2023

take(5)

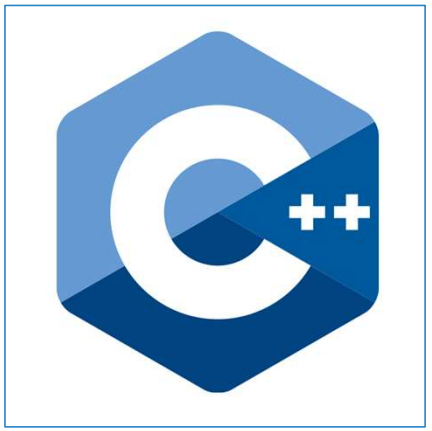
Adventures with Taking Elements from an Input Stream

Barry Revzin

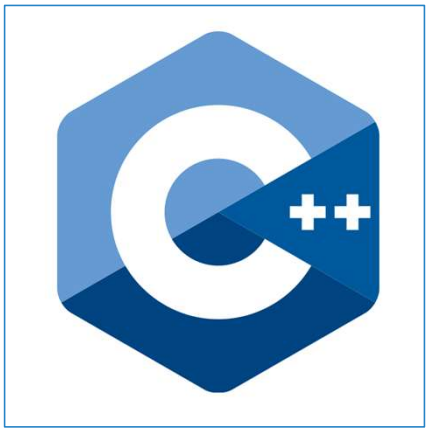
C++ now

About Me

About Me



About Me



https://brevzin.github.io/

@BarryRevzin

Barry

The Example

```
auto main() -> int {

}
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
}
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)) {
        print("loop: {}\n", i);
    }
}
```

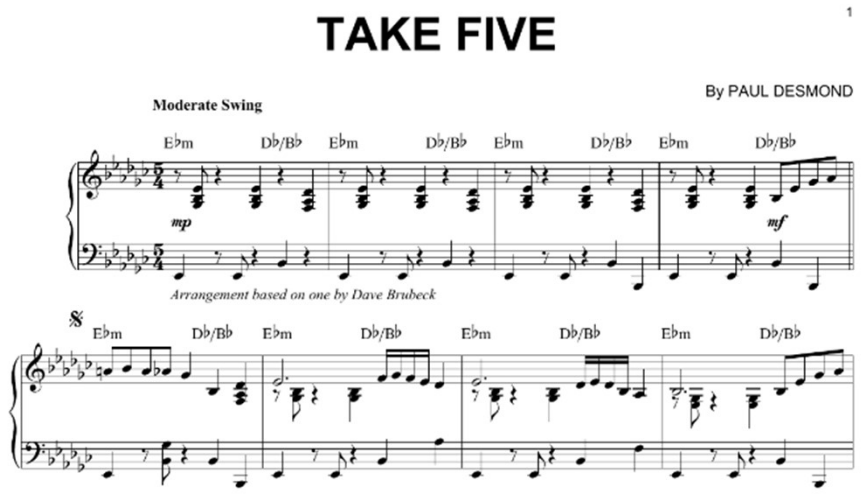
```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
loop: 6
loop: 7
loop: 8
loop: 9
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::take(views::istream<int>(input), 5)) {
        print("loop: {}\n", i);
    }
}
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }
}
```



The Example

```
auto main() -> int {
    istringstream input("1 2");
    for (int i : views::istream<int>(input)
               | views::take(5)) {
        print("loop: {}\n", i);
    }
}
```

```
loop: 1
loop: 2
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: ?
```

The Example

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 6
```

The Example

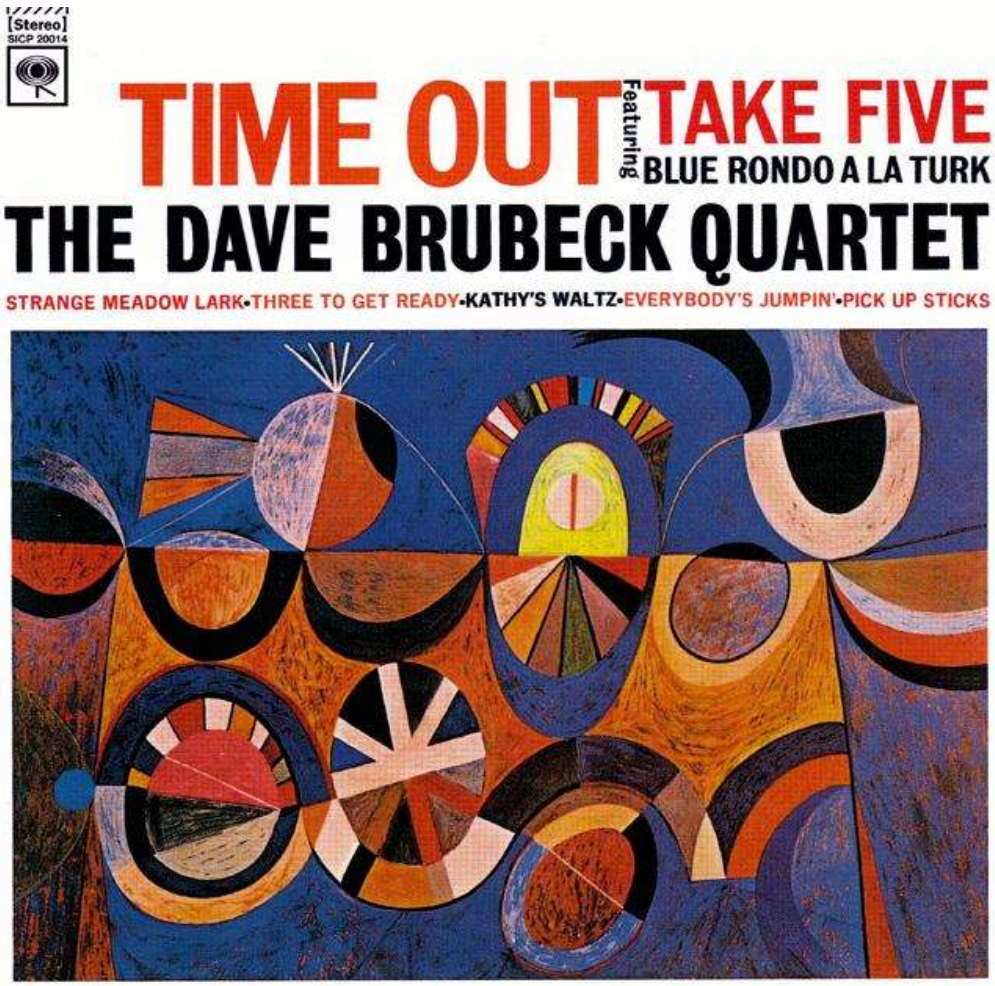
```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

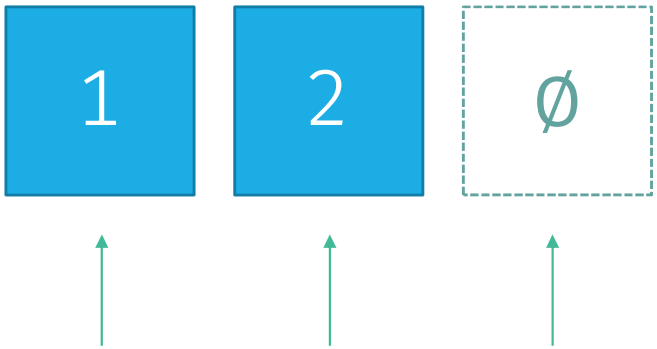
```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 7
```

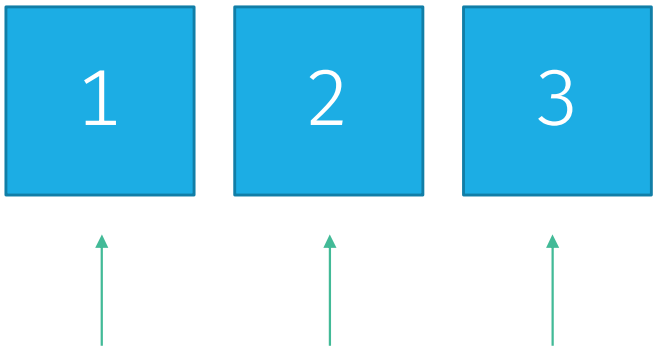
take(5): the search for the missing 6

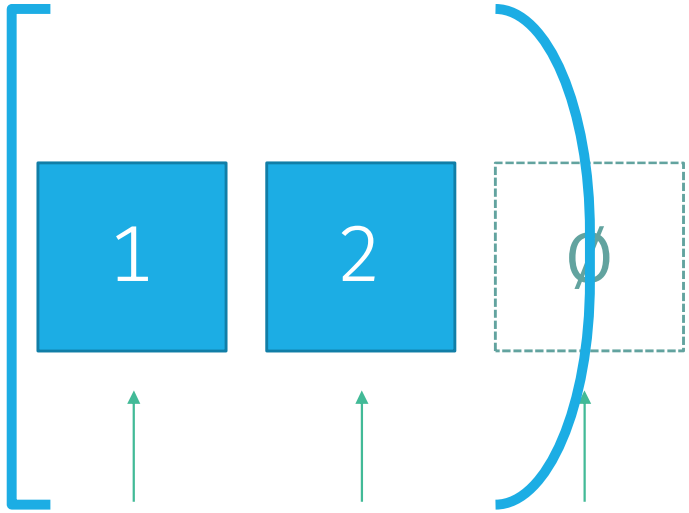
```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
               | views::take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

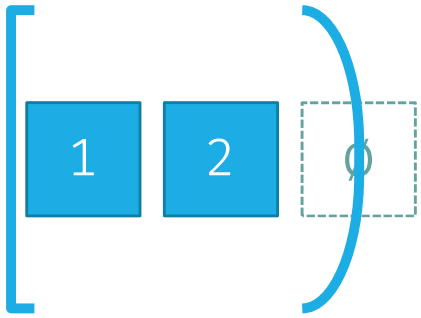
```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 7
```


[Stereo]
SICP 20014
TIME OUT
Featuring
TAKE FIVE
BLUE RONDO A LA TURK
THE DAVE BRUBECK QUARTET
STRANGE MEADOW LARK•THREE TO GET READY•KATHY'S WALTZ•EVERYBODY'S JUMPIN'•PICK UP STICKS

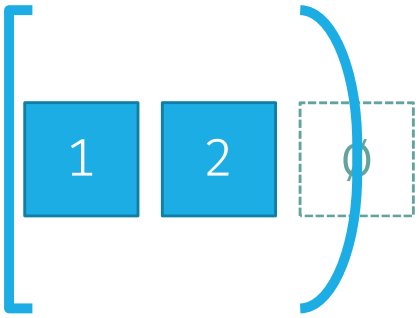

1
2
ø


1
2
3

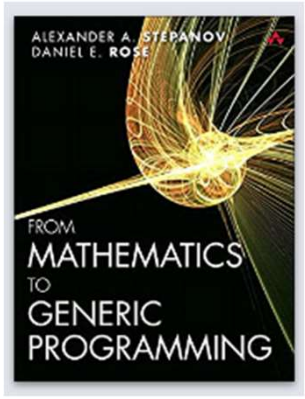

1
2
Ø




```
template <class I>
void destroy(I first, I last);
```



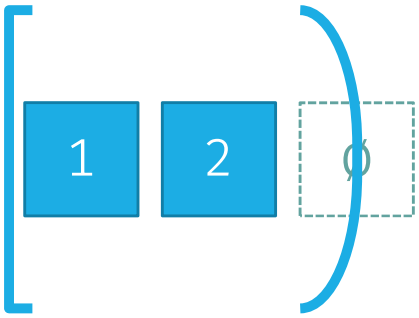

```
template <input_iterator I, sentinel_for<I> S>
void destroy(I first, S last);
```



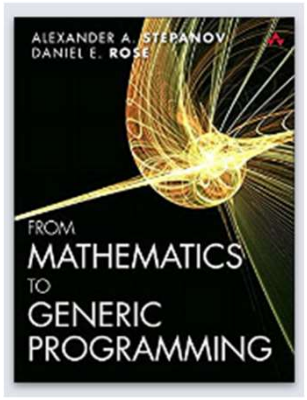
The Law of Useful Return, Revisited

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

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



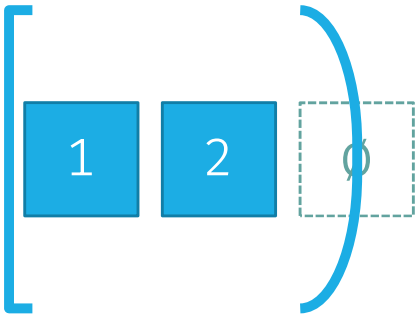

