Adobe

Algorithms - Preliminaries

Rubric: No Raw Loops

Sean Parent | Sr. Principal Scientist
Manager Software Technology Lab

Artwork by Dan Zucco

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*

A Simple Algorithm

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

- What does this line of code do?

A Simple Algorithm

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

A Simple Algorithm

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

Minimum

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

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.

Naming Functions

Operations with the same semantics should have the same name

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:

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

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

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

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

    //...
}
```

Implicit Preconditions

Object Lifetimes

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

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

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

Trivial vs Non-Trivial Algorithms

A *trivial* algorithm does not require iteration

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

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:

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());
}
```

Remove

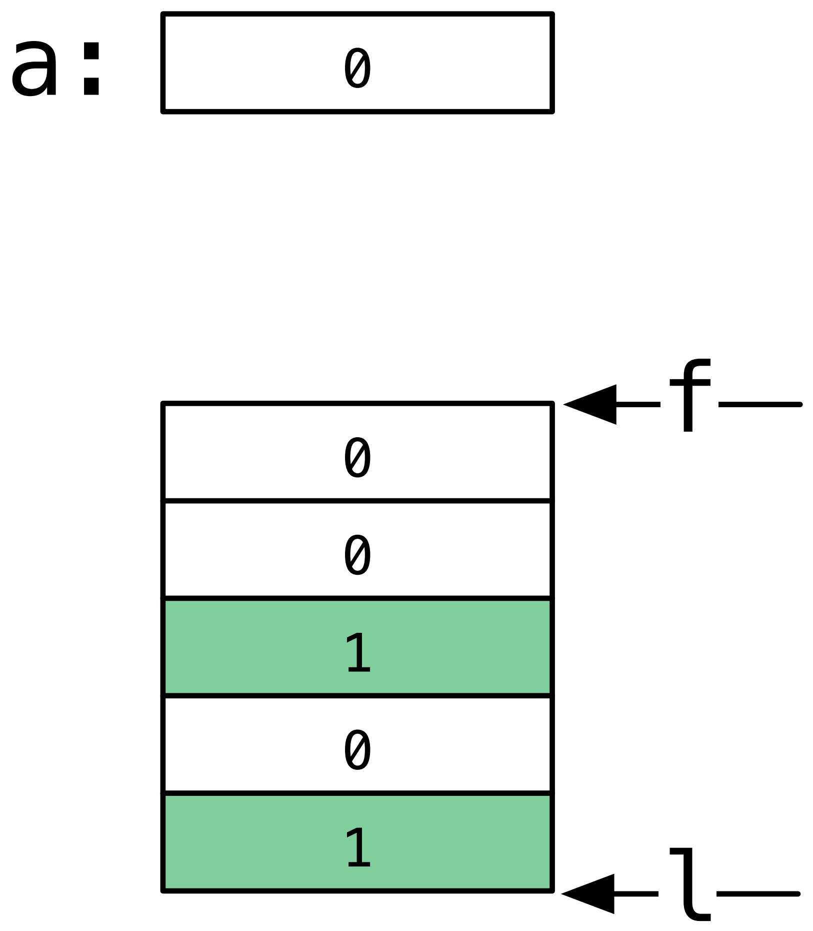
```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \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;
```

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]});
```

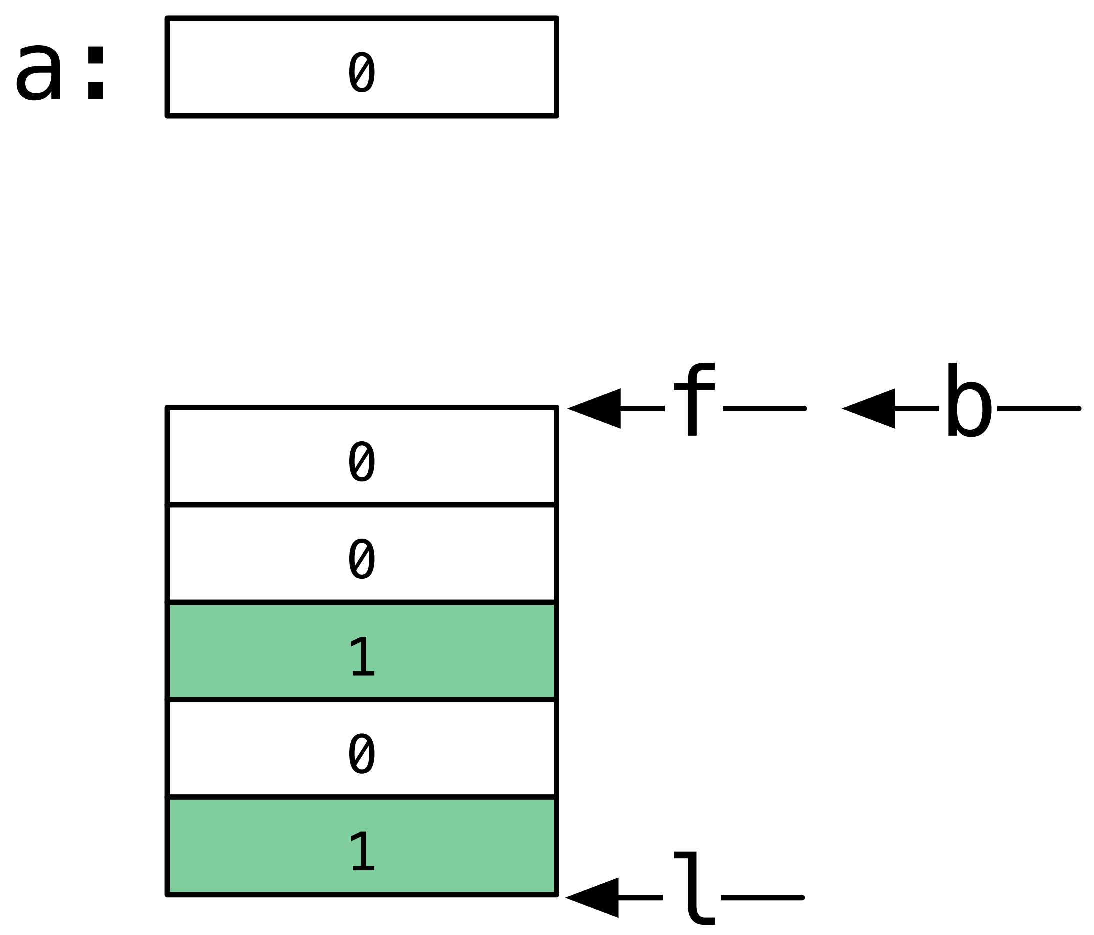
Remove

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



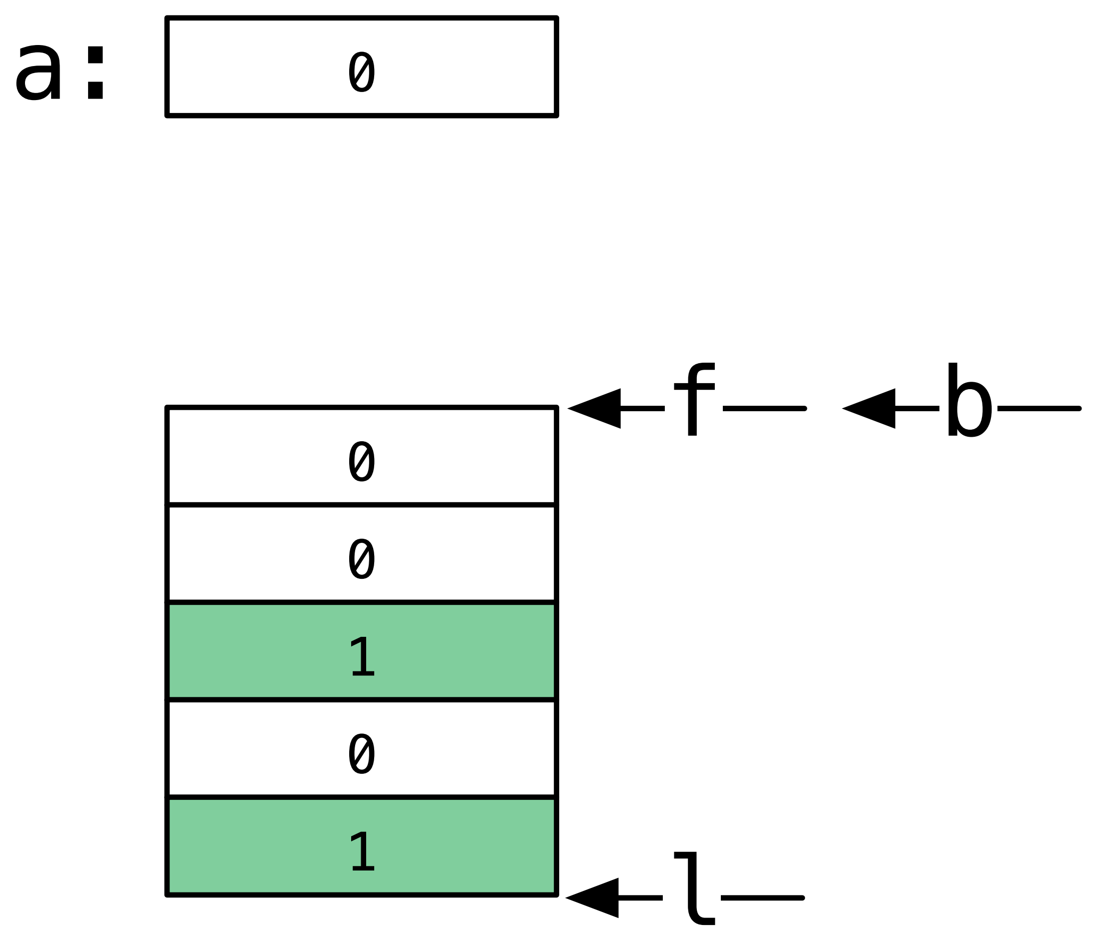
Remove

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



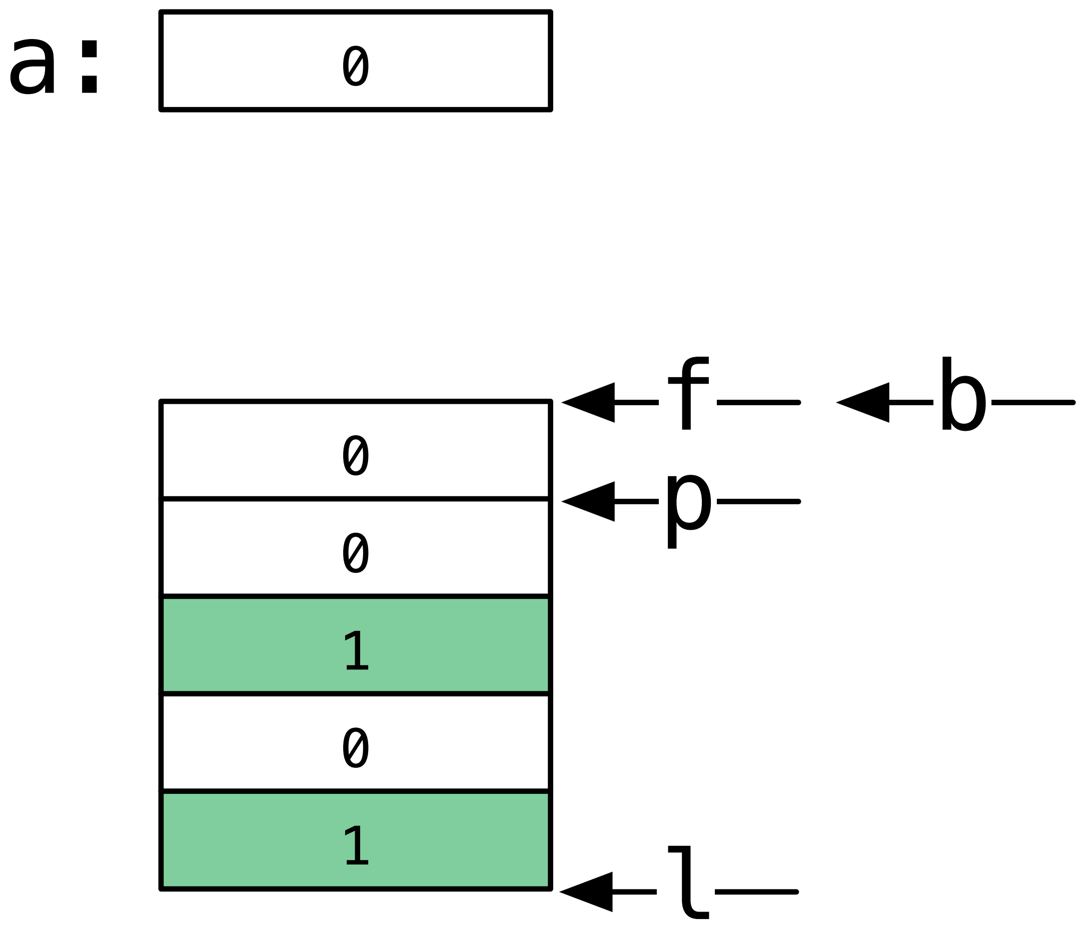
Remove

```
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;
```

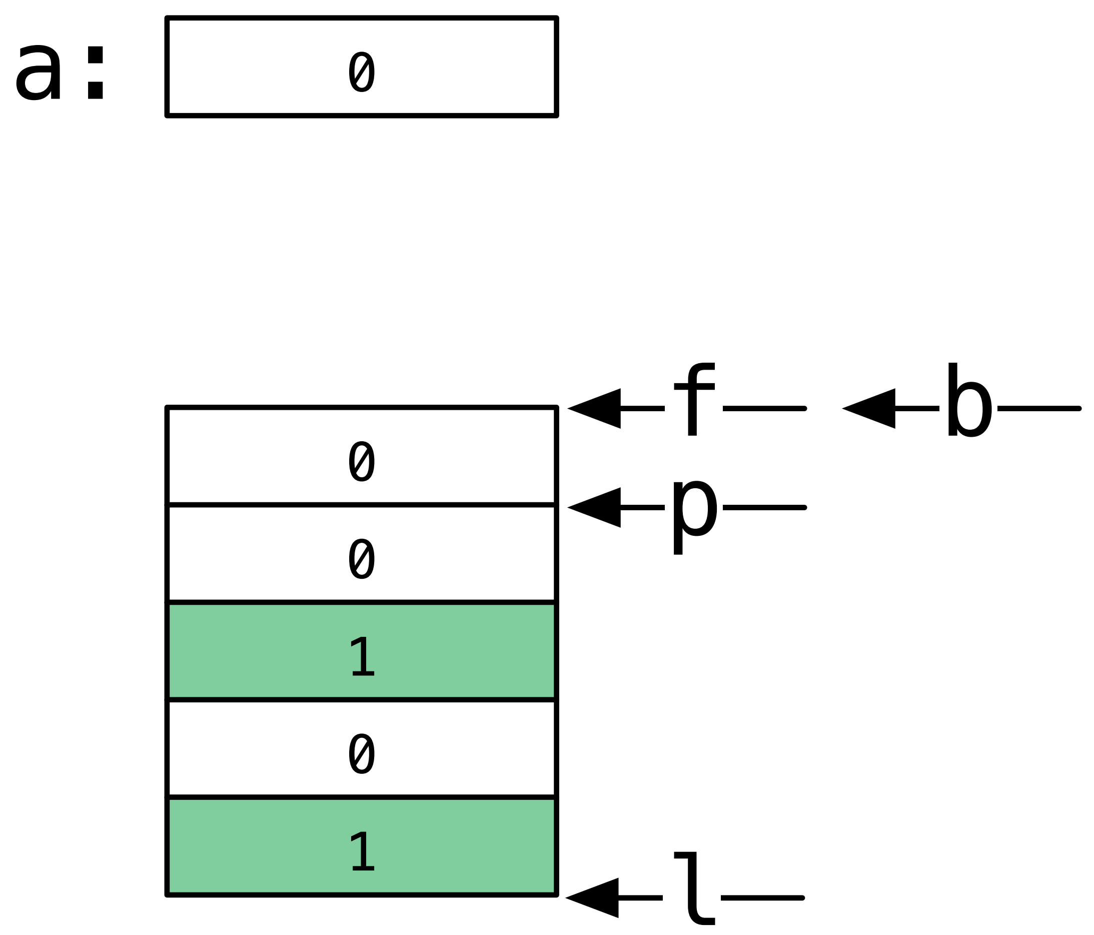


Remove

