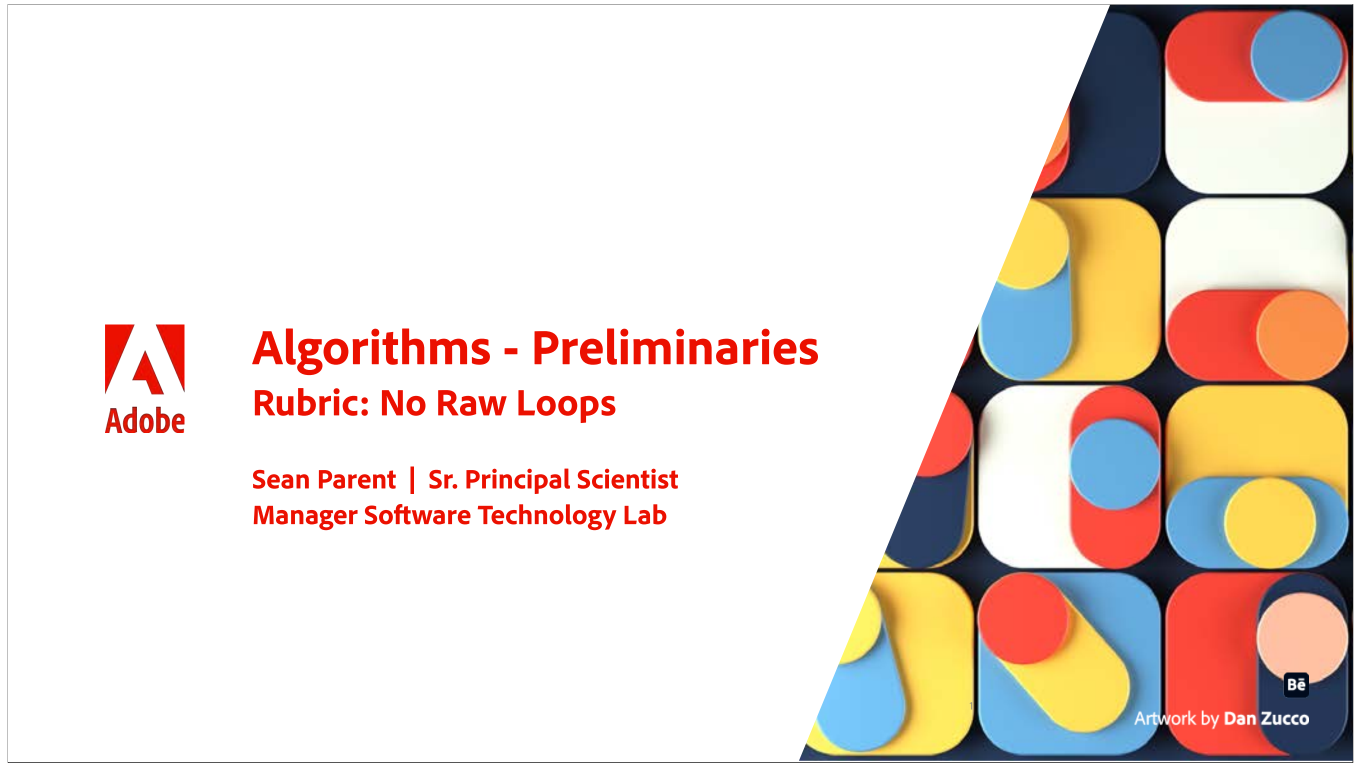


This talk will be a little tight - so I ask you to hold questions until the end. Slide numbers are provided so you can refer back.

A rubric is "a statement of purpose or function." As part of the Better Code seminar, we provide simple rubrics to help you write *Better Code*.

Definition

"An *Algorithm* is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer." – *New Oxford American Dictionary*

Programming is the construction of algorithms. I often hear, "I don't use or need algorithms." Or "I don't write algorithms." But all coding is the construction of algorithms. Sometimes working on a large project can feel like "plumbing" - just trying to connect components to make them do something. But that *is* creating an algorithm.

Often developers do not understand the algorithm they create.

[clarify how plumbing is creating an algorithm.]

A Simple Algorithm

```
int r = a < b ? a : b;
```

- What does this line of code do?

Consider this line of code <click>
This is not a trick question. <wait for answers>
Are you sure? <pause> When I asked, did you have to think about it and double-check?

A Simple Algorithm

```
// r is the minimum of `a` and `b`
int r = a < b ? a : b;
```

Adobe

Does a comment help you understand it? Maybe a little?

A Simple Algorithm

```
int r = min(a, b);
```


Is this more clear?

Functions are often ignored but are our most helpful abstraction for constructing software. We frequently focus on type hierarchies and object networks and ignore the basic function building block. In this talk, we're going to explore functions.

Factoring out simple algorithms can significantly impact readability, even for simple lines of code. A comment is not required where the function is used.

Minimum

```
/// returns the minimum of `a` and `b`
int min(int a, int b) {
    return a < b ? a : b;
}
```


You can write this comment once - or you can write the comment every time you compute the minimum.

Functions name algorithms. The last seminar introduced contracts to specify functions. Postconditions define the semantics or what the function does. Preconditions, not just the parameter types, define the domain of the operation. Many functions are *partial,* and the domain of a partial function is the values over which the function is defined.

Our `min()` function has no preconditions, which is another way of saying the domain of `min()` is the set of values representable by a pair of `int` types.

We state the postcondition in our specification - associating meaning with the name.

We are defining a vocabulary. We should avoid "making up words" and instead use established names within our domain if the semantics of our operation match.

`min()` is a well-established name for the minimum function. This justifies the use of the abbreviation.

Even for a one-line, trivial operation, the name and associated semantics can make the usage easier to reason about.