```
template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;
```



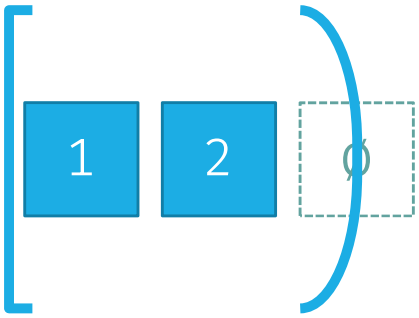
The Law of Useful Return, Revisited

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

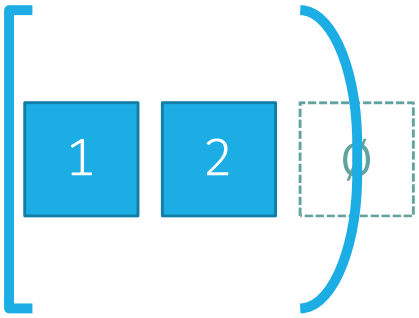
> *A procedure should return all the potentially useful information it computed.*



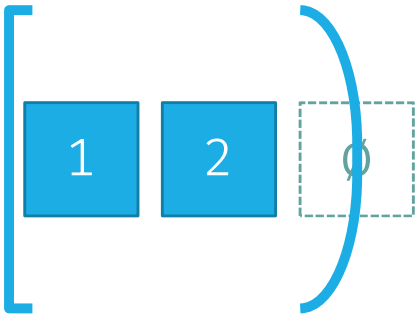

```
template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I;
```



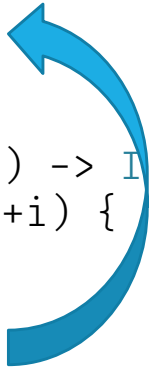

```
template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    for (iter_difference_t<I> i = 0; i != n; ++first, ++i) {
        destroy_at(addressof(*first));
    }
    return first;
}
```




```
template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    for (iter_difference_t<I> i = 0; i != n; ++i) {
        destroy_at(addressof(*first));
        ++first;
    }
    return first;
}
```



```
template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    for (iter_difference_t<I> i = 0; i != n; ++i) {
        destroy_at(addressof(*first));
        ++first;
    }
    return first;
}
```



`counted_iterator<I>`

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) {
        return *current;
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }


};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool {
        return length == 0;
    }
};
```

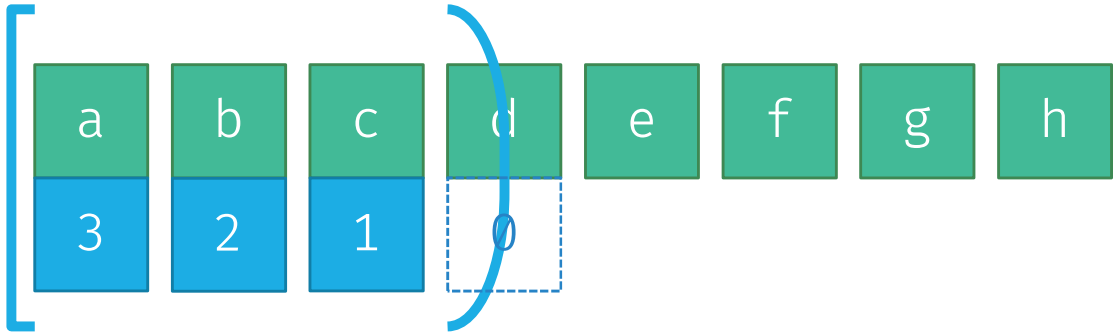
```
struct default_sentinel_t { };
inline constexpr default_sentinel_t default_sentinel{};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }


};
```

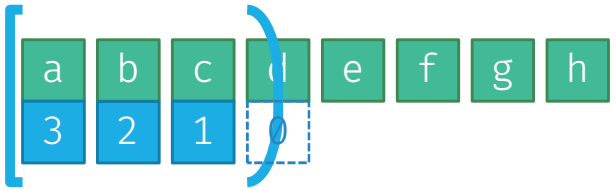
```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

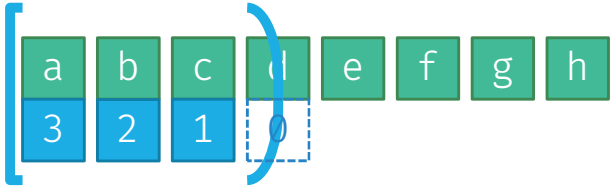
```
template <input_iterator I>
class counted_iterator { /* ... */ };
```



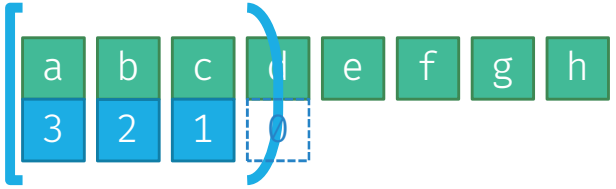
```
template <input_iterator I>
class counted_iterator { /* ... */ };

template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    // ...
}
```



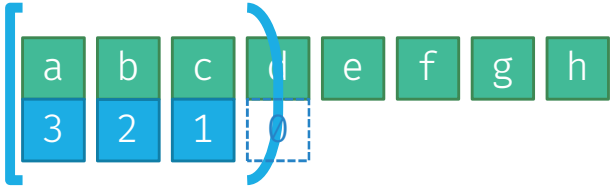
```
template <input_iterator I>
class counted_iterator { /* ... */ };

template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    auto last = destroy(
        counted_iterator<I>{first, n},
        default_sentinel);
}
```



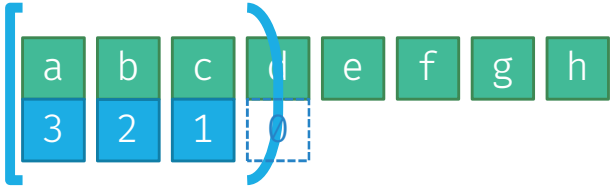
```
template <input_iterator I>
class counted_iterator { /* ... */ };

template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    counted_iterator<I> last = destroy(
        counted_iterator<I>{first, n},
        default_sentinel);
}
```



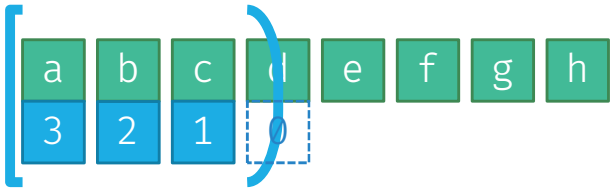
```
template <input_iterator I>
class counted_iterator { /* ... */ };

template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    counted_iterator<I> last = destroy(
        counted_iterator<I>{first, n},
        default_sentinel);
    return last.base();
}
```



```
template <input_iterator I>
class counted_iterator { /* ... */ };

template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    return destroy(
        counted_iterator<I>{first, n},
        default_sentinel
        ).base();
}
```



```
template <input_iterator I>
class counted_iterator { /* ... */ };

template <input_iterator I, sentinel_for<I> S>
auto destroy(I first, S last) -> I;

template <input_iterator I>
auto destroy_n(I first, iter_difference_t<I> n) -> I {
    if constexpr (random_access_iterator<I>) {
        return destroy(first, first + n);
    } else {
        return destroy(
            counted_iterator<I>{first, n},
            default_sentinel
            ).base();
    }
}
```



`views::take`

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
};
```

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;

public:
    auto begin() {
        return counted_iterator(ranges::begin(base), count);
    }
};
```

```
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }
};
```

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        return default_sentinel; // ??
    }
};
```

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        return default_sentinel; // would be take_exactly(count)
    }
};
```

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.base() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

taken `count` elements

reached `base.end()` before taking `count`

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        if constexpr (ranges::sized_range<V>) {
            return counted_iterator<I>(ranges::begin(base), std::min(count, ranges::size(base)));
        } else {
            return counted_iterator<I>(ranges::begin(base), count);
        }
    }

    auto end() {
        if constexpr (ranges::sized_range<V>) {
            return default_sentinel;
        } else {
            struct sentinel {
                ranges::sentinel_t<V> end;

                auto operator==(counted_iterator<I> const& rhs) const -> bool {
                    return rhs.count() == 0
                        or rhs.base() == end;
                }
            };

            return sentinel{ranges::end(base)};
        }
    }
};
```

`views::istream<T>`

```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {

};
```

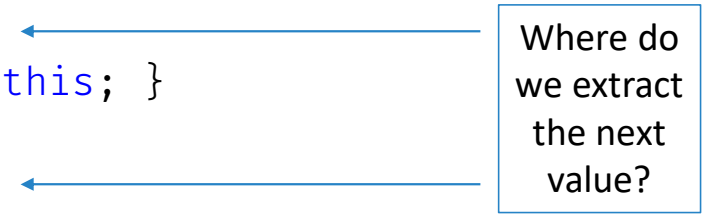
```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
};
```

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
    struct iterator;

public:
    auto begin() -> iterator;

};
```

```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
    struct iterator;

public:
    auto begin() -> iterator;

    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

public:
    auto begin() -> iterator;

    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

public:
    auto begin() -> iterator;

    auto end() -> default_sentinel_t { return default_sentinel; }
};
```



```
views::istream<int>(input)
```

```
views::istream<int>(input)
  | views::stride(2)
```

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

public:
    auto begin() -> iterator;

    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

    auto extract() -> void { *stream >> value; }

public:
    auto begin() -> iterator;

    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

    auto extract() -> void { *stream >> value; }

public:
    auto begin() -> iterator;
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

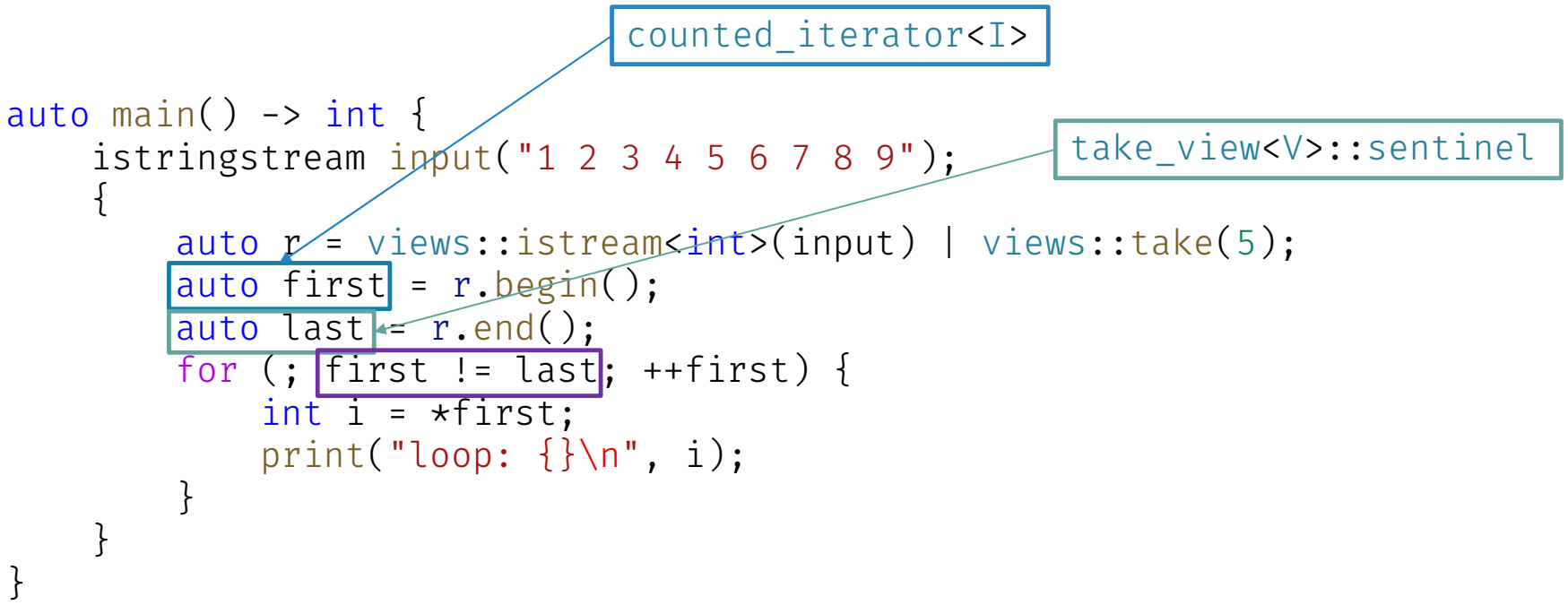
```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val& { return parent->value; }

        auto operator==(default_sentinel_t) const -> bool;
    };

    auto extract() -> void { *stream >> value; }

public:
    auto begin() -> iterator;
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

