Implementing Ranges and Views

Roi Barkan

Core C++ 2024

Slides

Hi, I'm Roi

- Roi Barkan (he/him) - רועי ברקן
- I live in Tel Aviv 🎗
- C++ developer since 2000
- SVP Development & Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com
- Always happy to learn and explore
  - Please - ask questions, make comments

Slides

Istra Research

Outline

- Ranges and Views - Brief Intro
  - What are they
  - What’s cool about them
  - Views we currently have
- Implementation Details - Several Perspectives
  - Object ⇌ Algorithm ⇌ Data
  - Concepts and Selection/Constraints
  - Lazy ⇌ Eager
- Case Study

Ranges and Composition

Ranges is a Breakthrough Library

- One of C++20 big-four features
- Rests on decades of existing libraries and experience
  - C++98 iterator-based algorithms
  - Fundamentals of functional / vectoric languages (APL, BQN, R, Julia, NumPy) Conor Hoekstra
  - Libraries of similar languages (D, Rust, Java) Barry Revzin, Alexandrescu BoostCon 2009.
- Main Innovation - Composability
  - Many algorithms take ranges as input and return ranges as output
    - Opposed to in-place or output-iterator nature of C++98 algorithms
  - Range Adaptors - algorithms encalsupated as ‘lazy ranges’ (views)
    - Algorithms as composable objects - ‘expression templates’
  - Projections - unary transformations of the ranges we inspect.

Terminology

- Range - Abstraction for a sequence of elements
  - begin-iterator and end-sentinel
- Range Algorithm - Function operating on ranges
  - Evolved from C++98 iterator based algorithms
  - Input: one or more ranges; potentially more arguments
  - Output: anything. If range: either in-place or via “output-iterator” or a subrange
- View - Ranges that are “cheap” to pass/hold
  - constant-time move, if-copyable-then-const-time (semantic nature → `enable_view<Rng>`)
- Range-Adaptor - range-to-range manipulations
  - Most adaptors are views and reside in `std::ranges::views`
  - View adaptors in the STL are ‘lazy’.
  - Adaptors are meant for chaining. The cheapness of views eases chain creation

Composability of Ranges

- Chaining algorithms due to range arguments and results
  ```
  ranges::reverse(ranges::search(str,"abc"sv));
  ```
  godbolt
- Views as composable lazy ranges
  ```
  str | views::split(' ') | views::take(2);
  ```
  godbolt
- Views have a value/algorithm duality
  ```
  auto square_evens =
        views::filter([](auto x) { return int(x) % 2 == 0; }) |
        views::transform([](auto x) { return x * x; });
  ```
  godbolt
- Simple combinations can enrich our vocabulary:
  ```
  auto histogram =
     views::chunk_by(std::equals{}) |
     views::transform([](const auto& rng) {
        return make_pair(begin(rng), distance(rng));};
  ```

The Views in the Standard (C++20/C++23*/C++26**)

- Factories/Generators: **empty**, **single**, **iota**, **repeat***, (**std::generator***)
- Rank preserving: **all**, **filter**, **transform**, **take{_while}**, **drop{_while}**, **reverse**, **stride***, **adjacent_transform***, (**counted**)
- Rank preserving - variadic→tuples: **zip***, **cartesian_product***
- Rank decreasing - tuples: **elements**, **keys**, **values**
- Rank decreasing - variadic: **zip_transform***, **concat****
- Rank decreasing - ranges: **join{_with*}**
- Rank increasing - tuples: **enumerate***, **adjacent***
- Rank increasing - ranges: **{lazy_}split**, **slide***, **chunk{_by}***
- Committee plan for C++26 is in [P2760]

[Details]

Adaptor Chain Fundamentals

Creating Composition Chains

- Adaptors support nesting as well as pipeline/infix composition
  - `views::take(views::split(str, ' '), 2)`

    equivalent to

    `str | views::split(' ') | views::take(2)` godbolt
- *RangeAdaptorClosure*: chains without a starting range
  - Objects that exist to be chained to some range
  - Semantically they are generic algorithms, not ranges
  - `std::ranges::range_adaptor_closure` is a CRTP helper for creating adaptors that have this nesting ⇌ pipeline duality.

