A view to a view

Peter Bindels / C++Now 2018

@dascandy42 / dascandy@slack

TOMTOM

Peter Bindels

@dascandy42

dascandy@Cpplang slack

Author of HippoMocks (C++Now 2017)

Co-author of Cpp-Dependencies

(MeetingC++ 2016/2017)

https://github.com/dascandy/presentations

-~- Making the world of C++ simpler -~-

A view to a view

- What is this presentation (not) ?
- Terminology
- Why views?
- What kinds of views?
- Thinking with views
- When to use views?
- Making a view

What is this presentation?

- This talk is about the general concept of views and their benefits/downsides
- Usable now
  - and 20 years ago, if you're a time traveller.
- Please try to avoid creating paradoxes…
- Usable if you cannot use Ranges-v3
  - Windows users

What is this presentation?

- Practice focused
- All examples are from actual use
  - Not all actually in production, but intended for it

What is this presentation not?

- It is not ranges-v3
  - See Eric Niebler’s talk from CppCon 2015

- I will try to keep as close as possible terminology-wise

A view to a view

- What is this presentation (not) ?
- **Terminology**
- Why views?
- What kinds of views?
- Thinking with views
- When to use views?
- Making a view

Terminology

- Input iterator

- Read
- Increment (without multiple passes)

Terminology

- Forward iterator

- Read
- Increment (without multiple passes)
- Increment (with multiple passes)

Terminology

- Bidirectional iterator

- Read
- Increment (without multiple passes)
- Increment (with multiple passes)
- Decrement

Terminology

- Random access iterator

- Read
- Increment (without multiple passes)
- Increment (with multiple passes)
- Decrement
- Random access

Terminology

- Contiguous iterator

- Read
- Increment (without multiple passes)
- Increment (with multiple passes)
- Decrement
- Random access
- Contiguous storage

Terminology

- Note that the iterator categories map to range categories
  - If your range returns an input iterator, it’s an input range
  - If your range returns a forward iterator, it’s a forward range

Terminology

- Range<T>
  - The concept of **multiple** T's, demarcated by begin/end
- Ref<T>
  - Non-owning type referring to a **singular** T
- View<T>
  - Non-owning type representing a lazy operation resulting in a range of T
  - Underlying input may be anything

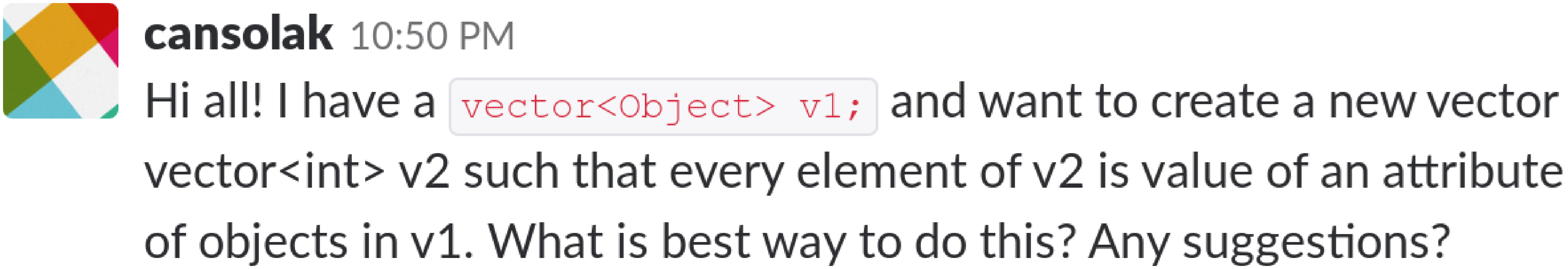
Terminology

- Container<T>
  - Owning type referring to a range of T
- Action<T>
  - Non-owning non-lazy operation resulting in a range of T
  - Actively outputs, so needs target storage

A view to a view

- What is this presentation (not) ?
- Terminology
- **Why views?**
- What kinds of views?
- Thinking with views
- When to use views?
- Making a view

Why views?



cansolak 10:50 PM

Hi all! I have a `vector<Object> v1;` and want to create a new vector vector<int> v2 such that every element of v2 is value of an attribute of objects in v1. What is best way to do this? Any suggestions?

Why views?

- Unicode transcoding between encodings
  - NxN problem

  - Or Nx1 + 1xN problem

Why views?