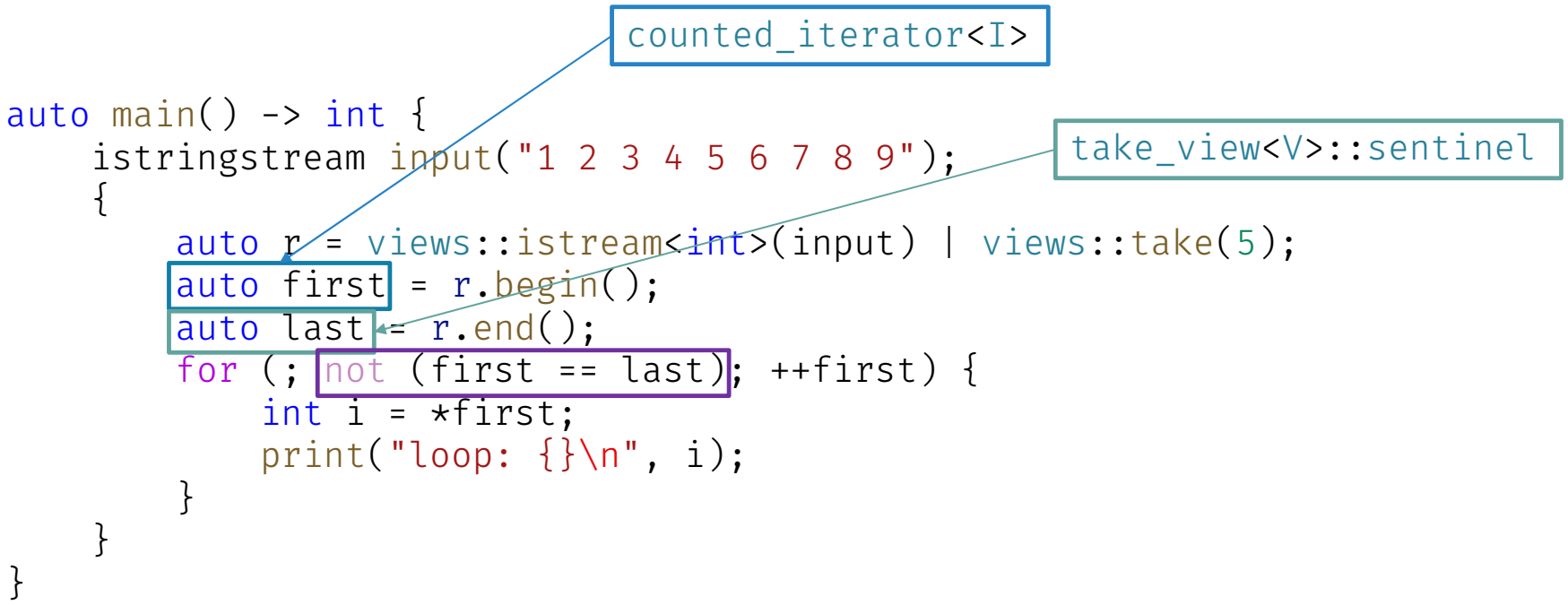
```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val& { return parent->value; }

        auto operator==(default_sentinel_t) const -> bool { return not *parent->stream; }
    };

    auto extract() -> void { *stream >> value; }

public:
    auto begin() -> iterator;
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

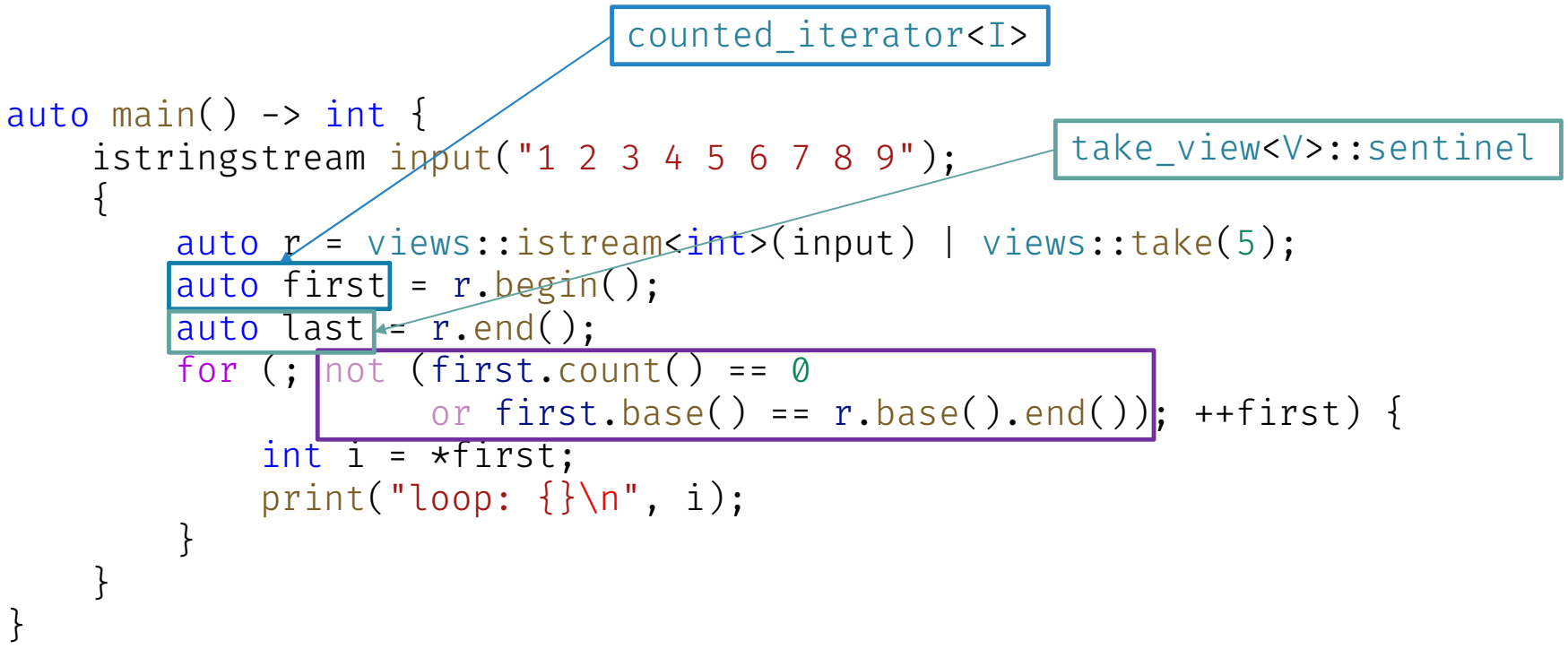
```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator& { parent->extract(); return *this; }
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val& { return parent->value; }

        auto operator==(default_sentinel_t) const -> bool { return not *parent->stream; }
    };

    auto extract() -> void { *stream >> value; }

public:
    auto begin() -> iterator { extract(); return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```
views::istream<int>(input)
    | views::take(5)
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }
}
```


counted_iterator<I>
take_view<V>::sentinel
auto main() -> int {
istringstream input("1 2 3 4 5 6 7 8 9");
{
auto r = views::istream<int>(input) | views::take(5);
auto first = r.begin();
auto last = r.end();
for (; first != last; ++first) {
int i = *first;
print("loop: {}\n", i);
}
}
}

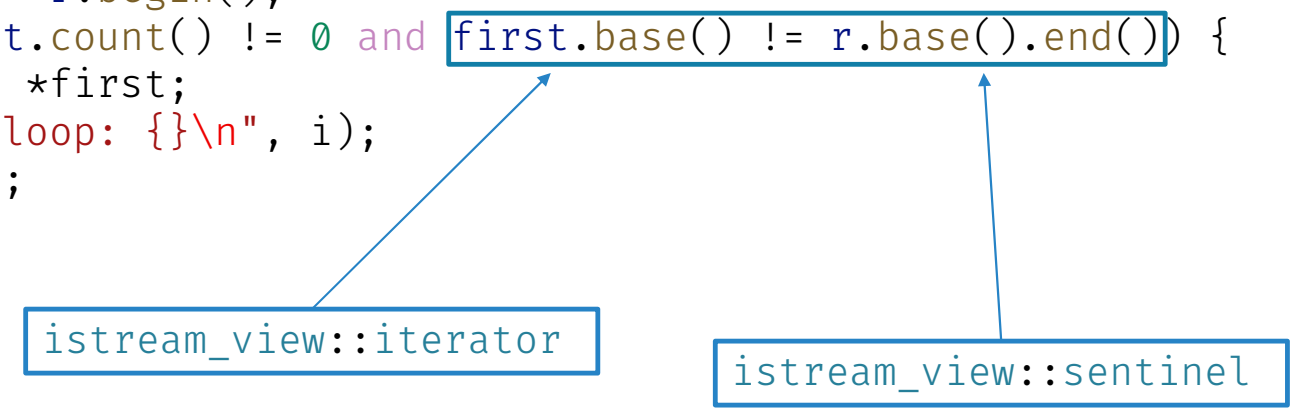

counted_iterator<I>
take_view<V>::sentinel
auto main() -> int {
istringstream input("1 2 3 4 5 6 7 8 9");
{
auto r = views::istream<int>(input) | views::take(5);
auto first = r.begin();
auto last = r.end();
for (; not (first == last); ++first) {
int i = *first;
print("loop: {}\n", i);
}
}
}


counted_iterator<I>
take_view<V>::sentinel
auto main() -> int {
istringstream input("1 2 3 4 5 6 7 8 9");
{
auto r = views::istream<int>(input) | views::take(5);
auto first = r.begin();
auto last = r.end();
for (; not (first.count() == 0
or first.base() == r.base().end()); ++first) {
int i = *first;
print("loop: {}\n", i);
}
}
}

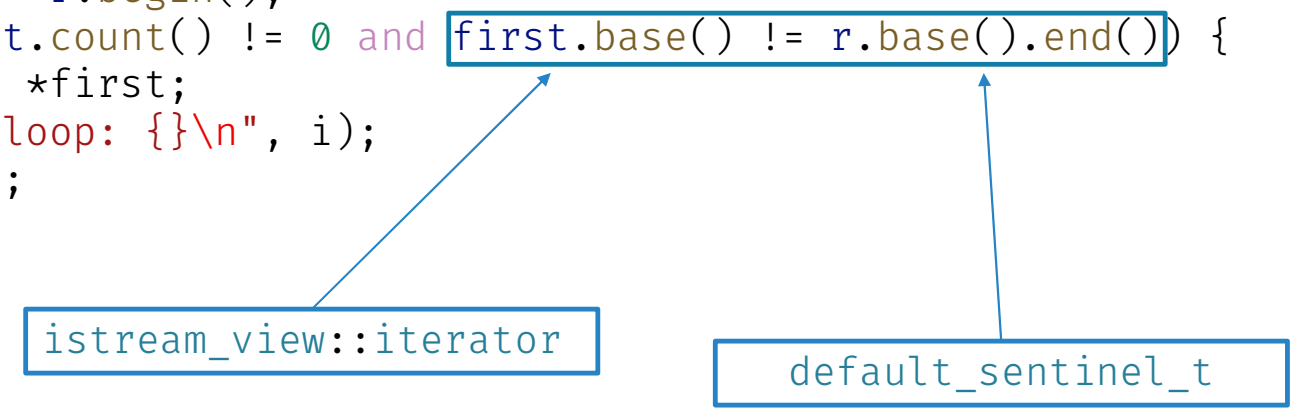
```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input) | views::take(5);
        auto first = r.begin();
        for (; not (first.count() == 0
                    or first.base() == r.base().end()); ++first) {
            int i = *first;
            print("loop: {}\n", i);
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input) | views::take(5);
        auto first = r.begin();
        for (; first.count() != 0
               and first.base() != r.base().end(); ++first) {
            int i = *first;
            print("loop: {}\n", i);
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input) | views::take(5);
        auto first = r.begin();
        while (first.count() != 0 and first.base() != r.base().end()) {
            int i = *first;
            print("loop: {}\n", i);
            ++first;
        }
    }
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input) | views::take(5);
        auto first = r.begin();
        while (first.count() != 0 and first.base() != r.base().end()) {
            int i = *first;
            print("loop: {}\n", i);
            ++first;
        }
    }
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input) | views::take(5);
        auto first = r.begin();
        while (first.count() != 0 and input) {
            int i = *first;
            print("loop: {}\n", i);
            ++first;
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input);
        auto first = counted_iterator(r.begin(), 5);
        while (first.count() != 0 and input) {
            int i = *first;
            print("loop: {}\n", i);
            ++first;
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        auto r = views::istream<int>(input);
        auto first = r.begin();
        int count = 5;
        while (count != 0 and input) {
            int i = *first;
            print("loop: {}\n", i);
            ++first;
            --count;
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        auto first = r.begin();
        int count = 5;
        while (count != 0 and input) {
            int i = *first;
            print("loop: {}\n", i);
            ++first;
            --count;
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        input >> __storage;
        int count = 5;
        while (count != 0 and input) {
            int i = *first;
            print("loop: {}\n", i);
            input >> __storage;
            --count;
        }
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        input >> __storage;
        int count = 5;
        while (count != 0 and input) {
            int i = __storage;
            print("loop: {}\n", i);
            input >> __storage;
            --count;
        }
    }
}
```

What happens when `count == 1`?

No access to `__storage`

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    auto r = views::istream<int>(input)
           | views::take(5);
    for (int i : r) {
        print("loop: {}\n", i);
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    auto r = views::istream<int>(input)
           | views::take(5);
    auto it = r.begin();
    for (; it != r.end(); ++it) {
        print("loop: {}\n", *it);
    }
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    auto r = views::istream<int>(input)
           | views::take(5);
    auto it = r.begin();
    for (; it != r.end(); ++it) {
        print("loop: {}\n", *it);
    }
    auto next = it.base();
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    auto r = views::istream<int>(input)
           | views::take(5);
    auto it = r.begin();
    for (; it != r.end(); ++it) {
        print("loop: {}\n", *it);
    }
    auto next = std::move(it).base();
}
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    auto r = views::istream<int>(input)
           | views::take(5);
    auto it = r.begin();
    for (; it != r.end(); ++it) {
        print("loop: {}\n", *it);
    }
    auto next = std::move(it).base();
    if (next != r.base().end()) {
        print("next: {}\n", *next);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 6
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    auto r = views::istream<int>(input)
           | views::take(5);
    for (int i : r) {
        print("loop: {}\n", i);
    }
    if (input) {
        print("next: {}\n", r.base().cached());
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 6
```

