think-cell

Why Iterators Got It All Wrong

and what we should use instead

Iterators

- part of C++ since the stone age
- modeled after pointer
- in D superseded by ranges

Iterators

- part of C++ since the stone age
- modeled after pointer
- in D superseded by ranges
- iterators can be elements v

```
vector<int> vec={3,2,1,3};

min_element(vec.begin(),vec.end())
          v
{ 3 , 2 , 1 , 3 }
```

Iterators

- part of C++ since the stone age
- modeled after pointer
- in D superseded by ranges
- iterators can be elements v

```
vector<int> vec={3,2,1,3};

min_element(vec.begin(),vec.end())
          v
{ 3 , 2 , 1 , 3 }
```

- iterators can be borders between elements |

```
vector<int> vec={0,0,1,1};

upper_bound(vec.begin(),vec.end(),0)
        | v
{ 0 , 0 , 1 , 1 }
```

Ranges

- anything that has iterators

```
for( auto it=begin(rng); it!=end(rng); ++it ){...}
```

Ranges

- anything that has iterators

```
for( auto it=begin(rng); it!=end(rng); ++it ){...}
```

- containers

```
vector
list
set
```

  - own elements
  - deep copying
    - copying copies elements in O(N)
  - deep constness
    - `const` objects implies `const` elements

Ranges

- anything that has iterators

```
for( auto it=begin(rng); it!=end(rng); ++it ){...}
```

- containers

```
vector
list
set
```

  - own elements
  - deep copying
    - copying copies elements in O(N)
  - deep constness
    - `const` objects implies `const` elements
- views

Views

- reference elements
- shallow copying
  - copying copies reference in O(1)
- shallow constness
  - view object `const` independent of element `const`

Views

- reference elements
- shallow copying
  - copying copies reference in O(1)
- shallow constness
  - view object `const` independent of element `const`

```
template<typename It>
struct iterator_view {
    It m_itBegin;
    It m_itEnd;
    It begin() const {
        return m_itBegin;
    }
    It end() const {
        return m_itEnd;
    }
};
```

Views (2)

- more compact code
  - with iterators:

```
std::vector<T> vec=...;
std::sort( vec.begin(), vec.end() );
vec.erase( std::unique( vec.begin, vec.end() ), vec.end() );
```

  - with ranges:

```
tc::unique_inplace(tc::sort(vec));
```

Views (3): Range Adaptors

```
std::vector<int> v={0,0,1,1};
auto it=find(
    v,
    0
); // first element of value 0.
```

Views (3): Range Adaptors

```
std::vector<int> v={0,0,1,1};
auto it=find(
    v,
    0
); // first element of value 0.
```

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};
auto it=find_if(
    v,
    [](A const& a){ return a.first==0; }
); // first element of value 0 in first
```

- related in semantics
- not at all related in syntax
- projection and search criterion lumped together

Transform Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};
auto it=find_if(
    v,
    [](A const& a){ return a.first==0; }
); // first element of value 0 in first
```

Transform Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};
auto it=find_if(
    v,
    [](A const& a){ return a.first==0; }
); // first element of value 0 in first
```

- separation of projection and search criterion

```
auto trans=transform(vec, mem_fn(&pair<int,char>::first)); // {0,0,1,1}

auto it=find(trans,0); // first element of value 0 in first
```

- `vec` not modified
- `trans` referencing `vec`
- transformation lazy
  - only pay for what you dereference
  - no extra heap memory

Transform Adaptor (2)

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};
find_if(
    v,
    [](A const& a){ return a.first==0; }
)->second; // 'a' !
```

Transform Adaptor (2)

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};
find_if(
    v,
    [](A const& a){ return a.first==0; }
)->second; // 'a' !
```

```
auto trans=transform(vec, mem_fn(&pair<int,char>::first)); // {0,0,1,1}

find(trans,0)->second // 'a' ?
```

Transform Adaptor (2)

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};
find_if(
    v,
    [](A const& a){ return a.first==0; }
)->second; // 'a' !
```

```
auto trans=transform(vec, mem_fn(&pair<int,char>::first)); // {0,0,1,1}

find(trans,0)->second // 'a' ?
```

