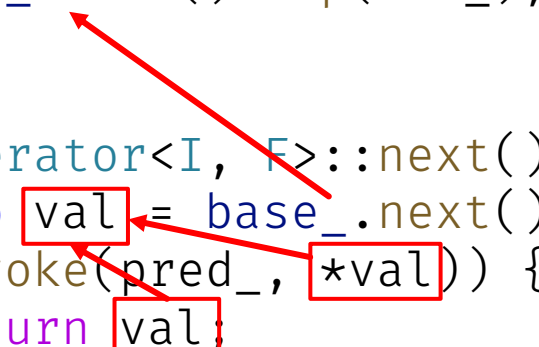
How many times is
some_operation
invoked??

Exactly 6.



Iterator Languages

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

A diagram illustrating the relationship between the two iterator types. A blue arrow points from the `base_.next()` call in the `filter_iterator` to the `invoke` call in the `map_iterator`. Another blue arrow points from the `val` variable in the `filter_iterator` to the `val` argument in the `invoke` call in the `filter_iterator`. The `val` variable is highlighted with a blue box in both locations.

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 NoSuchElementException();
        }
        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?
 - ▶ A copyable **Rust Iterator** is isomorphic to a **C++ Forward Range**
 - ▶ A non-copyable **Rust Iterator** is isomorphic to a **C++ Input Range**