Minimum

```
/// returns the minimum of `a` and `b`
int min(int a, int b) {
    return a < b ? a : b;
}
```

When implementing an algorithm, we need to reason through each statement

Minimum

```
/// returns the minimum of `a` and `b`
int min(int a, int b) {
    return a < b ? a : b;
}
```

When implementing an algorithm, we need to reason through each statement

- The preconditions of each statement must be satisfied by the statements before

Minimum

```
/// returns the minimum of `a` and `b`
int min(int a, int b) {
    return a < b ? a : b;
}
```

When implementing an algorithm, we need to reason through each statement

- The preconditions of each statement must be satisfied by the statements before
  - Or implied by the preconditions of the algorithm

Minimum

```
/// returns the minimum of `a` and `b`
int min(int a, int b) {
    return a < b ? a : b;
}
```

When implementing an algorithm, we need to reason through each statement

- The preconditions of each statement must be satisfied by the statements before
  - Or implied by the preconditions of the algorithm
- The postconditions for the algorithm must follow from the sequence of statements

Minimum

```
/// returns the minimum of `a` and `b`
int min(int a, int b);
```

Functions allow us to build a vocabulary focused on semantics.

After we have defined our function and are sure it is correct, we no longer have to worry about the implementation.

There is a myth that a limited vocabulary makes code easier to read - but this comes at the expense of limiting the ability to express ideas simply. A NAND gate is very simple and can describe all computations. But we don't program using only NANDs

Naming Functions

Operations with the same semantics should have the same name

Follow existing... The C++ standard library has a relatively rich vocabulary. The vocabulary and conventions in languages differ - defer to your language. C++ shouldn't read like Object Pascal. However, if a language lacks a convention, borrow from another before inventing a new term.

What follows are general recommendations. If your language has different conventions, use them.

Properties... Dictionary definition "an attribute, quality, or characteristic of something." - a non-mutating operation with a single argument.

consider a verb - Example std::list::size(), and adobe::forest::parent().

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

  The name should describe the postconditions and make the use clear

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

  The name should describe the postconditions and make the use clear

  For properties:

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

  The name should describe the postconditions and make the use clear

  For properties:

- nouns: `capacity`

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

  The name should describe the postconditions and make the use clear

  For properties:

- nouns: `capacity`
- adjectives: `empty` (ambiguous but used by convention)

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

  The name should describe the postconditions and make the use clear

  For properties:

- nouns: `capacity`
- adjectives: `empty` (ambiguous but used by convention)
- copular constructions: `is_blue`

Naming Functions

Operations with the same semantics should have the same name

- **Follow existing vocabulary and conventions**

  The name should describe the postconditions and make the use clear

  For properties:

- nouns: `capacity`
- adjectives: `empty` (ambiguous but used by convention)
- copular constructions: `is_blue`
- consider a verb if the complexity is greater than expected

Naming Functions

For mutating operations, use a verb:

<end>
Omit needless words.

Naming *is* hard. Focus on capturing the semantics and how it reads at the call site. When choosing a name, writing down your declaration and looking at it is not enough. Write usages of the name. Speak the language.

Naming Functions

For mutating operations, use a verb:

- verbs: `partition`

Naming Functions

For mutating operations, use a verb:

- verbs: `partition`

  For setting stable, readable properties, with *footprint complexity*

Naming Functions

For mutating operations, use a verb:

- verbs: `partition`

  For setting stable, readable properties, with *footprint complexity*

- Prefix with the verb, `set_`, i.e. `set_numerator`

Naming Functions

For mutating operations, use a verb:

- verbs: `partition`

  For setting stable, readable properties, with *footprint complexity*

- Prefix with the verb, `set_`, i.e. ``set_numerator``

  Clarity is of the highest priority. Don't construct unnatural verb phrases

Naming Functions

For mutating operations, use a verb:

- verbs: `partition`

For setting stable, readable properties, with *footprint complexity*

- Prefix with the verb, `set_`, i.e. ``set_numerator``

Clarity is of the highest priority. Don't construct unnatural verb phrases

- `intersection(a, b)` not `calculate_intersection(a, b)`

Naming Functions

For mutating operations, use a verb:

- verbs: `partition`

  For setting stable, readable properties, with *footprint complexity*

- Prefix with the verb, `set_`, i.e. `` `set_numerator` ``

  Clarity is of the highest priority. Don't construct unnatural verb phrases

- `intersection(a, b)` not `calculate_intersection(a, b)`
- `name()` not `get_name()`

Argument Types

let: the caller's argument is not modified

Adobe

[Split this up to be not so C++-centric. With definitions for each term. Maybe a table. Conventions - for construction.]

Three basic ideas in argument passing - this is how they reflect in C++; other languages will have a different mapping.

"Small" is "fits in a register." "Expected" means when used in a template.
Many languages don't have a notion of "sink" - develop or borrow a convention for this use.
Unfortunately, forwarding references have the same syntax as rvalue-references, and disambiguating with *enable_if* or *requires* clauses adds too much complexity.
Prefer return values to out arguments; otherwise, treat as *inout*.
Const in C++ is not transitive - treat it as if it were.

Argument Types

let: the caller's argument is not modified

- `const T&`

Argument Types

let: the caller's argument is not modified

- `const T&`
- T, for known or expected small types such as primitive types, iterators, and function objects

Argument Types

let: the caller's argument is not modified

- `const T&`
- `T`, for known or expected small types such as primitive types, iterators, and function objects

in-out: the caller's argument is modified

Argument Types

let: the caller's argument is not modified

- `const T&`
- `T`, for known or expected small types such as primitive types, iterators, and function objects

in-out: the caller's argument is modified

- `T&`

Argument Types

let: the caller's argument is not modified

- `const T&`
- `T`, for known or expected small types such as primitive types, iterators, and function objects

in-out: the caller's argument is modified

- `T&`
- Unless `sizeof(T)` is significant, prefer a sink argument and result to an in-out argument

Argument Types

let: the caller's argument is not modified

- `const T&`
- `T`, for known or expected small types such as primitive types, iterators, and function objects

in-out: the caller's argument is modified

- `T&`
- Unless `sizeof(T)` is significant, prefer a sink argument and result to an in-out argument

sink: the argument is consumed or escaped

Argument Types

let: the caller's argument is not modified

- `const T&`
- `T`, for known or expected small types such as primitive types, iterators, and function objects

in-out: the caller's argument is modified

- `T&`
- Unless `sizeof(T)` is significant, prefer a sink argument and result to an in-out argument

sink: the argument is consumed or escaped

- `T&&`, where `T` is not deduced

Argument Types

let: the caller's argument is not modified

- `const T&`
- T, for known or expected small types such as primitive types, iterators, and function objects

in-out: the caller's argument is modified

- T&
- Unless `sizeof(T)` is significant, prefer a sink argument and result to an in-out argument

sink: the argument is consumed or escaped

- T&&, where T is not deduced
- `T,` For known or expected small types and to avoid forward references

Ranges as Arguments

[Split this up to be not so C++-centric. With definitions for each term. Maybe a table. Conventions - for construction.]

