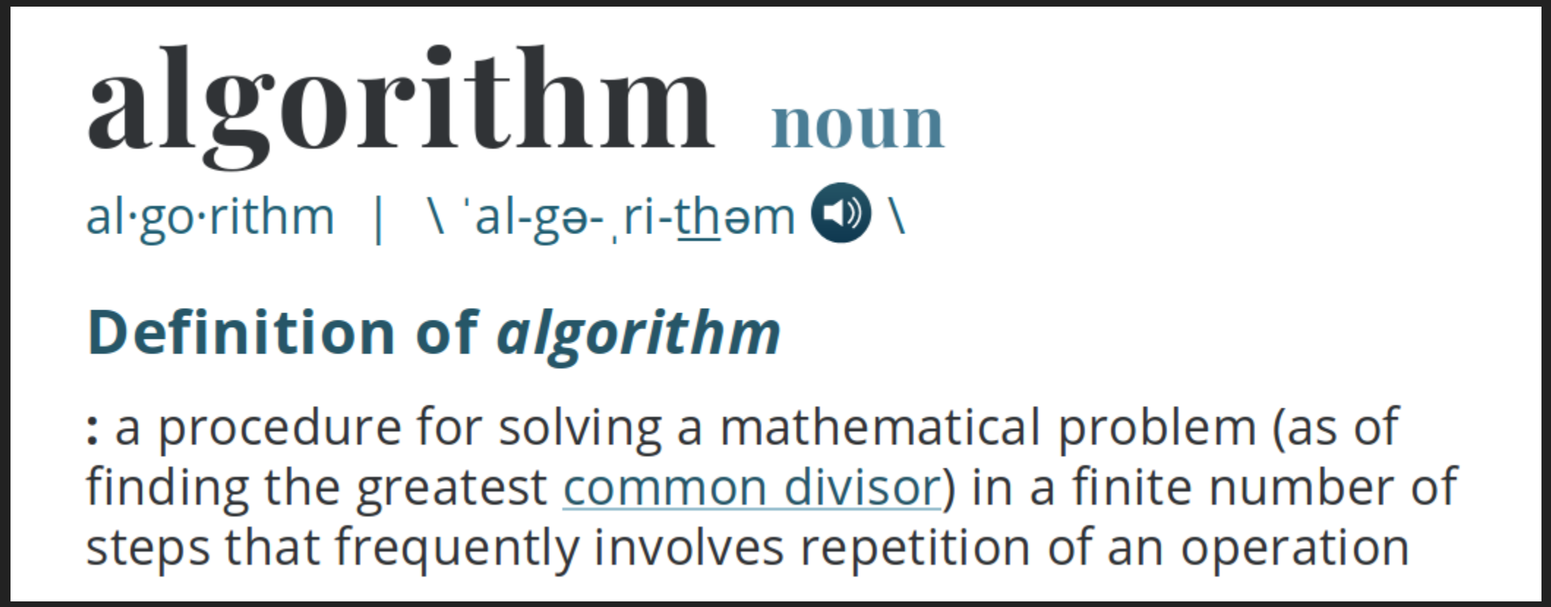
CONSTRUCTING GENERIC ALGORITHMS



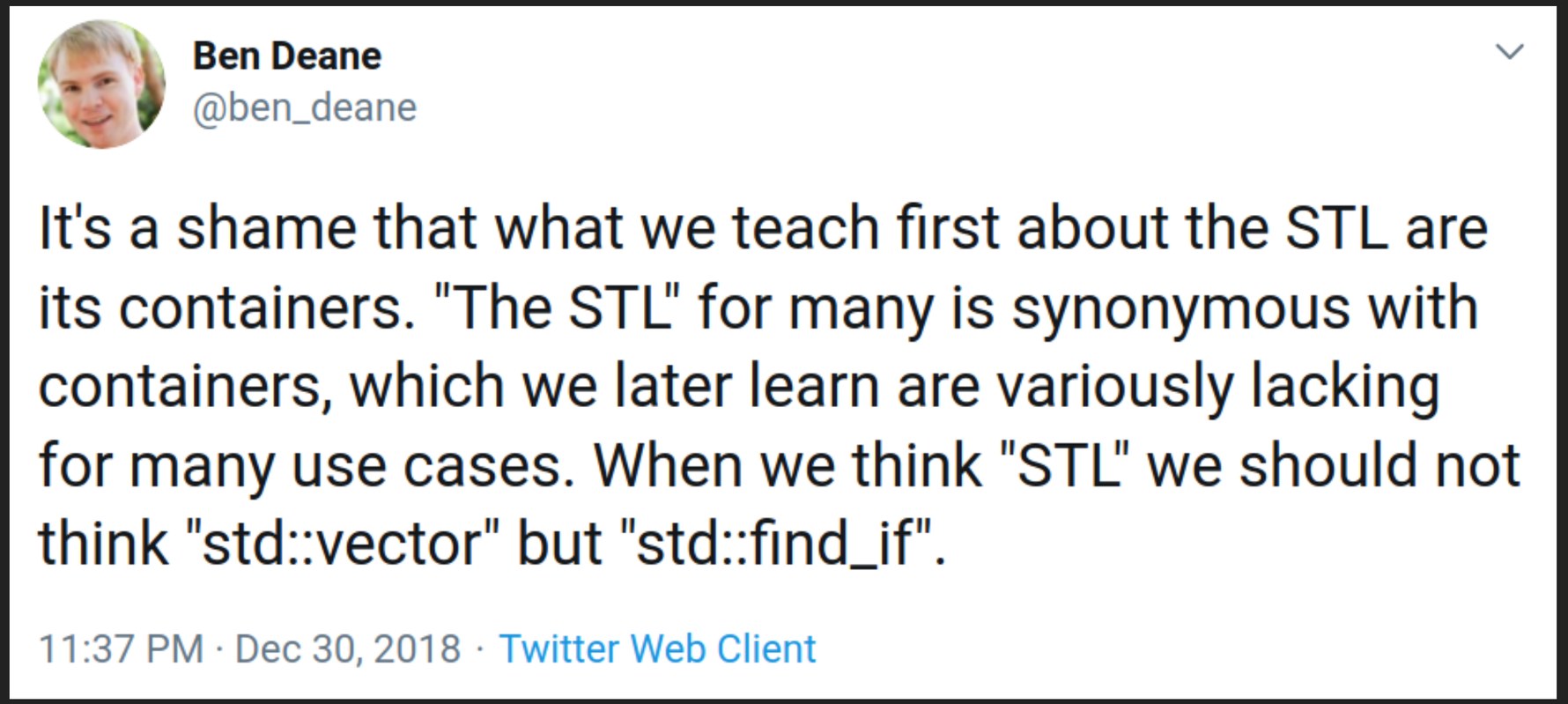
BEN DEANE / @ben_deane

16TH SEPTEMBER 2020

WHAT WE'LL COVER

- Preliminaries: motivations etc
- Case study: a nontrivial nonstandard algorithm
- Principles for algorithm design
- Some holes in the standard
- Ponters to further work

ALGORITHMS: THE SOUL OF THE STL



A WORD ABOUT RANGES

"STL algorithms are not composable."

-- Everyone (justifying ranges)

A WORD ABOUT RANGES

"STL algorithms are not composable."