- iterator points to `int`
- peel off `transform` to get iterator pointing to `pair<int,char>`

```
find(trans,0).base()->second // 'a'!

   v
{  0,       0,       1,       1    } // trans
   v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

Transform Adaptor (3)

- `find` returns iterator in role of element
- `.base()` must preserve identity of element

```
find(trans,0).base()->second // 'a'

   v
{  0,      0,      1,      1    } // trans
   v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

Transform Adaptor (3)

- find returns iterator in role of element
- .base() must preserve identity of element

```
find(trans,0).base()->second // 'a'

   v
{  0,      0,      1,      1    } // trans
   v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

- upper_bound returns iterator in role of border
- same base() preserves identity of border

```
upper_bound(trans,0).base()

                |  v
{  0,      0,      1,      1    } // trans
                |  v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

Filter Adaptor

think-cell

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto filt=filter(vec, [](auto const& p){return p.second=='b';});
    // {{0,'b'},{1,'b'}}
```

Filter Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto filt=filter(vec, [](auto const& p){return p.second=='b';});
    // {{0,'b'},{1,'b'}}
```

```
auto trans=transform(filt, mem_fn(&pair<int,char>::first)); // {0,1}

find(trans,0).base().base()

           v
{          0    ,          1    } // trans
           v // .base()
{        {0,'b'},        {1,'b'}} // filt
           v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

- `find` returns iterator in role of element
- result would be different without `filter`
- OK?

Filter Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto filt=filter(vec, [](auto const& p){return p.second=='b';});
    // {{0,'b'},{1,'b'}}
```

```
auto trans=transform(filt, mem_fn(&pair<int,char>::first)); // {0,1}

find(trans,0).base().base()

           v
{          0    ,          1    } // trans
           v // .base()
{        {0,'b'},        {1,'b'}} // filt
           v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

- `find` returns iterator in role of element
- irrelevant: result would be different without `filter`
- important: `.base()` preserves identity of element

Filter Adaptor (2)

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto filt=filter(vec, [](auto const& p){return p.second=='b';});
    // {{0,'b'},{1,'b'}}
```

```
auto trans=transform(filt, mem_fn(&pair<int,char>::first)); // {0,1}

upper_bound(trans,0).base().base()

                    |      v
{          0    ,          1    } // trans
                    |      v // .base()
{        {0,'b'},        {1,'b'}} // filt
                        |  v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

- `.base()` preserves identity of element
- OK?

Filter Adaptor (2)

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto filt=filter(vec, [](auto const& p){return p.second=='b';});
    // {{0,'b'},{1,'b'}}
```

```
auto trans=transform(filt, mem_fn(&pair<int,char>::first)); // {0,1}

upper_bound(trans,0).base().base()

                    |      v
{          0    ,          1    } // trans
                    |      v // .base()
{        {0,'b'},        {1,'b'}} // filt
                |???????|  v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

- `.base()` preserves identity of element
- identity of border not preservable
- `filter(...).base()` ambiguous if iterator in role of border

Filter Adaptor (2)

think-cell

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto filt=filter(vec, [](auto const& p){return p.second=='b';});
    // {{0,'b'},{1,'b'}}
```

```
auto trans=transform(filt, mem_fn(&pair<int,char>::first)); // {0,1}

upper_bound(trans,0).base().base()

                    |      v
{          0    ,          1    } // trans
                    |      v // .base()
{        {0,'b'},        {1,'b'}} // filt
                |???????|  v // .base()
{{0,'a'},{0,'b'},{1,'a'},{1,'b'}} // vec
```

- `.base()` preserves identity of element
- identity of border not preservable
- `filter(...).base()` ambiguous if iterator in role of border
- THEN DON'T CALL IT, DUMBA** !!!