Three basic ideas in argument passing - this is how they reflect in C++; other languages will have a different mapping.

"Small" is "fits in a register." "Expected" means when used in a template.
Many languages don't have a notion of "sink" - develop or borrow a convention for this use.
Unfortunately, forwarding references have the same syntax as rvalue-references, and disambiguating with *enable_if* or *requires* clauses adds too much complexity.
Prefer return values to out arguments; otherwise, treat as *inout*.
Const in C++ is not transitive - treat it as if it were.

Ranges as Arguments

spans, views, iterator pairs, and so on are a way to pass a range of objects as if they were a simple argument

Ranges as Arguments

spans, views, iterator pairs, and so on are a way to pass a range of objects as if they were a simple argument

let: `value_type` is `const` (i.e. `vector::const_iterator`)

Ranges as Arguments

spans, views, iterator pairs, and so on are a way to pass a range of objects as if they were a simple argument

let: `value_type` is `const` (i.e. `vector::const_iterator`)

in-out: `value_type` is *not* `const`

Ranges as Arguments

spans, views, iterator pairs, and so on are a way to pass a range of objects as if they were a simple argument

let: `value_type` is `const` (i.e. `vector::const_iterator`)

in-out: `value_type` is *not* `const`

sink: input range

Ranges as Arguments

spans, views, iterator pairs, and so on are a way to pass a range of objects as if they were a simple argument

let: `value_type` is `const` (i.e. `vector::const_iterator`)

in-out: `value_type` is *not* `const`

sink: input range

result: output iterator

Argument Types

Pointer types are ambiguous
const doesn't propagate
null vs. non-null
sink vs. not sink

Strengthen shared convention to sink, but if the convention is to copy, follow convention

A C++ proposal (part of TS 2/3) for a const propagating wrapper exists.

I don't have a good suggestion of a function result granting access - "access," "view", ...

Argument Types

Avoid pointer types as arguments

Argument Types

Avoid pointer types as arguments

- `const` doesn't propagate, act as if it does

Argument Types

Avoid pointer types as arguments

- `const` doesn't propagate, act as if it does

In reference-semantic languages, use conventions (examples)

Argument Types

Avoid pointer types as arguments

- `const` doesn't propagate, act as if it does

In reference-semantic languages, use conventions (examples)

- Arguments to "init", "set", and "assign" methods are *sink* arguments (caller cannot use after invoke)

Argument Types

Avoid pointer types as arguments

- `const` doesn't propagate, act as if it does

In reference-semantic languages, use conventions (examples)

- Arguments to "init", "set", and "assign" methods are *sink* arguments (caller cannot use after invoke)
- Functions with names with *unambiguous* verbs have *in-out* arguments

Argument Types

Avoid pointer types as arguments

- `const` doesn't propagate, act as if it does

In reference-semantic languages, use conventions (examples)

- Arguments to "init", "set", and "assign" methods are *sink* arguments (caller cannot use after invoke)
- Functions with names with *unambiguous* verbs have *in-out* arguments
- All other arguments are *let* (read-only, copied if escaped)

Argument Types

Avoid pointer types as arguments

- `const` doesn't propagate, act as if it does

In reference-semantic languages, use conventions (examples)

- Arguments to "init", "set", and "assign" methods are *sink* arguments (caller cannot use after invoke)
- Functions with names with *unambiguous* verbs have *in-out* arguments
- All other arguments are *let* (read-only, copied if escaped)
- Results of functions with names starting with "alloc," "new," "copy," or "create" are owned solely by the caller; other results are read-only

Argument Types

```
void display(const vector<unique_ptr<widget>>& a) {
    //...

    a[0]->set_name("displayed"); // DO NOT

    //...
}
```


Don't do this - we'll discuss value semantics more in future seminars, but there is no way to impose transitive const when using reference semantics.

Implicit Preconditions

Object Lifetimes

Adobe

Object lifetime can be broken with shared mutable references from shared structures, threads, callbacks, or reentrancy.

The implicit preconditions apply to the arguments passes and to all objects reachable through those arguments. If using reference instead of value semantics, this means the requirements are _deep_.

Implicit Preconditions

Object Lifetimes

- The caller must ensure that referenced arguments are valid for the duration of the call

Implicit Preconditions

Object Lifetimes

- The caller must ensure that referenced arguments are valid for the duration of the call
- The callee must copy (or move for sink arguments) an argument to retain it after returning

Implicit Preconditions

Object Lifetimes

- The caller must ensure that referenced arguments are valid for the duration of the call
- The callee must copy (or move for sink arguments) an argument to retain it after returning

Meaningless objects

Implicit Preconditions

Object Lifetimes

- The caller must ensure that referenced arguments are valid for the duration of the call
- The callee must copy (or move for sink arguments) an argument to retain it after returning

Meaningless objects

- A meaningless object should not be passed as an argument (i.e., an invalid pointer).

Implicit Preconditions

Law of Exclusivity

The _Law of Exclusivity_ is borrowed from Swift, and the term was coined by John McCall. C++ does not enforce this rule; it must be manually enforced.

No aliased object under mutation.

The C++ standard library is inconsistent in how it deals with aliasing. Unless aliasing is explicitly allowed, avoid it. Where it is allowed, document (with a comment) any code relying on the behavior.

Nearly every crash is caused by a violation of these implicit preconditions. dereferencing an invalid pointer, using an object after its lifetime, or aliasing a mutable object. Take care! This is a strong argument for why Rust or Val.