Simplest Range Adaptor

```
struct First : range_adaptor_closure<First> {
    constexpr auto operator()(forward_range auto&& rng) const {
        return subrange(begin(rng), empty(rng) ? begin(rng) : next(begin(rng)));
    }
};
constexpr First first;

int main() {
    string s = "aa bb cc";
    auto x = s | split(' ');
    println("{}", x | first);
    return 0;
}
```

Reference may dangle

```
Program returned: 0
  [['a', 'a']]
```

godbolt

Dealing with Dangling

- Chains involve creation (and destruction) of temporary objects
- Solution - aggregate the chain into “expression templates”:
    - `typeid("x"s | split(' ') | take(3))` ≈ **`take_view<split_view<string>>`**
- Adaptors themselves are typically small and cheap to pass as the chain grows
- Ranges can be expensive to pass → hence we use Views.

Digression: Best Implementation of `first`

```
namespace stdv = std::views;

constexpr auto first = stdv::take(1);

int main() {
    string s = "aa bb cc";
    auto x = s | split(' ') | first;
    println("{}", x);
    return 0;
}
```

Power of Composition

godbolt

Simplest Range Adaptor + View

```
template <view Inner> requires forward_range<Inner>
class FirstItemView : public view_interface<FirstItemView<Inner>> {
    [[no_unique_address]] Inner inner;
  public:
    constexpr FirstItemView(Inner inner_) : inner(std::move(inner_)) {}
    constexpr auto begin() { return std::ranges::begin(inner); }
    constexpr auto end() { return empty(inner) ? begin() : next(begin()); }
    constexpr std::size_t size() { return empty(inner) ? 0 : 1; }
};
template <forward_range Range>
FirstItemView(Range&&) -> FirstItemView<views::all_t<Range>>;
struct First : range_adaptor_closure<First> {
    template<forward_range Rng>
    constexpr auto operator()(Rng&& rng) const { return forward<Rng>(FirstItemView{rng});}
};
```

godbolt

Details About Views

- **`view_interface`** - helper CRTP which opts-in to the **`view`** concept
- Constructor - pass inner view by-value, **`std::move()`** inside
- **`begin()/end()`** - must be implemented.
  - **`const`** correctness is tricky (see [Nico Josuttis]())
- **`size()`** - constant-time, opt-in as a **`sized_range`**.
  - **`view_interface`** provides **`size()`** if **`{end() - begin();}`** is valid.
- Deduction guide - use **`views::all_t`** to allow non-view inputs
  - more about all_t in the next slide
- Range adaptor closure - simply return the view.
  - Some adaptors can have optimizations here, e.g. **`reverse | reverse`**.

Lifetime Management with `views::all`

- Chains of adaptors need to outlive their base range (otherwise UB).
- STL uses value categories (lvalue vs. rvalue) to try and avoid such cases
  - **`ref_view`** - A view that points to another range (reference semantics), and cannot be constructed if the range is rvalue (about to go away)
  - **`owning_view`** - A view that *takes ownership* of another range (moves it inside the view), and can be constructed only from rvalues. Move-only semantics (like **`unique_ptr`**).
- **`views::all(rng)`** will return one of 3 different types of views:
  - If **`rng`** is a view - simply return it
  - else-if **`rng`** is an lvalue - return a **`ref_view`** pointing to it (be careful of lifetimes
  - else return an **`owning_view`** that now owns the contents of the range.
- Range adaptor views in the STL use **`views:all`** to assist them.

Examples - views, all

```
//temporaries create an owning view
static_assert( not view<decltype(string{""}      )>);
static_assert(     view<decltype(string{""}| all)>);
static_assert(is_same_v<decltype(string{""}| all),
                        owning_view<string>.     >);

//lvalues create a ref view
string s = "some string";
static_assert( not view<decltype(s             )>);
static_assert(     view<decltype(s          | all)>);
static_assert(is_same_v<decltype(s          | all),
                        ref_view<string>.        >);
```