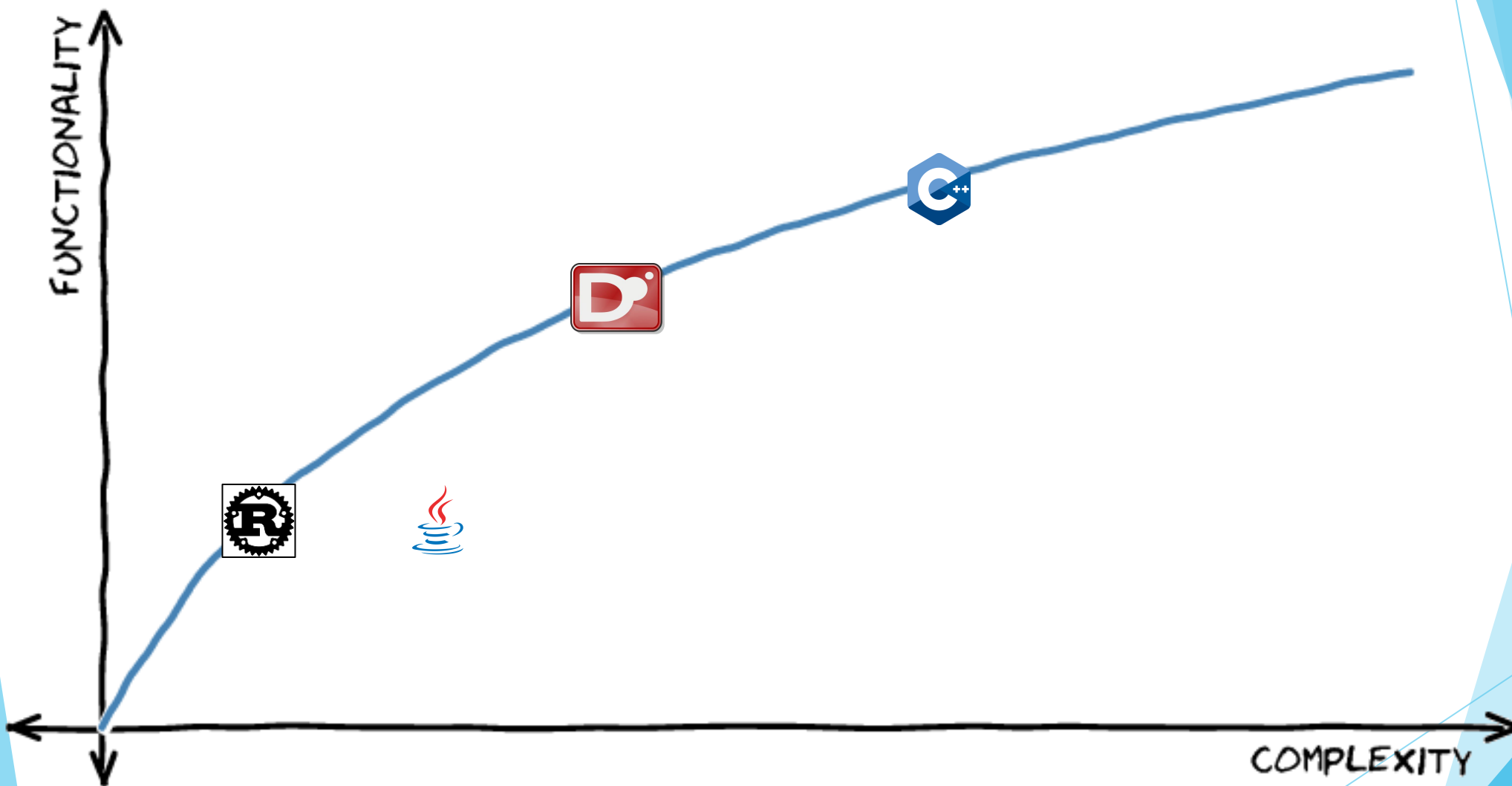


Iterator Languages: functional gaps



- ▶ No container/iterator cohesion
- ▶ What about algorithms? **Forward** only!

- ▶ No sort
 - `vector<string> names = {"Bob", "Steve", "Jane"};`
- ▶ No lower_bound, upper_bound, equal_range, binary_search
 - `vector<int> ages = {37, 27, 31};`
- ▶ No next_permutation, prev_permutation
 - `// after this:`
- ▶ No stable_partition or rotate (partition returns index)
 - `// ages = [27, 31, 37]`
- ▶ No min_element, max_element, minmax_element (questionable)
 - `// names = [Steve, Jane, Bob]`
 - `ranges::sort(views::zip(ages, names));`



Bonus Traversal Model



Basis Operations

- ▶ read
- ▶ advance
- ▶ done?

Not necessarily three **distinct** functions!

External Iteration Basis Operations

- ▶ read
- ▶ advance
- ▶ done?

Not necessarily three **distinct** functions!

Basic External Iteration Structure



```
template <input_iterator I, sentinel_for<I> S>
void print_all(I it, S last) {
    for (; it != last; ++it) {
        fmt::print("{}\n", *it);
    }
}
```

Basic External Iteration Structure



```
template <input_iterator I, sentinel_for<I> S>
void print_all(I it, S last) {
    for (; it != last; ++it) {
        fmt::print("{}\n", *it);
    }
}

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

Loop is
outside of
the iterator

Basic Internal Iteration Structure



```
template <stream S>
void print_all(S stream) {
    stream.for_each([](auto&& elem) {
        fmt::print("{}\n", elem);
    });
}
```

Loop is *inside*
the iterator

Stream from C++ Iterator Pair



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

public:
    using reference = iter_reference_t<I>;

    template <invocable<reference> F>
    void for_each(F f) {
        for (; first != last; ++first) {
            invoke(f, *first);
        }
    }
};
```

Stream from C++ Iterator Pair

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

public:
    using reference = iter_reference_t<I>;

    template <predicate<reference> P>
    void for_each(P pred) {
        for (; first != last; ) {
            if (not invoke(pred, *first++)) {
                break;
            }
        }
    }
};
```



Stream from C++ Iterator Pair

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

public:
    using reference = iter_reference_t<I>;

    template <predicate<reference> P>
    void while_(P pred) {
        while (first != last) {
            if (not invoke(pred, *first++)) {
                break;
            }
        }
    }
};
```



Stream from C++ Iterator Pair



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

public:
    using reference = iter_reference_t<I>;

    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        while (first != last) {
            if (not invoke(pred, *first++)) {
                return false;
            }
        }
        return true;
    }
};
```

Stream from C++ Iterator Pair

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

consumed

```
public:
    using reference = iter_reference_t<I>;

    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        while (first != last) {
            SCOPE_EXIT { ++first; };
            if (not invoke(pred, *first)) {
                return false;
            }
        }
        return true;
    }
};
```



Stream from Rust Iterator

```
template <rust_iterator I>
class rust_stream {
    I it;

public:
    using reference = I::reference;