Is This Broken?

Can It Be Fixed?

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

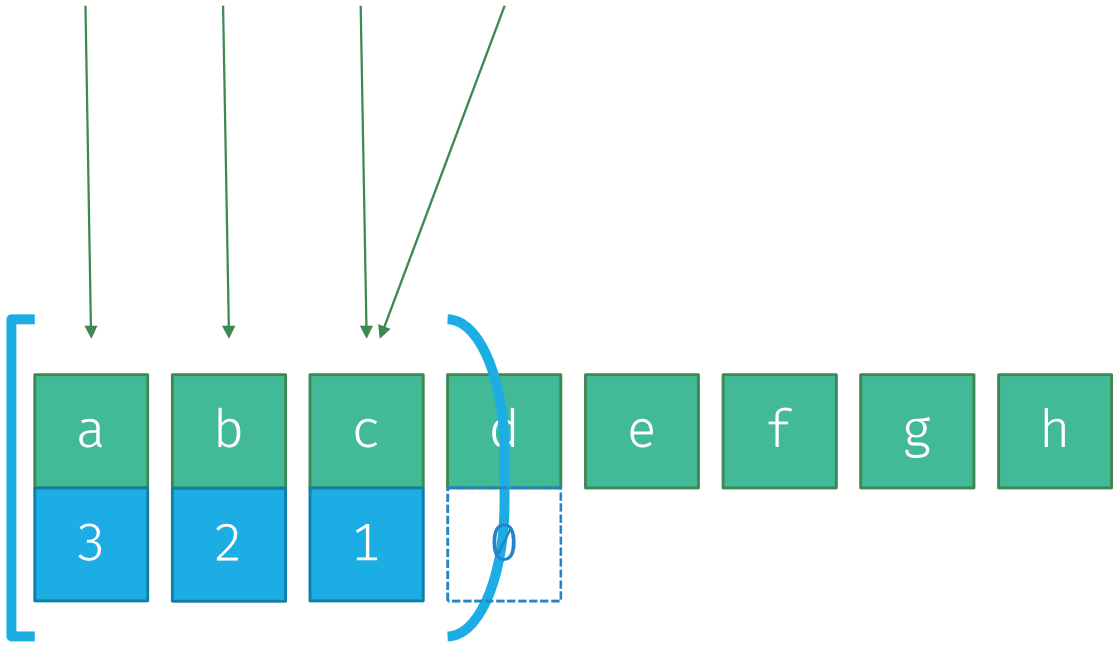
What happens when `length == 1`?

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

What happens when
length == 1
and not
random_access_iterator<I>?

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        --length;
        ++current;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

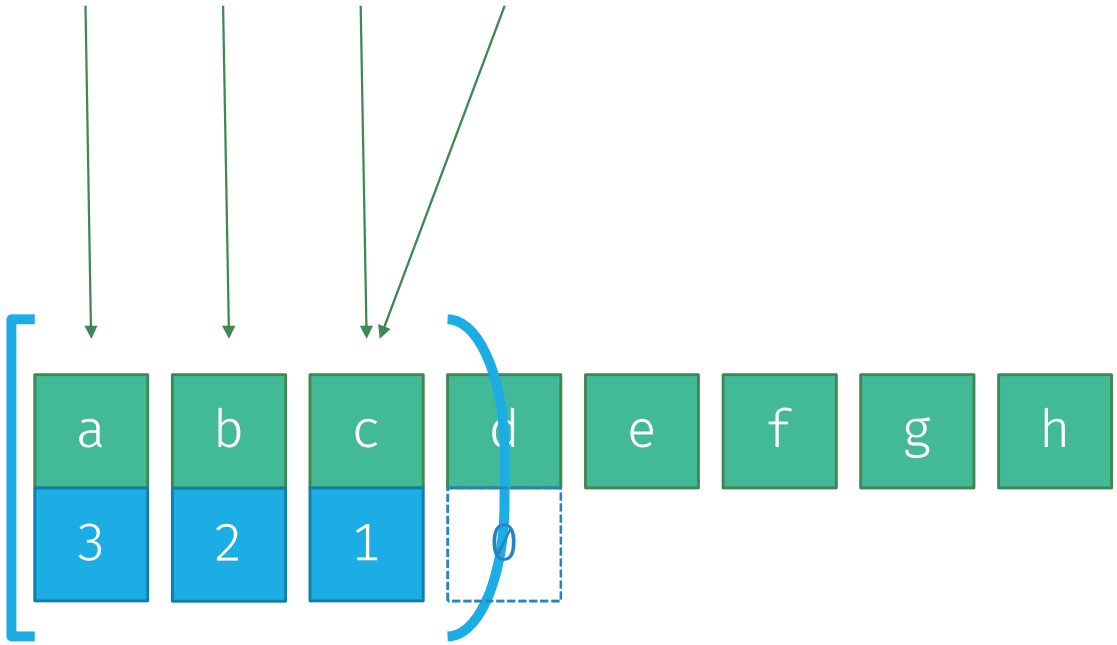

a
b
c
d
e
f
g
h
3
2
1
0

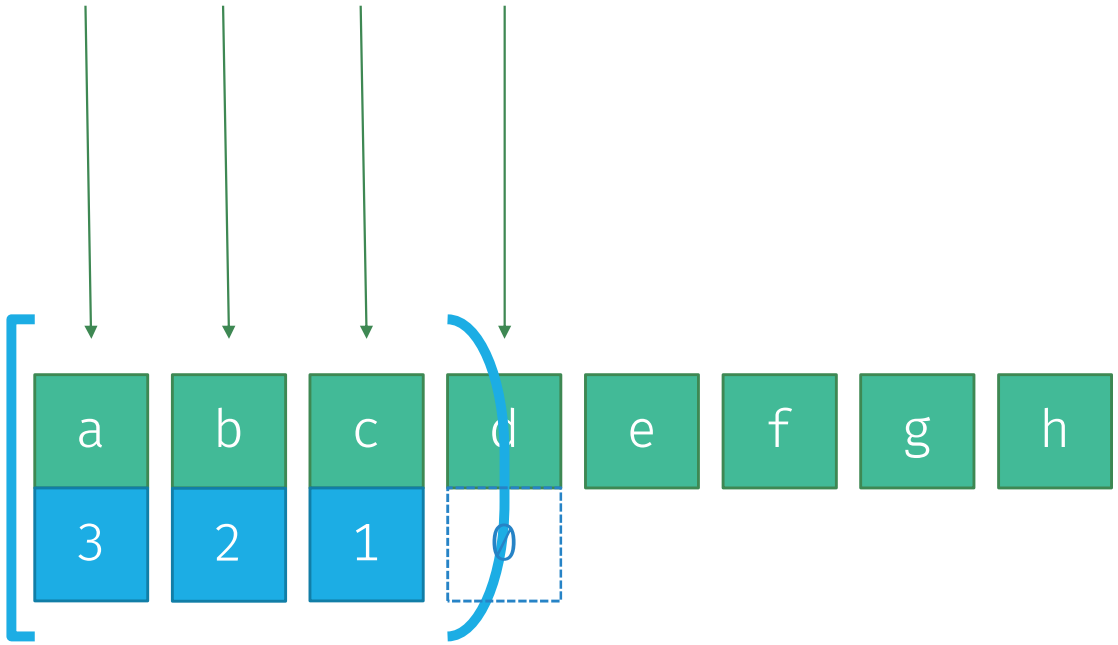
```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto base() const -> I { return current; }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto base() const -> I {
        return length == 0 ? ranges::next(current)
                           : current;
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto base() const -> I {
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return length == 0 ? ranges::next(current)
                               : current;
        }
    }
};
```


a
b
c
d
e
f
g
h
3
2
1
0


a
b
c
d
e
f
g
h
3
2
1
0

```
auto ci = counted_iterator<I>(i, N);
auto j  = ci.base();
```

```
auto ci = counted_iterator<I>(i, N);
auto j  = ci.base();
assert(j == i);
```

```cpp
auto ci = counted_iterator<I>(i, 0);
auto j  = ci.base();
assert(j == i);
```

```
auto end = destroy_n(begin_, size_);
```

```
auto end = destroy(
    counted_iterator<I>(begin_, size_),
    default_sentinel
).base();
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto base() const -> I
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return length == 0 ? ranges::next(current)
                               : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need_to_advance = false;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto base() const -> I
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return length == 0 ? ranges::next(current)
                               : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need_to_advance = false;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        } else {
            need_to_advance = true;
        }
        return *this;
    }

    auto base() const -> I
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return length == 0 ? ranges::next(current)
                               : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need_to_advance = false;

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        } else {
            need_to_advance = true;
        }
        return *this;
    }

    auto base() const -> I
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return need_to_advance ? ranges::next(current)
                                   : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need-to-advance = false; // only present if not random_access_iterator<I>

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        } else {
            need-to-advance = true;
        }
        return *this;
    }

    auto base() const -> I
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return need-to-advance ? ranges::next(current)
                                   : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need-to-advance = false; // only present if not random_access_iterator<I>

public:
    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        } else {
            need-to-advance = true;
        }
        return *this;
    }

    auto operator--() -> counted_iterator& requires bidirectional_iterator<I> {
        ++length;
        // ???
        return *this;
    }

    auto base() const -> I;
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need-to-advance = false; // only present if not random_access_iterator<I>

public:
    auto operator++() -> counted_iterator&;

    auto base() const -> I
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return need-to-advance ? ranges::next(current)
                                   : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need-to-advance = false; // only present if not random_access_iterator<I>

public:
    auto operator++() -> counted_iterator&;

    auto base() const& -> I const& {
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return need-to-advance ? ranges::next(current)
                                   : current;
        }
    }

    auto base() && -> I {
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return need-to-advance ? ranges::next(current)
                                   : current;
        }
    }
};
```