-- Everyone (justifying ranges)

What, just because `transform_copy_if` doesn't exist?

STL ALGORITHMS: A STUDY IN COMPOSABILITY

"The algorithms fit together like puzzle pieces."

-- Sean Parent

The original standard set of algorithms is designed to support `stable_sort`.

WHY WRITE GENERIC ALGORITHMS?

Isn't the standard set good enough?

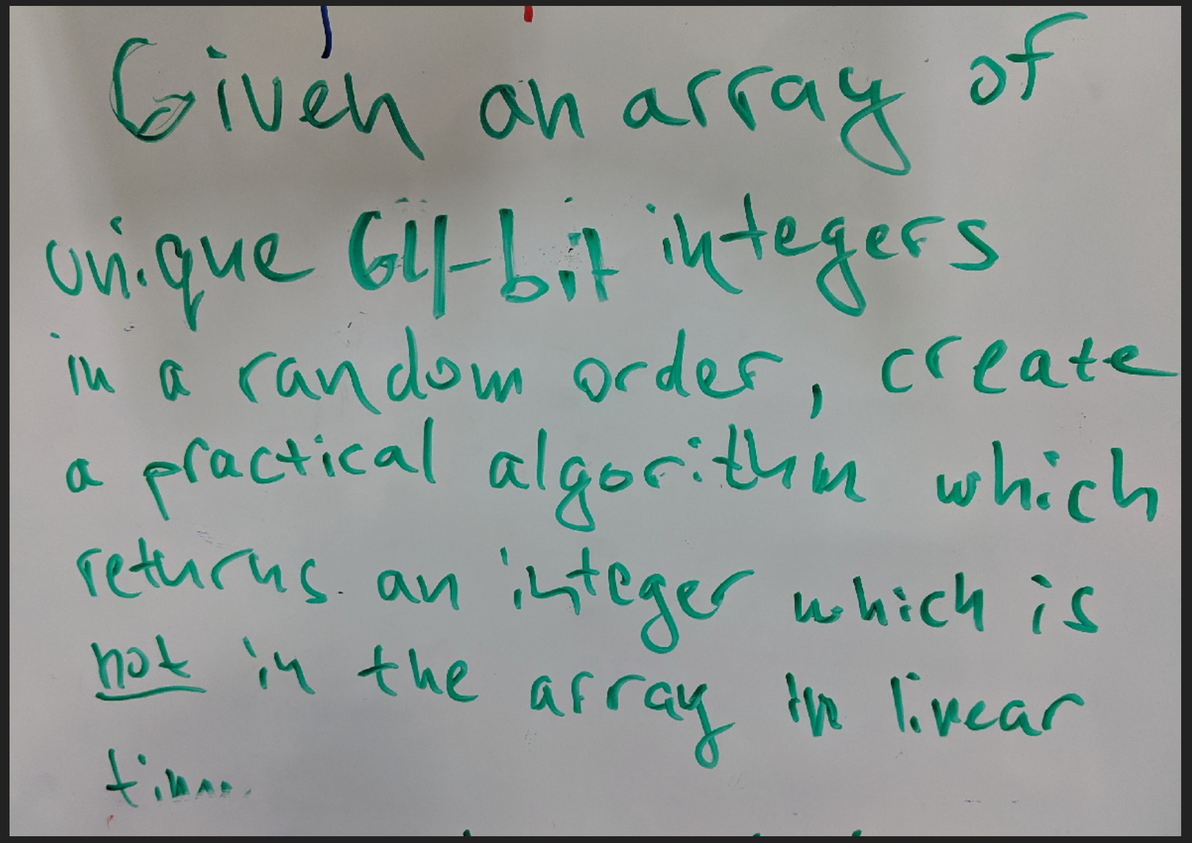
Frequently, yes. But not always.

And the standard set was *never designed to be complete*.

The whole point of the STL is that decoupling containers (with the iterator abstraction) allows us to write generic algorithms that work on *all containers*.

So let's write some!

THE PROBLEM



From the board at CppCon 2019

THE PROBLEM

unused number = ???

RESTATE THE PROBLEM CONSTRAINTS

RESTATE THE PROBLEM CONSTRAINTS

- 64-bit integers
  - ⇒ we can't just do "largest + 1" or similar
  - ⇒ totally ordered

RESTATE THE PROBLEM CONSTRAINTS

- 64-bit integers
  - ⇒ we can't just do "largest + 1" or similar
  - ⇒ totally ordered
- unique elements

RESTATE THE PROBLEM CONSTRAINTS

- 64-bit integers
  - ⇒ we can't just do "largest + 1" or similar
  - ⇒ totally ordered
- unique elements
- random order
  - ⇒ we are free to permute the input (it's arbitrary anyway)
  - ⇒ we can do this in place

RESTATE THE PROBLEM CONSTRAINTS

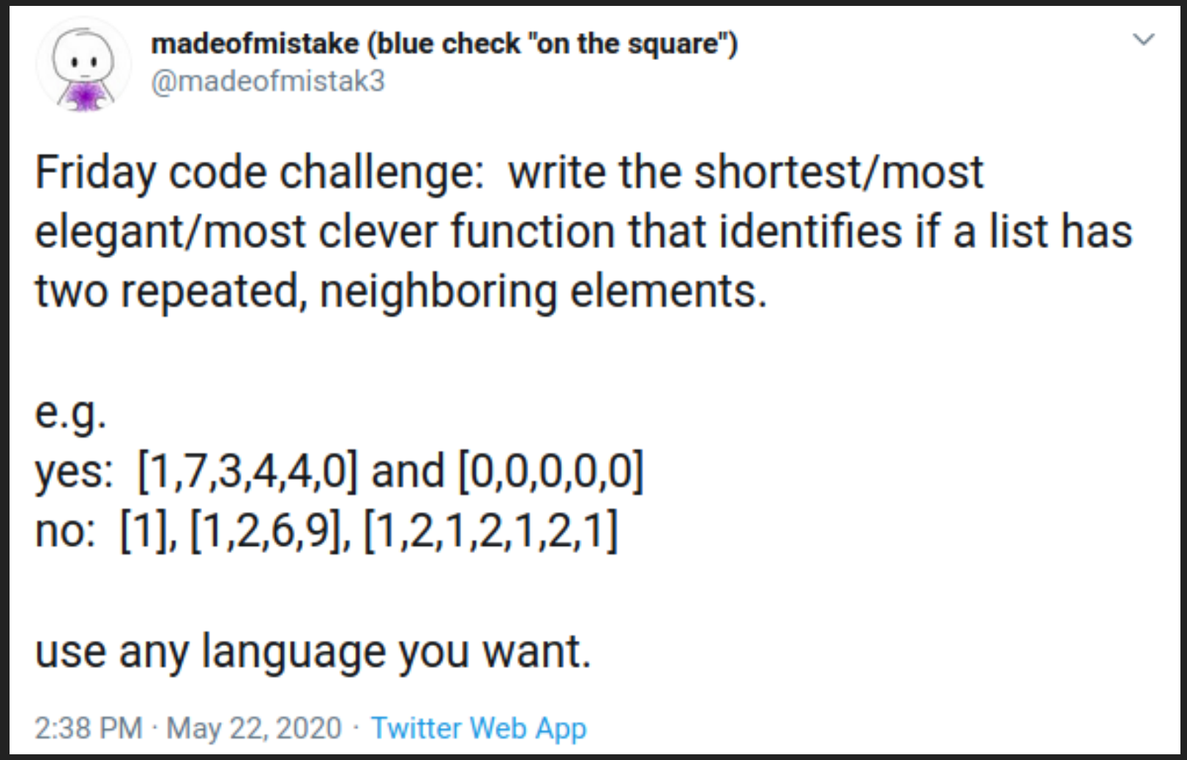
- 64-bit integers
  - ⇒ we can't just do "largest + 1" or similar
  - ⇒ totally ordered
- unique elements
- random order
  - ⇒ we are free to permute the input (it's arbitrary anyway)
  - ⇒ we can do this in place
- linear time

SIMPLIFIED SOLUTION

```
auto find_missing_element(const std::vector<unsigned>& v) {
  const auto it = std::adjacent_find(
    std::cbegin(v), std::cend(v),
    [] (auto x, auto y) { return y-x > 1; });
  return *it + 1;
}
```

If the input were sorted, we'd be done already with a standard algorithm.