Implicit Preconditions

Law of Exclusivity

- To modify a variable, exclusive access to that variable is required

Implicit Preconditions

Law of Exclusivity

- To modify a variable, exclusive access to that variable is required
- This applies to *in-out* and *sink* arguments and is the caller's responsibility

Implicit Preconditions

Law of Exclusivity

- To modify a variable, exclusive access to that variable is required
- This applies to *in-out* and *sink* arguments and is the caller's responsibility

```
vector a{0, 0, 1, 0, 1 };
erase(a, a[0]);
display(a);
```

Implicit Preconditions

Law of Exclusivity

- To modify a variable, exclusive access to that variable is required
- This applies to *in-out* and *sink* arguments and is the caller's responsibility

```
vector a{0, 0, 1, 0, 1 };
erase(a, a[0]);
display(a);

{ 1, 0 }
```

Implicit Preconditions

Law of Exclusivity

- To modify a variable, exclusive access to that variable is required
- This applies to *in-out* and *sink* arguments and is the caller's responsibility

```
vector a{0, 0, 1, 0, 1 };
erase(a, int{a[0]});
display(a);
```

Implicit Preconditions

Law of Exclusivity

- To modify a variable, exclusive access to that variable is required
- This applies to *in-out* and *sink* arguments and is the caller's responsibility

```
vector a{0, 0, 1, 0, 1 };
erase(a, int{a[0]});
display(a);

{ 1, 1 }
```

Implicit Postconditions

Any internal references to (possibly) remote parts held by the caller to an object are assumed to be invalided on return from a mutating operation

Internal references include pointers, iterators, even indices, etc.
Unless the container docs specifically say the iterator is not invalidated, assume it is. Reliance on a class guarantee for reference stability should be noted in a comment at the use site.

The reference returned from vector::back is good until the vector is modified or its lifetime ends

Implicit Postconditions

Any internal references to (possibly) remote parts held by the caller to an object are assumed to be invalided on return from a mutating operation

- Part of the Law of Exclusivity

Implicit Postconditions

Any internal references to (possibly) remote parts held by the caller to an object are assumed to be invalided on return from a mutating operation

- Part of the Law of Exclusivity

```
container<int> a{ 0, 1, 2, 3 };
auto f = begin(a);
a.push_back(5);
// `f` is now invalid and cannot be used
```

Implicit Postconditions

Any internal references to (possibly) remote parts held by the caller to an object are assumed to be invalided on return from a mutating operation

- Part of the Law of Exclusivity

```
container<int> a{ 0, 1, 2, 3 };
auto f = begin(a);
a.push_back(5);
// `f` is now invalid and cannot be used
```

A returned reference must be to one (or a part of one) of the arguments to the function and is valid until the argument is modified or its lifetime ends

Implicit Postconditions

Any internal references to (possibly) remote parts held by the caller to an object are assumed to be invalided on return from a mutating operation

- Part of the Law of Exclusivity

```
container<int> a{ 0, 1, 2, 3 };
auto f = begin(a);
a.push_back(5);
// `f` is now invalid and cannot be used
```

A returned reference must be to one (or a part of one) of the arguments to the function and is valid until the argument is modified or its lifetime ends

- Example: the reference returned from `vector::back()`

Trivial vs Non-Trivial Algorithms

A *trivial* algorithm does not require iteration

iteration and recursion and interchangeable - from now on we will just call it "iteration" but statements apply to both.

Trivial vs Non-Trivial Algorithms

A *trivial* algorithm does not require iteration

- Examples: `swap()`, `exchange()`, `min()`, `max()`, `clamp()`, `tolower()`…

Adobe

Trivial vs Non-Trivial Algorithms

A *trivial* algorithm does not require iteration

- Examples: `swap()`, `exchange()`, `min()`, `max()`, `clamp()`, `tolower()`…

A *non-trivial* algorithm requires iteration

Trivial vs Non-Trivial Algorithms

A *trivial* algorithm does not require iteration

- Examples: `swap()`, `exchange()`, `min()`, `max()`, `clamp()`, `tolower()`…

A *non-trivial* algorithm requires iteration

- iteration may be implemented as a loop or recursion

Reasoning About Iteration

To show that a loop or recursion is correct, we need to demonstrate two things:

A finite decreasing property - there must be a mapping of the loop onto natural numbers. You may not know the numbers - but you must prove the mapping exists and that the numbers are decreasing.

Reasoning About Iteration

To show that a loop or recursion is correct, we need to demonstrate two things:

- An invariant that holds at the start and after each iteration

Reasoning About Iteration

To show that a loop or recursion is correct, we need to demonstrate two things:

- An invariant that holds at the start and after each iteration
- A finite decreasing property where termination happens when the property is zero

Reasoning About Iteration

To show that a loop or recursion is correct, we need to demonstrate two things:

- An invariant that holds at the start and after each iteration
- A finite decreasing property where termination happens when the property is zero

The postcondition of the iteration is the above invariant when the decreasing property reaches zero

Erase

```
template <class T>
void erase(vector<T>& c, const T& value) {
    c.erase(remove(begin(c), end(c), value), c.end());
}
```


We used standard `erase` a moment ago. erase is built using the `remove()` algorithm. If you have tried to roll your code to erase elements from a container, you might know it can be tricky. Erasing each element going forward gets complex because positions keep moving. Going backward and erasing each element is more straightforward, but both approaches are quadratic. Let's build the remove algorithm to see how to do it.

Remove

```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \return the position, `b`, such that `[f, b)` contains all the
        values in `[f, l)` not equal to `a` in the original order

    values in `[b, l)` are unspecified
*/

template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I;
```

Adobe

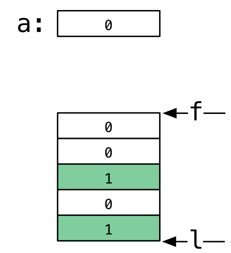
We used `erase` a moment ago. erase is built using the `remove()` algorithm. If you have tried to roll your code to erase elements from a container, you might know it can be tricky. Erasing each element going forward gets complex because positions keep moving. Going backward and erasing each element is more straightforward, but both approaches are quadratic. Let's build the remove algorithm to see how to do it.
In order

Remove

```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \return the position, `b`, such that `[f, b)` contains all the
        values in `[f, l)` not equal to `a` in the original order

    values in `[b, l)` are unspecified
*/

template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I;

vector a{0, 0, 1, 0, 1 };
erase(a, int{a[0]});
```