- Parsing a file
  - All inputs already were allocated in memory
  - Save a ton of copying
  - You can refer back to the inputs for file line numbers and offsets directly

Why views?

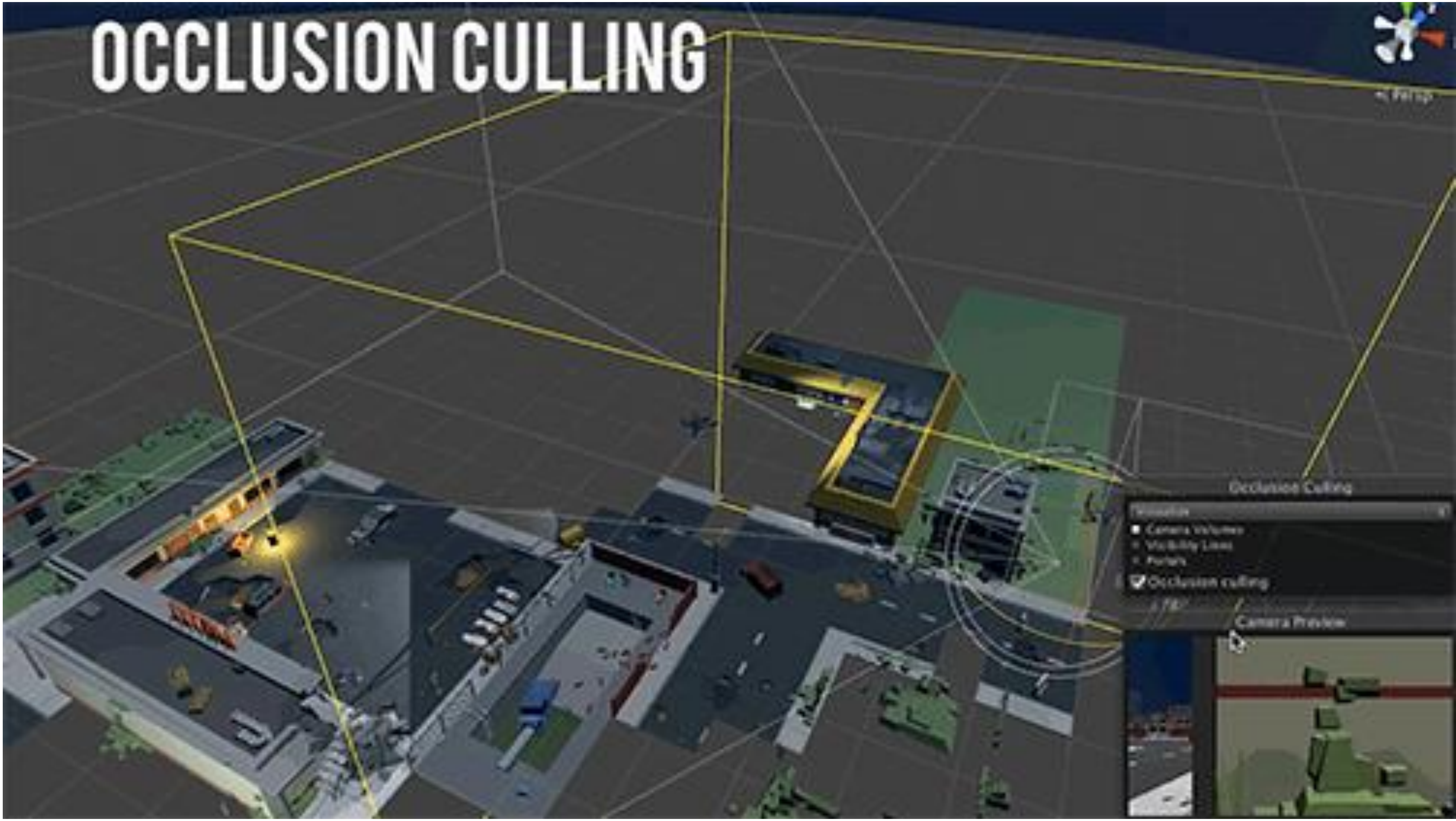
- Maximize the amount of work not done
- Much more readable code

- Lifetime and ownership

Maximize the amount of work not done

- In games, if you don't see part of the world, it's not even loaded
- If you can avoid sending something to the GPU, it's avoided where reasonably possible
- If you do render it, and know that it'll be obscured, render the obscurer first so you will render less pixels total