ASIDE: adjacent_find



madeofmistake (blue check "on the square")
@madeofmistak3

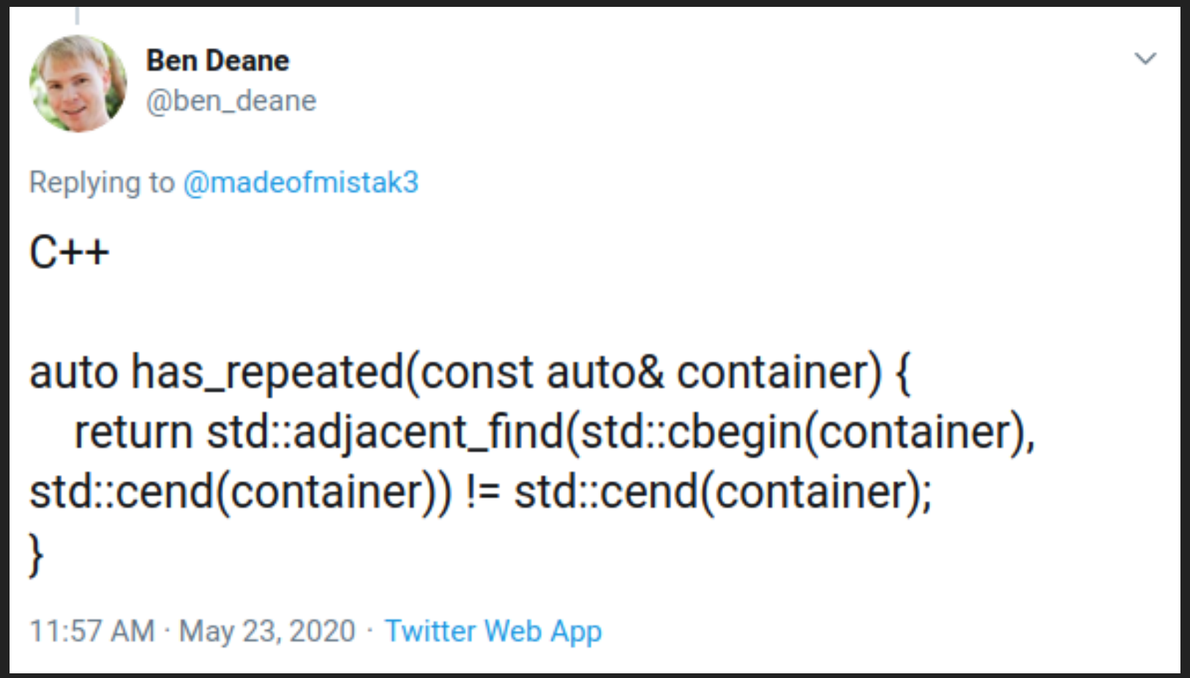
Friday code challenge: write the shortest/most elegant/most clever function that identifies if a list has two repeated, neighboring elements.

e.g.
yes: [1,7,3,4,4,0] and [0,0,0,0,0]
no: [1], [1,2,6,9], [1,2,1,2,1,2,1]

use any language you want.

2:38 PM · May 22, 2020 · Twitter Web App

ASIDE: adjacent_find



Ben Deane
@ben_deane

Replying to @madeofmistak3

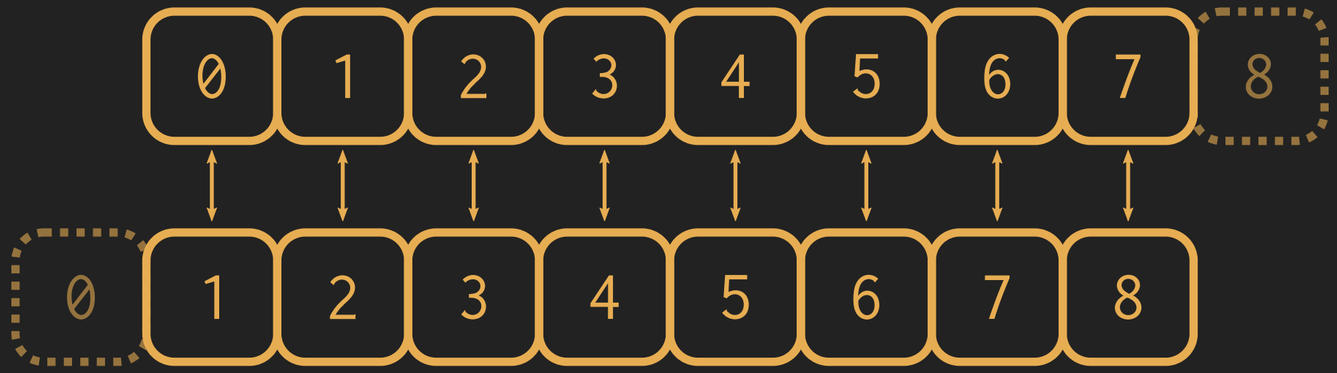
C++

```
auto has_repeated(const auto& container) {
   return std::adjacent_find(std::cbegin(container),
std::cend(container)) != std::cend(container);
}
```

11:57 AM · May 23, 2020 · Twitter Web App

ZIPPING A RANGE WITH ITSELF

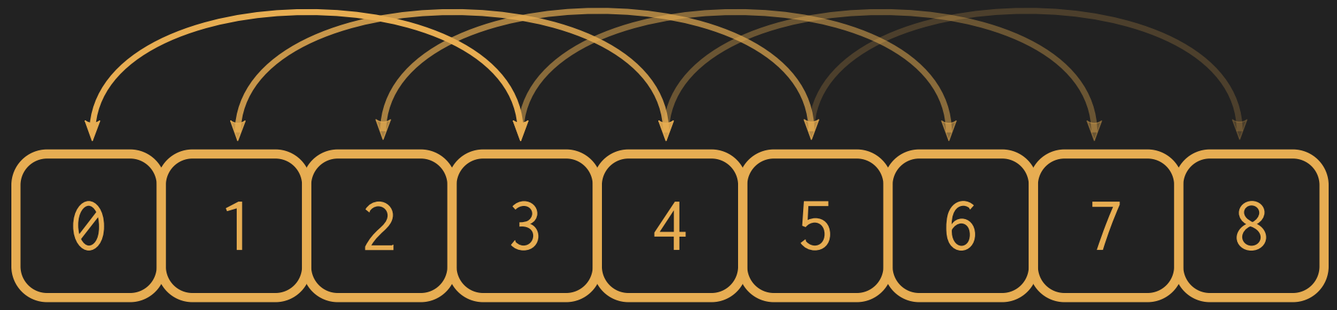
Zipping a range with itself (often shifted)
is an important and common algorithmic pattern.



zip with 1-shift (tail) is `adjacent_find` / `mismatch` / `adjacent_difference`

ZIPPING A RANGE WITH ITSELF

Zipping a range with itself (often shifted)
is an important and common algorithmic pattern.



zip with n-shift is a sliding window like `views::sliding` (`range-v3`, not C++20)

and zip is of course the binary version of `transform`

BACK TO THE PROBLEM

</aside>

BACK TO THE PROBLEM

We need a new approach.