Remove

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
```



[rework - shorter example direct from erase - show erase implementation first. Execute a second time showing the aliasing issue.]

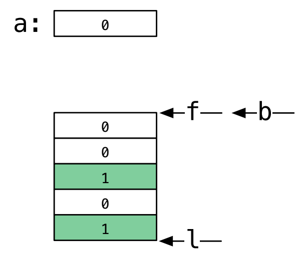
Say "in order" when reading the invariant

[At end, reread the invariant and decreasing]

Because at termination p equals l, it follow that `[f, b)` contains all the values in `[f, l)` not equal to `a`.

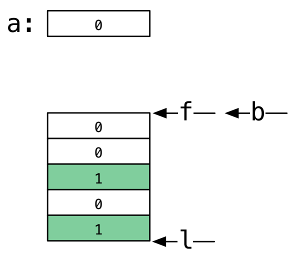
Remove

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
```

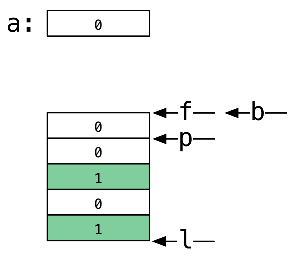


Remove

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
```

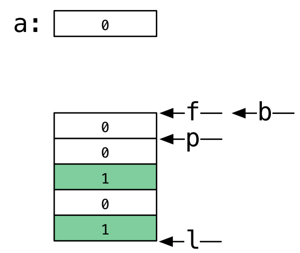


Remove



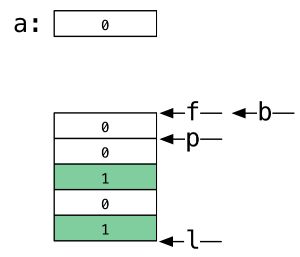

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
```

Remove



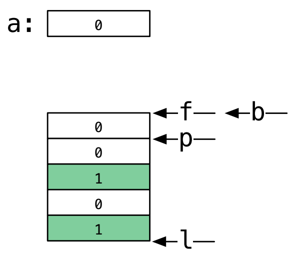

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
```

Remove



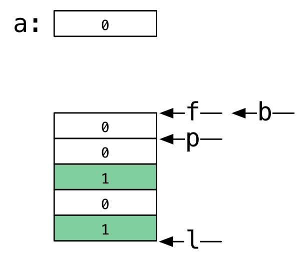

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
```

Remove



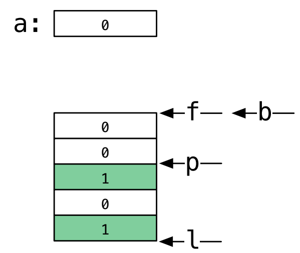

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
```

Remove



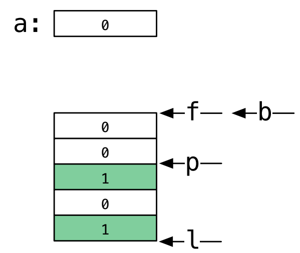

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
```

Remove



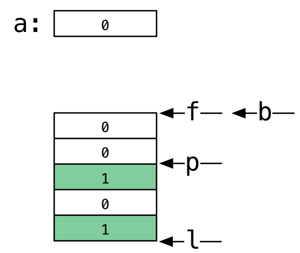

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
```

Remove



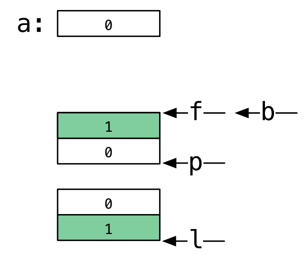

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



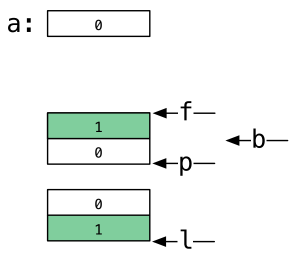

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



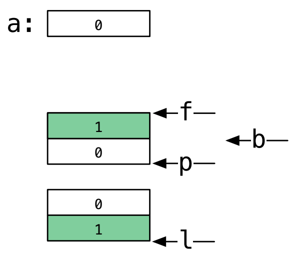

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



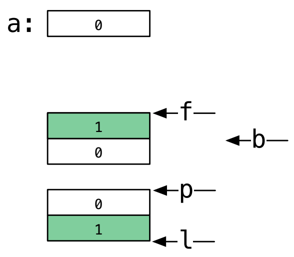

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



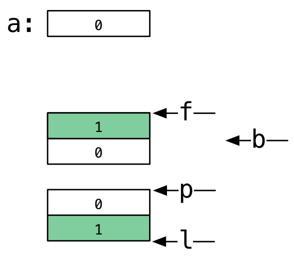

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



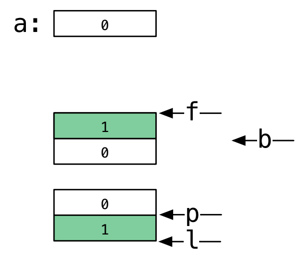

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



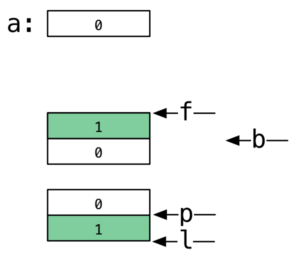

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



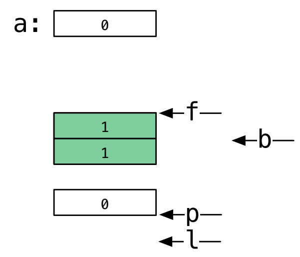

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



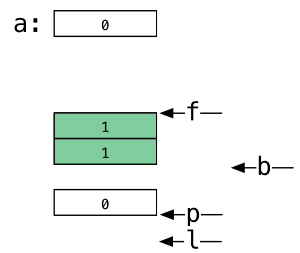

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



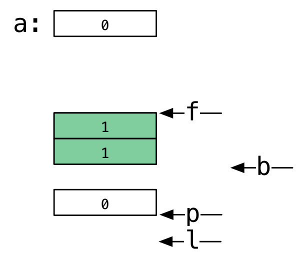

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



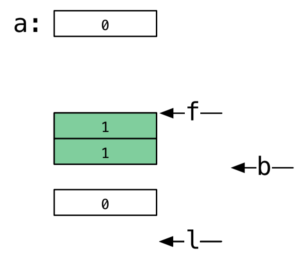

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



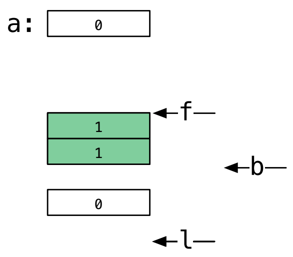

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



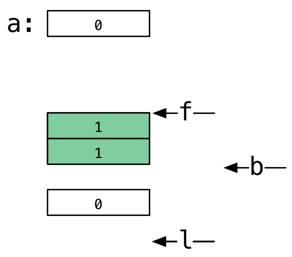

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove




```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
    return b;
```

Remove




```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
    return b;
}
```

Remove

```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \return the position, `b`, such that `[f, b)` contains all the
        values in `[f, l)` not equal to `a` in the original order

    values in `[b, l)` are unspecified
*/

template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I;
```

Adobe

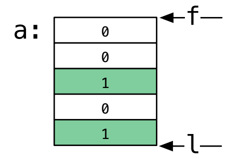
We used `erase` a moment ago. erase is built using the `remove()` algorithm. If you have tried to roll your code to erase elements from a container, you might know it can be tricky. Erasing each element going forward gets complex because positions keep moving. Going backward and erasing each element is more straightforward, but both approaches are quadratic. Let's build the remove algorithm to see how to do it.
In order

Remove

```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \return the position, `b`, such that `[f, b)` contains all the
        values in `[f, l)` not equal to `a` in the original order

    values in `[b, l)` are unspecified
*/

template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I;

vector a{0, 0, 1, 0, 1 };
erase(a, a[0]);
```

Remove

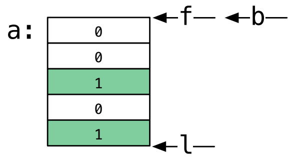
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
```



Say "in order" when reading the invariant
[after overwriting a - at end of loop] : How is the invariant doing - it is violated!

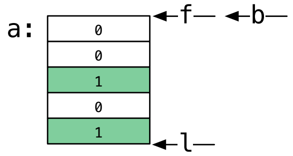
Remove

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
```



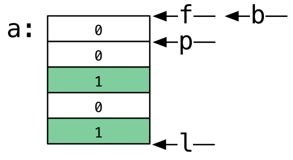
Remove

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
```

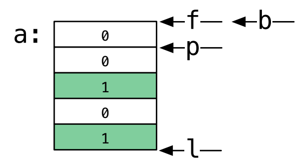


Remove

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
```

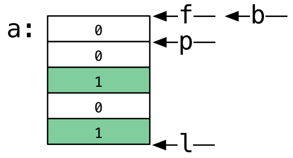


Remove



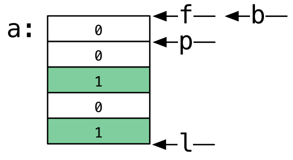

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
```

Remove



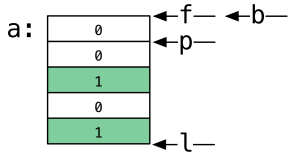

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
```

Remove



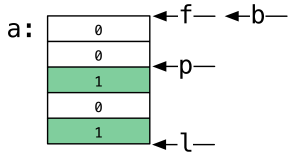

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
```

Remove



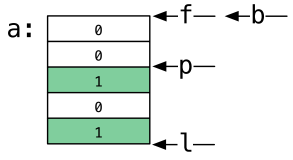

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
```

Remove



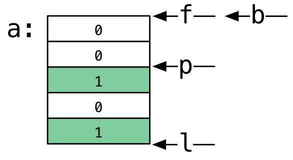

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
```

Remove



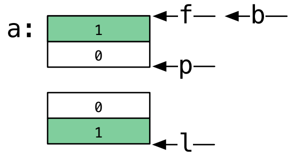

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



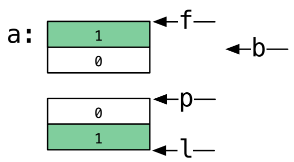

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



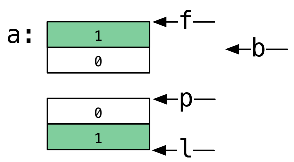

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



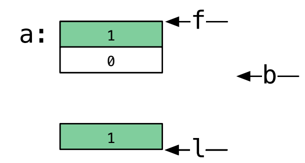

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove



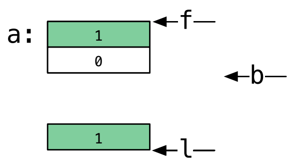

```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
```

Remove




```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
    return b;
```

Remove




```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l) {
        if (*p != a) {
            *b = std::move(*p);
            ++b;
        }
        ++p;
    }
    return b;
}
```

Remove

```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \return the position, `b`, such that `[f, b)` contains all the
        values in `[f, l)` not equal to `a` in the original order

    values in `[b, l)` are unspecified
*/

template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I;

vector a{0, 0, 1, 0, 1 };
erase(a, a[0]);
```


We used `erase` a moment ago. erase is built using the `remove()` algorithm. If you have tried to roll your code to erase elements from a container, you might know it can be tricky. Erasing each element going forward gets complex because positions keep moving. Going backward and erasing each element is more straightforward, but both approaches are quadratic. Let's build the remove algorithm to see how to do it.
In order

Remove

```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \return the position, `b`, such that `[f, b)` contains all the
        values in `[f, l)` not equal to `a` in the original order

    values in `[b, l)` are unspecified
*/

template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I;

vector a{0, 0, 1, 0, 1 };
erase(a, a[0]);
```

Sequences

For a sequence of *n* elements, there are *n + 1* positions

Iteration and recursion imply some form of sequencing. It is essential to understand the properties of sequences for reasoning about loops and iterations. The ideas of sequences extend to multidimensional and numeric algorithms as well. There must existing a mapping to natural numbers for all loops and iterations.

A closed interval cannot represent an empty interval and is missing one position.
An open interval has one extra position. In an open interval, `f` and `l ` cannot be equal. The empty range of discrete elements is (f, f + 1). Open and closed intervals are mathematic constructs and are most helpful when dealing with continuous values.