```
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)};
```

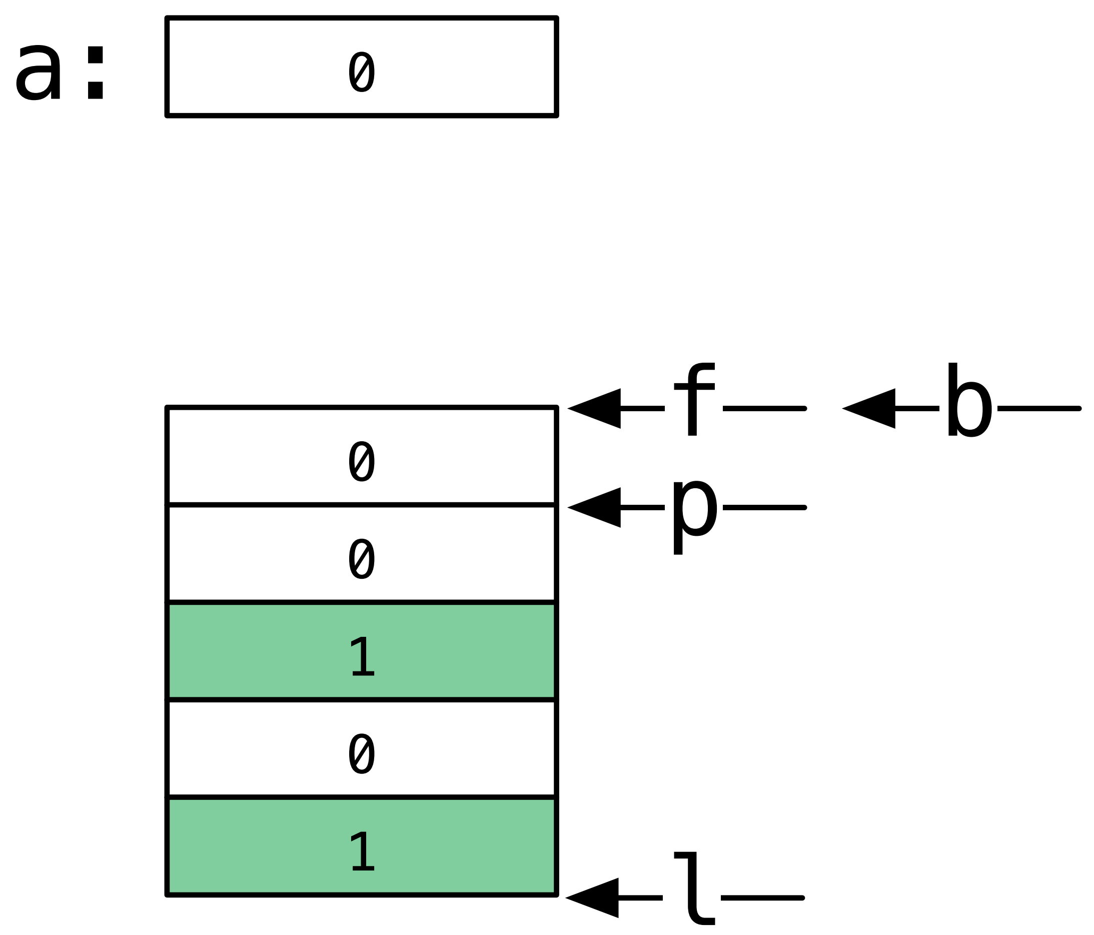


Remove



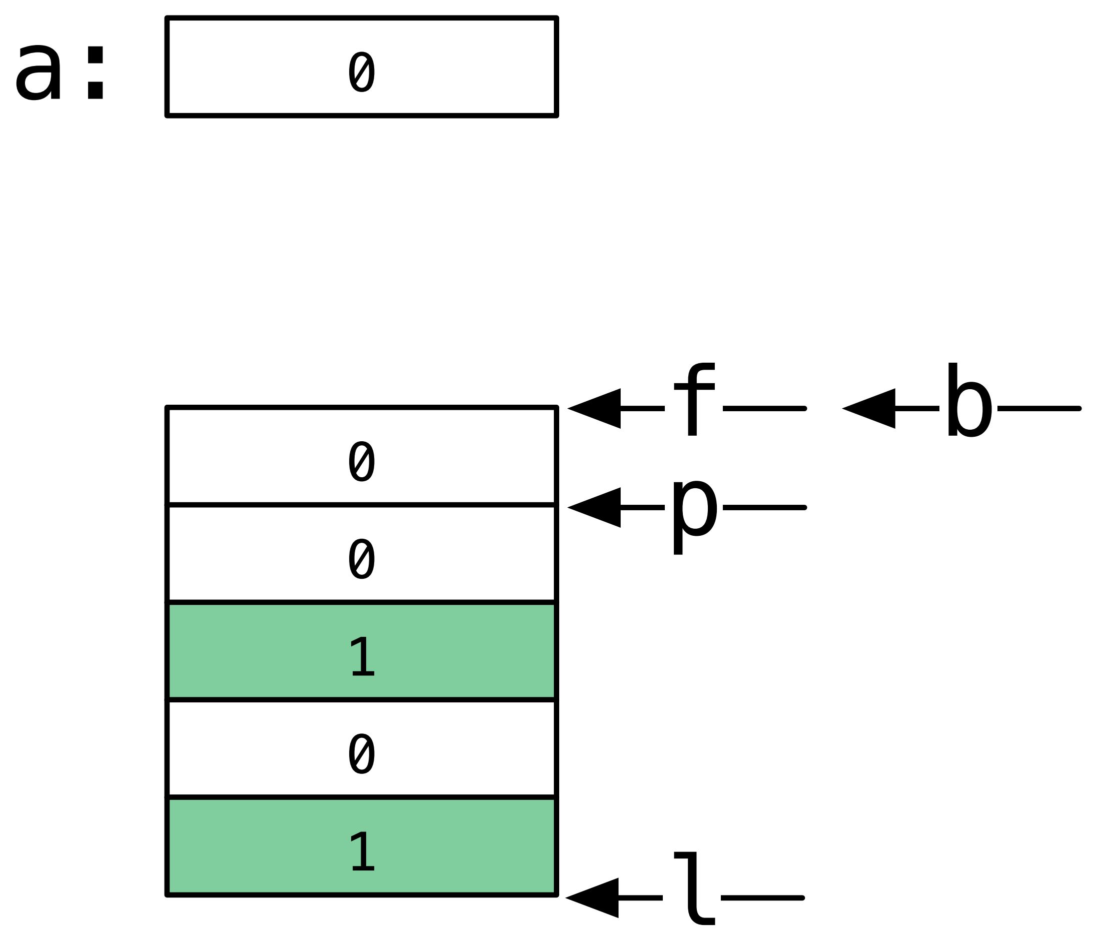
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
```

Remove



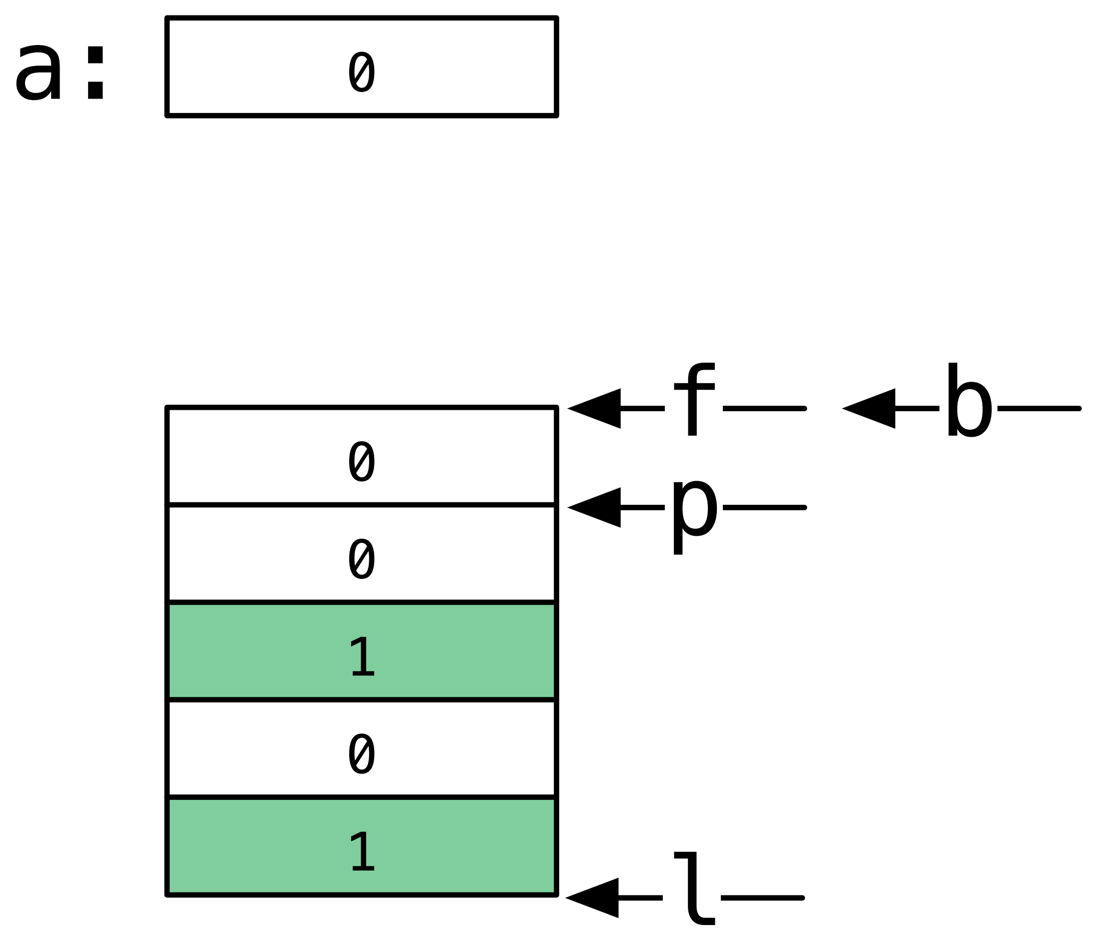
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
```

Remove



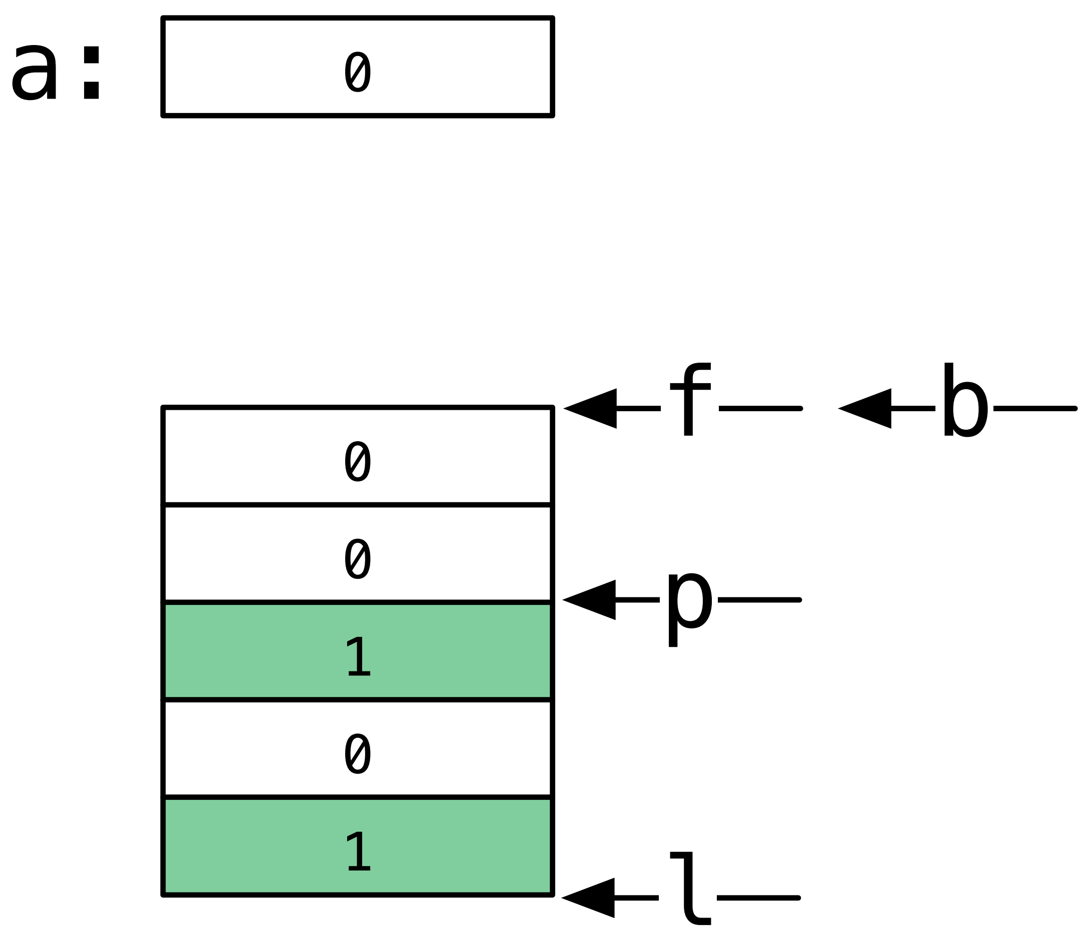
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
```