BACK TO THE PROBLEM

We need a new approach.

How to get one?

BACK TO THE PROBLEM

We need a new approach.

How to get one?

~~Cheat~~ Stand on the shoulders of giants!

TRY A NEW APPROACH

A new approach: divide and conquer

TRY A NEW APPROACH

A new approach: divide and conquer

TRY A NEW APPROACH

A new approach: divide and conquer

`m > p` ⇒ gap in `[0, m)` ⇒ recurse on bottom half

TRY A NEW APPROACH

A new approach: divide and conquer

`m > p` ⇒ gap in `[0, m)` ⇒ recurse on bottom half

`m == p` ⇒ gap in `[m, k)` ⇒ recurse on top half

TRY A NEW APPROACH

A new approach: divide and conquer

`m > p` ⇒ gap in `[0, m)` ⇒ recurse on bottom half

`m == p` ⇒ gap in `[m, k)` ⇒ recurse on top half

when `size(sequence) == 1`, answer is `value + 1`

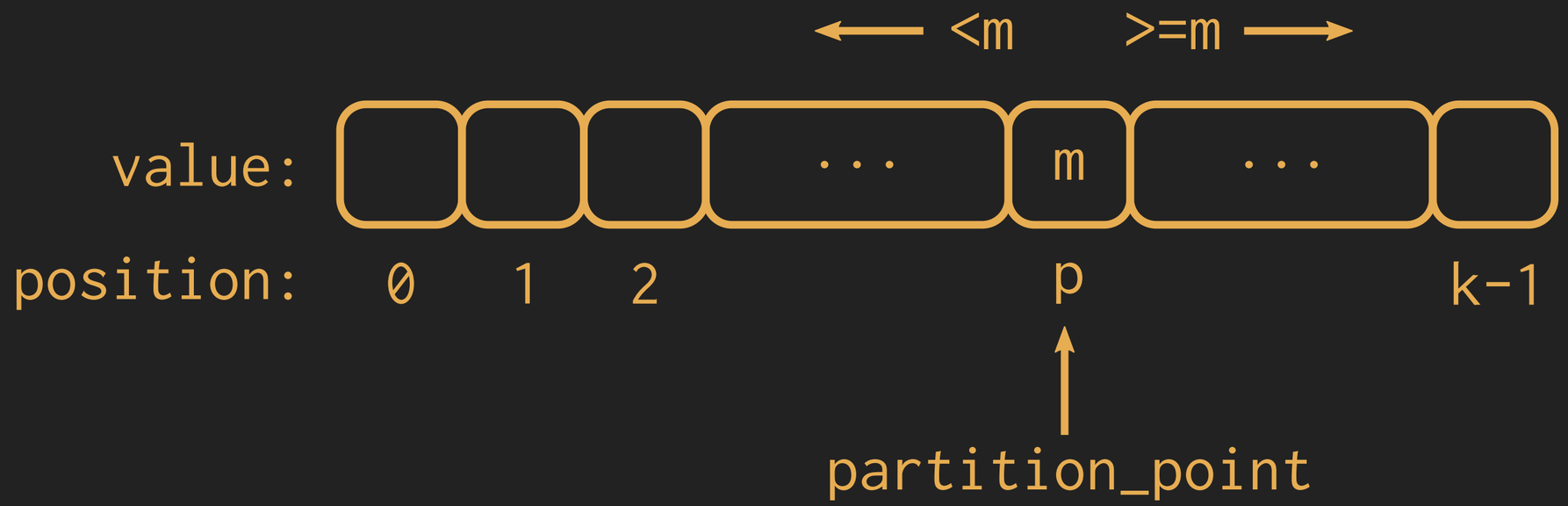
"SOLUTION" FIRST CUT

Assuming a sorted sequence.

```
unsigned find_missing_element(unsigned* first, unsigned* last) {
  // base case
  if (last - first == 1) {
    return *first + 1;
  }

  // recursive step
  int m = (last - first) / 2;
  if (*(first + m) == *first + m) {
    return find_missing_element(first + m, last);
  } else {
    return find_missing_element(first, first + m);
  }
}
```

PARTITION



ACTUAL SOLUTION, FIRST CUT

Not assuming a sorted sequence.

```
unsigned find_missing_element(unsigned* first, unsigned* last, unsigned value) {
  if (last == first) {
    return value;
  }

  unsigned half = (last - first + 1) / 2;
  unsigned m = value + half;
  auto p = std::partition(first, last, [&](auto x) { return x < m; });
  if (p == first + half) {
    return find_missing_element(p, last, m);
  } else {
    return find_missing_element(first, p, value);
  }
}
```

REQUIREMENTS CHECK

- linear time? ✓
- in-place ("no extra memory")? ✓
- "practical"? ✓
- 64-bit integers? **wave hands** we're coming to that

If this were an interview question, we've got a good start.

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence
- even-length sequence with a gap

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence
- even-length sequence with a gap
- odd-length sequence with a gap

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence
- even-length sequence with a gap
- odd-length sequence with a gap
- sequence starting at 0 with no gap

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence
- even-length sequence with a gap
- odd-length sequence with a gap
- sequence starting at 0 with no gap
- sequence starting at N>0 with no gap

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence
- even-length sequence with a gap
- odd-length sequence with a gap
- sequence starting at 0 with no gap
- sequence starting at N>0 with no gap
- sequence with a gap >1

TESTING

Now that we have a first draft working algorithm, let's talk about testing.

What are some test cases we'd want to write?

- zero-length sequence
- even-length sequence with a gap
- odd-length sequence with a gap
- sequence starting at 0 with no gap
- sequence starting at N>0 with no gap
- sequence with a gap >1
- exhaustive tests for sequence length 0..N with gaps at 0..N?