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;
    bool need-to-advance = false; // only present if not random_access_iterator<I>

public:
    auto operator++() -> counted_iterator&;

    auto base() const& -> I const& {
        if constexpr (random_access_iterator<I>) {
            return current;
        } else {
            return need-to-advance ? ranges::next(current)
                                   : current;
        }
    }

    auto base() && -> I
        if constexpr (random_access_iterator<I>) {
            return std::move(current);
        } else {
            return need-to-advance ? ranges::next(std::move(current))
                                   : std::move(current);
        }
    }
};
```

```cpp
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator++() -> counted_iterator&;

    auto base() const& -> I const& requires random_access_iterator<I> {
        return current;
    }

    auto base() const& -> I {
        return need-to-advance ? ranges::next(current)
                               : current;
    }

    auto base() && -> I {
        if constexpr (random_access_iterator<I>) {
            return std::move(current);
        } else {
            return need-to-advance ? ranges::next(std::move(current))
                                   : std::move(current);
        }
    }
};
```

```cpp
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator++() -> counted_iterator&;

    auto base() const& -> I const& requires random_access_iterator<I> {
        return current;
    }

    auto base() const& -> I requires forward_iterator<I> {
        return need-to-advance ? ranges::next(current)
                               : current;
    }

    auto base() && -> I {
        if constexpr (random_access_iterator<I>) {
            return std::move(current);
        } else {
            return need-to-advance ? ranges::next(std::move(current))
                                   : std::move(current);
        }
    }
};
```

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.base() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

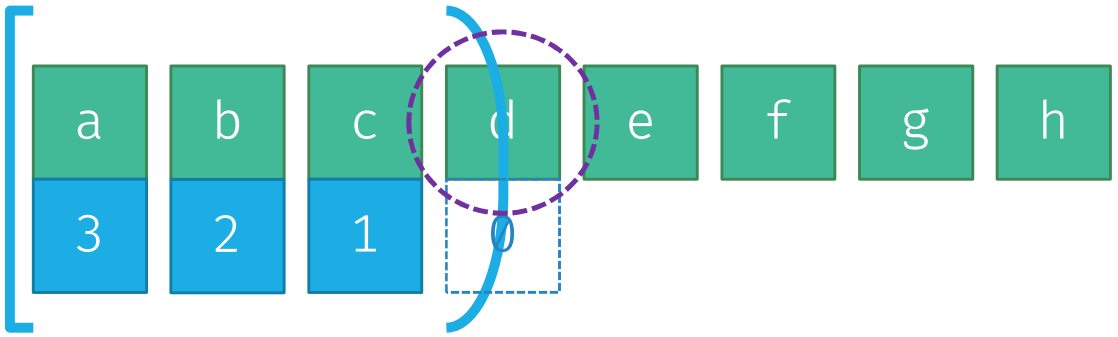
taken `count` elements

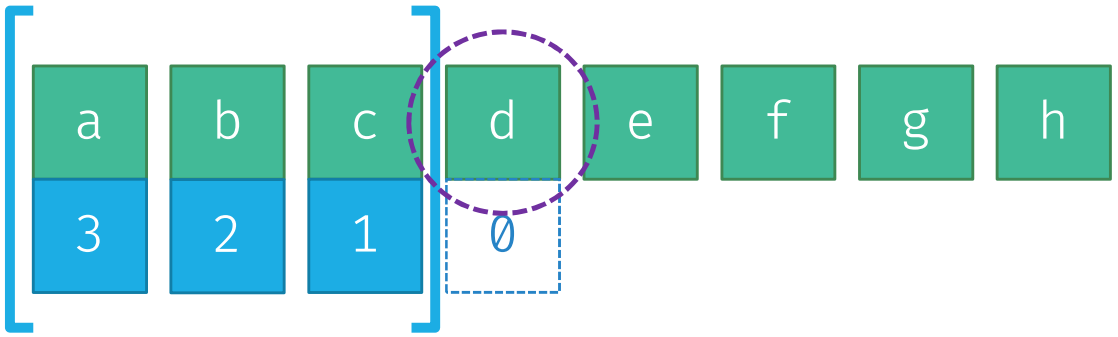
reached `base.end()` before taking `count`

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.base?() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

taken `count` elements

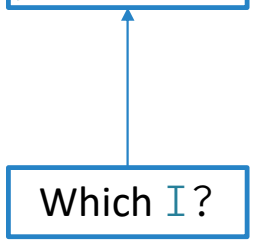
reached `base.end()` before taking `count`


a
b
c
d
e
f
g
h
3
2
1
0


a
b
c
d
e
f
g
h
3
2
1
0

```
template <class I, class Size, class O>
auto copy_n(I first, Size n, O result) -> O;
```

```
template <class I, class Size, class O>
auto copy_n(I first, Size n, O result) -> O;

namespace ranges {
    template <input_iterator I, output_iterator<iter_reference_t<I>> O>
    auto copy_n(I first, iter_difference_t<I> n, O result) -> pair<I, O>;
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    vector<int> out;
    NAMESPACE::copy_n(istream_iterator<int>(input), 5, back_inserter(out));
    print(" out: {}\n", out);

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```cpp
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    vector<int> out;
    std::ranges::copy_n(istream_iterator<int>(input), 5, back_inserter(out));
    print(" out: {}\n", out);

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
 out: [1, 2, 3, 4, 5]
next: 7
```

```cpp
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    vector<int> out;
    auto [i, o] = std::ranges::copy_n(istream_iterator<int>(input), 5, back_inserter(out));
    print("out: {}\n", out);

    print("  *i: {}\n", *i);

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
 out: [1, 2, 3, 4, 5]
  *i: 6
next: 7
```

```cpp
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    vector<int> out;
    auto i = std::ranges::copy_n(istream_iterator<int>(input), 5, back_inserter(out)).in;
    print("out: {}\n", out);

    print("  *i: {}\n", *i);

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
 out: [1, 2, 3, 4, 5]
  *i: 6
next: 7
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    vector<int> out;
    std::copy_n(istream_iterator<int>(input), 5, back_inserter(out));
    print(" out: {}\n", out);