godbolt

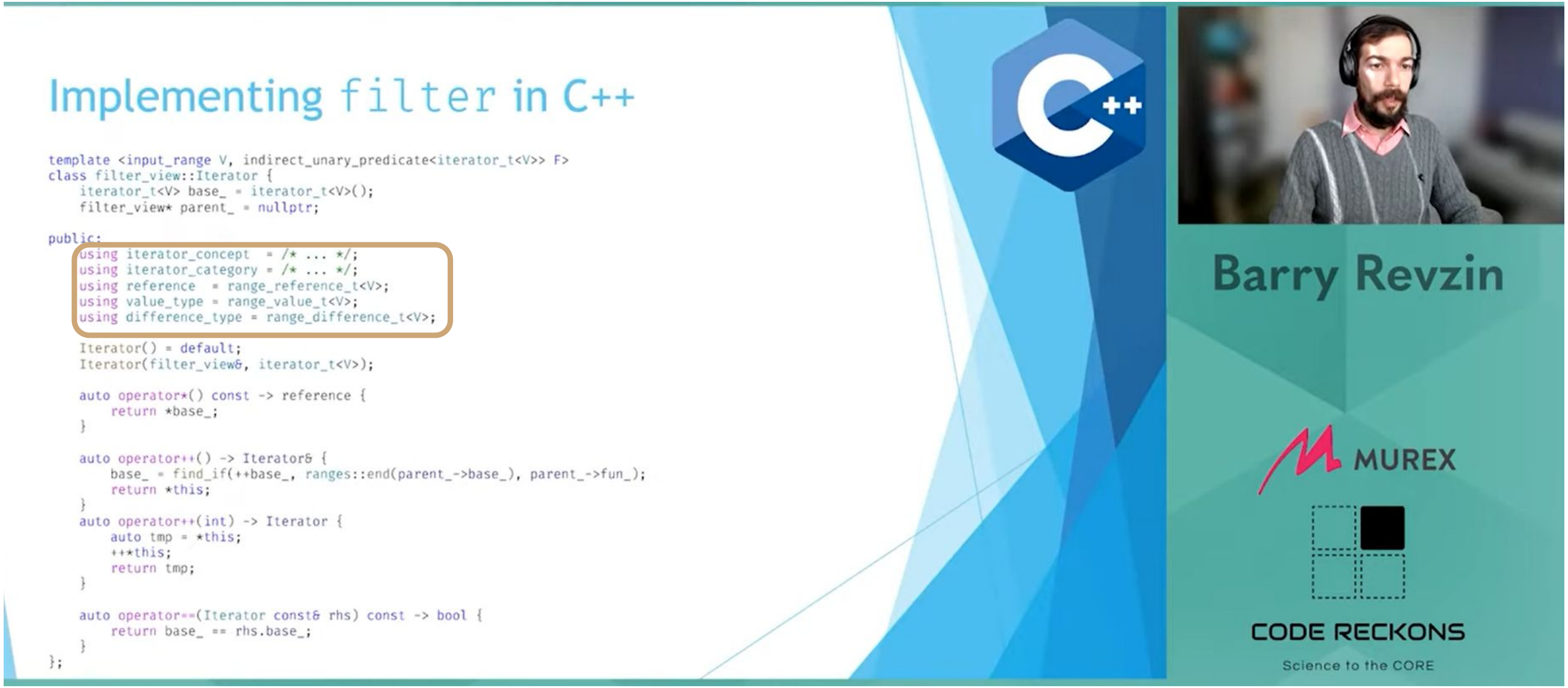
Examples - views, all (2)

```
//views stay views
auto x = s | split(' ');
static_assert(     view<decltype(x               )>);
static_assert(     view<decltype(x          | all)>);
static_assert(is_same_v<decltype(x          | all),
                        decltype(x               )>);
//Careful - all_t<array> can be expensive-to-move
static_assert( not view<decltype(array<int,1000>{}     )>);
static_assert(     view<decltype(array<int,1000>{}| all)>);
static_assert(is_same_v<decltype(array<int,1000>{}| all),
                        owning_view<array<int,1000>>.   >);
static_assert(sizeof(decltype(array<int,1000>{}| all)) >= 4000);
```
godbolt

Range Adaptor Iterators - Being Lazy

- Most views implement their own iterator (and/or sentinel) types, and achieve their functionality through the iterator member functions
  - `transform` - utilizing `operator*()`
  - `filter/stride/reverse` - utilizing `operator++()`
  - `take_while` - utilizing `operator!=(const sentinel&)`
  - `chunk/split` - utilizing `operator*()` and `operator++()`.
- The lazy approach has many benefits
  - Pay only for what you need
  - Better support for potentially infinite ranges
  - More data locality and less need for extra RAM
  - Compiler known expression-templates have potential for performance gains.