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;
        }
```

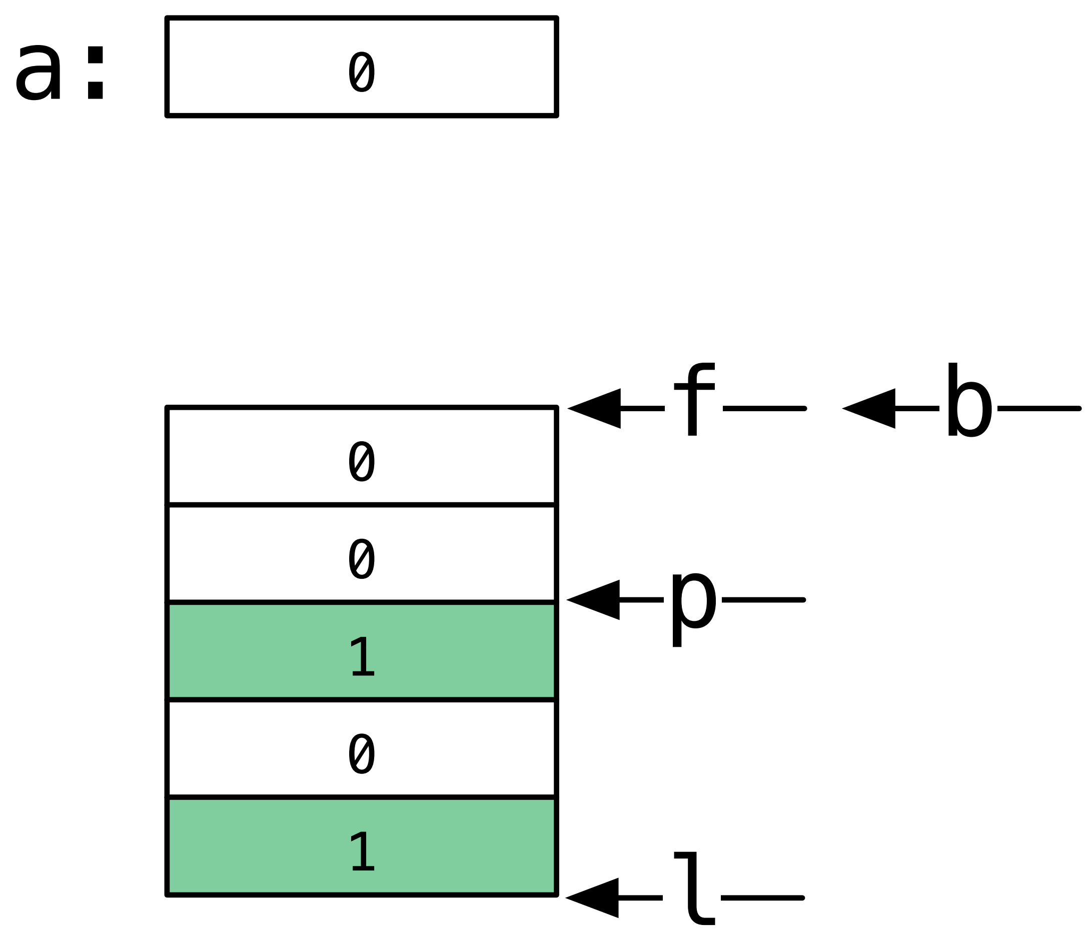
Remove