    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        while (auto elem = it.next()) {
            if (not invoke(pred, *elem)) {
                return false;
            }
        }
        return true;
    }
};
```



Implementing map with Streams



```
template <stream S, invocable<stream_reference_t<S>> F>
class map_stream {
    S stream;
    F proj;

public:
    using reference = invoke_result_t<F&, stream_reference_t<S>>;

    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        return stream.while_([&](stream_reference t<S> elem){
            return invoke(pred, invoke(proj, elem));
        });
    }
};
```

Implementing map with Streams



```
template <stream S, invocable<stream_reference_t<S>> F>
class map_stream {
    S stream;
    F proj;

public:
    using reference = invoke_result_t<F&, stream_reference_t<S>>;

    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        return stream.while_(compose(pred, proj));
    }
};
```

Implementing filter with Streams



```
template <stream S, predicate<stream_reference_t<S>> F>
class filter_stream {
    S stream;
    F filt;
```

```
public:
    using reference = stream_reference_t<S>;
```

```
    template <predicate<reference> P>
    auto while_(P pred) -> b
        return stream.while_
            if (invoke(filt,
                return invoke
            } else {
                return true;
            }
        });
};
```

For a work in progress implementation, see
<https://github.com/brevzin/rivers/>

Inspired by Joaquín M López Muñoz's
<https://github.com/joaquintides/transrangers>



Why Streams?

Is there some advantage to internal iteration?

Consuming a pipeline



```
ranges::for_each(r | views::filter(pred), f);
```

Consuming a pipeline



```
auto filtered = r | views::filter(pred);  
auto first = ranges::begin(filtered);  
auto last = ranges::end(filtered);  
  
for (; first != last; ++first) {  
    invoke(f, *first);  
}
```

Consuming a pipeline



```
auto filtered = r | views::filter(pred);  
auto first = ranges::begin(filtered);  
auto last = ranges::end(filtered);  
  
while (first != last) {  
    invoke(f, *first);  
    ++first;  
}
```

Consuming a pipeline



```
auto first = ranges::begin(r);
auto last = ranges::end(r);
first = ranges::find_if(first, last, pred);

while (first != last) {
    invoke(f, *first);
    ++first;
    first = ranges::find_if(first, last, pred);
}
```

Consuming a pipeline



```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}

while (first != last) {
    invoke(f, *first);
    ++first;
    first = ranges::find_if(first, last, pred);
}
```

Consuming a pipeline

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}
```

```
while (first != last) {
    invoke(f, *first);
    ++first;
    while (first != last) {
        if (invoke(pred, *first)) {
            break;
        }
        ++first;
    }
}
```



Consuming a pipeline

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}

while (first != last) {
    invoke(f, *first);
    ++first;
    while (first != last) {
        if (invoke(pred, *first)) {
            break;
        }
        ++first;
    }
}
```

```
rvr::from(r)
    .filter(pred)
    .for_each(f);
```



Consuming a pipeline

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}

while (first != last) {
    invoke(f, *first);
    ++first;
    while (first != last) {
        if (invoke(pred, *first)) {
            break;
        }
        ++first;
    }
}
```

```
rvr::from(r)
    .filter(pred)
    .while_([&](auto&& elem) {
        invoke(f, elem);
        return true;
    });
```



Consuming a pipeline

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}

while (first != last) {
    invoke(f, *first);
    ++first;
    while (first != last) {
        if (invoke(pred, *first)) {
            break;
        }
        ++first;
    }
}
```

```
rvr::from(r)
    .while_([&](auto&& elem) {
        if (invoke(pred, elem)) {
            invoke(f, elem);
            return true;
        } else {
            return true;
        }
    });
```



Consuming a pipeline

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}

while (first != last) {
    invoke(f, *first);
    ++first;
    while (first != last) {
        if (invoke(pred, *first)) {
            break;
        }
        ++first;
    }
}
```

```
rvr::from(r)
    .while_([&](auto&& elem) {
        if (invoke(pred, elem)) {
            invoke(f, elem);
        }
        return true;
    });
```