    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
 out: [1, 2, 3, 4, 5]
next: 6
```

2471. `copy_n`'s number of `InputIterator` increments unspecified

Section: 27.7.1 [alg.copy] **Status:** Open **Submitter:** Jonathan Wakely **Opened:** 2015-01-28 **Last modified:** 2018-06-22

Discussion:

It's unspecified how many times `copy_n` increments the `InputIterator`. `uninitialized_copy_n` is specified to increment it exactly `n` times, which means if an `istream_iterator` is used then the next character after those copied is read from the stream and then discarded, losing data.

I believe all three of Dinkumware, libc++ and libstdc++ implement `copy_n` with `n - 1` increments of the `InputIterator`, which avoids reading and discarding a character when used with `istream_iterator`, but is inconsistent with `uninitialized_copy_n` and causes surprising behaviour with `istreambuf_iterator` instead, because `copy_n(in, 2, copy_n(in, 2, out))` is not equivalent to `copy_n(in, 4, out)`

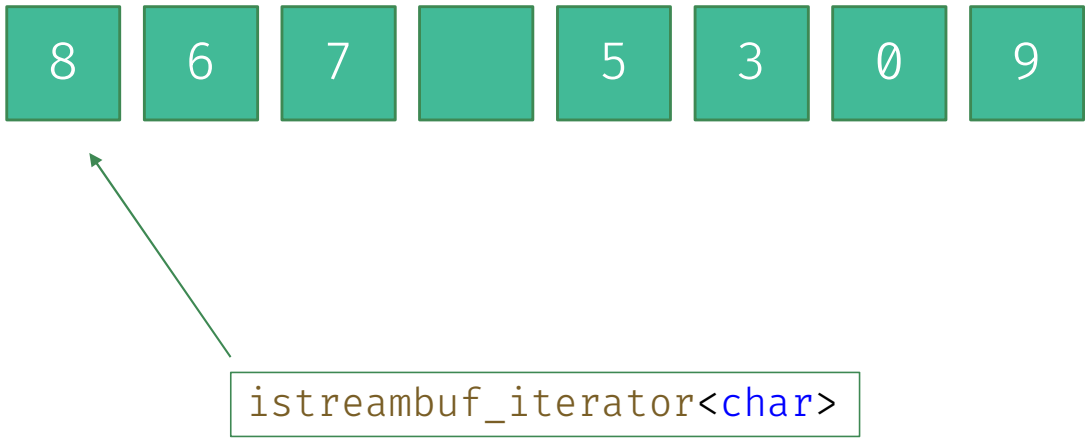
This is a mess. Append to Effects: If the InputIterator is not a forward iterator, increments n-1 times. Otherwise the number of increments is not more than n. (ncm) The preceding proposition is unsatisfactory, because it is wrong for istreambuf_iterator, which is much more useful than istream_iterator. Proposing instead: Append to Effects: If InputIterator is istream_iterator for some T, increments n-1 times. Otherwise, increments n times. Want a paper exploring what the implementations actually do, and what non-uniformity is "right".

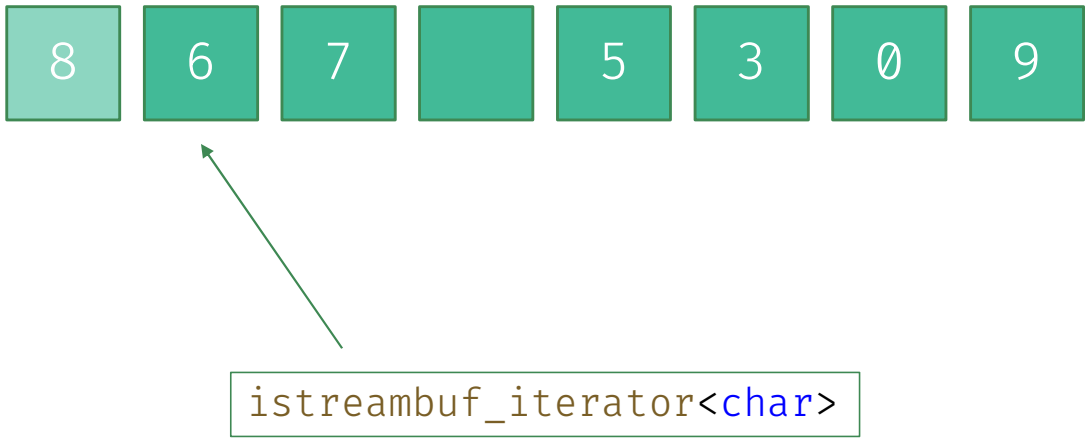
`istream_iterator(istream_type& s);`

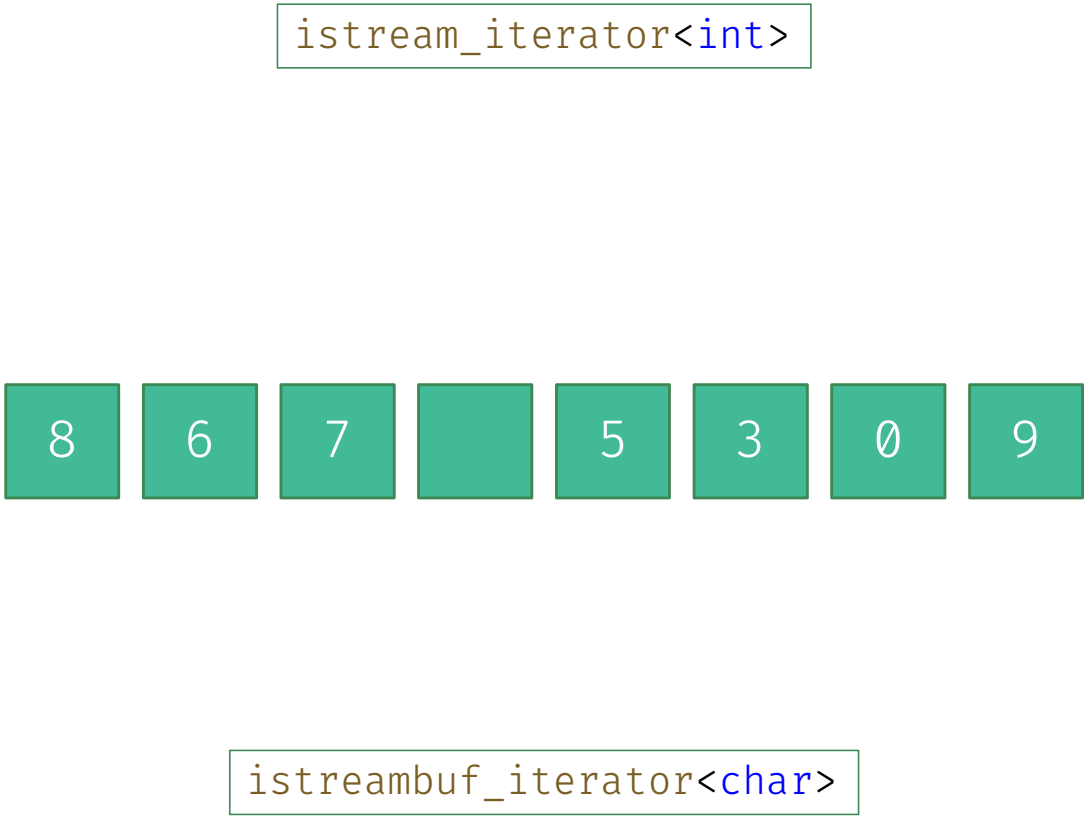
4 *Effects*: Initializes `in_stream` with `addressof(s)`, value-initializes `value`, and then calls `operator++()`.

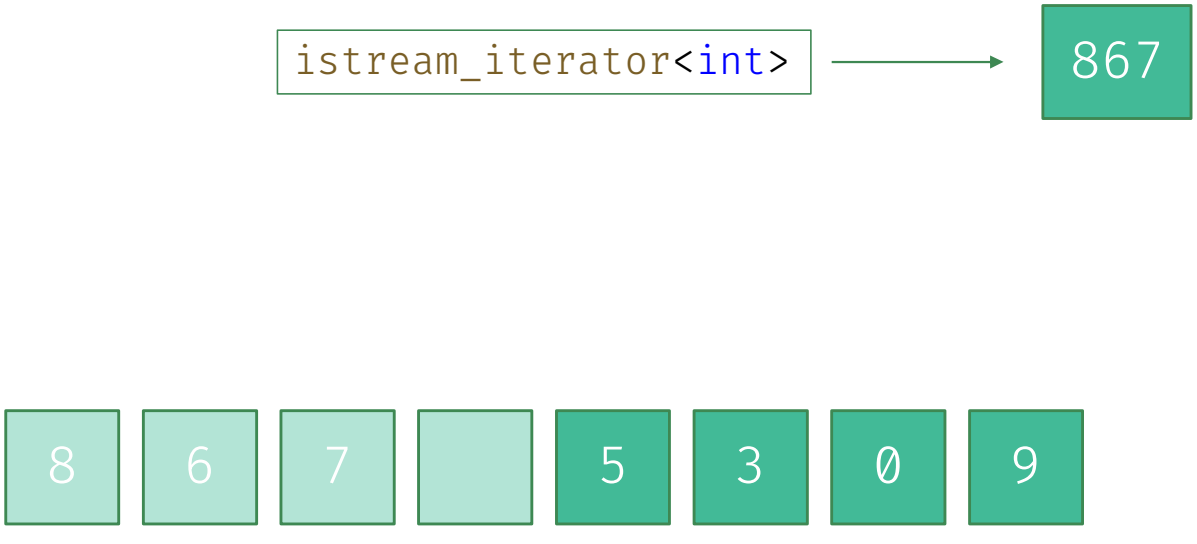
`istreambuf_iterator(istream_type& s) noexcept;`

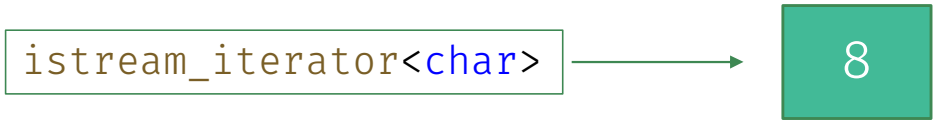
3 *Effects*: Initializes `sbuf_` with `s.rdbuf()`.


8
6
7
5
3
0
9
istreambuf_iterator<char>


8
6
7
5
3
0
9
istreambuf_iterator<char>

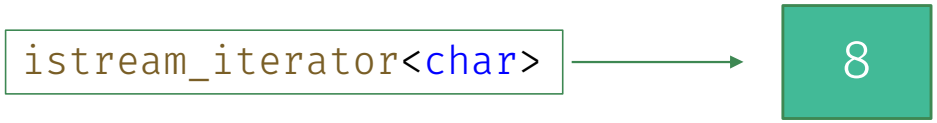

istream_iterator<int>
8
6
7
5
3
0
9
istreambuf_iterator<char>


istream_iterator<int>
867
8
6
7
5
3
0
9
istreambuf_iterator<char>


istream_iterator<char>
8

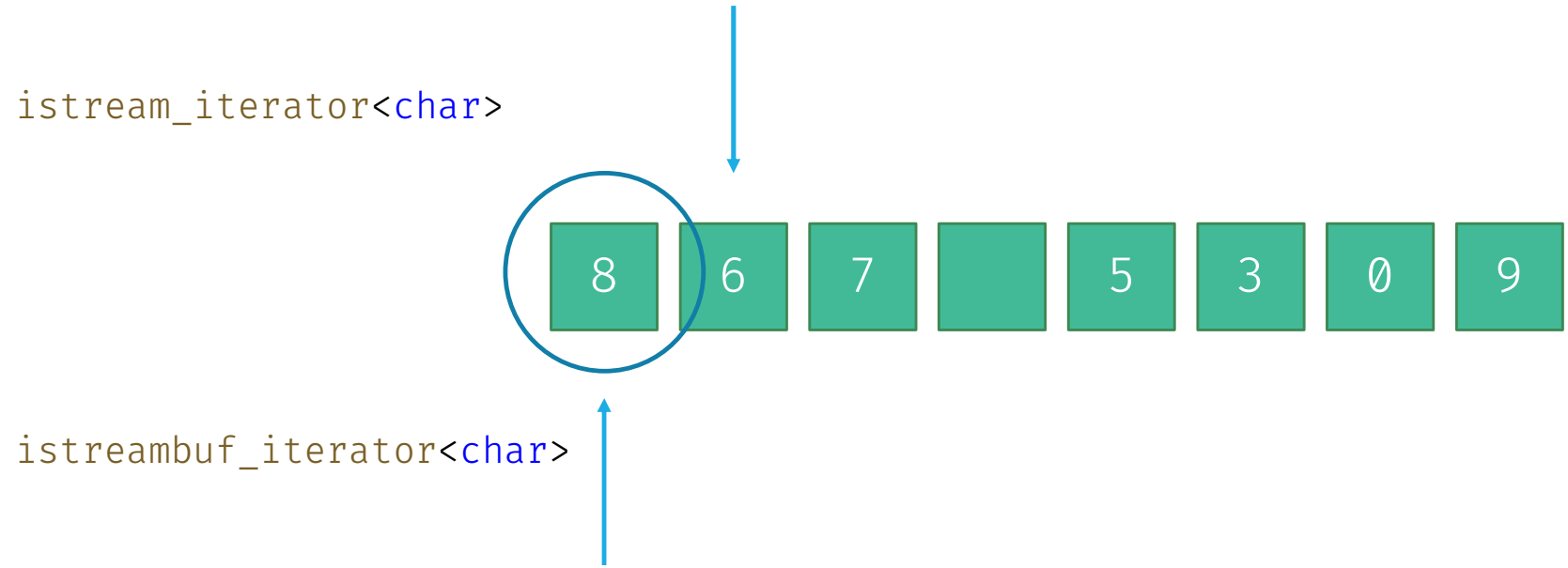

8
6
7
5
3
0
9


istreambuf_iterator<char>

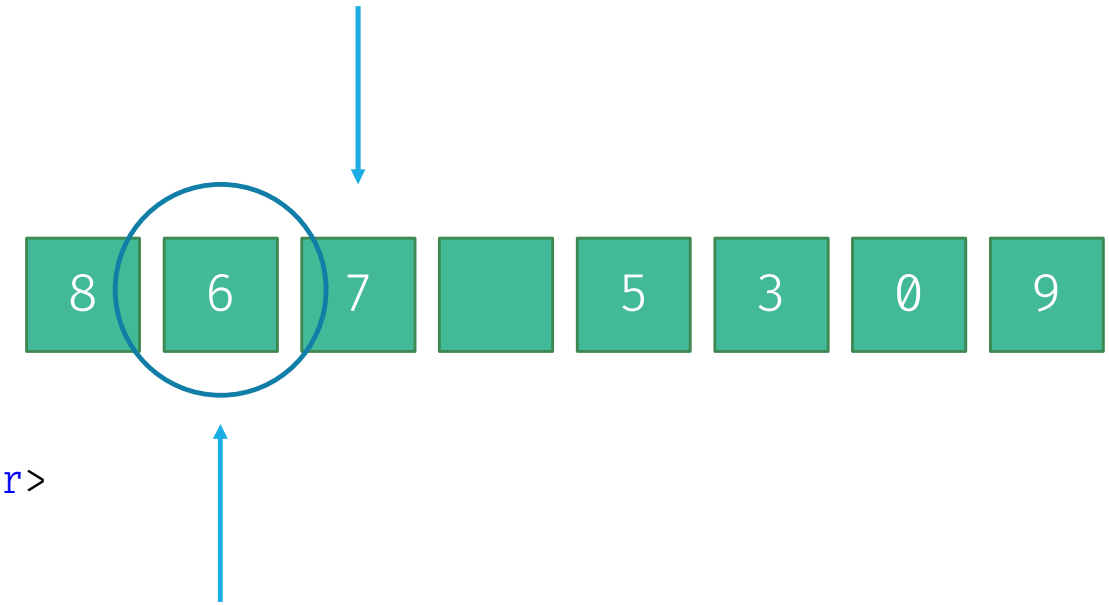

istream_iterator<char>
8


8
6
7
5
3
0
9


istreambuf_iterator<char>

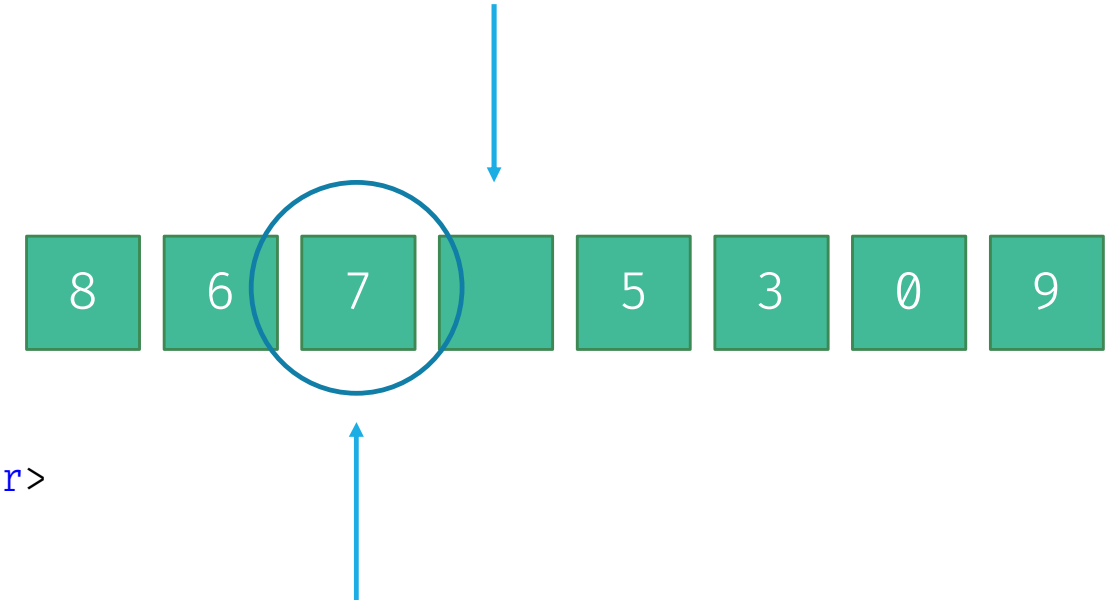

istream_iterator<char>
8
6
7
5
3
0
9
istreambuf_iterator<char>

`istream_iterator<char>`



`istreambuf_iterator<char>`

`istream_iterator<char>`



`istreambuf_iterator<char>`

~~Is This Broken?~~

Can It Be Fixed?