LET'S REMOVE THE RECURSION (0)

Original function (recursive).

```
unsigned find_missing_element(unsigned* first, unsigned* last, unsigned value) {
  if (last == first) {
    return value;
  }

  unsigned half = (last - first + 1) / 2;
  unsigned m = value + half;
  auto p = std::partition(first, last, [&](auto x) { return x < m; });
  if (p == first + half) {
    return find_missing_element(p, last, m);
  } else {
    return find_missing_element(first, p, value);
  }
}
```

LET'S REMOVE THE RECURSION (1)

Make the base case into a `while` loop; `return` after it.

```
unsigned find_missing_element(unsigned* first, unsigned* last, unsigned value) {
  while (last != first) {

    unsigned half = (last - first + 1) / 2;
    unsigned m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      return find_missing_element(p, last, m);
    } else {
      return find_missing_element(first, p, value);
    }
  }
  return value;
}
```

LET'S REMOVE THE RECURSION (2)

Change `return` statements to update variables.

```
unsigned find_missing_element(unsigned* first, unsigned* last, unsigned value) {
  while (last != first) {
    unsigned half = (last - first + 1) / 2;
    unsigned m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

TURN RECURSIVE INTO ITERATIVE

TURN RECURSIVE INTO ITERATIVE

1. Add an "accumulator" variable to the signature
    - return the accumulator in the base case
    - this is "tail recursion"

TURN RECURSIVE INTO ITERATIVE

1. Add an "accumulator" variable to the signature
    - return the accumulator in the base case
    - this is "tail recursion"
2. Convert the base case condition to a loop
    - typically `while (first != last)`
    - `return` the accumulator value after the loop

TURN RECURSIVE INTO ITERATIVE

1. Add an "accumulator" variable to the signature
    - return the accumulator in the base case
    - this is "tail recursion"
2. Convert the base case condition to a loop
    - typically `while (first != last)`
    - `return` the accumulator value after the loop
3. Replace recursive calls with variable updates
    - instead of (re)binding variables, assign them

WHAT NEXT?

```
unsigned find_missing_element(unsigned* first, unsigned* last, unsigned value) {
  while (last != first) {
    unsigned half = (last - first + 1) / 2;
    unsigned m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

TEMPLATES!

```
template <typename It, typename T>
T find_missing_element(It first, It last, T value) {
  while (first != last) {
    T half = (last - first + 1) / 2;
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

ITERATOR CATEGORIES (CONCEPTS)

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
- `contiguous_iterator`

C++17 iterator categories got a concepts specification in C++20.

All of these expose `iterator_traits<It>::value_type`.

ITERATOR CONCEPT QUESTIONS

ITERATOR CONCEPT QUESTIONS

1. Am I using any standard algorithms? Which concept(s) do they require?
    - also informs complexity guarantees, maybe different for different concepts

ITERATOR CONCEPT QUESTIONS

1. Am I using any standard algorithms? Which concept(s) do they require?
    - also informs complexity guarantees, maybe different for different concepts
2. Do I look at an element after moving past it?

ITERATOR CONCEPT QUESTIONS

1. Am I using any standard algorithms? Which concept(s) do they require?
    - also informs complexity guarantees, maybe different for different concepts
2. Do I look at an element after moving past it?
3. Do I return an element after moving past it?
    - must be at least `forward_iterator`

ITERATOR CONCEPT QUESTIONS

1. Am I using any standard algorithms? Which concept(s) do they require?
    - also informs complexity guarantees, maybe different for different concepts
2. Do I look at an element after moving past it?
3. Do I return an element after moving past it?
    - must be at least `forward_iterator`
4. Do I need to decrement the iterator?
    - must be at least `bidirectional_iterator`

ITERATOR CONCEPT QUESTIONS

1. Am I using any standard algorithms? Which concept(s) do they require?
    - also informs complexity guarantees, maybe different for different concepts
2. Do I look at an element after moving past it?
3. Do I return an element after moving past it?
    - must be at least `forward_iterator`
4. Do I need to decrement the iterator?
    - must be at least `bidirectional_iterator`
5. Do I need to increment/decrement by more than one?
    - *might* need `random_access_iterator`

WHAT SORT OF ITERATOR?

```cpp
template <typename It, typename T>
T find_missing_element(It first, It last, T value) {
  while (first != last) {
    T half = (last - first + 1) / 2;
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

WHAT SORT OF ITERATOR?

```
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value);

// or prior to C++20:

template <typename ForwardIt,
          typename T = typename std::iterator_traits<ForwardIt>::value_type>
T find_missing_element(ForwardIt first, ForwardIt last, T value);
```

USING FORWARD ITERATOR

So we can't do general iterator arithmetic. But we can use:

- `std::next`
- `std::distance`
- `std::advance`
- `std::prev` (on bidirectional iterators)

FORWARD ITERATOR RELAXATION (0)

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value) {
  while (first != last) {
    T half = (last - first + 1) / 2;
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

FORWARD ITERATOR RELAXATION (1)

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value) {
  while (first != last) {
    T half = (std::distance(first, last) + 1) / 2; // relax
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == first + half) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

FORWARD ITERATOR RELAXATION (2)

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value) {
  while (first != last) {
    T half = (std::distance(first, last) + 1) / 2;
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == std::next(first, half)) { // relax
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

ITERATOR COMPLEXITY GUARANTEES

By relaxing the operations, we make our algorithm more useful.

What was only usable with `vector`, `array`, etc. is now useful with `list` and `forward_list`, without affecting the runtime complexity for containers with stronger iterators.

ITERATOR CATEGORY MISTAKES? (1)

```cpp
// SGI STL
template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator first,
                          ForwardIterator last, Predicate pred);

// C++98
template <class BidirIt, class UnaryPredicate>
BidirIt partition(BidirIt first, BidirIt last, UnaryPredicate p);

// C++11
template <class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p);
```

Why did partition require a bidirectional iterator in C++98?

(`stable_partition` still does!)

ITERATOR CATEGORY MISTAKES? (2)

```
template <class BidirIt>
void inplace_merge(BidirIt first, BidirIt middle, BidirIt last);
```

Again, this algorithm is doable with `forward_iterator`.

STRENGTH REDUCTION

Iterator category relaxation is an important step that is a specific form of *strength reduction*.

> *"In compiler construction, strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations."*
>
> *-- https://en.wikipedia.org/wiki/Strength_reduction*

OPERATIONS TO CONSIDER CAREFULLY

OPERATIONS TO CONSIDER CAREFULLY

- decrement

OPERATIONS TO CONSIDER CAREFULLY

- decrement
- addition (separate from increment)

OPERATIONS TO CONSIDER CAREFULLY

- decrement
- addition (separate from increment)
- replace `it++` with `++it`

OPERATIONS TO CONSIDER CAREFULLY

- decrement
- addition (separate from increment)
- replace `it++` with `++it`
- equality/ordering requirements