Sequences

For a sequence of n elements, there are $n + 1$ positions

Ways to represent a range of elements

Sequences

For a sequence of *n* elements, there are *n + 1* positions

Ways to represent a range of elements

- Closed interval [f, l]

Sequences

For a sequence of *n* elements, there are *n + 1* positions

Ways to represent a range of elements

- Closed interval [f, l]
- Open interval (f, l)

Sequences

For a sequence of *n* elements, there are *n + 1* positions

Ways to represent a range of elements

- Closed interval [f, l]
- Open interval (f, l)
- Half-open interval [f, l)

Sequences

For a sequence of *n* elements, there are *n + 1* positions

Ways to represent a range of elements

- Closed interval [f, l]
- Open interval (f, l)
- Half-open interval [f, l)
  - By strong convention, open on the right

Half-Open Intervals

[p, p) represents an empty range at position p

- All empty ranges are not equal

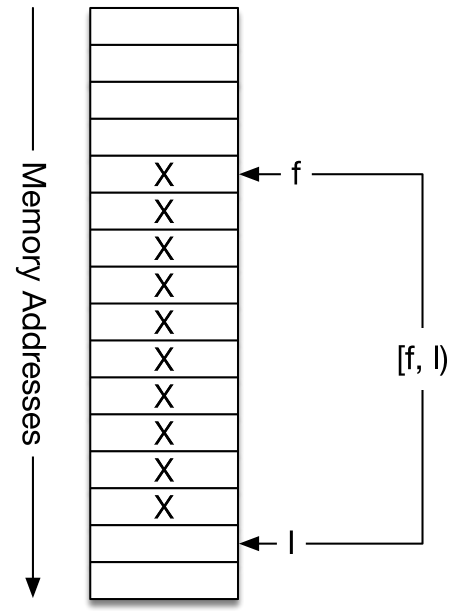
Cannot express the last item in a set with positions of the same set type

- i.e., `[INT_MIN, INT_MAX]` is not expressible as a half-open interval with type `int`

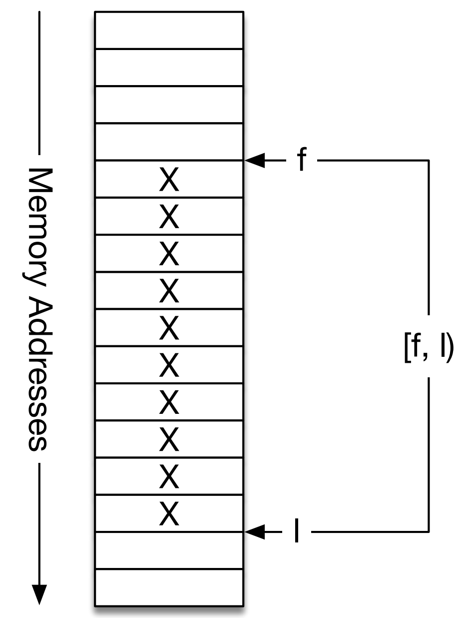
Think of the positions as the lines between the elements

Or fence posts.

Half-Open Intervals



Half-Open Intervals



first and *last* or *begin()* and *end()* are the first and last *positions*, not the first and last elements.

Half-Open Intervals

In this model, there is a symmetry with reverse ranges *(l, f]*

- The dereference operation is asymmetric. dereferencing at a position *p* is the value in *[p, p + 1)*

Half-open intervals avoid off-by-one errors and confusion about *before* or *after*

In C and C++, half-open intervals are built into the language. For any object, `a`, `&a` is a pointer to the object, and `&a + 1` is a valid pointer but may not be dereferenceable.

- Any object can be treated as a range of one element

Alex Stepanov (the creator of STL) would like "while first does not equal last" engraved on his tombstone.

[slightly longer pause on code - build for result.]

Half-Open Intervals

In this model, there is a symmetry with reverse ranges *(l, f]*

- The dereference operation is asymmetric. dereferencing at a position *p* is the value in *[p, p + 1)*

Half-open intervals avoid off-by-one errors and confusion about *before* or *after*

In C and C++, half-open intervals are built into the language. For any object, `a`, `&a` is a pointer to the object, and `&a + 1` is a valid pointer but may not be dereferenceable.

- Any object can be treated as a range of one element

```
int a{42};
```

Half-Open Intervals

In this model, there is a symmetry with reverse ranges *(l, f]*

- The dereference operation is asymmetric. dereferencing at a position *p* is the value in *[p, p + 1)*

Half-open intervals avoid off-by-one errors and confusion about *before* or *after*

In C and C++, half-open intervals are built into the language. For any object, `a`, `&a` is a pointer to the object, and `&a + 1` is a valid pointer but may not be dereferenceable.

- Any object can be treated as a range of one element

```
int a{42};
copy(&a, &a + 1, ostream_iterator<int>(cout));
```

Half-Open Intervals

In this model, there is a symmetry with reverse ranges *(l, f]*

- The dereference operation is asymmetric. dereferencing at a position *p* is the value in *[p, p + 1)*

Half-open intervals avoid off-by-one errors and confusion about *before* or *after*

In C and C++, half-open intervals are built into the language. For any object, `a`, `&a` is a pointer to the object, and `&a + 1` is a valid pointer but may not be dereferenceable.

- Any object can be treated as a range of one element

```
int a{42};
copy(&a, &a + 1, ostream_iterator<int>(cout));
42
```

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

Positions could be pointers, iterators, indices...
Null Terminated Byte String

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

- pair of positions: *[f, l)*

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

- pair of positions: *[f, l)*
- position and count: *[f, f + n)*, use `_n` suffix

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

- pair of positions: *[f, l)*
- position and count: *[f, f + n)*, use `_n` suffix
- position and predicate: *[f, predicate)*, use `_until` suffix

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

- pair of positions: *[f, l)*
- position and count: *[f, f + n)*, use `_n` suffix
- position and predicate: *[f, predicate)*, use `_until` suffix
- position and sentinel: *[f, is_sentinel)*, i.e. NTBS (C string)

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

- pair of positions: *[f, l)*
- position and count: *[f, f + n)*, use `_n` suffix
- position and predicate: *[f, predicate)*, use `_until` suffix
- position and sentinel: *[f, is_sentinel)*, i.e. NTBS (C string)
- unbounded: *[f, ...)*, limit is dependent on an extrinsic relationship