```
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;
```

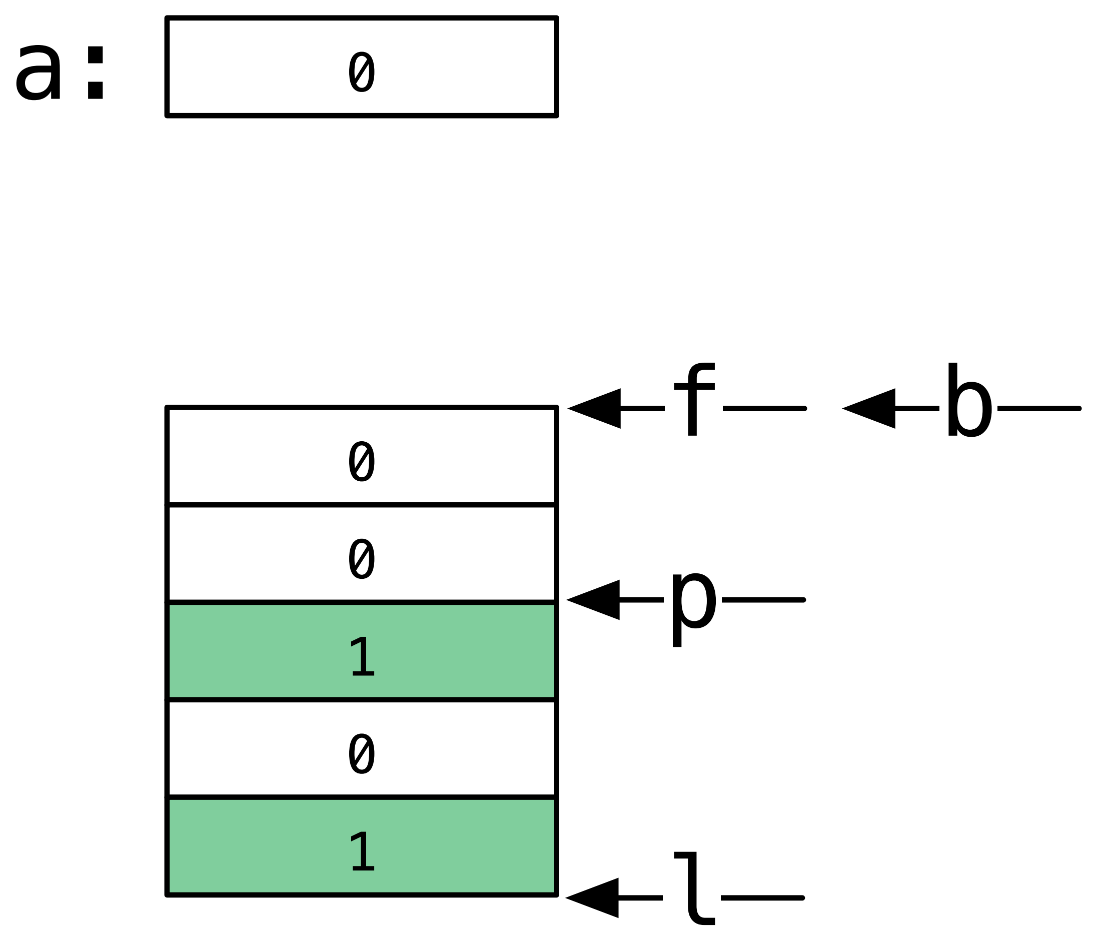
Adobe

Remove



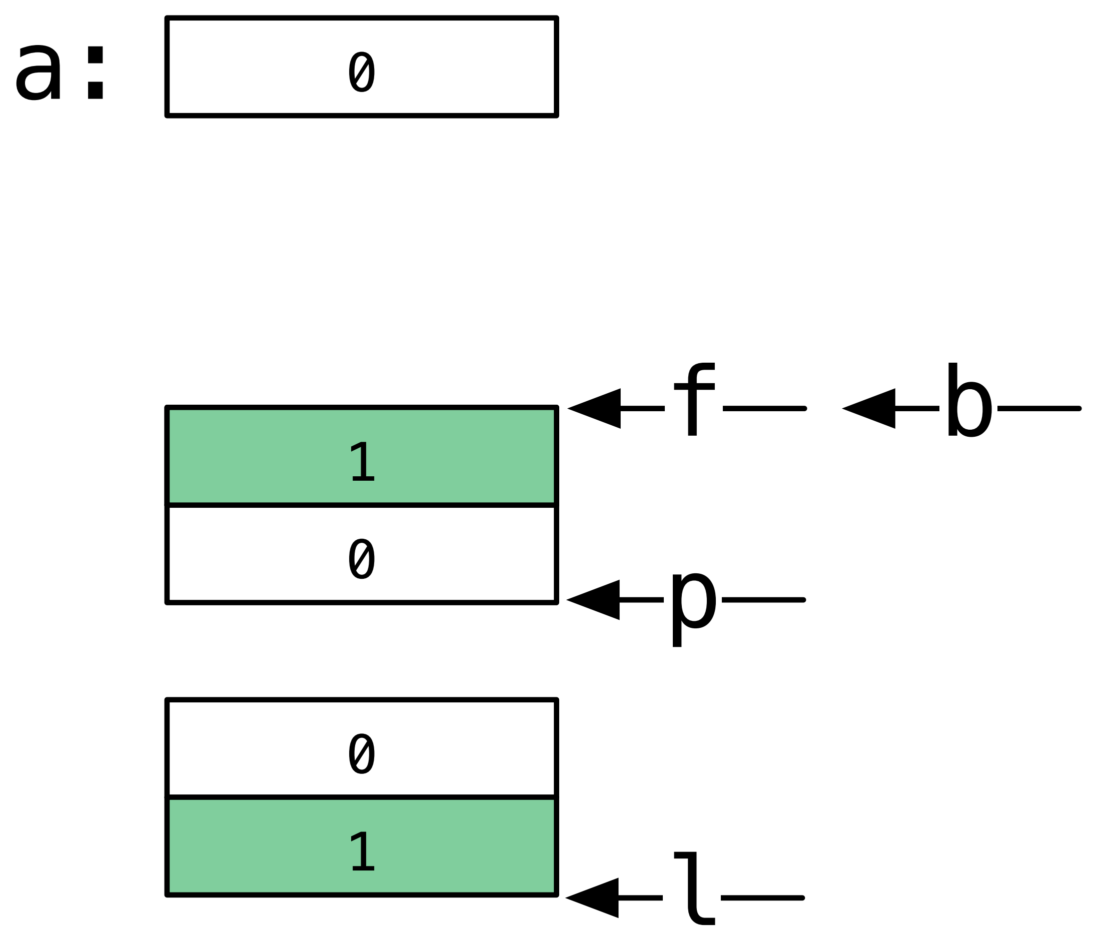
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



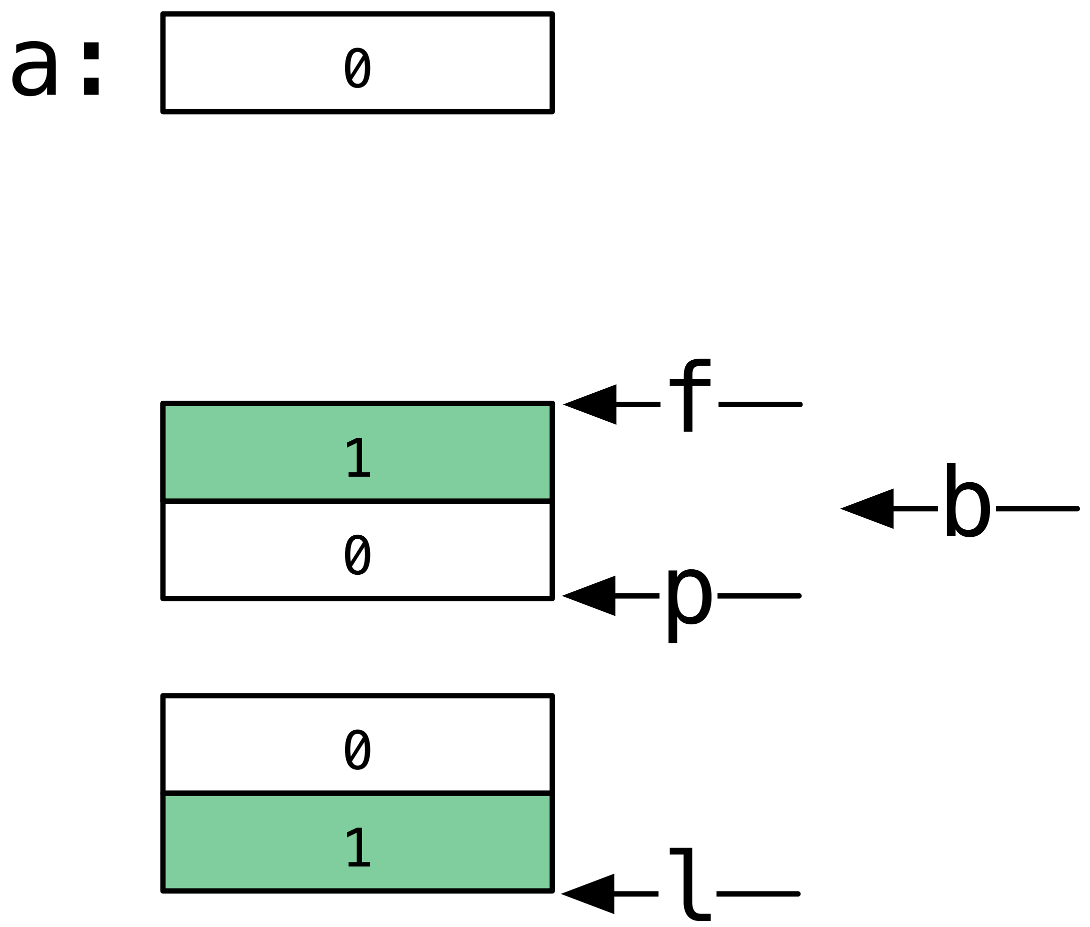
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



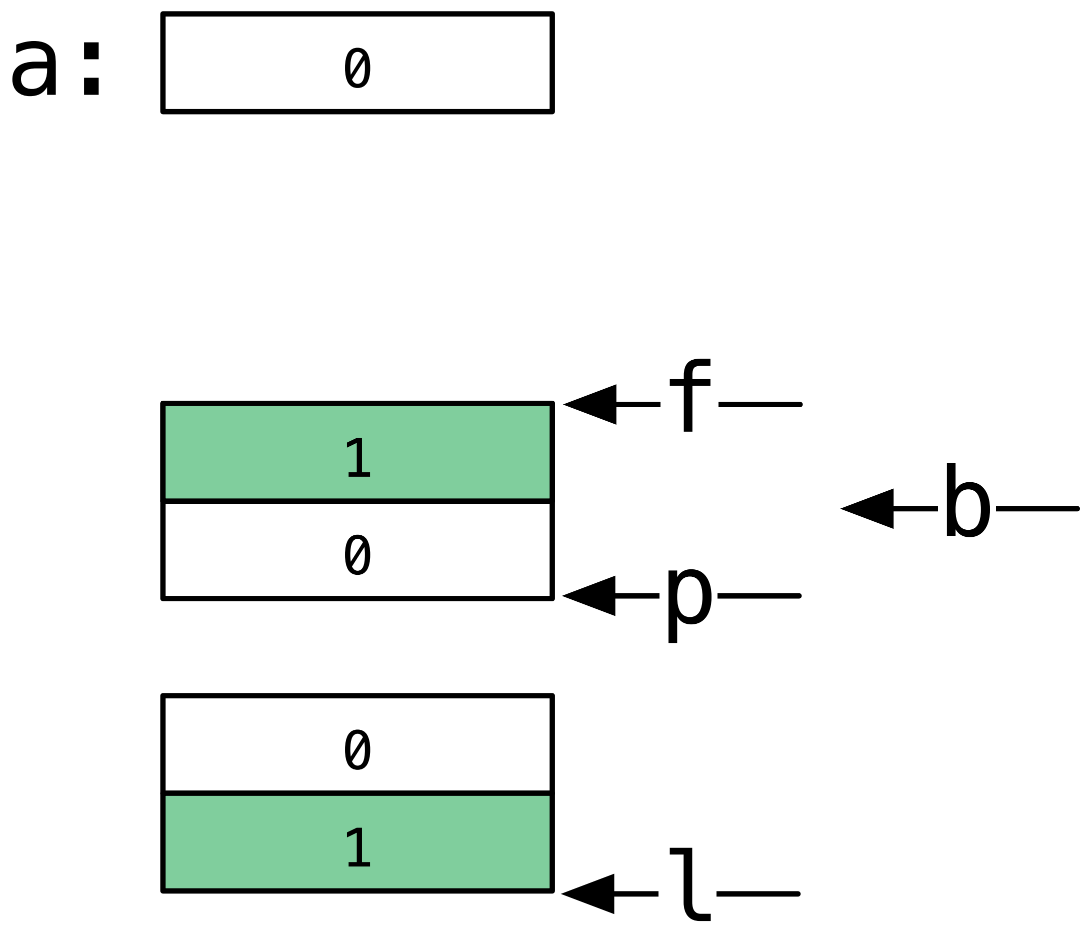
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



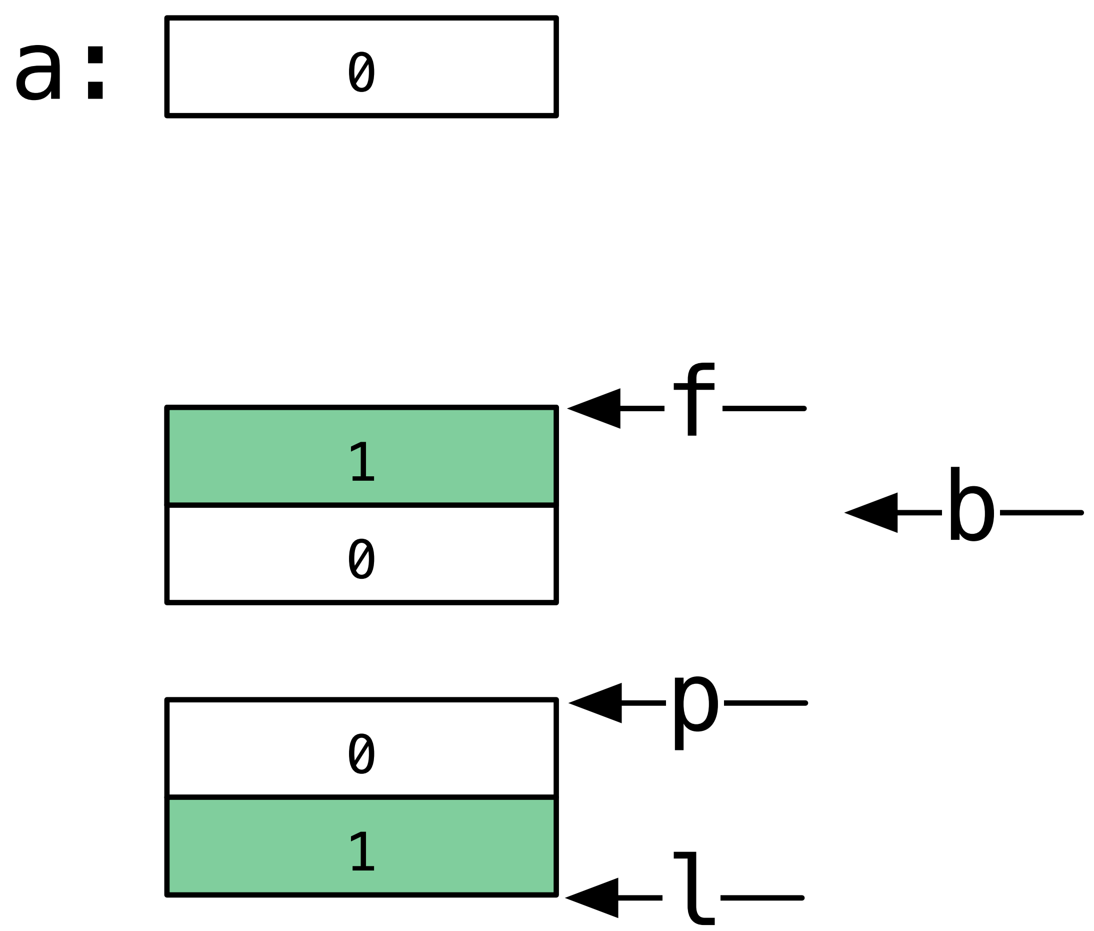
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



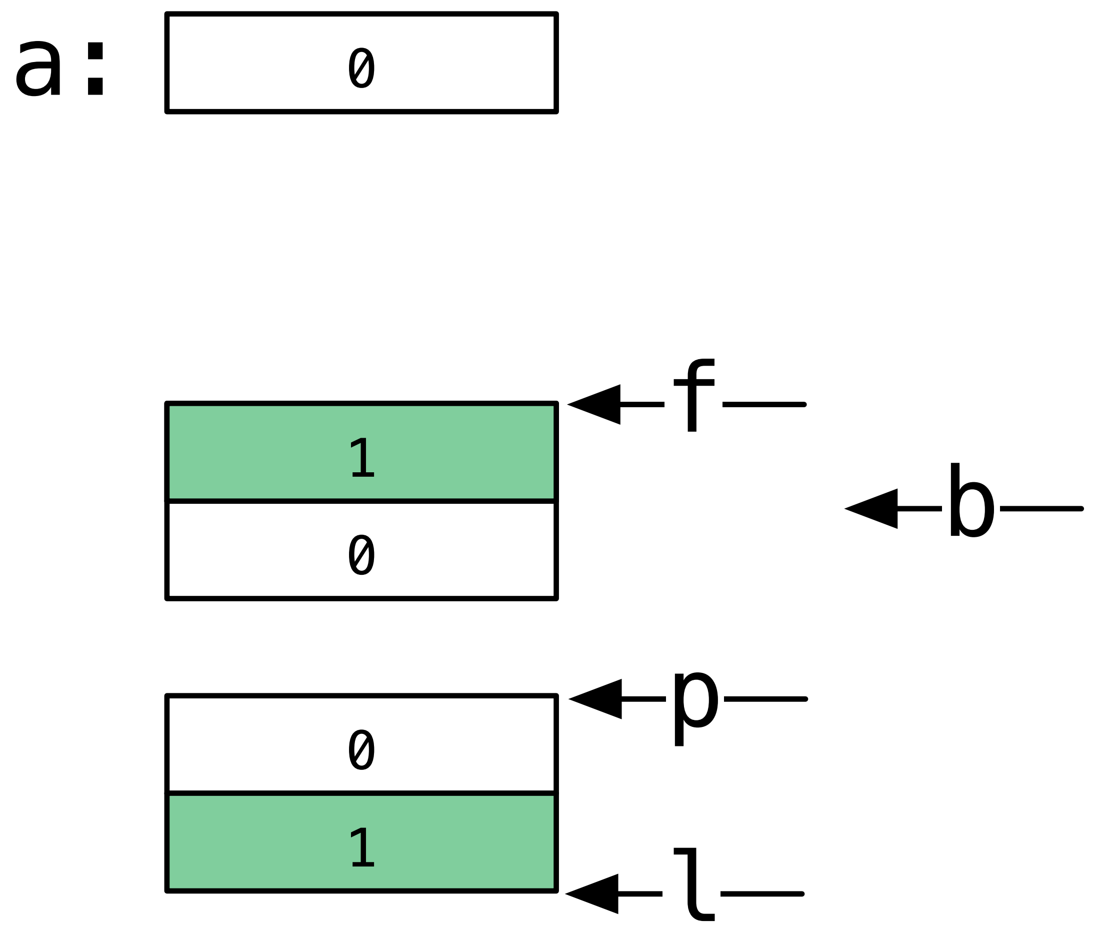
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



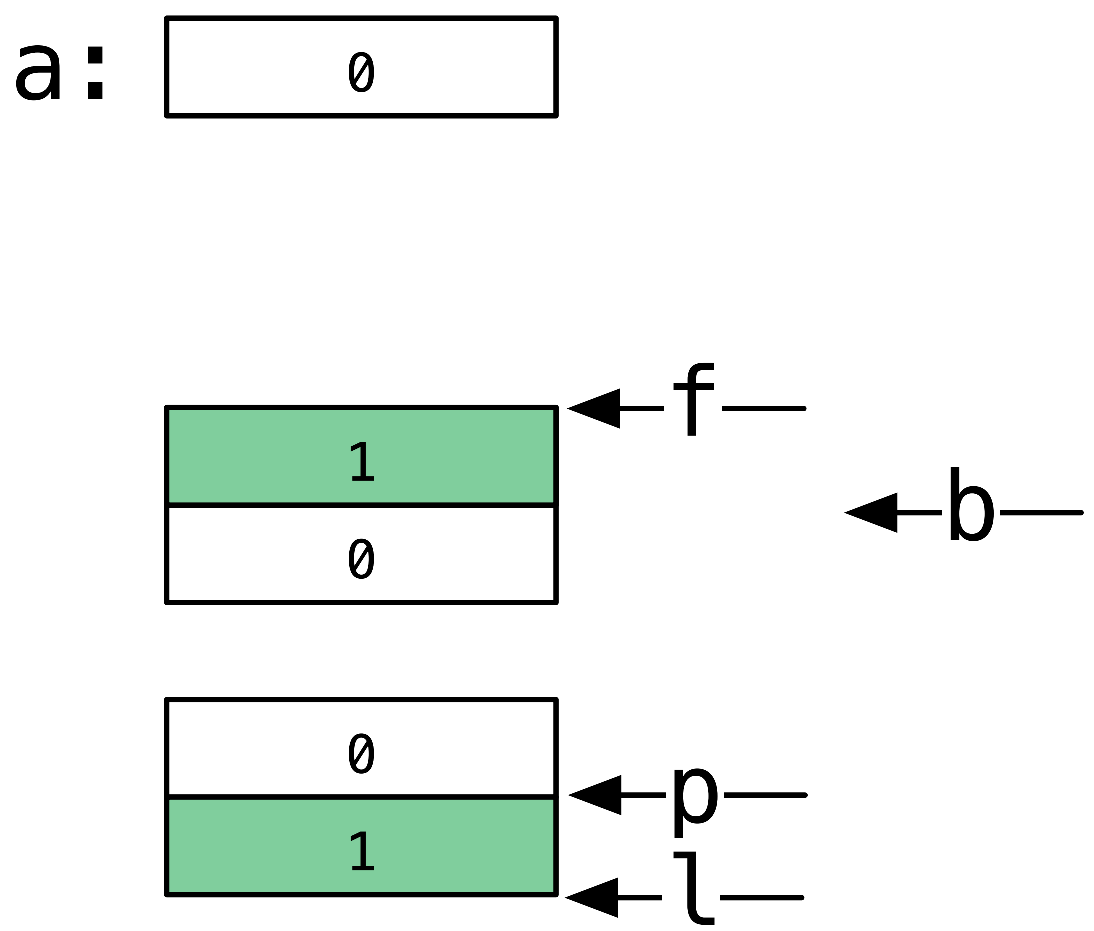
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



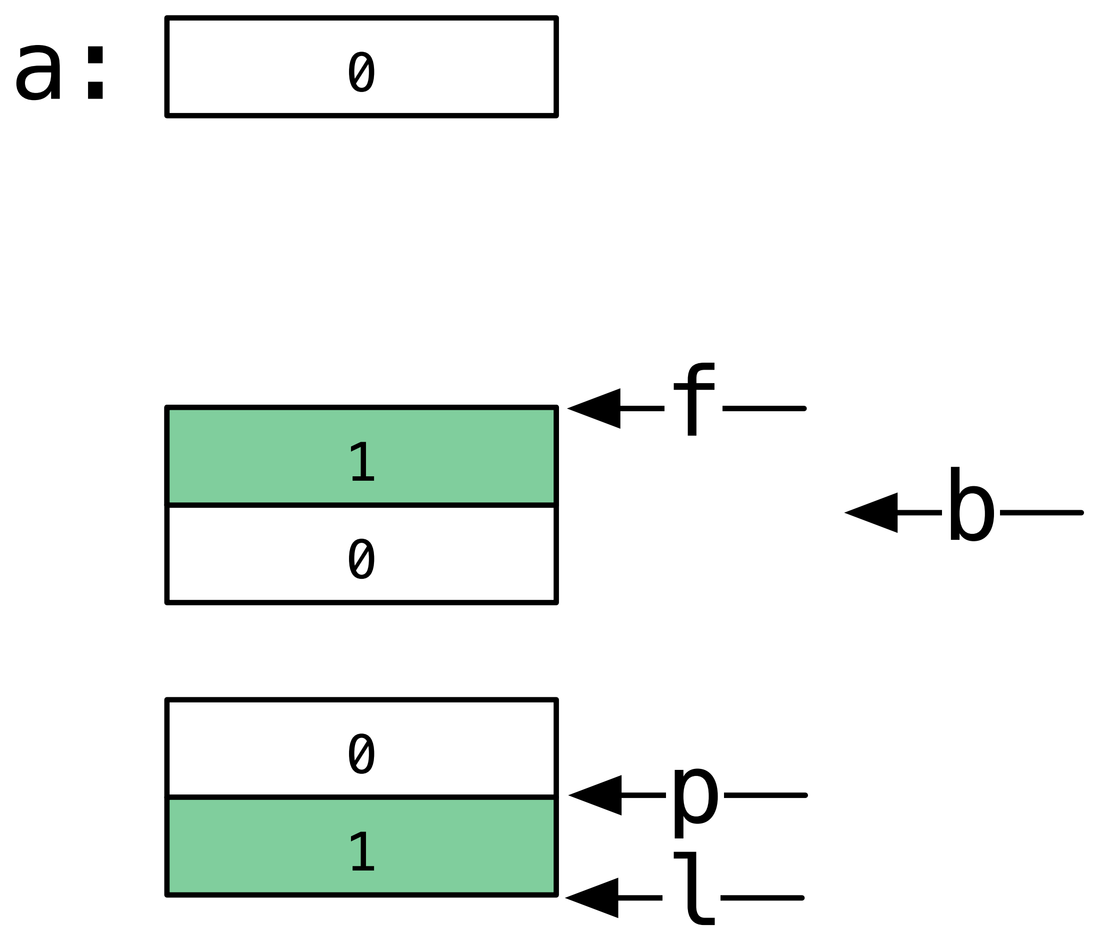