Consuming a pipeline

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *first)) {
        break;
    }
    ++first;
}

while (first != last) {
    invoke(f, *first);
    ++first;
    while (first != last) {
        if (invoke(pred, *first)) {
            break;
        }
        ++first;
    }
}
```

```
auto first = ranges::begin(r);
auto last = ranges::end(r);
while (first != last) {
    if (invoke(pred, *elem)) {
        invoke(f, *elem);
    }
    ++first;
}
```



Implementing chain in C++



- Concatenate multiple ranges together

```
std::vector<int> a = {1, 2, 3};  
std::list<int> b = {4, 5};  
std::vector<int> c = {6, 7, 8};
```

```
auto stuff = views::chain(a, b, c); // {1, 2, 3, 4, 5, 6, 7, 8}
```

Implementing chain in C++

```
template <input_view... Vs>
class chain_view<Vs...>::iterator {
    variant<iterator_t<Vs>...> cur_;
    tuple<Vs...>* parent_;
};
```



Implementing chain in C++



```
template <input_view... Vs>
class chain_view<Vs...>::iterator {
    variant<iterator_t<Vs>...> cur_;
    tuple<Vs...>* parent_;

public:
    auto operator*() const -> reference { /* ... */ }

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

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



Implementing chain in C++

```
template <input_view... Vs>
class chain_view<Vs...>::iterator {
    variant<iterator_t<Vs>...> cur_;
    tuple<Vs...>* parent_;

public:
    auto operator*() const -> reference {
        return std::visit([](auto it) -> reference { return *it; }, cur_);
    }

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

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



Implementing chain in C++

```
template <input_view... Vs>
class chain_view<Vs...>::iterator {
    variant<iterator_t<Vs>...> cur_;
    tuple<Vs...>* parent_;

public:
    auto operator*() const -> reference {
        return std::visit([](auto it) -> reference { return *it; }, cur_);
    }

    auto operator++() -> iterator& {
        return std::visit([](auto& it) {
            // how do you differentiate *which* vector<int>::iterator?
        }, cur_);
        return *this;
    }

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




Implementing chain in C++

```
template <input_view... Vs>
class chain_view<Vs...>::iterator {
    variant<iterator_t<Vs>...> cur_;
    tuple<Vs...>* parent_;

public:
    auto operator*() const -> reference {
        return std::visit([](auto it) -> reference { return *it; }, cur_);
    }

    auto operator++() -> iterator& {
        mp_with_index<sizeof...(Vs)>(cur_.index(), [&](auto I){
            ++std::get<I>(cur_);
            maybe_increment<I>();
        });
        return *this;
    }

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



Implementing chain in C++

```
template <input_view... Vs>
class chain_view<Vs...>::iterator {
    variant<iterator_t<Vs>...> cur_;
    tuple<Vs...>* parent_;

    template <size_t I> void maybe_increment() {
        if constexpr (I + 1 < sizeof...(Vs)) {
            if (std::get<I>(cur_) == ranges::end(std::get<I>(parent_))) {
                cur_ = ranges::begin(std::get<I+1>(parent_));
                maybe_increment<I+1>();
            }
        }
    }

public:
    auto operator*() const -> reference {
        return std::visit([](auto it) -> reference { return *it; }, cur_);
    }

    auto operator++() -> iterator& {
        mp_with_index<sizeof...(Vs)>(cur_.index(), [&](auto I){
            ++std::get<I>(cur_);
            maybe_increment<I>();
        });
        return *this;
    }

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

branch

branch

branch

branch

Implementing chain in Rust

```
template <input_iterator... Is>
class chain_iterator {
    tuple<Is...> cur_;
    size_t which_;
};
```



Implementing chain in Rust

```
template <input_iterator... Is>
class chain_iterator {
    tuple<Is...> cur_;
    size_t which_;

public:
    auto next() -> Optional<reference> {
        return mp_with_index<sizeof...(Is)>(which_, [&](auto I) {
            // ???
        });
    }
};
```



Implementing chain in Rust

```
template <input_iterator... Is>
class chain_iterator {
    tuple<Is...> cur_;
    size_t which_;

public:
    auto next() -> Optional<reference> {
        return mp_with_index<sizeof...(Is)>(which_, [&](auto I) {
            auto elem = std::get<I>(cur_).next();
            if (elem) {
                return elem;
            } else {
                // ???
            }
        });
    }
};
```