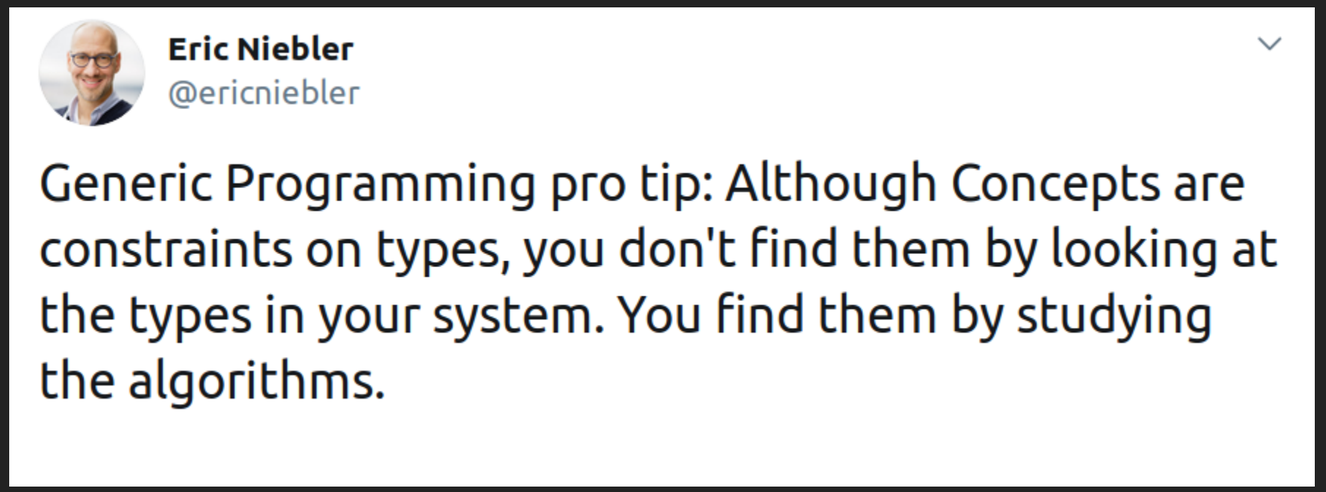
OPERATIONS TO CONSIDER CAREFULLY

- decrement
- addition (separate from increment)
- replace `it++` with `++it`
- equality/ordering requirements
- halving/doubling

OPERATIONS TO CONSIDER CAREFULLY

- decrement
- addition (separate from increment)
- replace `it++` with `++it`
- equality/ordering requirements
- halving/doubling
- ~~replace divisions/multiplications with shifts~~

REQUIREMENTS ON TYPES



AFTER RELAXATION

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value) {
  while (first != last) {
    T half = (std::distance(first, last) + 1) / 2;
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

AFTER RELAXATION

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value) {
  while (first != last) {
    T half = (std::distance(first, last) + 1) / 2;
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

```
$ clang++ -Wconversion main.cpp
```

SIGNED/UNSIGNED CONVERSION

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
T find_missing_element(I first, I last, T value) {
  while (first != last) {
    auto half = static_cast<T>(std::distance(first, last) + 1) / 2; // here
    T m = value + half;
    auto p = std::partition(first, last, [&](auto x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

`distance_type` is signed...

const / constexpr ALL THE THINGS

```
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T find_missing_element(I first, I last, T value) {
  while (first != last) {
    const auto half = static_cast<T>(std::distance(first, last) + 1) / 2;
    const T m = value + half;
    const auto p = std::partition(first, last, [&](const auto& x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

All right, already.

DEFAULT ARGUMENT IS "ZERO"

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T find_missing_element(I first, I last, T value = {}) {
  while (first != last) {
    const auto half = static_cast<T>(std::distance(first, last) + 1) / 2;
    const T m = value + half;
    const auto p = std::partition(first, last, [&](const auto& x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

For easier call sites?

PRECONDITIONS/POSTCONDITIONS?

Let's document them, even if we can't express them in contracts yet.

PRECONDITIONS/POSTCONDITIONS?

Let's document them, even if we can't express them in contracts yet.

- (Pre) [`first`, `last`) does not contain all possible values of `T`

PRECONDITIONS/POSTCONDITIONS?

Let's document them, even if we can't express them in contracts yet.

- (Pre) [`first`, `last`) does not contain all possible values of `T`
- (Pre) `last` is *reachable* from `first`

PRECONDITIONS/POSTCONDITIONS?

Let's document them, even if we can't express them in contracts yet.

- (Pre) [`first`, `last`) does not contain all possible values of `T`
- (Pre) `last` is *reachable* from `first`
- (Pre?) [`first`, `last`) does not contain duplicate values of `T`

PRECONDITIONS/POSTCONDITIONS?

Let's document them, even if we can't express them in contracts yet.

- (Pre) [`first`, `last`) does not contain all possible values of `T`
- (Pre) `last` is *reachable* from `first`
- (Pre?) [`first`, `last`) does not contain duplicate values of `T`
- (Pre?) `value` will not overflow if it is signed

PRECONDITIONS/POSTCONDITIONS?

Let's document them, even if we can't express them in contracts yet.

- (Pre) [`first`, `last`) does not contain all possible values of `T`
- (Pre) `last` is *reachable* from `first`
- (Pre?) [`first`, `last`) does not contain duplicate values of `T`
- (Pre?) `value` will not overflow if it is signed
- (Post) [`first`, `last`) elements are permuted

A BETTER NAME

`find_missing_element` is OK, but verbose, and not completely descriptive.

- we find the *smallest* missing element
- there isn't just one "missing" element

Suggestions?

A BETTER NAME

`find_missing_element` is OK, but verbose, and not completely descriptive.

- we find the *smallest* missing element
- there isn't just one "missing" element

Suggestions?

I propose `min_absent`.

min_absent

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T min_absent(I first, I last, T value = {}) {
  while (first != last) {
    const auto half = static_cast<T>(std::distance(first, last) + 1) / 2;
    const T m = value + half;
    const auto p = std::partition(first, last, [&](const auto& x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

Final version. Or is it?

TESTING WITH MORE TYPES

```
using namespace std::chrono_literals;
std::vector<std::chrono::seconds> v = {0s, 1s, 2s, 3s, 5s};
assert(min_absent(std::begin(v), std::end(v)) == 4s);
```

TESTING WITH MORE TYPES

```
using namespace std::chrono_literals;
std::vector<std::chrono::seconds> v = {0s, 1s, 2s, 3s, 5s};
assert(min_absent(std::begin(v), std::end(v)) == 4s);
```

```
main.cpp:17:14: error: no matching function for call to 'next'
    if (p == std::next(first, half)) {
             ^~~~~~~~~
```

MORE TYPE SUPPORT (0)

```
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T min_absent(I first, I last, T value = {}) {
  while (first != last) {
    const auto half = static_cast<T>(std::distance(first, last) + 1) / 2;
    const T m = value + half;
    const auto p = std::partition(first, last, [&](const auto& x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

`half` isn't the right type.

MORE TYPE SUPPORT (1)

```
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T min_absent(I first, I last, T value = {}) {
  while (first != last) {
    const auto half = (std::distance(first, last) + 1) / 2;
    const T m = value + static_cast<T>(half);
    const auto p = std::partition(first, last, [&](const auto& x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

Requirement: T must be constructible from a (signed) integral type.

TESTING WITH EVEN MORE TYPES

```
using namespace std::chrono_literals;
using T = std::chrono::time_point<std::chrono::system_clock,
                                  std::chrono::seconds>;
std::vector v = {T{0s}, T{1s}, T{2s}, T{3s}, T{5s}};
assert(min_absent(std::begin(v), std::end(v)) == T{4s});
```

TESTING WITH EVEN MORE TYPES

```
using namespace std::chrono_literals;
using T = std::chrono::time_point<std::chrono::system_clock,
                                  std::chrono::seconds>;
std::vector v = {T{0s}, T{1s}, T{2s}, T{3s}, T{5s}};
assert(min_absent(std::begin(v), std::end(v)) == T{4s});
```

```
main.cpp:16:28: error: no matching conversion for static_cast from
'const long' to 'std::chrono::time_point<std::chrono::_V2::system_clock,
std::chrono::duration<long, std::ratio<1, 1> > >'
    const auto m = value + static_cast<T>(half);
                           ^~~~~~~~~~~~~~~~~~~~
```

AFFINE SPACE TYPES

Sometimes, a "point" type and a "difference" type are not the same type, but together form an *affine space*.

Standard examples:

- `std::chrono::time_point` and `std::chrono::duration`
- pointer and `std::ptrdiff_t`

Note: it's possible (common?) for the "point type" to be unsigned, while the "vector type" is signed.

FINALLY*

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T min_absent(I first, I last, T value = {}) {
  using diff_t = decltype(value - value);
  while (first != last) {
    const auto half = (std::distance(first, last) + 1) / 2;
    const T m = value + static_cast<diff_t>(half);
    const auto p = std::partition(first, last, [&](const auto& x) { return x < m; });
    if (p == std::next(first, half)) {
      first = p;
      value = m;
    } else {
      last = p;
    }
  }
  return value;
}
```

*Until we think of something else. But this is pretty good.