Reverse Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto rev=reverse(vec); // {{1,'b'},{1,'a'},{0,'b'},{0,'a'}}
```

Reverse Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto rev=reverse(vec); // {{1,'b'},{1,'a'},{0,'b'},{0,'a'}}
```

```
auto trans=transform(rev, mem_fn(&pair<int,char>::first)); // {1,1,0,0}

find(trans,0).base().base()

                   v
{  1    ,  1    ,  0    ,  0    } // trans
                   v          // .base()
{{1,'b'},{1,'a'},{0,'b'},{0,'a'}} // rev
                   v          // .base()
        {{0,'a'},{0,'b'},{1,'a'},{1,'b'}}; // vec
```

- `.base()` must preserve identity of element

Reverse Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto rev=reverse(vec); // {{1,'b'},{1,'a'},{0,'b'},{0,'a'}}
```

```
auto trans=transform(rev, mem_fn(&pair<int,char>::first)); // {1,1,0,0}

lower_bound(trans,0,std::greater<>()).base().base()

                   v
{  1    ,  1    ,  0    ,  0    } // trans
                   v         // .base()
{{1,'b'},{1,'a'},{0,'b'},{0,'a'}} // rev
                   v         // .base()
        {{0,'a'},{0,'b'},{1,'a'},{1,'b'}}; // vec
```

- `.base()` must preserve identity of element

Reverse Adaptor

```
vector<pair<int,char>> vec={{0,'a'},{0,'b'},{1,'a'},{1,'b'}};

auto rev=reverse(vec); // {{1,'b'},{1,'a'},{0,'b'},{0,'a'}}
```

```
auto trans=transform(rev, mem_fn(&pair<int,char>::first)); // {1,1,0,0}

lower_bound(trans,0,std::greater<>()).base().base()

                |  v
{  1    ,  1    ,  0    ,  0    } // trans
                |  v         // .base()
{{1,'b'},{1,'a'},{0,'b'},{0,'a'}} // rev
                   v    |  v // .base()
        {{0,'a'},{0,'b'},{1,'a'},{1,'b'}}; // vec
```

- `.base()` must preserve identity of element
- `.base()` must also preserve identity of border

!!! the same base() cannot do both !!!

Reverse Adaptor - how come?

```
struct reverse_adaptor {
  struct iterator {
    BaseIt m_it;
    operator++() { --m_it; }
    operator--() { ++m_it; }
    operator*() { return *(m_it-1); } // why?
  };
  begin() { return iterator{m_base.end()}; }
  end() { return iterator{m_base.begin()}; }
};
```

Reverse Adaptor - how come?

```
struct reverse_adaptor {
  struct iterator {
    BaseIt m_it;
    operator++() { --m_it; }
    operator--() { ++m_it; }
    operator*() { return *(m_it-1); } // why?
  };
  begin() { return iterator{m_base.end()}; }
  end() { return iterator{m_base.begin()}; }
};
```

- `iterator` at `begin()` stores `m_it=m_base.end()`
  - may be dereferenced
  - must return `*(m_base.end()-1)`

Reverse Adaptor - how come?

```
struct reverse_adaptor {
  struct iterator {
    BaseIt m_it;
    operator++() { --m_it; }
    operator--() { ++m_it; }
    operator*() { return *(m_it-1); } // why?
  };
  begin() { return iterator{m_base.end()}; }
  end() { return iterator{m_base.begin()}; }
};
```

- `iterator` at `begin()` stores `m_it=m_base.end()`
  - may be dereferenced
  - must return `*(m_base.end()-1)`
- `iterator` at `end()` stores `m_it=m_base.begin()`
  - won't be dereferenced

Reverse Adaptor - how come?

```
struct reverse_adaptor {
  struct iterator {
    BaseIt m_it;
    operator++() { --m_it; }
    operator--() { ++m_it; }
    operator*() { return *(m_it-1); } // why?
  };
  begin() { return iterator{m_base.end()}; }
  end() { return iterator{m_base.begin()}; }
};
```

- `iterator` at `begin()` stores `m_it=m_base.end()`
    - may be dereferenced
    - must return `*(m_base.end()-1)`
- `iterator` at `end()` stores `m_it=m_base.begin()`
    - won't be dereferenced
- decrementing `iterator` to `end()-1` makes `m_it=m_base.begin()+1`
    - returns `*(m_base.begin()+1-1)` - OK!

Reverse Adaptor - how come?