Maximize the amount of work not done



Maximize the amount of work not done

- Total amount of work is reduced to the minimum.
  - Sometimes by doing more work, to avoid doing heavy work elsewhere
  - Sometimes by just not doing something at all
- Minecraft forest fires pause if you run away
- You don’t need food if you log out

Maximize the amount of work not done

- If you're only using half of the output, you can get away with not transforming half of your input
- You can stack multiple views together and only pay for the copy operation once

Maximize the amount of work not done

- You can avoid heavy allocations for an intermediate result if your eventual result is (much) smaller
  - Unicode transcoding
  - You can use streaming outputs in some cases as well, for example in file transformations and network operations, without storing the whole file/network stream at any point

Maximize the amount of work not done

```
s2::basic_string_view<s2::encoding::cp437> string437
  ("Victor jagt zw\x94lf Boxk\x84mpfer quer \x81"
   "ber den gro\xE1" "en Sylter Deich");

s2::basic_string_view<s2::encoding::utf8> ustring
  ("Victor jagt zwölf Boxkämpfer quer ü"
   "ber den großen Sylter Deich");

REQUIRE(string437 == ustring);

s2::basic_string<s2::encoding::utf16>
  u16string = ustring;
```

Maximize the amount of work not done

```
s2::basic_string_view<s2::encoding::cp437> string437
  ("Victor jagt zw\x94lf Boxk\x84mpfer quer \x81"
   "ber den gro\xE1" "en Sylter Deich");

s2::basic_string_view<s2::encoding::utf8> ustring
  ("Victor jagt zwölf Boxkämpfer quer ü"
   "ber den großen Sylter Deich");

REQUIRE(string437 == ustring); <-- No conversion needed

s2::basic_string<s2::encoding::utf16>
  u16string = ustring; <-- No UTF32 code points stored
```

Much more readable code

- Raise abstraction level
  - Express what you do, rather than how
  - Operations are what you want to do, not copy or move storage
- Do complicated operations with a stack of views
  - Much less data stored & copied
  - In some contexts, avoid storing the whole data streaming contexts; you can avoid storing the whole stream at any point ever.

Lifetime and ownership

- The major power of C++ is having controlled lifetime and ownership.
  - Views subvert this by not owning their contents.
  - Main risk is dangling references

Lifetime and ownership

```
std::string_view GetEntry(int index) {
  auto it = list.find(index);
  if (it != list.end()) {
    return it->second;
  }
  return “No such entry”s;
}
```

Lifetime and ownership

```
std::string_view GetEntry(int index) {
  auto it = list.find(index);
  if (it != list.end()) {
    return it->second;
  }
  return “No such entry”s;
}
```

Lifetime and ownership

- Design principle: Design your API such that you making it hard to create dangling views
  - You can’t prevent it

Views and iterator types

- A view shouldn't raise the iterator type for its underlying range
  - This would imply caching the full resulting sequence
  - Then it’s not a view

Views and iterator types

- Often, a view will have to lower the iterator type to a simpler one
  - Only basic views are consecutive
  - All others are at best bidirectional
  - Maybe going back is prohibitively expensive or impossible

Views and iterator types

- It can be beneficial to make the view cache a subsection
  - A tar view, converting a byte stream into a range of files
- themselves again byte streams
  - Each file itself can be fully cached, so a consecutive range
- Simpler to pass along
- Requires a full cache of that file

A view to a view

- What is this presentation (not) ?
- Terminology
- Why views?
- **What kinds of views?**
- Thinking with views
- When to use views?
- Making a view

What kinds of views are there?

- Basic views
- Generators
- Rope
- Transform
- Filter
- Zip

Basic view types

- Non-owning type referring to a range

- std::string_view (C++17)
- std::span (C++20)

Generators

- Mostly similar to functional programming languages
- Defines a (possibly infinite) range from limited inputs

- In my opinion mostly funky to show, but not actually all that useful in production code.