Implementing chain in Rust

```
template <input_iterator... Is>
class chain_iterator {
    tuple<Is...> cur_;
    size_t which_;

    auto next_one() -> Optional<reference> {
        return mp_with_index<sizeof...(Is)>(which_, [&](auto I) {
            return std::get<I>(cur_).next();
        });
    }

public:
    auto next() -> Optional<reference> {
        auto elem = next_one();
        if (elem) {
            return elem;
        } else {
            // ???
        }
    }
};
```



Implementing chain in Rust

```
template <input_iterator... Is>
class chain_iterator {
    tuple<Is...> cur_;
    size_t which_;

    auto next_one() -> Optional<reference> {
        return mp_with_index<sizeof...(Is)>(which_, [&](auto I) {
            return std::get<I>(cur_).next();
        });
    }

public:
    auto next() -> Optional<reference> {
        if (auto elem = next_one()) { return elem; }
        ++which_;
        return next();
    }
};
```



Implementing chain in Rust

```
template <input_iterator... Is>
class chain_iterator {
    tuple<Is...> cur_;
    size_t which_;

    auto next_one() -> Optional<reference> {
        return mp_with_index<sizeof...(Is)>(which_, [&](auto I) {
            return std::get<I>(cur_).next();
        });
    }
}
```

branch

```
public:
    auto next() -> Optional<reference> {
        for (; which_ < sizeof...(Is); ++which_) {
            if (auto elem = next_one()) { return elem; }
        }
        return nullopt;
    }
};
```

branch



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <invocable<reference> F>
    void for_each(F f) {
        // ???
    }
};
```



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <invocable<reference> F>
    void for_each(F f) {
        auto apply_one = [&](auto& s){
            s.for_each(f);
        };

        std::apply([&](S&... streams){
            (apply_one(streams), ...);
        }, streams_);
    }
};
```



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <invocable<reference> F>
    void for_each(F f) {
        auto apply_one = [&](auto& s){
            s.for_each(f);
        };

        tuple_for_each(streams_, apply_one);
    }
};
```



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <invocable<reference> F>
    void for_each(F f) {
        tuple_for_each(streams_, [&](auto& s){
            s.for_each(f);
        });
    }
};
```

zero branches!



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        tuple_for_each(streams_, [&](auto& s){
            s.while_(pred);
        });
        return true; // :-)
    }
};
```



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        bool keep_going = true;

        tuple_for_each(streams_, [&](auto& s){
            if (keep_going) {
                s.while_(pred);
            }
        });

        return true;
    }
};
```



Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        bool keep_going = true;

        tuple_for_each(streams_, [&](auto& s){
            if (keep_going) {
                s.while_([&](reference elem){
                    if (not invoke(pred, elem)) {
                        keep_going = false;
                        return false;
                    } else {
                        return true;
                    }
                });
            }
        });

        return keep_going;
    }
};
```



Implementing chain with Streams



```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        bool keep_going = true;

        tuple_for_each(streams_, [&](auto& s){
            if (keep_going) {
                keep_going = s.while_(pred);
            }
        });

        return keep_going;
    }
};
```

branch

Implementing chain with Streams



```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        bool keep_going = true;

        tuple_for_each(streams_, [&](auto& s){
            if (keep_going) {
                keep_going = s.while_(pred);
            }
        });

        return keep_going;
    }
};
```

branch
per stream

Implementing chain with Streams



```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        bool keep_going = true;

        tuple_for_each(streams_, [&](auto& s){
            if (keep_going) {
                keep_going = s.while_(pred);
            }
        });

        return keep_going;
    }
};
```

branch
per stream
easily optimizable

Implementing chain with Streams

```
template <stream... S>
class chain_stream {
    tuple<S...> streams_;