```
struct reverse_adaptor {
  struct iterator {
    BaseIt m_it;
    operator++() { --m_it; }
    operator--() { ++m_it; }
    operator*() { return *(m_it-1); }
    base() { return m_it-1; } // correct for element
    base() { return m_it; } // correct for border
  };
  begin() { return iterator{m_base.end()}; }
  end() { return iterator{m_base.begin()}; }
};
```

- element *after* border in `reverse` sequence is element *before* border in original sequence

Scope of Problem: Order

```
        | v
{ b , c , a , d }
          v // .base()
        { a , b , c , d }
```

- adaptor changes order of elements

Scope of Problem: Order

```
    | v
{ b , c , a , d }
     v // .base()
    { a , b , c , d }
```

- adaptor changes order of elements
  - `base()` of element well-defined
  - `base()` of border in general undefined
  - example: `sort` adaptor

Scope of Problem: Order

```
        | v
{ b , c , a , d }
          v // .base()
        { a , b , c , d }
```

- adaptor changes order of elements
  - base() of element well-defined
  - base() of border in general undefined
  - example: sort adaptor
- reverse adaptor
  - exceptional: everything changes sides
  - base() of border well-defined, but different from base() of element

Scope of Problem: Removal

```
          |   v
    { b ,     d }
        |???| v // .base()
{ a , b , c , d }
```

- adaptor removes elements

Scope of Problem: Removal

```
          |   v
    { b ,     d }
        |???| v // .base()
{ a , b , c , d }
```

- adaptor removes elements
  - elements may collapse into border
  - `base()` of element well-defined
  - `base()` of border ambiguous
  - ex.: `filter`, `sorted_intersection`, `sorted_difference`

Scope of Problem: Adding

```
        | v
{ a , b , c , d }
        | // .base()
{ a ,         d }
```

- adaptor adds elements

Scope of Problem: Adding

```
        | v
{ a , b , c , d }
        | // .base()
{ a ,         d }
```

- adaptor adds elements
  - elements appear that were not present in base
  - `base()` of border well-defined
  - `base()` of element in general undefined
  - ex.: `sorted_union`

What do we do?

- Separate functions `border_base` and `element_base`
  - (+) small change
  - (-) no safety against wrong choice

What do we do?

- Separate functions `border_base` and `element_base`
  - (+) small change
  - (-) no safety against wrong choice
- Separate concepts Border and Element
  - (-) big change: no more iterator (at least in user code)
  - (+) `base()` always does right thing

What do we do?

- Separate functions `border_base` and `element_base`
  - (+) small change
  - (-) no safety against wrong choice
- Separate concepts Border and Element
  - (-) big change: no more iterator (at least in user code)
  - (+) `base()` always does right thing

Q: Do Iterators really have assigned roles Border/Element?

- Hypothesis tested against our codebase (~1M LOC)

Border vs Element

Q: Do Iterators really have assigned roles Border/Element?

- Hypothesis tested against our codebase (~1M LOC)
- `find`
  - 201 single match
    - 1 border role
    - 1 incremented to get border after
    - others element role
  - 98 first match
    - 7 border role
    - 5 incremented to get border after
    - others element role

Border vs Element (2)

Q: Do Iterators really have assigned roles Border/Element?

- Hypothesis tested against our codebase (~1M LOC)
- `find_if`
    - 67 single match
        - all element role
    - 75 first match
        - 3 border role
        - others element role

Border vs Element (3)

Q: Do Iterators really have assigned roles Border/Element?

- Hypothesis tested against our codebase (~1M LOC)
- `lower_bound`
  - 2 no further use of predicate
    - border role
  - 89 use predicate to find single match
    - all element role
  - 19 use predicate to find first match
    - all element role

Border vs Element (3)

Q: Do Iterators really have assigned roles Border/Element?

- Hypothesis tested against our codebase (~1M LOC)
- `lower_bound`
  - 2 no further use of predicate
    - border role
  - 89 use predicate to find single match
    - all element role
  - 19 use predicate to find first match
    - all element role
- `upper_bound`
  - 24 total
    - 17 border role
    - 7 decremented to get element before

Border vs Element (3)

Q: Do Iterators really have assigned roles Border/Element?

- Hypothesis tested against our codebase (~1M LOC)
- `lower_bound`
  - 2 no further use of predicate
    - border role
  - 89 use predicate to find single match
    - all element role
  - 19 use predicate to find first match
    - all element role
- `upper_bound`
  - 24 total
    - 17 border role
    - 7 decremented to get element before

→ Iterator instances have distinct roles Border/Element

Iterators were always ugly

- `begin()` and `end()` asymmetric
  - can dereference `begin()`
  - cannot dereference `end()`