Rope

- Logical concatenation of multiple ranges
  - Take multiple ranges satisfying the same concept
  - Represents a single range with the same concept
  - Logically contains the sequence of its inner ranges

Rope - Parsing code

```
#pragma once

int f();

#include “b.h”

int g();

#ifdef _WIN32
#include “windows.h”
#endif

class A {};
```

```
#pragma once

class B {
public:
  virtual ~B() {}
};
```

Rope - Parsing code

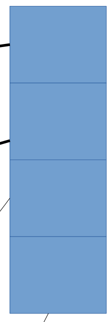
```
#pragma once

int f();

#include “b.h”

int g();

#ifdef _WIN32
#include “windows.h”
#endif

class A {};
```

```
#pragma once

class B {
public:
  virtual ~B() {}
};
```

Rope - Parsing code

```
#pragma once

int f();

#include “b.h”

int g();

#ifdef _WIN32
#include “windows.h”
#endif

class A {};
```

```
#pragma once

class B {
public:
  virtual ~B() {}
};
```



Rope

- This is not a new trick to do
  - PCI Scatter-gather buffers

Rope

- Can be expression template-like construct
  - https://github.com/dascandy/s2
- Can be runtime list/tree of segments
  - https://github.com/tzlaine/text
- Both implement the generic concept of having a second-order collection of things, being viewed as a first-order thing.

Transform

- Original input after performing a smaller or larger transformation.
  - Converting a span to an iteration of files viewed as a TAR or AR file
  - Converting UTF8 data to a UTF32 view
  - Converting a compressed input stream into an uncompressed stream

Transform

- Escaping or unescaping an input string
- Taking only the keys, or values, from a map
- to_string from a date, integer or such
- String split

Filter

- Range containing a subset of its input
  - skipping duplicate values
  - taking only numbers that are prime
  - taking only the talks you’ve selected
  - taking only those entries that are currently enabled (by some ruleset)

Zip

- Multiple ranges pairwise (tuplewise?) taken together to form a new range

- Theoretically important concept
- I haven’t actually needed it / used it

What's so hard about views to views?

- Pre-C++17 `is_same(decltype(begin(x)), decltype(end(x)))` in a range-based for loop
- This means that an N-th order view’s iterators will be 2x the size of a N-1th order view’s iterator
- A 5-high stack of views-to-views-to-views has at least 97% memory wasted to this

What's so hard about views to views?

- C++17 allows `end(x)` in a range-based for loop to be a different type
- This makes views-to-views much smaller
  - Make the end type a one-byte sentinel
- There is a pre-C++17 “workaround” with caveats

What's so hard about views to views?

- Who actually owns what?
- Can you store a view as itself?
  - Do you want to?

A view to a view

- What is this presentation (not) ?
- Terminology
- Why views?
- What kinds of views?
- **Thinking with views**
- When to use views?
- Making a view

Thinking with views

- For any given operation
  - Can you represent the desired output as an iteratable conversion of the input?
  - Does the operation require amortized O(1) work to increment the view iterator?
- Then a view is a great idea

Thinking with views

- Creating a view
  - Create iterator state that maps your output position to the input domain
  - Allow constructing a view from a valid input range
  - Given an output position, extract the value for it from the iterator state

Thinking with views

- Many of these views are conceptually other things
- This foreshadows the need for concepts, as this is a "String-like object" that no string / text designer could anticipate
- You can implement concepts (pretty much), see ranges-v3
- Not implementing it loses type safety

A view to a view

- What is this presentation (not) ?
- Terminology
- Why views?
- What kinds of views?
- Thinking with views
- **When to use views?**
- Making a view

When to use views

- Often!
  - Culling in graphics
  - Parsing inputs / files
  - Destructuring
  - Lazy conversions from one type or representation into another
- Makes your code much easier to read

When to use views

- But only if you know lifetimes will be good
  - All data must be owned by **something**

A view to a view

- What is this presentation (not) ?
- Terminology
- Why views?
- What kinds of views?
- Thinking with views
- When to use views?
- **Making a view**

How to make a simple view

- This is **not** a ranges-v3 view. It’s “just a class” that happens to do something really similar.
- Major benefit is simplicity in making new views
- Does not interact with ranges-v3 style views directly
- Think of it as a “gateway” to using views

How to make a simple view

- For new people, these are the “nothing up my sleeve” views
  - There’s no library code to hide behind
  - There’s no magic happening
  - They’re short and simple enough that we can do three

How to make a simple view

- Keys
- Int-as-a-string_view
- Basic lambda filter

How to make a simple view

```
template <typename Container>
struct keys_view {

    keys_view(Container& c)
        : it_(std::begin(c))
        , end_(std::end(c))
    {}

    decltype(std::begin(::std::declval<Container&>())) it_;
    decltype(std::end(::std::declval<Container&>())) end_;
...
```

How to make a simple view

```
template <typename Container>
struct keys_view {
...
    struct sentinel {};
    keys_view& begin() {
        return *this;
    }
    sentinel end() {
        return sentinel();
    }
...
```