DOCUMENTATION

If we want an algorithm worthy of the STL, in addition to behaviour of the algorithm, we need to specify:

DOCUMENTATION

If we want an algorithm worthy of the STL, in addition to behaviour of the algorithm, we need to specify:

- type requirements
  - concepts modelled by the iterator(s)
  - concepts modelled by the other argument(s)

DOCUMENTATION

If we want an algorithm worthy of the STL, in addition to behaviour of the algorithm, we need to specify:

- type requirements
  - concepts modelled by the iterator(s)
  - concepts modelled by the other argument(s)
- return value

DOCUMENTATION

If we want an algorithm worthy of the STL, in addition to behaviour of the algorithm, we need to specify:

- type requirements
  - concepts modelled by the iterator(s)
  - concepts modelled by the other argument(s)
- return value
- complexity guarantees
  - how many applications of comparison
  - how many swaps/copies/constructions/conversions
  - may vary by iterator category

DOCUMENTATION

If we want an algorithm worthy of the STL, in addition to behaviour of the algorithm, we need to specify:

- type requirements
  - concepts modelled by the iterator(s)
  - concepts modelled by the other argument(s)
- return value
- complexity guarantees
  - how many applications of comparison
  - how many swaps/copies/constructions/conversions
  - may vary by iterator category
- exception behaviour

ASIDE: RANGES

Ranges don't give us too many extra algorithms (yet).

Ranges do give us:

- laziness
- algorithms-in-the-iterators

ASIDE: RANGES

Some traditional standard algorithms that ranges embed in iterators ("views"):

- unary transform (projection functions everywhere)
- reverse
- copy (take, drop, etc)
- remove_if (filter)

ASIDE: RANGES

And the "iterator runes"

- counted (_n)
- reverse
- move iterator
- stride, sliding

OVERLOADS

We may also want to provide different versions.

```
template <std::forward_iterator I>
constexpr auto your_algorithm(I first, I last);

template <std::forward_iterator I, std::predicate P = std::less<>>
constexpr auto your_algorithm(I first, I last, P pred = {});

template <std::ranges::forward_range R>
constexpr auto your_algorithm(R range);
```

Or even a parallel version? (Left as an exercise!)

NOT FINALLY*

We implemented `min_absent` with `partition`.

It's also possible to implement it with `nth_element`.

http://quick-bench.com/DaoMWaqBSHHm-ZfhjVh-NiWsejc

*I thought of something else.

min_absent WITH nth_element

```cpp
template <std::forward_iterator I, typename T = std::iter_value_t<I>>
constexpr T min_absent(I first, I last, T value = {}) {
  using diff_t = decltype(value - value);
  while (first != last) {
    const auto half = std::distance(first, last) / 2;
    const auto mid = std::next(first, half);
    std::nth_element(first, mid, last);
    if (*mid == init + static_cast<diff_t>(half)) {
      init = *mid + static_cast<diff_t>(1);
      first = std::next(mid);
    } else {
      last = mid;
    }
  }
  return init;
}
```

NOW WE *ARE* DONE WITH `min_absent`

</case study>

And now for something ~~completely~~ different…

LET'S TALK ABOUT ARGUMENTS

A man walks into an office.

Man: (Michael Palin) Ah. I'd like to have an argument, please.

Receptionist: Certainly sir. Have you been here before?

...

PARAMETER ORDERING

What order should we use for parameters?

Order of parameters is an important consideration.

It determines many things about functions, up to and including their names (as we shall see).

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)
2. `first`, `last` (iterator pair) or `first`, `n` (iterator & length)

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)
2. `first`, `last` (iterator pair) or `first`, `n` (iterator & length)
3. other input iterators (if more than one input range)

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)
2. `first`, `last` (iterator pair) or `first`, `n` (iterator & length)
3. other input iterators (if more than one input range)
4. output iterator(s)

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)
2. `first`, `last` (iterator pair) or `first`, `n` (iterator & length)
3. other input iterators (if more than one input range)
4. output iterator(s)
5. initialization value

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)
2. `first`, `last` (iterator pair) or `first`, `n` (iterator & length)
3. other input iterators (if more than one input range)
4. output iterator(s)
5. initialization value
6. predicate/comparison

PARAMETER ORDERING: OPTION 1

Do as the standard does. Generally the standard follows convention:

1. `ExecutionPolicy` comes first (if applicable)
2. `first`, `last` (iterator pair) or `first`, `n` (iterator & length)
3. other input iterators (if more than one input range)
4. output iterator(s)
5. initialization value
6. predicate/comparison
7. projection/transformation

PARAMETER ORDERING: OPTION 2

We have `bind_front` now. So:

> *"Order arguments according to the partial application you wish to see in the world."*
>
> *-- @aymannadeem*

Or generally: order parameters according to frequency of change.

PARAMETER ORDERING: OPTION 3

Sometimes the order is imposed by the language.

PARAMETER ORDERING: OPTION 3

Sometimes the order is imposed by the language.

- parameters with default values must be last
  - so use overloads instead of default parameters?

PARAMETER ORDERING: OPTION 3

Sometimes the order is imposed by the language.

- parameters with default values must be last
  - so use overloads instead of default parameters?
- variadic arg packs must* come last
  - there aren't any variadic algorithms (yet)

PARAMETER ORDERING: OPTION 3

Sometimes the order is imposed by the language.

- parameters with default values must be last
  - so use overloads instead of default parameters?
- variadic arg packs must* come last
  - there aren't any variadic algorithms (yet)

```
template <typename InputIt, typename OutputIt,
          typename Operation, typename... InputItN>
OutputIt transform(InputIt first, InputIt last, OutputIt d_first,
                   Operation op, InputItN... first_n);
```

PARAMETER ORDER INFORMS NAMING

For binary functions, once you have decided the parameter order according to principles, name it as if it were infix (or member function/UFCS).

```
// with parameters this way around...
bool starts_with(const std::string& s, const std::string_view prefix);
// ...the right name is "starts_with"
// because s "starts_with" prefix
```

```
// with parameters this way around...
bool is_prefix_of(const std::string_view prefix, const std::string& s);
// ...the right name is "is_prefix_of"
// because prefix "is_prefix_of" s
```

EPILOGUE: FOUR ALGORITHMIC PRINCIPLES

- The Law of Useful Return
- The Law of Separating Types
- The Law of Completeness
- The Law of Interface Refinement

USEFUL RETURN

When writing code, it's often the case that you end up computing a value that the calling function doesn't currently need. Later, however, this value may be important when the code is called in a different situation. In this situation, you should obey the law of useful return: A procedure should return all the potentially useful information it computed.