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



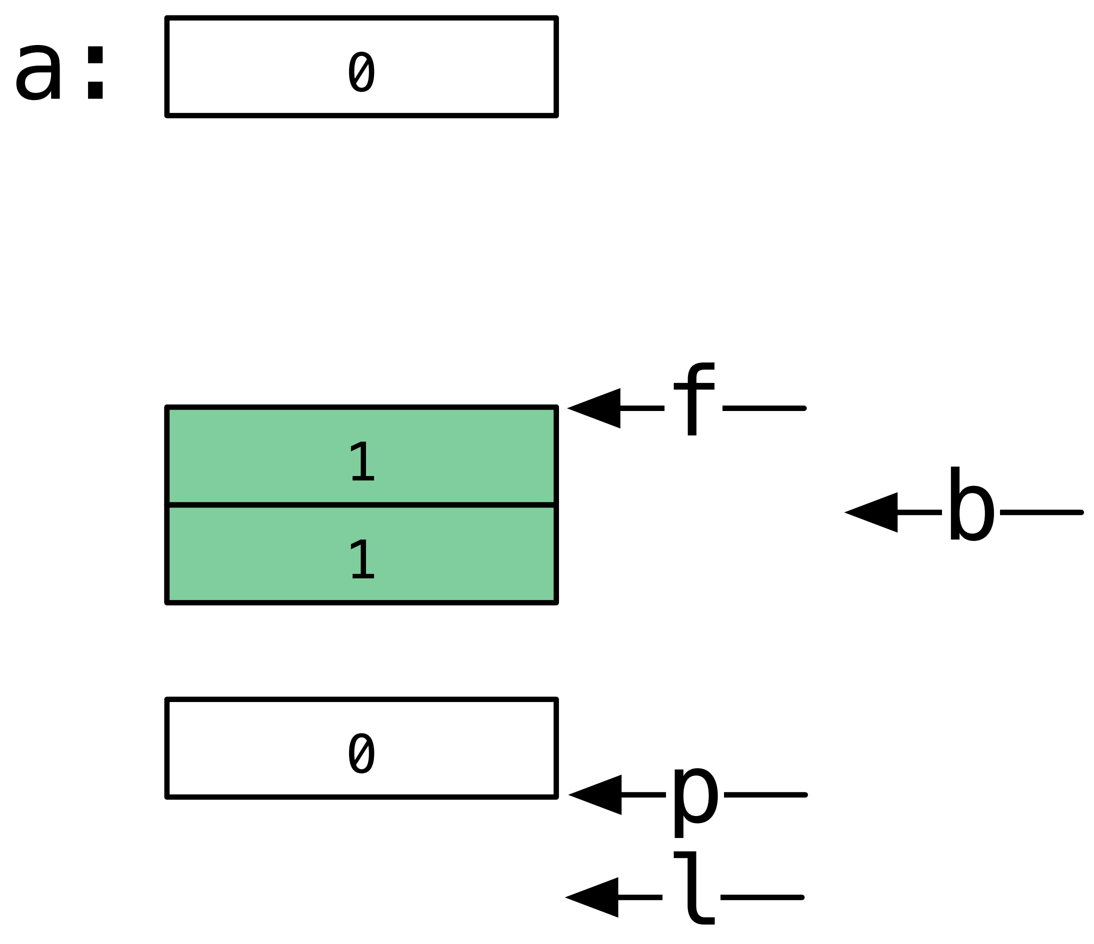
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



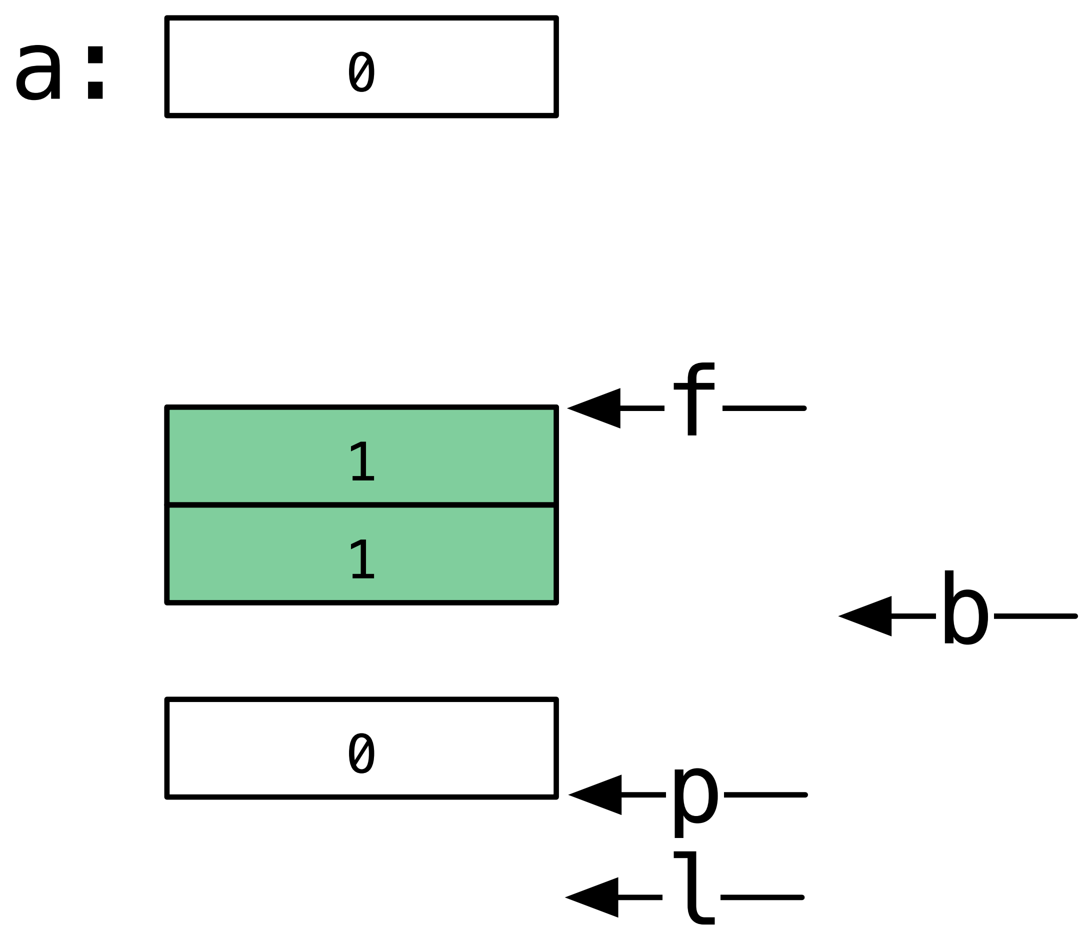
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



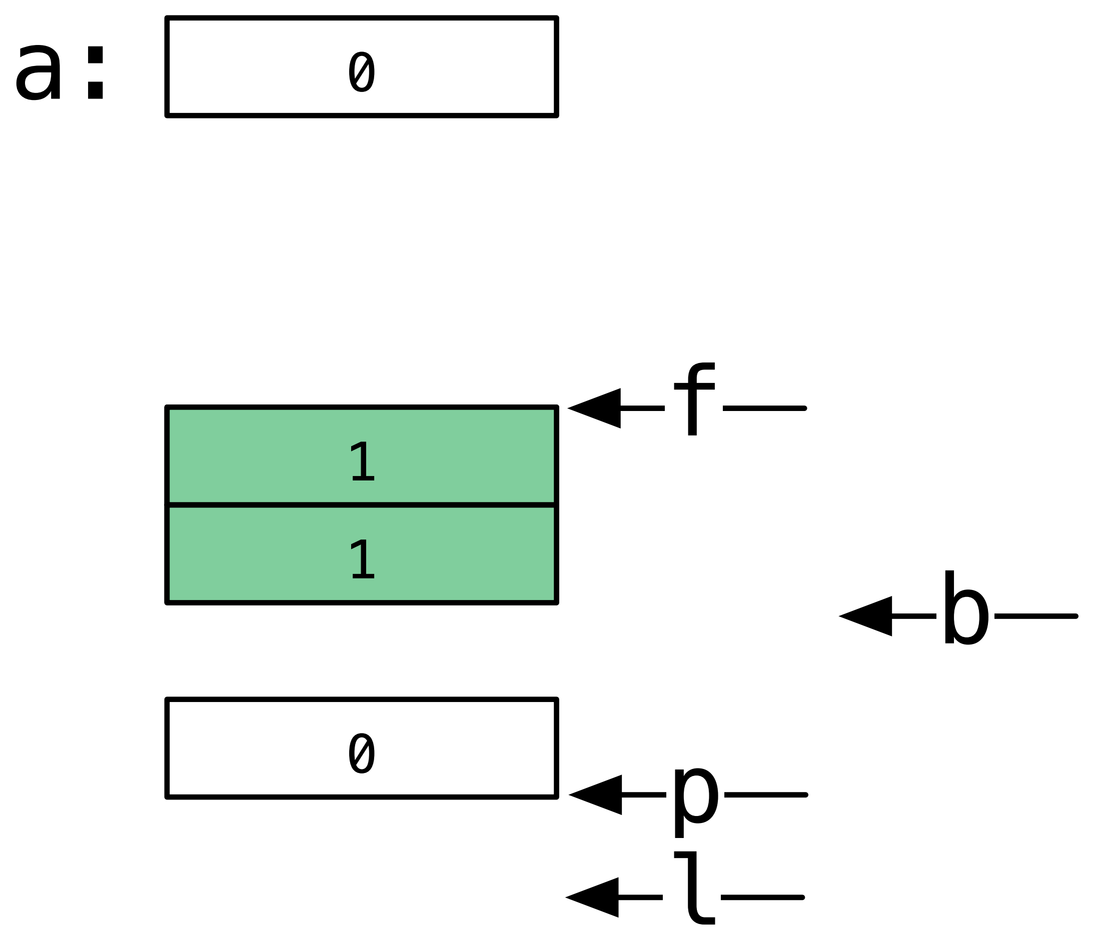
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



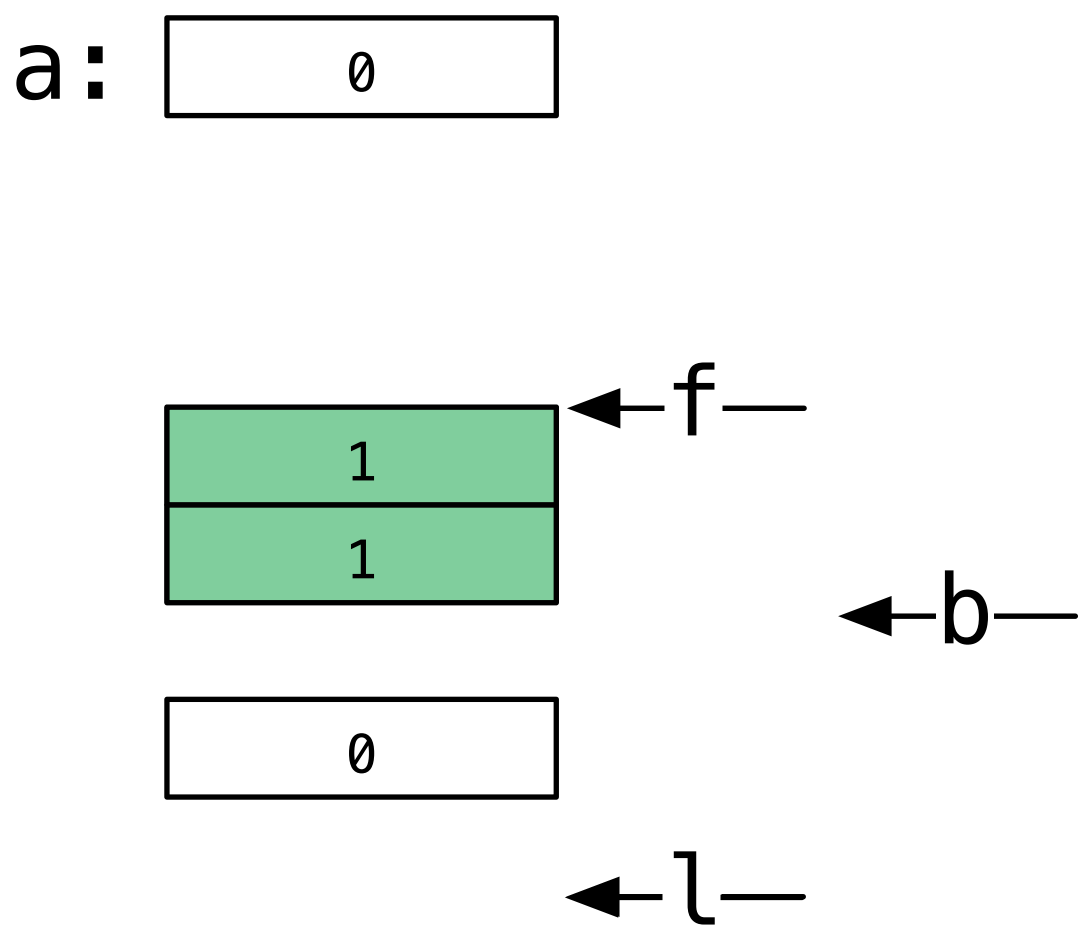
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



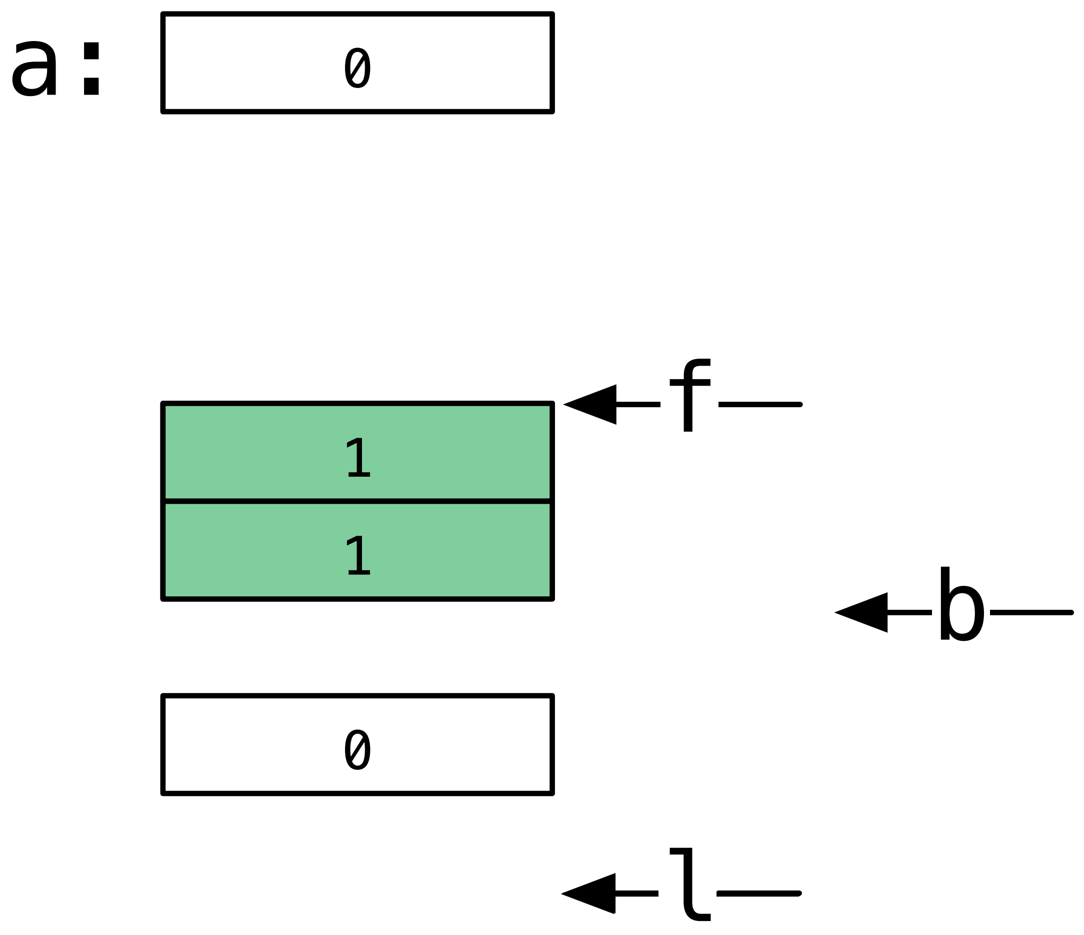
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



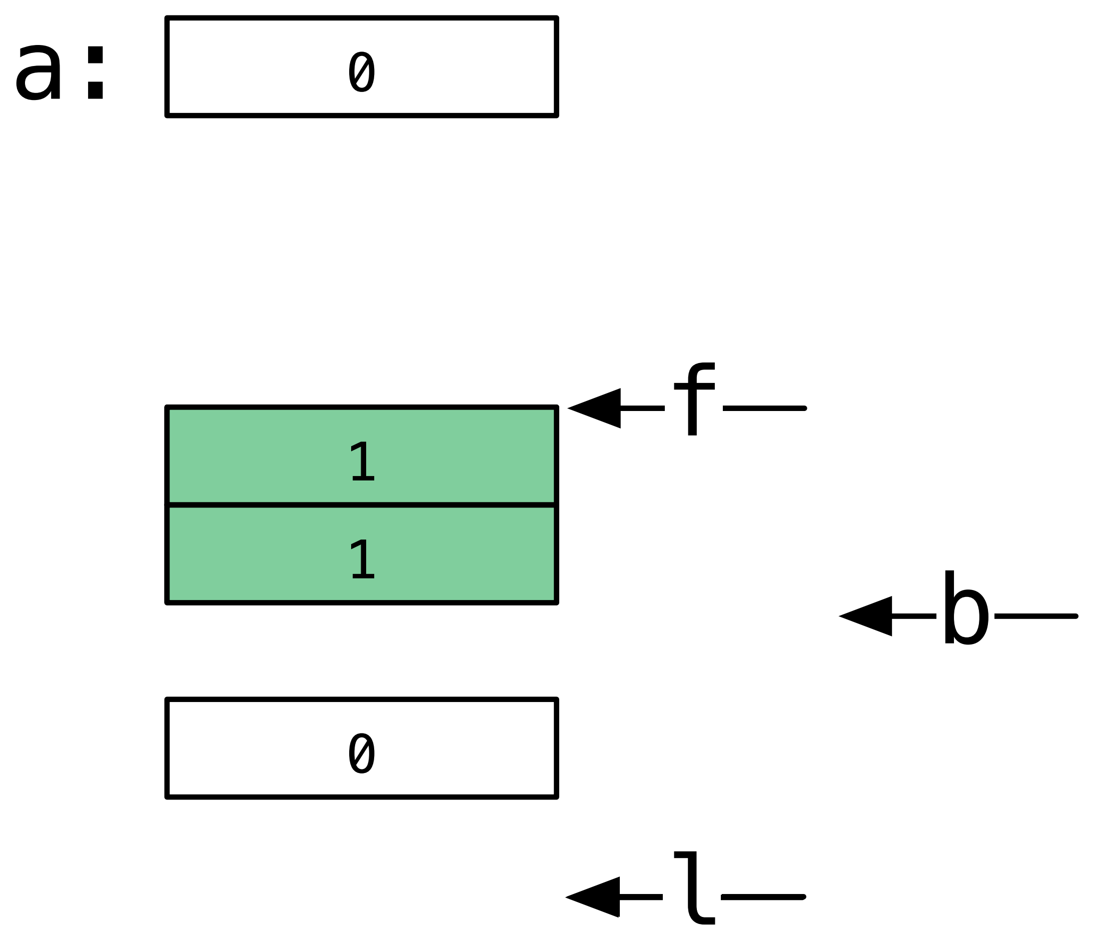
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

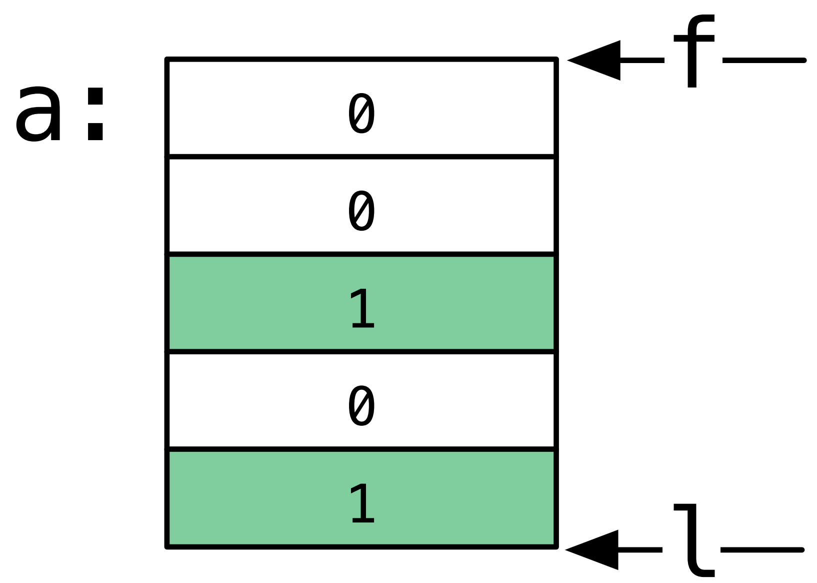
```
/**
    Removes values equal to `a` in the range `[f, l)`.