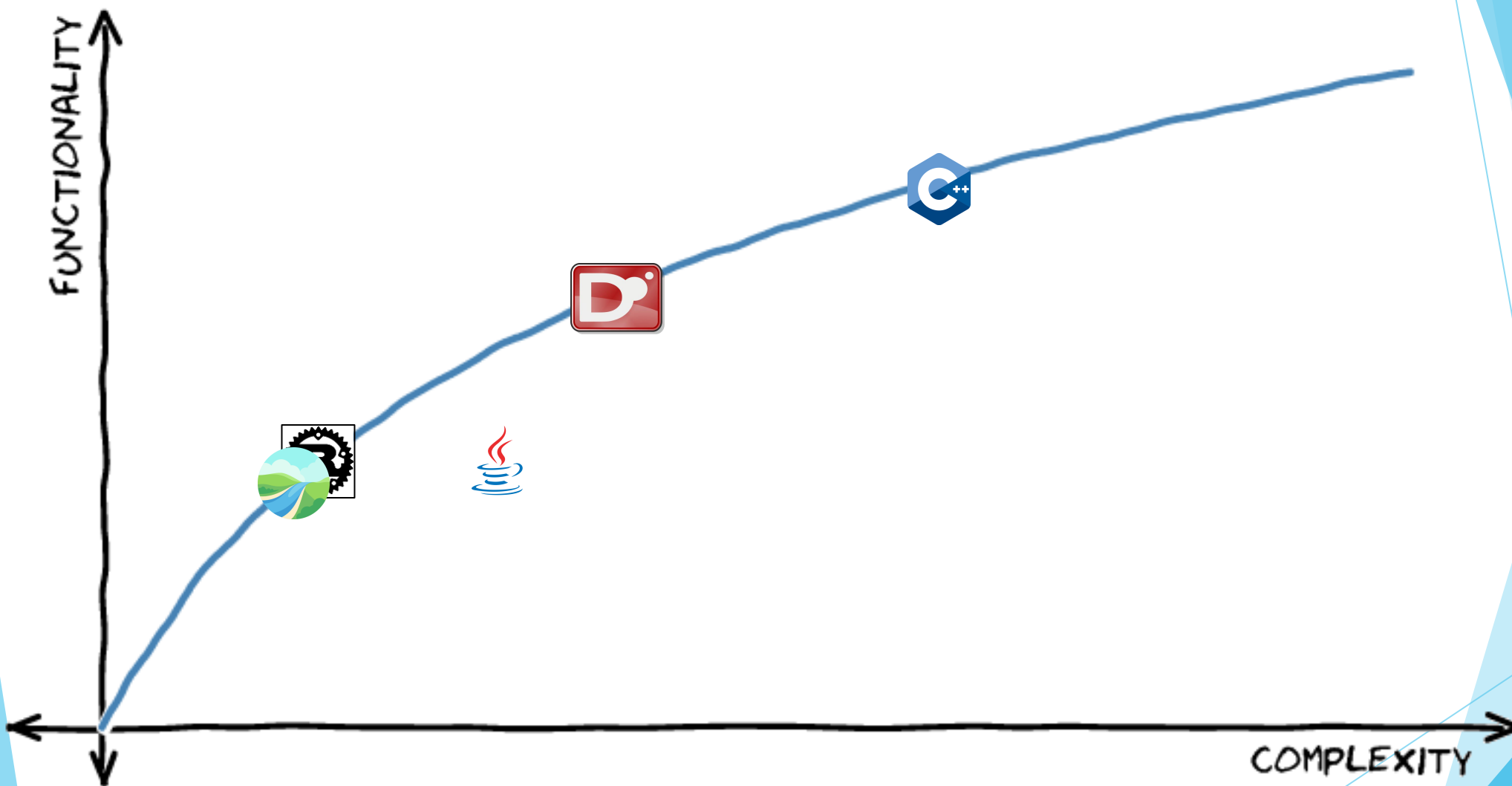
public:
    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        return std::apply([&](S&... streams){
            return (streams.while_(pred) and ...);
        }, streams_);
    }
};
```



Internal vs External Iteration



- ▶ Equivalent to hand-written loops
- ▶ Can be much faster than C++ or Rust model in pathological cases (e.g. `chain`)
- ▶ ... but would be slower if you aren't just doing a terminal



Thank You!



<https://brevzin.github.io/>



@BarryRevzin



Barry



<https://github.com/brevzin/rivers>