– Alex Stepanov, *From Mathematics to Generic Programming*

USEFUL RETURN: rotate

```cpp
// C++98
template <class ForwardIt>
void rotate(ForwardIt first, ForwardIt n_first, ForwardIt last);

// C++11
template <class ForwardIt>
ForwardIt rotate(ForwardIt first, ForwardIt n_first, ForwardIt last);
```

With C++11, `rotate` was fixed to return the iterator pointing to where `first` ends up.

USEFUL RETURN: copy_n

```
template <class InputIt, class Size, class OutputIt>
OutputIt copy_n(InputIt first, Size count, OutputIt result);
```

Oh dear. *_n algorithms require extra care with the law of Useful Return.

USEFUL RETURN: copy_n

```
template <class InputIt, class Size, class OutputIt>
OutputIt copy_n(InputIt first, Size count, OutputIt result);
```

Oh dear. *_n algorithms require extra care with the law of Useful Return.

```
template<input_iterator I, weakly_incrementable O>
  requires indirectly_copyable<I, O>
constexpr ranges::copy_n_result<I, O>
  ranges::copy_n(I first, iter_difference_t<I> n, O result);
```

THE LAW OF USEFUL RETURN

A procedure should return all the potentially useful information it computed.

This doesn't mean:

- do extra work to return "potentially useful information"
- return things the caller already knows ("just in case?")

It does mean:

- pay attention to non-random-access iterators
- look out for free stuff (e.g. quotient & remainder)

SEPARATING TYPES

Print out your code and take a highlighter to it!

- which variables interact with which other variables?
- which operations do they use?
- are there disjoint subsets?
- think about affine space types
- what iterator strength is required?

SEPARATING TYPES

See "Programming Conversations", Lecture 5 parts 1 & 2

https://www.youtube.com/watch?v=IzNtM038JuI

https://www.youtube.com/watch?v=vxv74Mjt9_0

```
template <class BidirIt, class UnaryPredicate>
BidirIt stable_partition(BidirIt first, BidirIt last, UnaryPredicate p);
```

THE LAW OF SEPARATING TYPES

Do not assume that two types are the same when they may be different.

We saw some of this already with `min_absent` development.

Also, see ranges (iterator pair becomes iterator & sentinel).

LAW OF COMPLETENESS

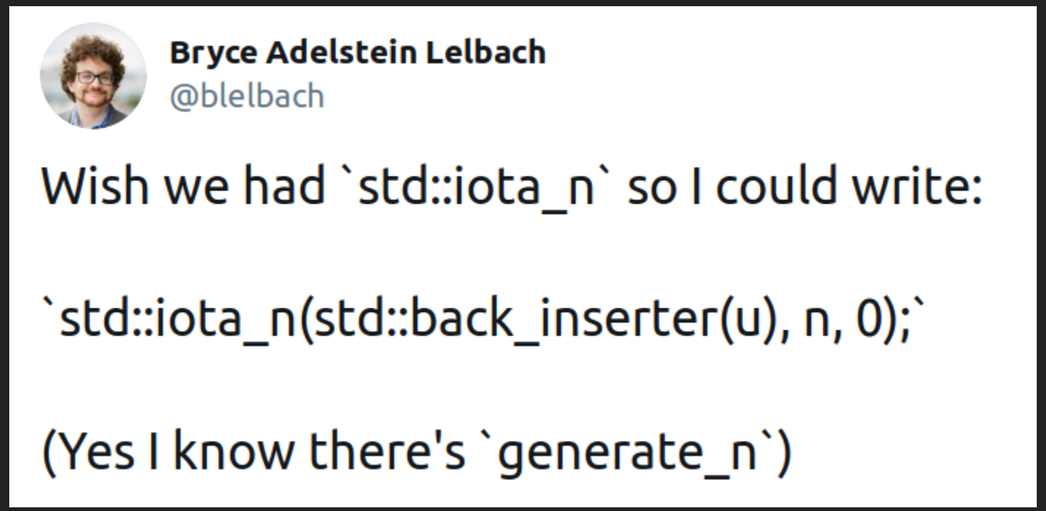
When designing an interface, consider providing all the related procedures.

For example:

- iterator-pair version and iterator-count version
- default to `equal_to`, `less` or take a user-provided predicate
- versions that deal with `string`, `string_view`, `const char *`, single `char`

This sort of design-space exploration often comes out of the Law of Separating Types.

COMPLETENESS VIOLATIONS (1)



Many algorithms could do with the addition of `_n` variants.

We have `copy_n`, `generate_n`, `fill_n`, `for_each_n`.

Oh, and… er… `search_n`.

COMPLETENESS VIOLATIONS (2)

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
constexpr OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
```

Why does this output everything to one place?

I'd like to know the "before" and "after" differences…

COMPLETENESS VIOLATIONS (3)

```
template <class InputIterator, class OutputIterator, class T,
          class BinaryOperation, class UnaryOperation>
OutputIterator transform_exclusive_scan(
    InputIterator first, InputIterator last,
    OutputIterator result, T init,
    BinaryOperation binary_op, UnaryOperation unary_op);
```

`transform_{in,ex}clusive_scan` can't zip two ranges?

`transform` can, `transform_reduce` can.

LAW OF INTERFACE REFINEMENT

Designing interfaces, like designing programs, is a multi-pass activity.

It's hard to know what the right formulation of interface is,
until we have experience using it.

INTERFACE REFINEMENTS (1)

```
template <class T>
constexpr const T& max(const T& a, const T& b);
```

Remarks: Returns the first argument when the arguments are equivalent.

Unfortunately, we're stuck with this one.

INTERFACE REFINEMENTS (2)

```cpp
// C++98
template <class ForwardIt>
void rotate(ForwardIt first, ForwardIt n_first, ForwardIt last);

// C++11
template <class ForwardIt>
ForwardIt rotate(ForwardIt first, ForwardIt n_first, ForwardIt last);
```

Another example of refinement coming years later, and only being recognized in the context of using the algorithms.

FOUR ALGORITHMIC PRINCIPLES

Useful return: A procedure should return all the potentially useful information it computed.

Separating types: Do not assume that two types are the same when they may be different.

Completeness: When designing an interface, consider providing all the related procedures.

Interface refinement: Designing interfaces, like designing programs, is a multi-pass activity.

EPILOGUE: WHY?

We started out with necessity:

- the standard set of algorithms isn't complete

And with the advice:

- no raw loops

EPILOGUE: WHY?

Building and examining algorithms gives us insights into the true nature of the problems we are trying to solve.

It allows us to generalize and/or specialize to see how one problem is like another. (This is "algorithmic intuition".)

It shows how our data actually interacts and can allow reformulations.

It allows us to identify efficiencies at the machine level and the API level.

EPILOGUE: WHAT TO EXPECT FROM RANGES

> *"Have many variants of simple, common algorithms such as find() and copy()."*
>
> *-- Sean Parent*

Ranges will expand the algorithm variations that we can easily use.

This isn't a reason not to continue writing variations and new algorithms.

HAPPY CODING!

- ✓ Study the algorithms
- ✓ Write your own
- ✓ Study the types and concepts
- ✓ Have fun!