    \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;
```

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]);
```

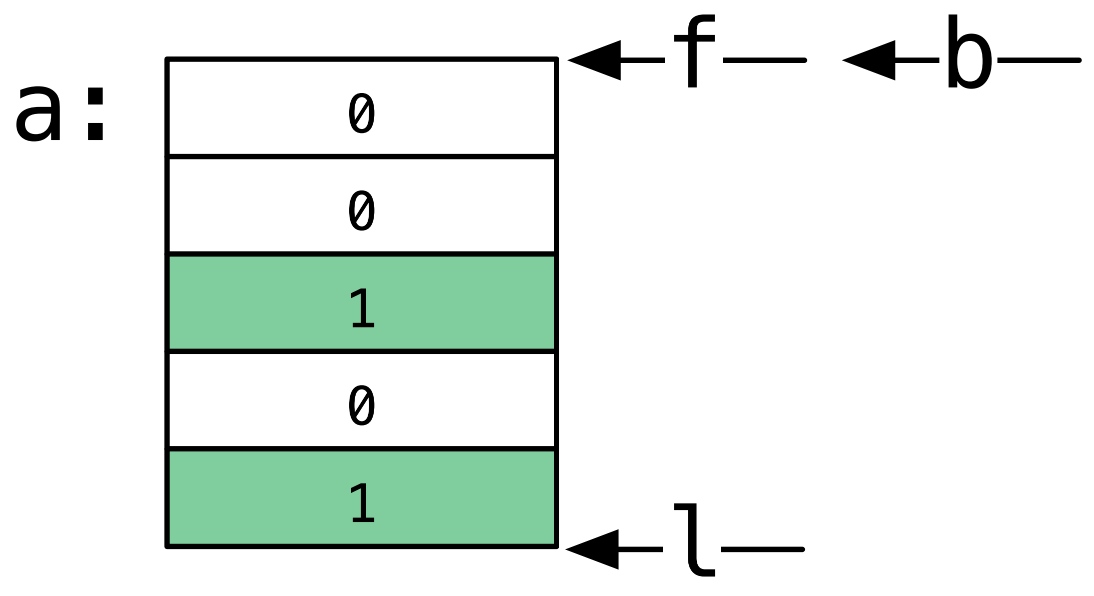
Remove

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



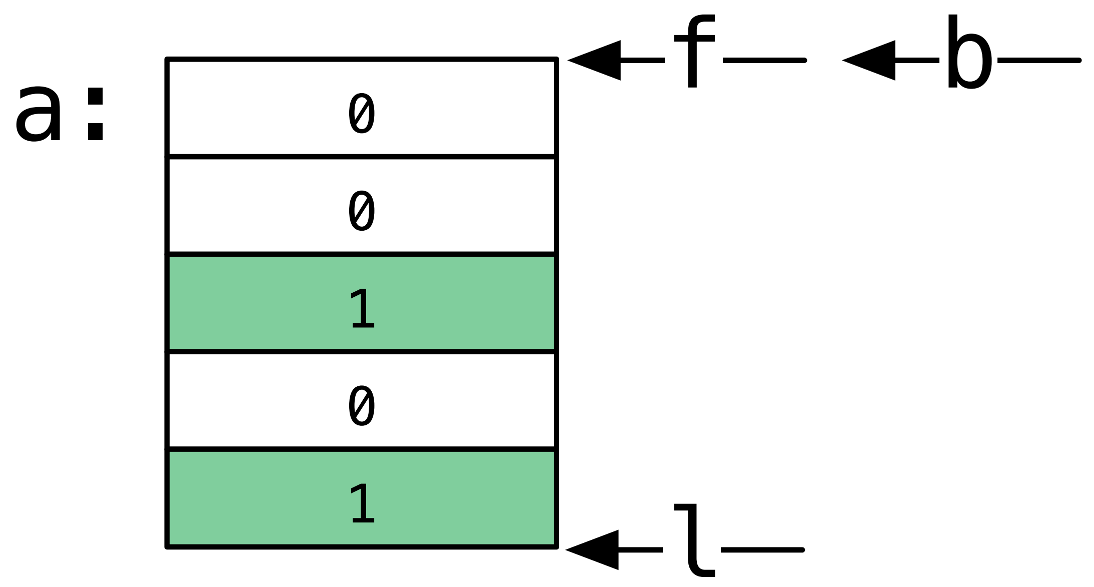
Remove

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



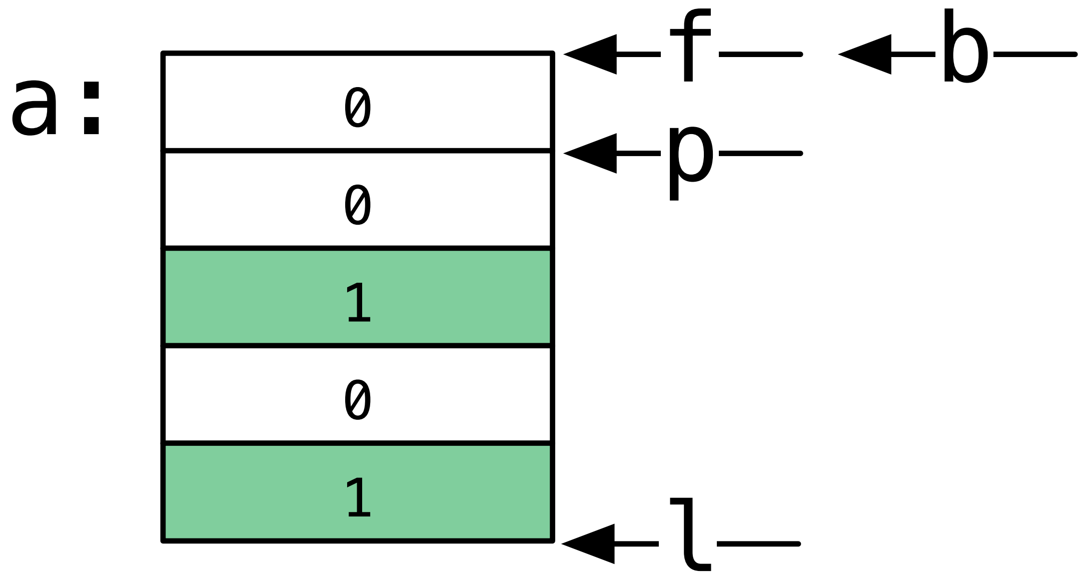
Remove

```
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;
```