```
begin            end
  v   v   v   v   v
{ a , b , c , d }
```

Iterators were always ugly

- begin() and end() asymmetric
  - can dereference begin()
  - cannot dereference end()

```
begin            end
  v   v   v   v   v
{ a , b , c , d }
```

- elements and borders are symmetric

```
begin            end
  | v | v | v | v |
  { a , b , c , d }
```

Iterators were always ugly (2)

```
auto it=find(rng, t);
if(it!=end(rng)) {...}
```

- end()'s meaning depends on role
  - if border, border after all elements
  - if element, magic value to say "none"

Iterators were always ugly (2)

```
auto it=find(rng, t);
if(it!=end(rng)) {...}
```

- `end()`'s meaning depends on role
  - if border, border after all elements
  - if element, magic value to say "none"
- have to mention rng twice
  - cannot write range expression inline

Iterators were always ugly (2)

think-cell

```
auto it=find(rng, t);
if(it!=end(rng)) {...}
```

- `end()`'s meaning depends on role
  - if border, border after all elements
  - if element, magic value to say "none"
- have to mention rng twice
  - cannot write range expression inline
- why not

  ```
  if( auto it=find(rng, t) ) {...}
  ```

→ **Introduce Border and Element concepts!**

Border Concept

```
begin            end
  | v | v | v | v |
  { a , b , c , d }
```

- Border | : like Iterator but
  - cannot be dereferenced
  - if not at begin, has `element_before()`
  - if not at end, has `element_after()`

Border Concept

```
begin            end
  | v | v | v | v |
  { a , b , c , d }
```

- Border | : like Iterator but
  - cannot be dereferenced
  - if not at begin, has `element_before()`
  - if not at end, has `element_after()`
- range `begin()` and `end()` are borders

Border Concept

```
begin               end
  | v | v | v | v |
  { a , b , c , d }
```

- Border | : like Iterator but
  - cannot be dereferenced
  - if not at begin, has `element_before()`
  - if not at end, has `element_after()`
- range `begin()` and `end()` are borders
- all iterators going into `<algorithm>` are borders
  - begin or end of input range

Border Concept

```
begin            end
  | v | v | v | v |
  { a , b , c , d }
```

- Border | : like Iterator but
  - cannot be dereferenced
  - if not at begin, has `element_before()`
  - if not at end, has `element_after()`
- range `begin()` and `end()` are borders
- all iterators going into `<algorithm>` are borders
  - begin or end of input range
- all output iterators are borders
  - begin or end of output

Border Concept

```
begin            end
  | v | v | v | v |
  { a , b , c , d }
```

- Border | : like Iterator but
  - cannot be dereferenced
  - if not at begin, has `element_before()`
  - if not at end, has `element_after()`
- range `begin()` and `end()` are borders
- all iterators going into `<algorithm>` are borders
  - begin or end of input range
- all output iterators are borders
  - begin or end of output
- iterators returned from `<algorithm>`
  - depends on algorithm

Border Concept (2)

- returned iterators are borders:

```
mismatch                           // end of matching prefix
search                             // begin of matching range
lower_bound                        // begin of equal range
upper_bound                        // end of equal range
equal_range                        // lower and upper bound together
partition_point/[stable_]partition // border between first part
                                   // and second part
unique                             // end of compacted range
```

Element Concept

```
begin            end
  | v | v | v | v |
  { a , b , c , d }
```

- Element v : like Iterator but
  - never end(), cannot ++ beyond last element
  - has border_before()/border_after()
- following algorithms return element:

```
[max_|min_]element // max/min element of a range
```

- range_of_elements utility to get all elements inside borders

```
for_each( range_of_elements(range), [&]( auto element ){...} );
```

Element Concept (2)

- make Element nullable
  - compatible with pointer: pointer satisfies Element concept
  - contextually convertible to bool
  - `null` state reached through value initialization `Element{}`
  - functions returning Element return `null` instead of `.end()`