```
template <input_iterator I>
class counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class closed_counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        ++current;
        --length;
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class closed_counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class closed_counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto unsafe_base() const -> I { return current; }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class closed_counted_iterator {
    I current;
    iter_difference_t<I> length;

public:
    auto operator*() const -> decltype(auto) { return *current; }

    auto operator++() -> counted_iterator& {
        --length;
        if (random_access_iterator<I> or length > 0) {
            ++current;
        }
        return *this;
    }

    auto operator==(default_sentinel_t) const -> bool { return length == 0; }

    auto unsafe_base() const& -> I const& {
        assert(length > 0);
        return current;
    }
    auto count() const -> iter_difference_t<I> { return length; }
};
```

```
template <input_iterator I>
class closed_counted_iterator {
public:
  auto operator++() -> counted_iterator& {
    --length;
    if (random_access_iterator<I> or length > 0) {
      ++current;
    }
    return *this;
  }

  auto unsafe_base() const& -> I const& {
    assert(length > 0);
    return current;
  }
};
```

N−1 increments

base() invalid at end

```
template <input_iterator I>
class counted_iterator {
public:
  auto operator++() -> counted_iterator& {
    --length;
    ++current;
    return *this;
  }

  auto base() const& -> I const& {
    return current;
  }

};
```

N increments

base() valid at end

```cpp
template <view V>
class take_view : public ranges::view_interface<take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.base() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

```cpp
template <view V>
class closed_take_view : public ranges::view_interface<closed_take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.base() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

```
template <view V>
class closed_take_view : public ranges::view_interface<closed_take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return closed_counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.base() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

```cpp
template <view V>
class closed_take_view : public ranges::view_interface<closed_take_view<V>> {
    V base;
    ranges::range_difference_t<V> count;
    using I = ranges::iterator_t<V>;

public:
    auto begin() {
        return closed_counted_iterator<I>(ranges::begin(base), count);
    }

    auto end() {
        struct sentinel {
            ranges::sentinel_t<V> end;

            auto operator==(counted_iterator<I> const& rhs) const -> bool {
                return rhs.count() == 0
                    or rhs.unsafe_base() == end;
            }
        };

        return sentinel{ranges::end(base)};
    }
};
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 7
```

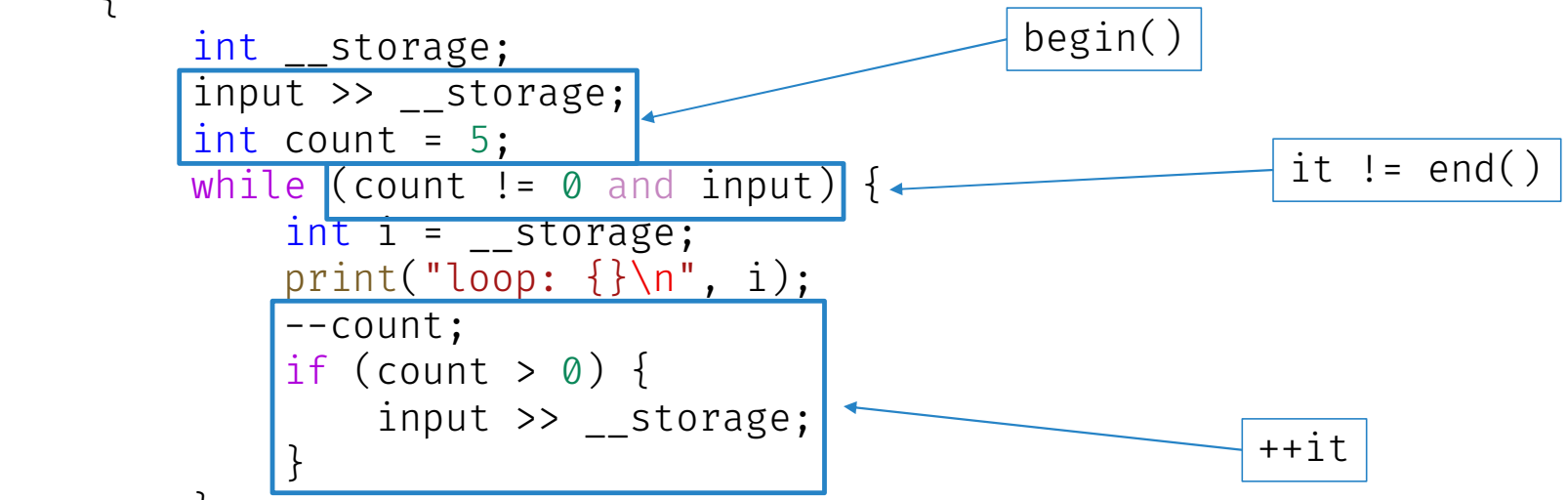
```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream<int>(input)
                | views::closed_take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 6
```

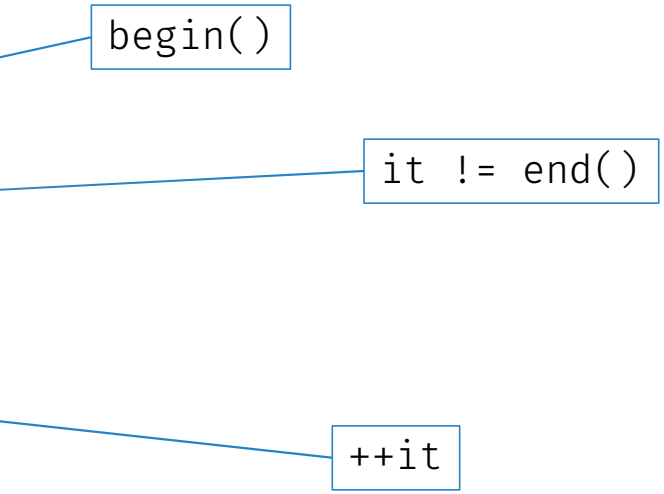
Is This Broken?
Can It Be Fixed?

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        input >> __storage;
        int count = 5;
        while (count != 0 and input) {
            int i = __storage;
            print("loop: {}\n", i);
            input >> __storage;
            --count;
        }
    }
}
```

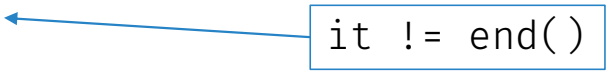
```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        input >> __storage;
        int count = 5;
        while (count != 0 and input) {
            int i = __storage;
            print("loop: {}\n", i);
            --count;
            if (count > 0) {
                input >> __storage;
            }
        }
    }
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        input >> __storage;
        int count = 5;
        while (count != 0 and input) {
            int i = __storage;
            print("loop: {}\n", i);
            --count;
            if (count > 0) {
                input >> __storage;
            }
        }
    }
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        int count = 5;
        while (count != 0 and input >> __storage) {
            int i = __storage;
            print("loop: {}\n", i);
            --count;
        }
    }
}
```



```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator& { parent->extract(); return *this; }
        auto operator++(int) -> void { ++*this; }

        auto operator*() const -> Val& { return parent->value; }

        auto operator==(default_sentinel_t) const -> bool { return not *parent->stream; }
    };

    auto extract() -> void { *stream >> value; }

public:
    auto begin() -> iterator { extract(); return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

public:
    auto begin() -> iterator;
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

Where do we extract the next value?

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

public:
    auto begin() -> iterator;
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

Where do we extract the next value?

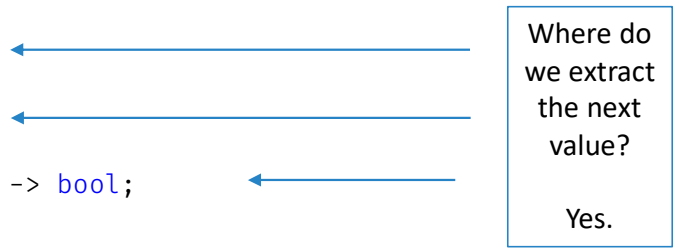
Yes.

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };

public:
    auto begin() -> iterator { return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

Where do we extract the next value?

Yes.

```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
    bool dirty = true;

    auto prime() -> void {
        if (dirty) {
            *stream >> value;
            dirty = false;
        }
    }

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;

        auto operator*() const -> Val&;

        auto operator==(default_sentinel_t) const -> bool;
    };


public:
    auto begin() -> iterator { return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```



```
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
    bool dirty = true;

    auto prime() -> void {
        if (dirty) {
            *stream >> value;
            dirty = false;
        }
    }

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;

        auto operator*() const -> Val& { parent->prime(); return parent->value; }

        auto operator==(default_sentinel_t) const -> bool;
    };


public:
    auto begin() -> iterator { return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

Where do we extract the next value?

Yes.

```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
    bool dirty = true;

    auto prime() -> void {
        if (dirty) {
            *stream >> value;
            dirty = false;
        }
    }

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator&;

        auto operator*() const -> Val& { parent->prime(); return parent->value; }

        auto operator==(default_sentinel_t) const -> bool {
            parent->prime();
            return not *parent->stream;
        }
    };

public:
    auto begin() -> iterator { return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

Where do we extract the next value?

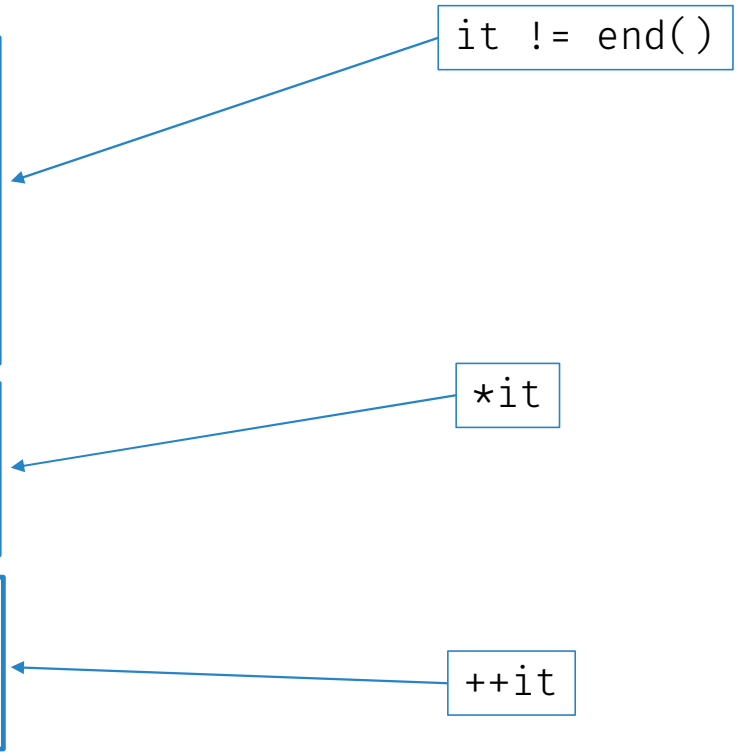
Yes.