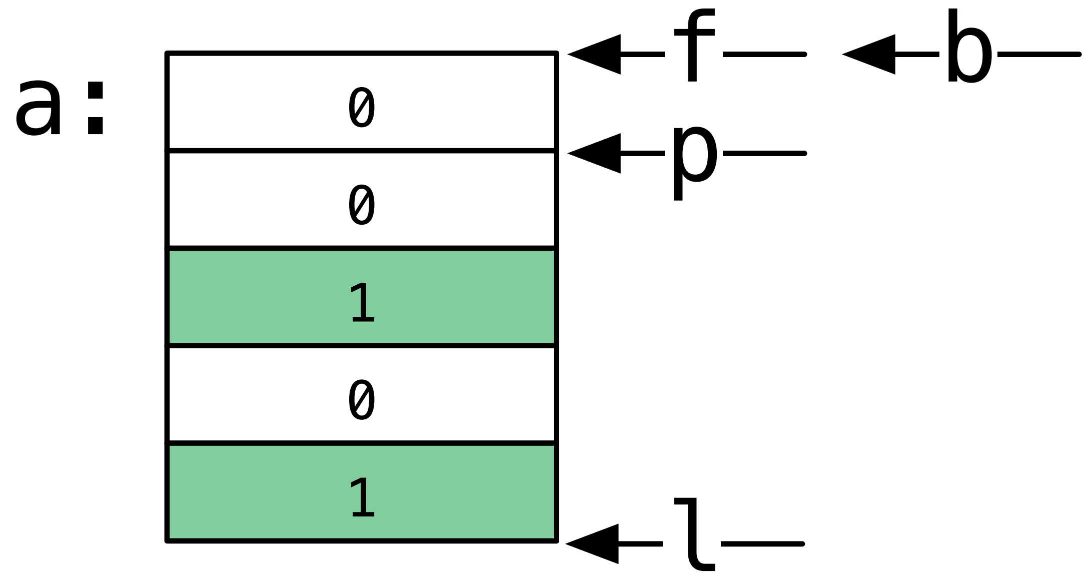


Remove

```
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)};
```

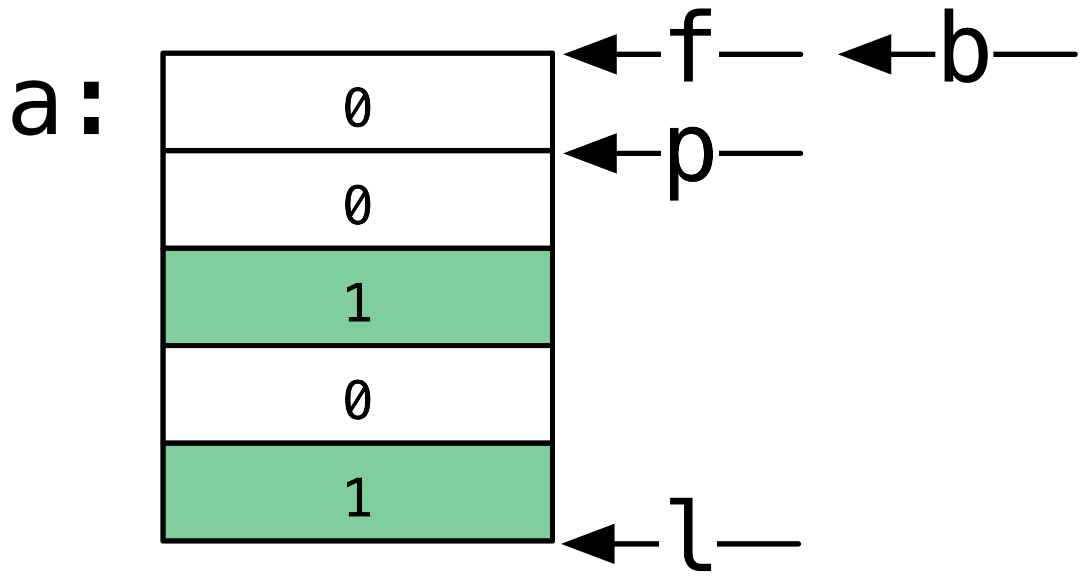


Remove



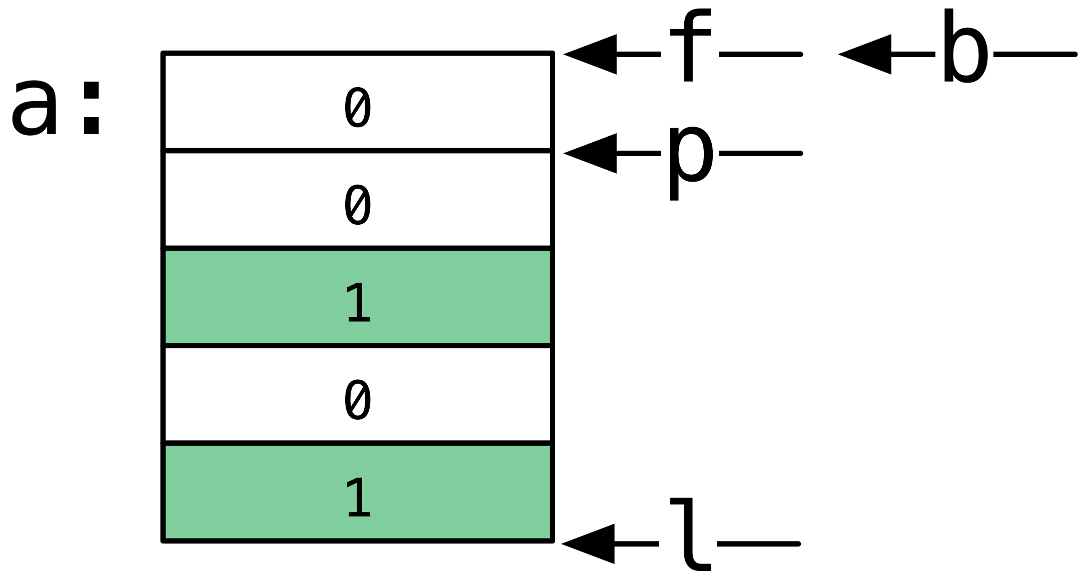
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
```

Remove



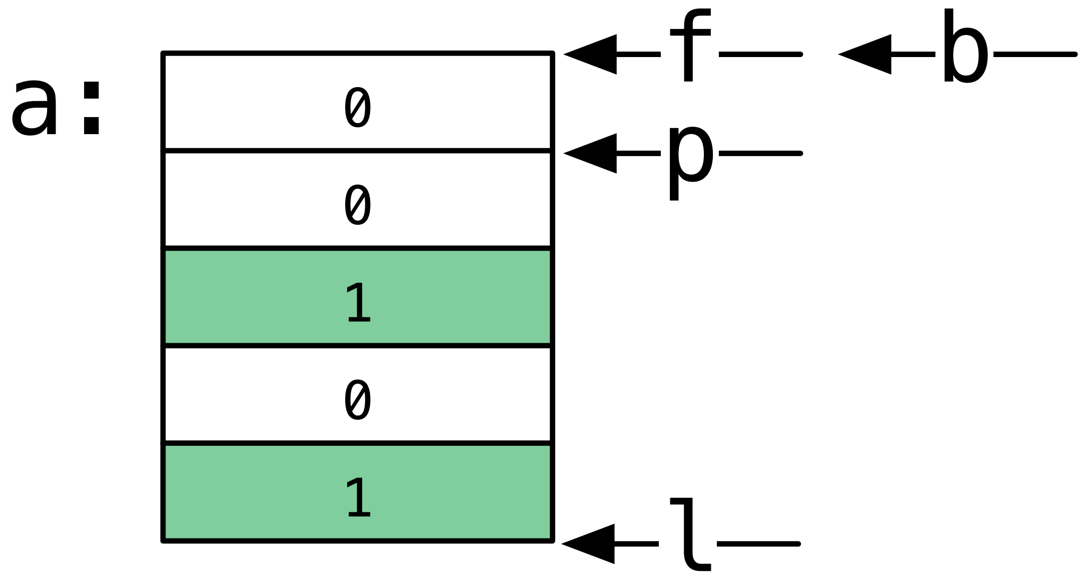
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
```

Remove



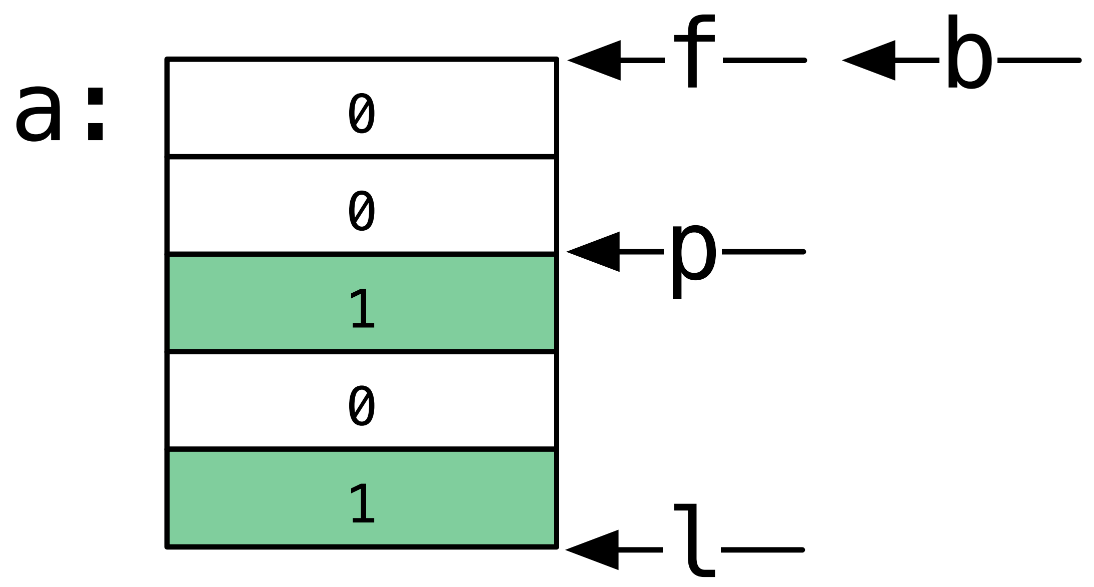
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
```

Remove



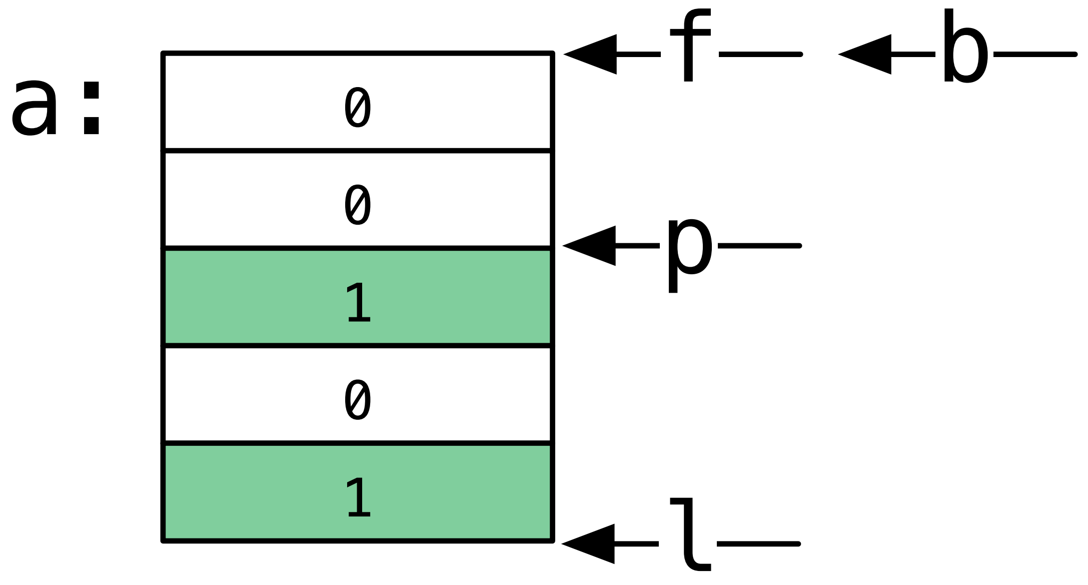
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



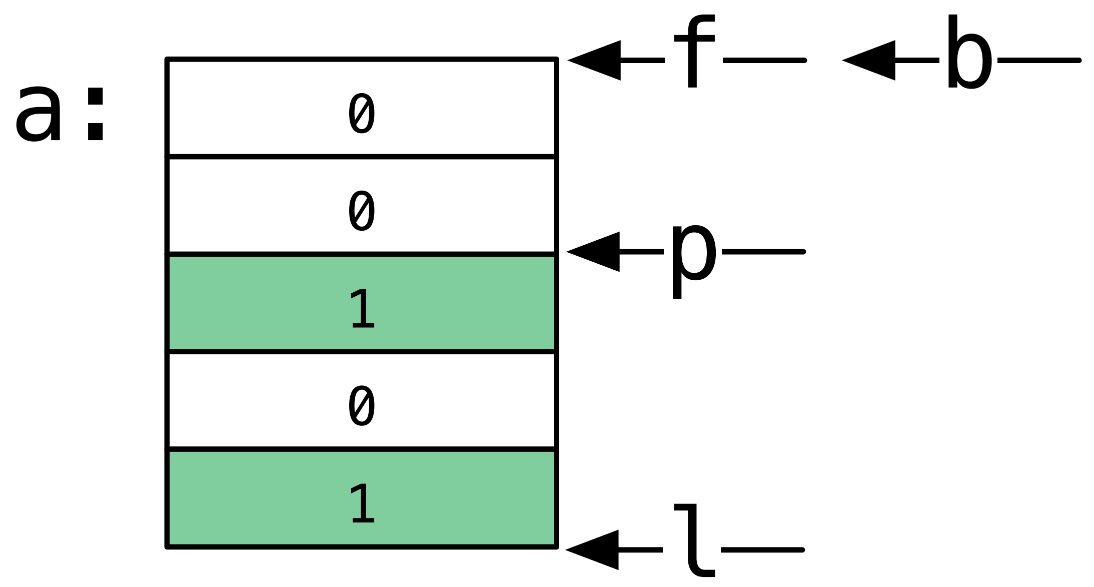
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



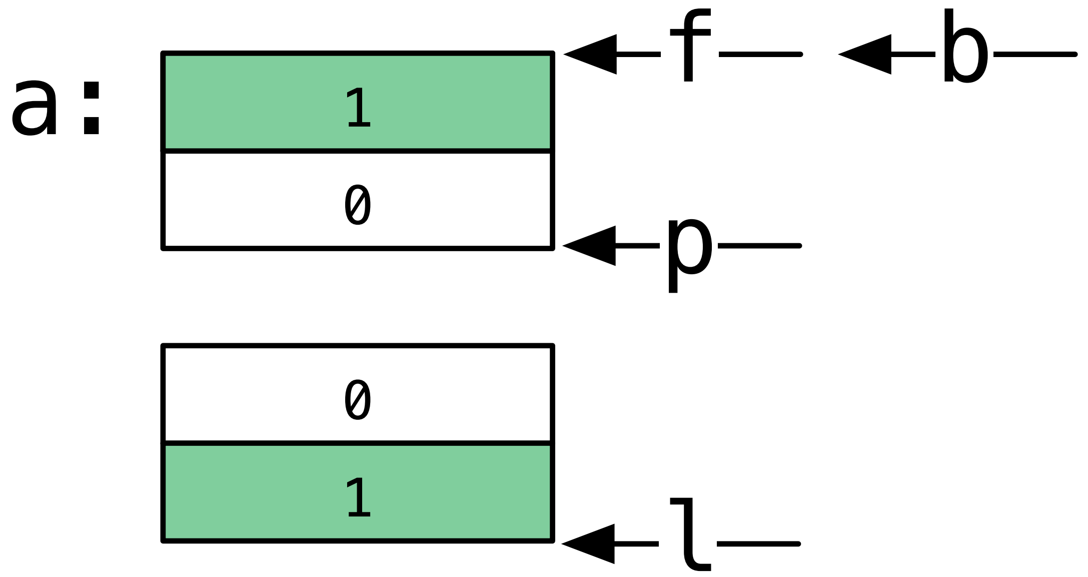
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



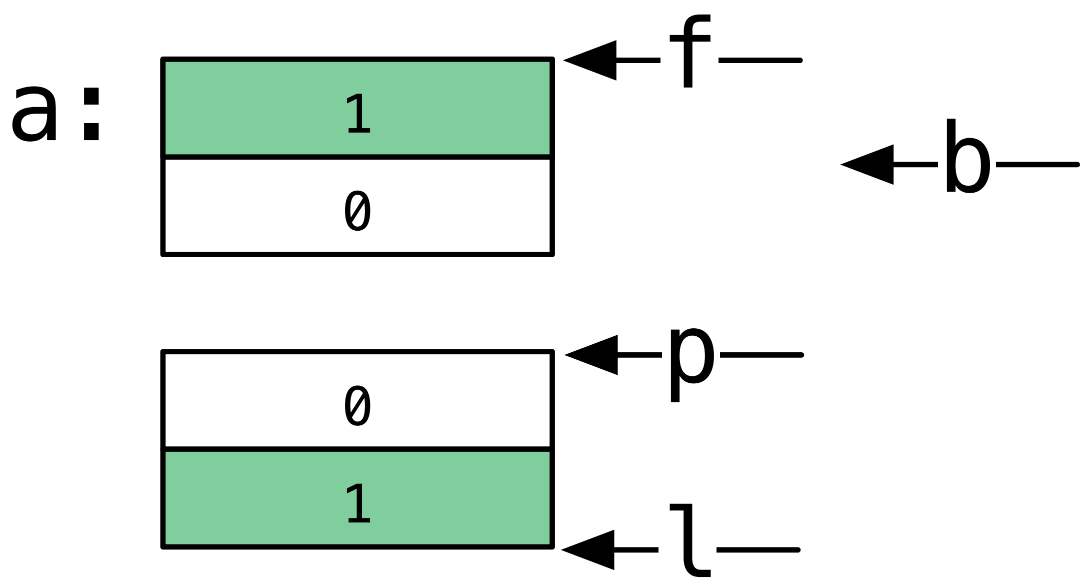
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



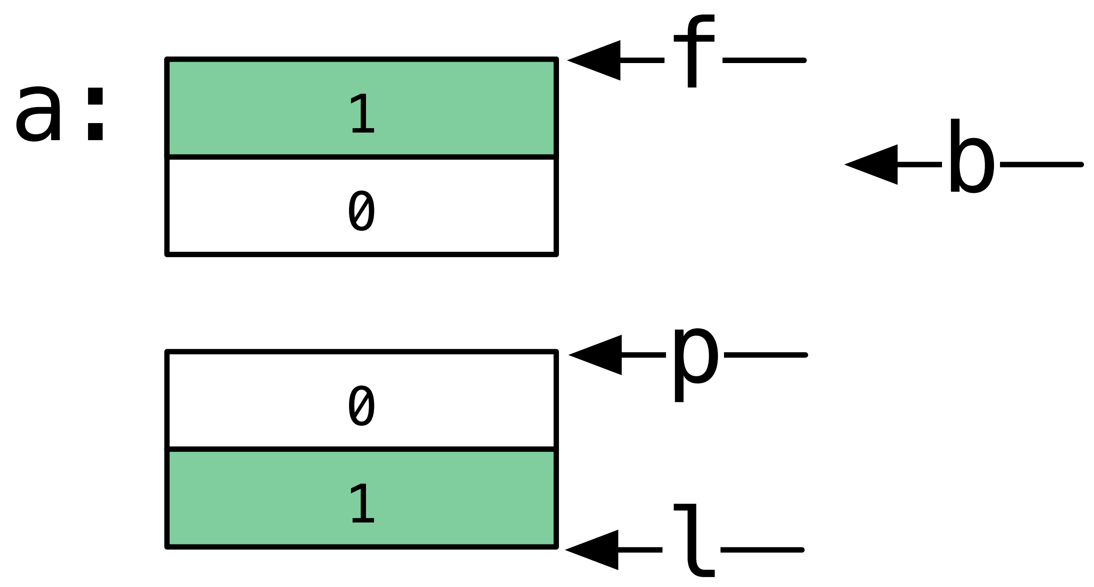
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



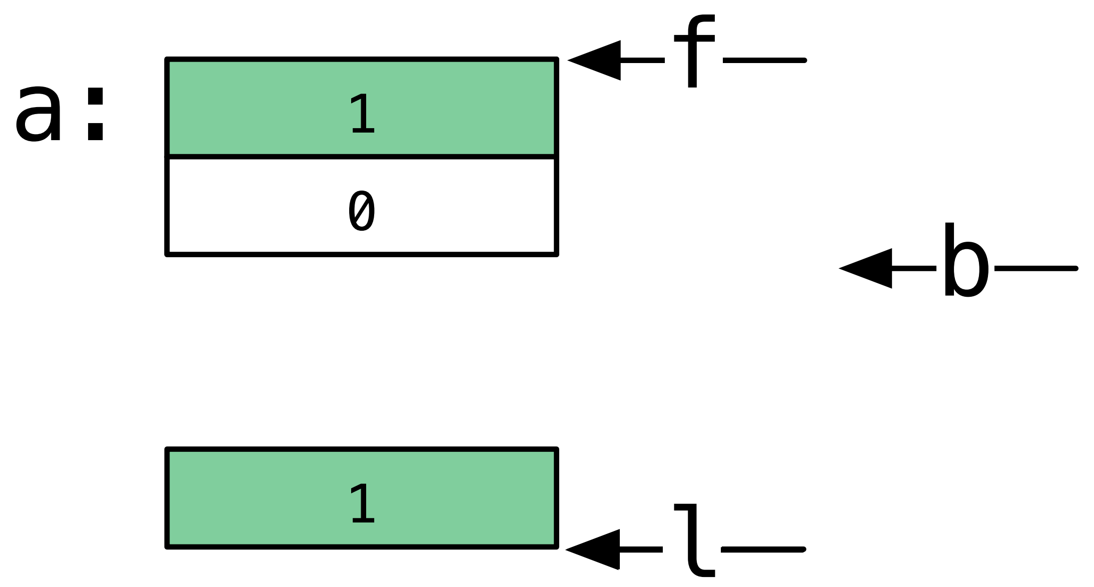
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

Remove



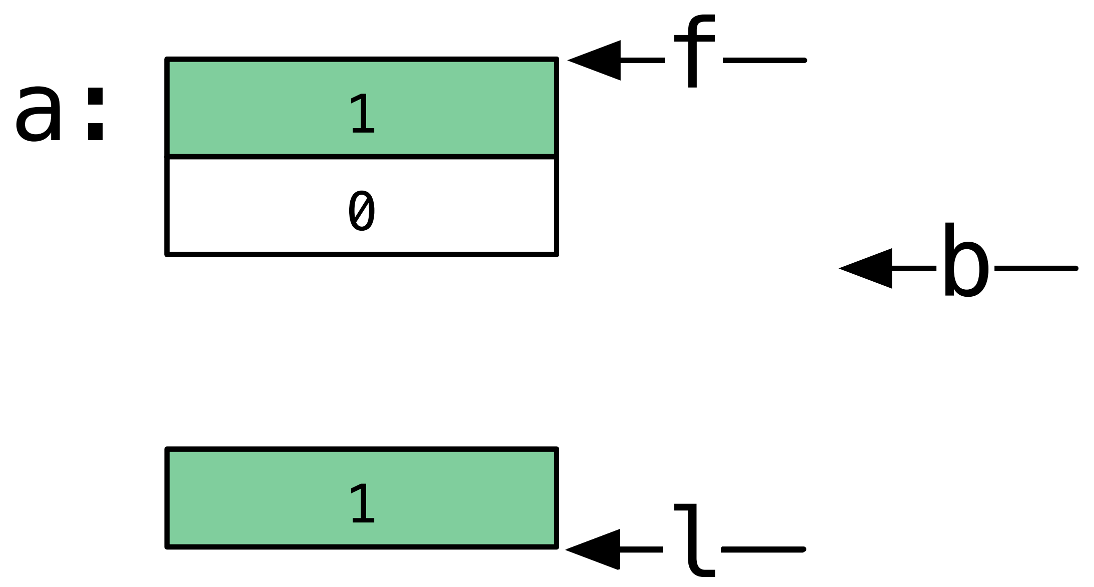
```
template <forward_iterator I, class T>
auto remove(I f, I l, const T& a) -> I {
    auto b{find(f, l, a)};
    if (b == l) return b;
    auto p{next(b)};
    // invariant: `[f, b)` contain all the
    //    values in `[f, p)` not equal to `a`
    // decreasing: `distance(p, l)`
    while (p != l)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