Half-Open Intervals

Half-open intervals can be represented in a variety of forms

- pair of positions: *[f, l)*
- position and count: *[f, f + n),* use `_n` suffix
- position and predicate: *[f, predicate)*, use `_until` suffix
- position and sentinel: *[f, is_sentinel)*, i.e. NTBS (C string)
- unbounded: *[f, ...),* limit is dependent on an extrinsic relationship
  - i.e., the range is require to be the same length or greater than another range

Much More

Composing Algorithms

Sort maps the relationship
We'll talk more about structured data and relationships in future seminars

[There will be a part 2]

Much More

Composing Algorithms

Complexity and efficiency

Much More

Composing Algorithms

Complexity and efficiency

Sorting and heap algorithms

Much More

Composing Algorithms

Complexity and efficiency

Sorting and heap algorithms

- Encoding relationships between properties into structural relationships to create *structured data*

Much More

Composing Algorithms

Complexity and efficiency

Sorting and heap algorithms

- Encoding relationships between properties into structural relationships to create *structured data*
- i.e., $a < b$ *implies* $position(a) < position(b)$

Why No Raw Loops?

Difficult to reason about and difficult to prove post conditions

This brings us back to our rubric

Why No Raw Loops?

Difficult to reason about and difficult to prove post conditions

Error prone and likely to fail under non-obvious conditions

Why No Raw Loops?

Difficult to reason about and difficult to prove post conditions

Error prone and likely to fail under non-obvious conditions

Introduce non-obvious performance problems

Why No Raw Loops?

Difficult to reason about and difficult to prove post conditions

Error prone and likely to fail under non-obvious conditions

Introduce non-obvious performance problems

Complicates reasoning about the surrounding code

Alternatives to Raw Loops

Use an existing algorithm

Most of the standard algorithms have all been machine proven to be correct - this is not Adobe's policy publishing provides the same bonus as a patent bonus and some legal protections.

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library
  - Preferably open source

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library
  - Preferably open source

Invent a new algorithm

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library
  - Preferably open source

Invent a new algorithm

- Write a paper

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library
  - Preferably open source

Invent a new algorithm

- Write a paper
- Give talks

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library
  - Preferably open source

Invent a new algorithm

- Write a paper
- Give talks
- Become famous!

Alternatives to Raw Loops

Use an existing algorithm

- Prefer standard algorithms if available

Implement a known algorithm as a general function

- Contribute it to a library
  - Preferably open source

Invent a new algorithm

∅ Patents

- Write a paper
- Give talks
- Become famous!

Q & A

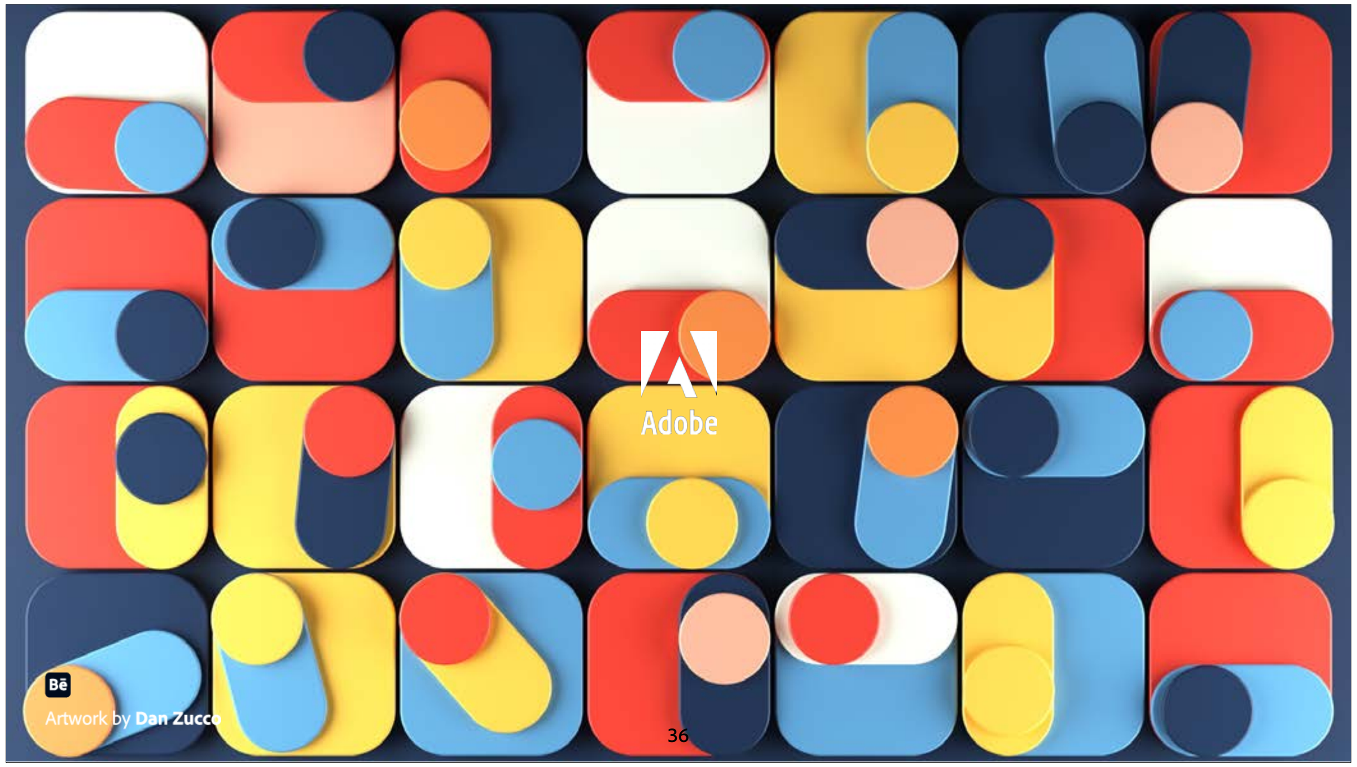
About the artist

Dan Zucco

London-based 3D art and motion director Dan Zucco creates repeating 2D patterns and brings them to life as 3D animated loops. Inspired by architecture, music, modern art, and generative design, he often starts in Adobe Illustrator and builds his animations using Adobe After Effects and Cinema 4D. Zucco's objective for this piece was to create a geometric design that felt like it could have an infinite number of arrangements.

Made with

Adobe Illustrator Adobe After Effects


Adobe
Artwork by Dan Zucco