```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;
    bool dirty = true;

    auto prime() -> void {
        if (dirty) {
            *stream >> value;
            dirty = false;
        }
    }

    struct iterator {
        istream_view* parent;
        auto operator++() -> iterator& {
            parent->prime();
            parent->dirty = true;
            return *this;
        }

        auto operator*() const -> Val& { parent->prime(); return parent->value; }

        auto operator==(default_sentinel_t) const -> bool {
            parent->prime();
            return not *parent->stream;
        }
    };

public:
    auto begin() -> iterator { return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

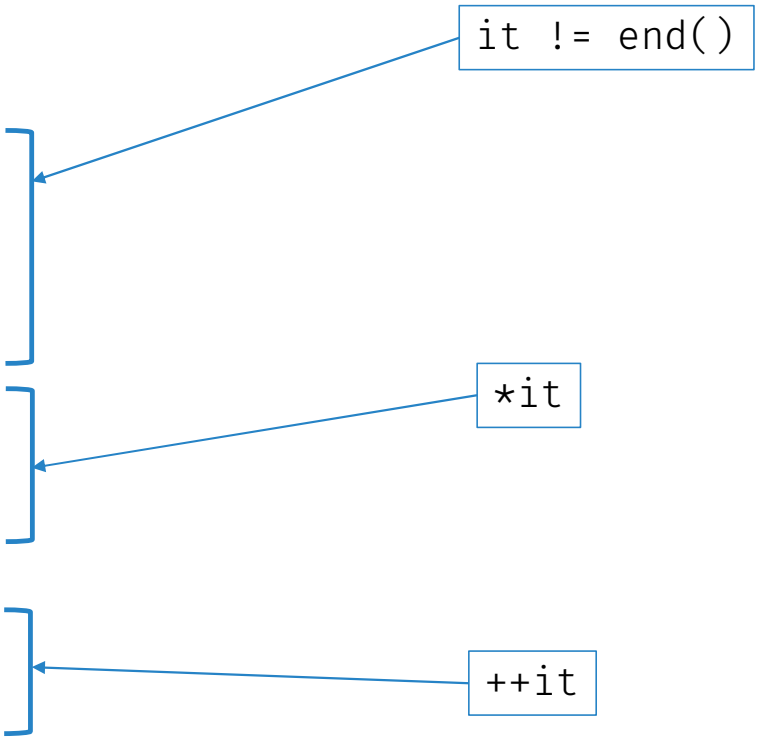
```cpp
template <class Val>
class istream_view : public ranges::view_interface<istream_view<Val>> {
    istream* stream;
    Val value;

    struct iterator {
        istream_view* parent;
        mutable bool dirty = true;
        auto prime() const -> void {
            if (dirty) {
                *parent->stream >> parent->value;
                dirty = false;
            }
        }

        auto operator++() -> iterator& {
            prime();
            dirty = true;
            return *this;
        }

        auto operator*() const -> Val& { prime(); return parent->value; }

        auto operator==(default_sentinel_t) const -> bool {
            prime();
            return not *parent->stream;
        }
    };

public:
    auto begin() -> iterator { return iterator{this}; }
    auto end() -> default_sentinel_t { return default_sentinel; }
};
```

```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        int count = 5;
        while (count != 0 and input >> __storage) {
            int i = __storage;
            print("loop: {}\n", i);
            --count;
        }
    }
}
```

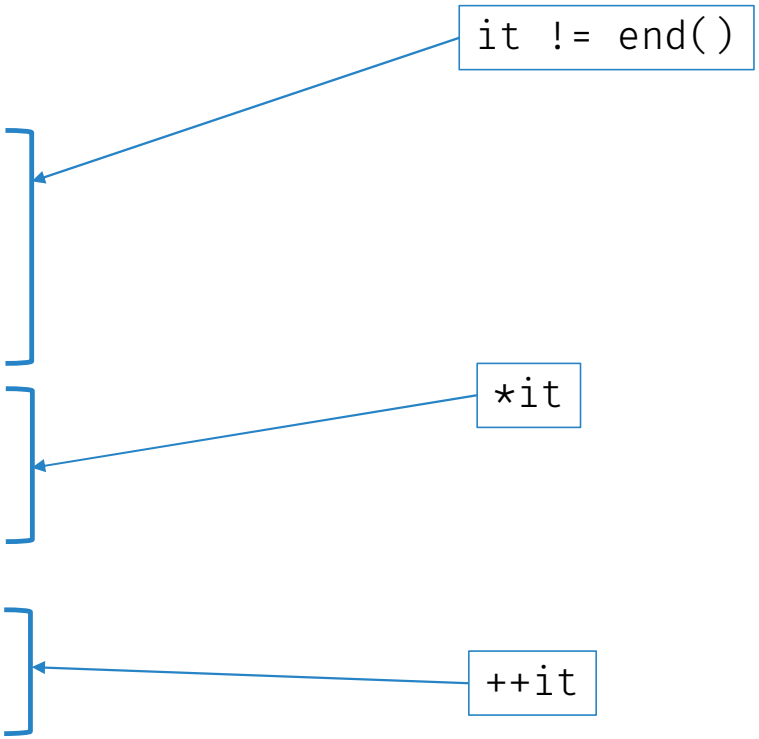
```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        int count = 5;
        bool dirty = true;
        for (;;) {
            if (count == 0) {
                break;
            }
            if (dirty) {
                input >> __storage;
                dirty = false;
            }
            if (not input) {
                break;
            }
            if (dirty) {
                input >> __storage;
                dirty = false;
            }
            int i = __storage;
            print("loop: {}\n", i);
            --count;
            if (dirty) {
                input >> __storage;
            }
            dirty = true;
        }
    }
}
```



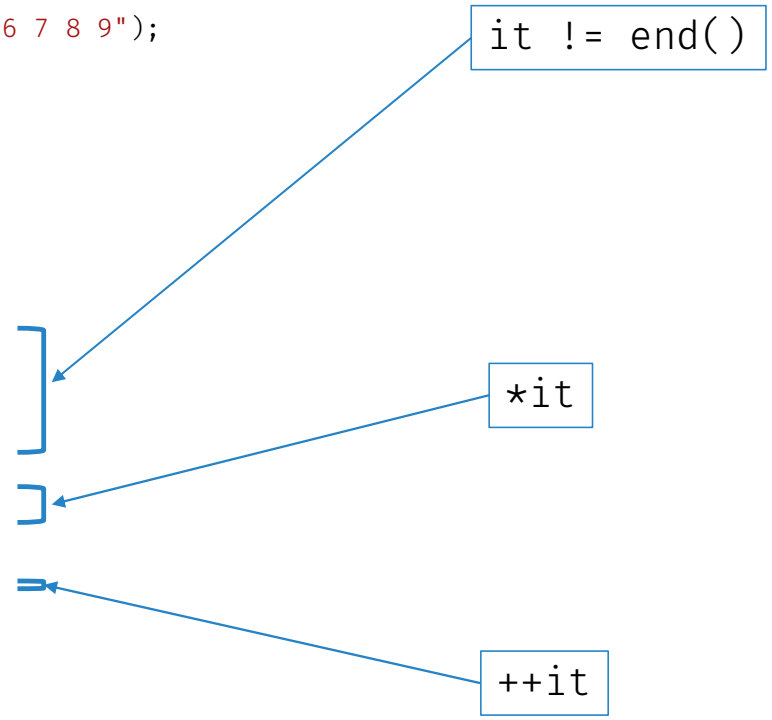
```cpp
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        int count = 5;
        bool dirty = true;
        for (;;) {
            if (count == 0) {
                break;
            }
            if (dirty) {
                input >> __storage;
                dirty = false;
            }
            if (not input) {
                break;
            }

            if (dirty) {
                input >> __storage;
                dirty = false;
            }
            int i = __storage;
            print("loop: {}\n", i);
            --count;
            if (dirty) {
                input >> __storage;
            }
            dirty = true;
        }
    }
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int __storage;
        int count = 5;

        for (;;) {
            if (count == 0) {
                break;
            }
                        {
                input >> __storage;

            }
            if (not input) {
                break;
            }



            int i = __storage;
            print("loop: {}\n", i);
            --count;



        }
    }
}
```



```
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    {
        int   storage;
        int count = 5;

        for (;;) {
            if (count == 0) {
                break;
            }

            input >> __storage;
            if (not input) {
                break;
            }

            int i = __storage;
            print("loop: {}\n", i);
            --count;
        }
    }
}
```



```cpp
auto main() -> int {
    istringstream input("1 2 3 4 5 6 7 8 9");
    for (int i : views::istream2<int>(input)
                | views::take(5)) {
        print("loop: {}\n", i);
    }

    int next;
    if (input >> next) {
        print("next: {}\n", next);
    }
}
```

```
loop: 1
loop: 2
loop: 3
loop: 4
loop: 5
next: 6
```

Additional Reading

P2406

Fix `counted_iterator` interaction with input iterators
Add `lazy_counted_iterator`

Yehezkel and Yehuda Bernat

P2799

Closed ranges may be a problem;
Breaking `counted_iterator` is not the solution

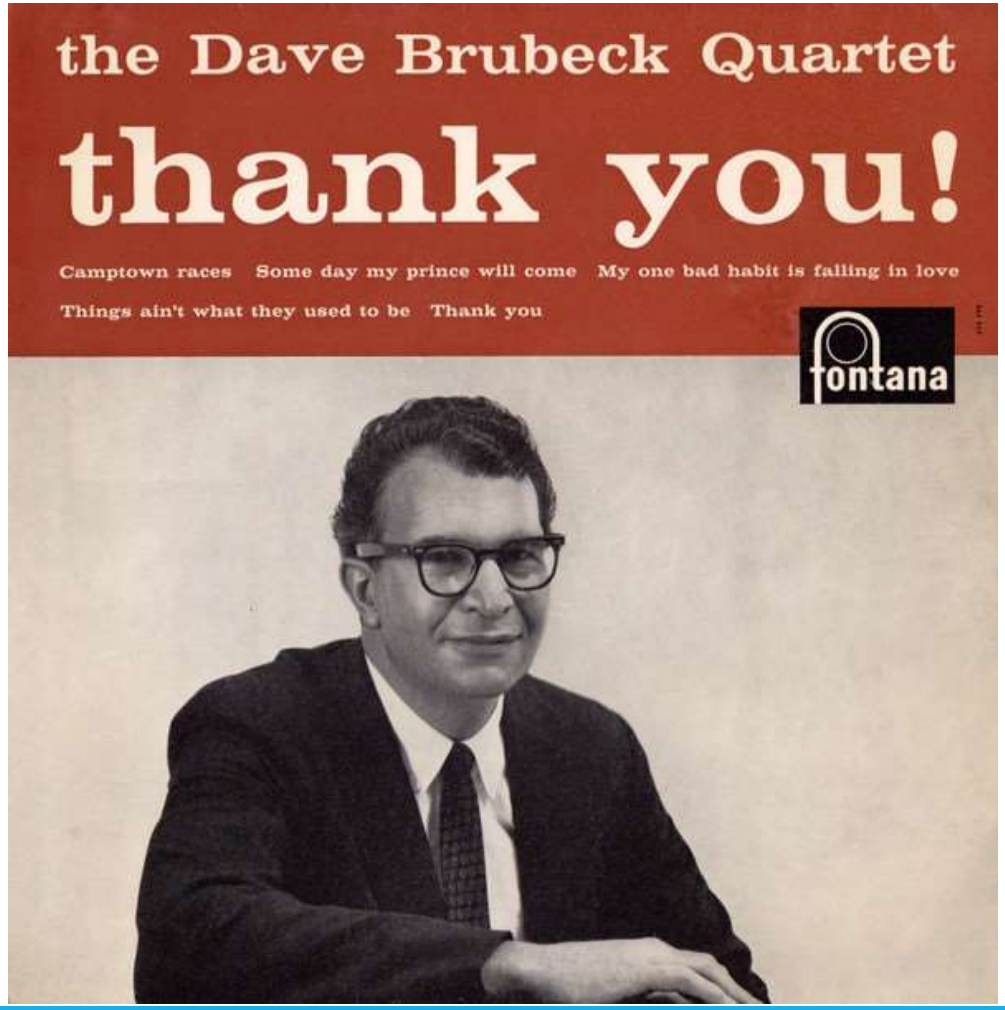
Tim Song

Bonus Example

```
auto main() -> int {
    for (int i : views::iota(0)
               | views::filter([](int i){ return i < 5; })
               | views::take(5)) {
        print("{}\n", i);
    }
}
```

```
auto main() -> int {
    for (int i : views::iota(0)
                | views::filter([](int i){ return i < 5; })
                | views::take(5)) {
        print("{}\n", i);
    }
}
```

```
auto main() -> int {
    for (int i : views::iota(0)
                | views::filter([](int i){ return i < 5; })
                | views::closed_take(5)) {
        print("{}\n", i);
    }
}
```


the Dave Brubeck Quartet
thank you!
Camptown races Some day my prince will come My one bad habit is falling in love
Things ain't what they used to be Thank you
fontana