```
Element elem{}; // null element
assert(!elem);

if( auto it=find_unique(rng, t) ) {...}
```

Element Concept (3)

- let programmer encode her intent

Element Concept (3)

- let programmer encode her intent
- `std::find[_if]` gets refined to

```
tc::find_unique[_if] -> Element
tc::find_first[_if] -> Element
tc::find_last[_if] -> Element
tc::trim_left[_if] -> Border
tc::trim_right[_if] -> Border
```

Element Concept (3)

- let programmer encode her intent
- std::find[_if] gets refined to

```
tc::find_unique[_if] -> Element
tc::find_first[_if] -> Element
tc::find_last[_if] -> Element
tc::trim_left[_if] -> Border
tc::trim_right[_if] -> Border
```

- std::lower_bound gets refined to

```
tc::binary_find_unique -> Element
tc::binary_find_first -> Element
tc::binary_find_last -> Element
tc::lower_bound -> Border
```

Element Concept (3)

- let programmer encode her intent
- `std::find[_if]` gets refined to

```
tc::find_unique[_if] -> Element
tc::find_first[_if] -> Element
tc::find_last[_if] -> Element
tc::trim_left[_if] -> Border
tc::trim_right[_if] -> Border
```

- `std::lower_bound` gets refined to

```
tc::binary_find_unique -> Element
tc::binary_find_first -> Element
tc::binary_find_last -> Element
tc::lower_bound -> Border
```

- can always use `border_before()`/`border_after()` to convert Element to Border

Element Concept (3)

- let programmer encode her intent
- std::find[_if] gets refined to

```
tc::find_unique[_if] -> Element
tc::find_first[_if] -> Element
tc::find_last[_if] -> Element
tc::trim_left[_if] -> Border
tc::trim_right[_if] -> Border
```

- std::lower_bound gets refined to

```
tc::binary_find_unique -> Element
tc::binary_find_first -> Element
tc::binary_find_last -> Element
tc::lower_bound -> Border
```

- can always use border_before()/border_after() to convert Element to Border
- _unique functions assert single match

Algorithm Return Specification

```
if( auto it=find_unique(rng, t) ) {...}
```

- want to mention rng only once
  - can write range expression inline

Algorithm Return Specification

think-cell

```
if( auto it=find_unique(rng, t) ) {...}
```

- want to mention rng only once
    - can write range expression inline
- let programmer write intent
    - algorithms get template parameter to control return value

Algorithm Return Specification

```
if( auto it=find_unique(rng, t) ) {...}
```

- want to mention rng only once
  - can write range expression inline
- let programmer write intent
  - algorithms get template parameter to control return value
- algorithm returning border

```
// return border
lower_bound<return_border> -> border

// return view beginning/ending at border
lower_bound<return_take> -> view
lower_bound<return_drop> -> view

// return border's adjacent element
lower_bound<return_element_after> -> element
upper_bound<return_element_before> -> element
```

Algorithm Return Spec. (2)

- algorithm returning element

```
// return element, which may not be there
find<return_element_or_null>

// return element, which must be there
find<return_element>

// return element's adjacent border [or alternative if not there]
find<return_border_before[_or_begin|_or_end]>
find<return_border_after[_or_begin|_or_end]>

// return view beginning/ending adjacent to element
// [or alternative if not there]
find<return_take_before[_or_empty|_or_all]>
find<return_take_after[_or_empty|_or_all]>
find<return_drop_before[_or_empty|_or_all]>
find<return_drop_after[_or_empty|_or_all]>
```

Return Spec. Implementation

```
template< typename Rng >
struct return_take_before_or_empty {
  template<typename It>
  static auto pack_element(It it, Rng&& rng) noexcept {
    return tc::take(std::forward<Rng>(rng), it);
  }
  static auto pack_no_element(Rng&& rng) noexcept {
    return tc::take(std::forward<Rng>(rng), boost::begin(rng));
  }
};
```

Conclusion

- Iterator modeled after pointers
  - low level machine concept
- Element and Border stronger semantics
  - intent already in programmer's head
  - express intent in code
  - needed for correctness of important range functions
- think-cell range library https://github.com/think-cell/think-cell-library
  - Element nullable
  - algorithm refinements
  - return specifications
- still missing
  - Border not dereferencable
  - no implicit conversion Element→Border

THANK YOU!