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

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

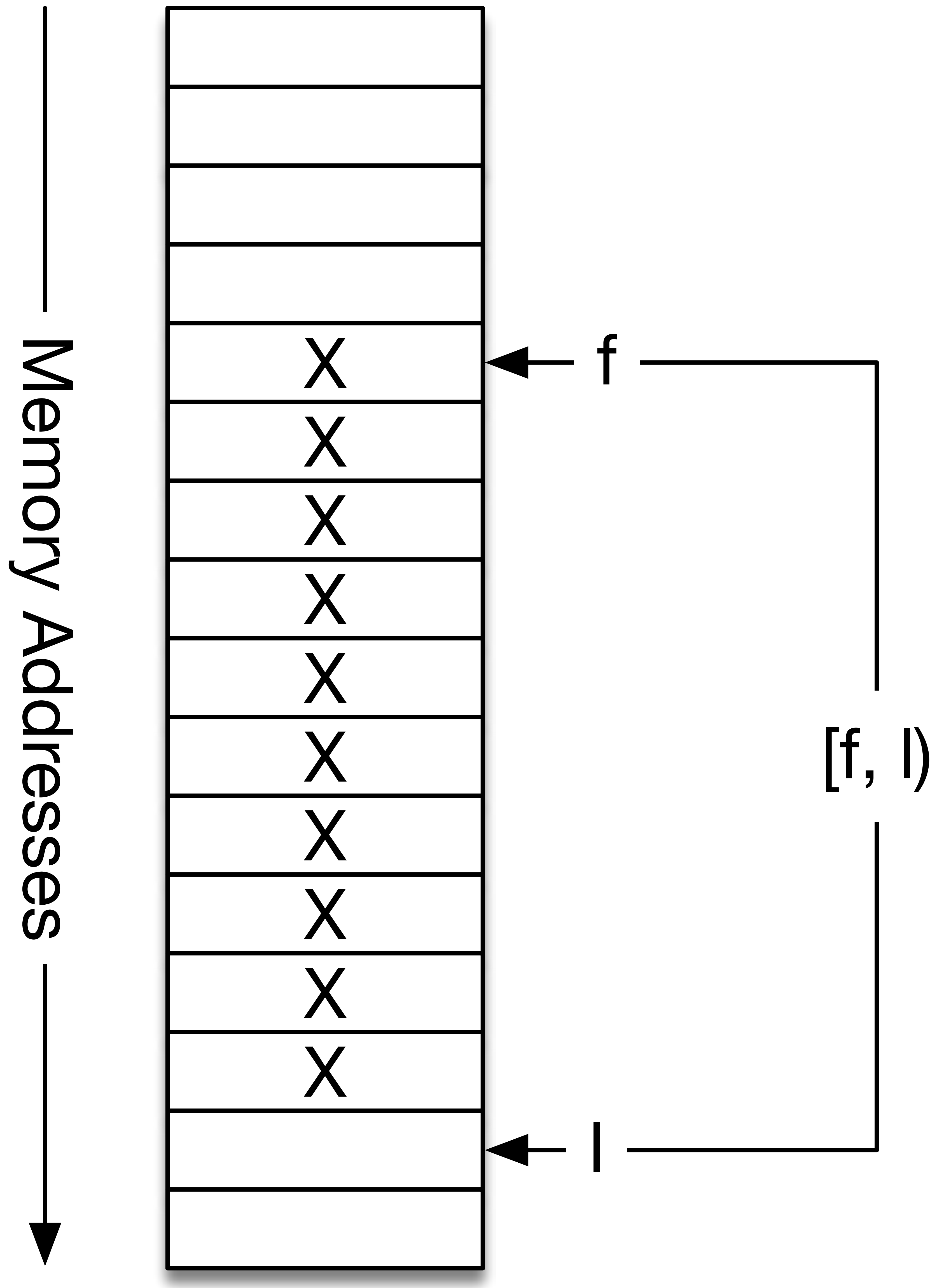
- All empty ranges are not equal

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

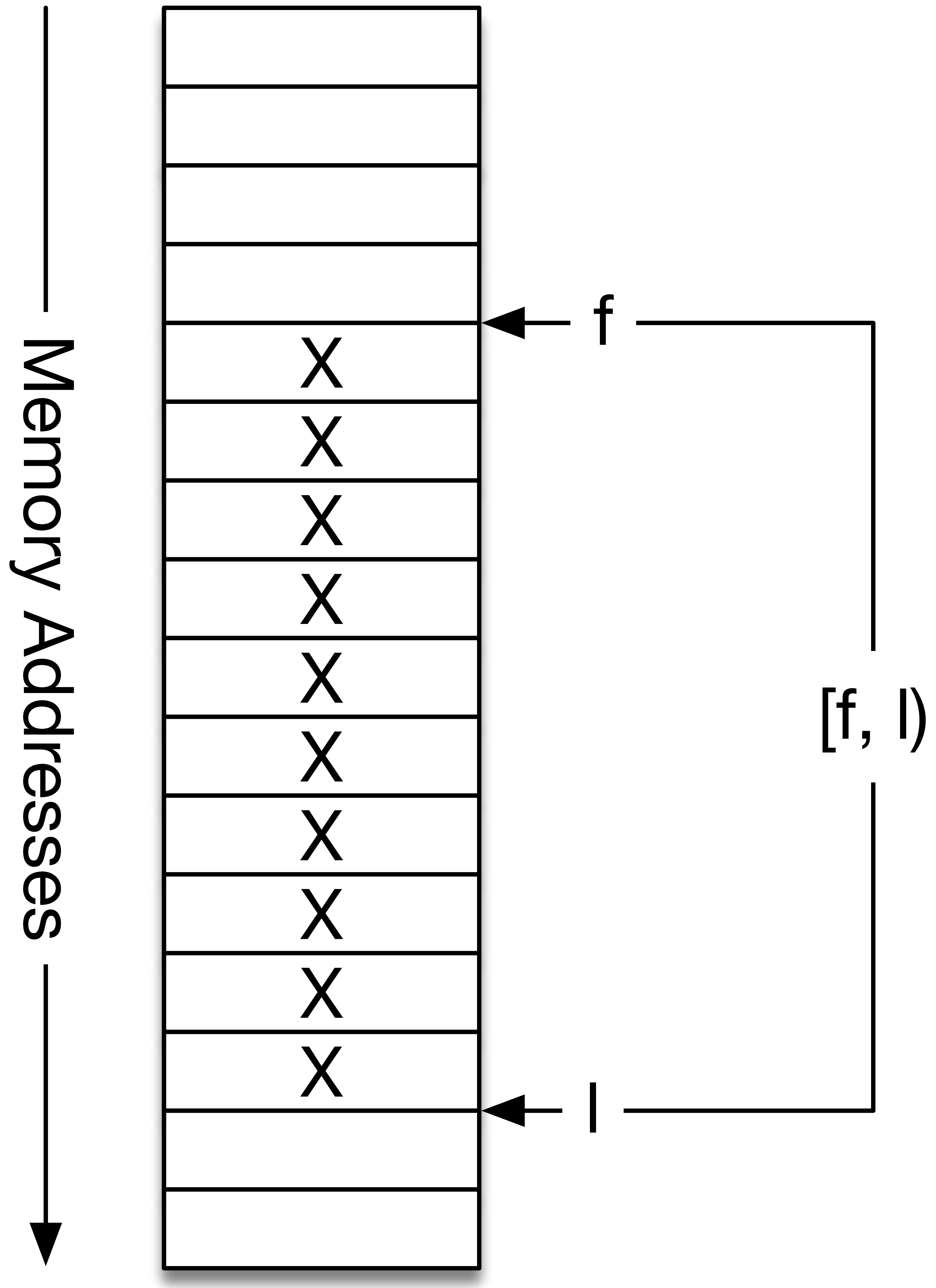
- i.e., `[INT_MIN, INT_MAX]` is not expressible as a half-open interval with type `int`

  Think of the positions as the lines between the elements

Half-Open Intervals



Half-Open Intervals



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

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

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

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

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

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

- Write a paper
- Give talks
- Become famous!

∅ Patents

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

Ai **Adobe Illustrator** Ae **Adobe After Effects**

Adobe

Artwork by **Dan Zucco**