How to make a simple view

```
template <typename Container>
struct keys_view {
...
    keys_view<Container>& operator++() {
        ++it_;
        return *this;
    }
    auto &operator*() {
        return it_->first;
    }
    bool operator!=(const sentinel&) {
        return it_ != end_;
    }
    bool operator==(const sentinel&) {
        return it_ == end_;
    }
};
```

How to make a simple view

```
std::map<std::string, int> numbers;
for (auto& key : keys(numbers)) {
    std::cout << key << “\n”;
}
```

How to make a simple view

```
std::map<std::string, int> numbers;
for (auto& key : keys(numbers)) {
    std::cout << key << “\n”;
}

std::string hi = “Hello C++Now 2018 attendees!”;
for (const auto& str : split(hi, ‘ ‘)) {
    std::cout << str << “\n”;
}
```

How to make a simple view

```
struct int_view {
    int_view(int n)
        : n(n)
        , index(0)
    {
        if (n >= 0) ++(*this); // skip minus sign space
    }

    int n, index;
...
```

How to make a simple view

```
struct int_view {
...
    struct sentinel {};
    int_view& begin() {
        return *this;
    }
    sentinel end() {
        return sentinel();
    }
    bool operator!=(const sentinel&) {
        return index != 11;
    }
    bool operator==(const sentinel&) {
        return index == 11;
    }
...
```

How to make a simple view

```
struct int_view {
...
    int_view& operator++() {
        ++index;
        if (index == 1) {
            while (**this == ‘0’ && index < 10) index++;
        }
        return *this;
    }
    auto &operator*() {
        if (index == 0) return (n < 0) ? ‘-’ : ‘+’;
        int tmp = n;
        for (int i = index; i < 10; i++) tmp /= 10;
        return (tmp % 10) + ‘0’;
    }
};
```

How to make a simple view

```
int_view i(42195);

std::string s(i.begin(), i.end());

// s is now “42195”
```

How to make a simple view

```
template <typename Container, typename Pred>
struct filter_view {
    struct sentinel {};
    filter_view& begin() {
        return *this;
    }
    sentinel end() {
        return sentinel();
    }
    decltype(std::begin(::std::declval<Container&>())) it_;
    decltype(std::end(::std::declval<Container&>())) end_;
    auto &operator*() {
        return *it_;
    }
    bool operator!=(const sentinel&) {
        return it_ != end_;
    }
    bool operator==(const sentinel&) {
        return it_ == end_;
    }
...
```

How to make a simple view

```
template <typename Container, typename Pred>
struct filter_view {
...
    filter_view<Container, Pred>& operator++() {
        do {
            ++it_;
        } while (it_ != end_ && !pred_(*it_));
        return *this;
    }
    filter_view(Container& c, Pred&& p)
        : it_(std::begin(c))
        , end_(std::end(c))
        , pred_(std::move(p))
    {
        while (it_ != end_ && !pred_(*it_)) ++it_;
    }
};
```

How to make a simple view

```
for (auto& key : filter(keys(myMap),
          [](auto& k) { return k.hash < 42; }
   )) {

    std::cout << key << “\n”;

}
```

How to make a simple view

- You **can** cheat
  - Make your sentinel the same object as your iterator
  - Make your comparison pretend to always compare to the end iterator
  - Works in C++11 range-for, no size overhead

How to make a simple view

- You **can** cheat
  - Make your iterator the same as your range
  - Avoids a copy
  - If somebody copies the iterator or the range, it will still work
  - Easy to undo – remove the `&` on the return type of begin
- Alternatively, split off the iterator logic

How to make a simple view

- This is risky
  - If anybody tries to treat it as a forward iterator it'll fail horribly and be very hard to debug

Questions?

References

- https://github.com/dascandy/view
  - General view types that are easy to understand
- https://github.com/dascandy/s2
  - Std2 playground, currently with a string that uses views & ropes
- https://github.com/dascandy/compiler
  - C++ lexer that lexes as a pure view. All tokens are string_views into the input

References

- https://github.com/tzlaine/text
  - Zach Laine’s full-fledged Unicode text library. Uses views where possible.
- https://github.com/ericniebler/range-v3
  - Eric Niebler’s Range-v3 library. It’s the mathematically-complete counterpart to this bare-bones view style.

References

- https://cpplang.slack.com/
  - Join at https://cpplang.now.sh/
  - #learn
  - #cppnow
  - #plug_worthy
  - #speakerscorner