See Barry About the Iterators



[CPPP21](#)

[Take(5)](#)

Range Categories and Refinements

- Ranges are categorized by their power of iteration, similar to the C++98 iterator category model
  - output, input → forward → bidirectional → random-access → contiguous
  - Similarly to C++98 category is associated via opt-in of iterator_category tags.
- On top of the power of iteration, ranges have additional *orthogonal* refinements:
  - borrowed - iterators can outlive the range. opt-in **`enable_borrowed_range`**
  - sized - number of elements in amortized constant time. opt-out **`disable_sized_range`**
  - common - **`begin()`** and **`end()`** return the same type
  - constant - range into read-only values.
- Range Adaptors must correctly publish their effect on their input.

Motivation of the Categories- Algorithm Selection

- Sometimes the same goal can be achieved in several ways
  - `ranges::ssize` - returns a signed integer equal to the size of a range
  - `ranges::distance` - returns the distance between the beginning and end of a range
  - `ssize` only works for *sized* ranges (constant-time calculation)
    `distance` allows linear calculation if necessary. Ben Deane recommends it.
- The library uses concepts to constrain which ranges are applicable for which algorithm/view, and to know the best method of reaching the intended goal
- Before C++20 other mechanisms were used to achieve this goal - and with concepts we have a way to be more precise and more flexible where needed.

Digression - How Lazy are We

- Recall histogram. How many passes does it perform over the data

```
auto histogram =
   views::chunk_by(std::equals{}) |
   views::transform([](const auto& rng) {
      return make_pair(begin(rng), distance(rng));}
```

- Intuitively a single pass is enough.
- Depends on if **`range_reference_t<chunk_by_view<...>>`** is sized
  - i.e. depends on if **`subrange<...>`** is sized.
  - Could potentially be controlled via **`subrange_kind`** but not possible in existing adaptors
- Alternative implementation can **`enumerate`** and then **`chunk`** the pairs and **`transform`** the **`subrange`**s with a single pass.

Range/Iterator `const` Correctness

- Remember that iterators have indirect semantics.
- Still, ranges were meant to differentiate between **`iterator`** and **`const_iterator`** for ‘deep’ constness.
- Views are thus allowed to differentiate and have 2 different iterator types.
- C++23 now has **`std::basic_const_iterator`** which can be used as a drop in iterator adaptor.
- Views are notoriously tricky (bad) when it comes to const-correctness
  - Due to caching behavior
  - Due to **`owning_view`** vs. **`ref_view`** being so interchangeable
  - See Nico Josutis.

Iterator Customization Points

- Apart of the basic operators (**`*`**, **`!=`**, **`++`**, **`–`**, **`+=`**, ...), iterators are allowed implement two more functions, which the ranges library must use for their purpose:
- **`iter_move(iterator)`** - instead if **`std::move(*iterator)`**
- **`iter_swap(it1, it2)`** - instead if **`std::ranges::swap(*it1, *it2)`**
- Main motivation: proxy-iterators (e.g. **`zip_view`**)
  - More on that from Jacob Rice.
- Typically implemented as “hidden friends” and invoked via **`std::ranges::iter_{move,swap}`** - which are CPOs

CPO - Customization Point Objects

- Customization points - ways in which a library (ranges) allows its users (specific range-adaptor implementers) to dictate how it behaves in certain cases.
- Before C++20 the STL had “clunky” customization point mechanisms
  - Template specialization (e.g. **`std::hash`**) [unord.hash]
  - Overload resolution and ADL (e.g. **`std::swap`**) [swappable.requirements].
- CPOs are actually objects (global variables) with template **`operator()`** function which knows to perform the correct search for customized implementations (typically via **`if constexpr`** or **`requires`** clauses)
  - More on that from Gašper Ažman.

Case Study

Views for Sorted Ranges ([More Ranges Please](#))

- Suggestion - views for **`merge`**, **`set_union`**, **`set_intersection`**, **`set_{symmetric_}difference`**
  - Most algorithms can benefit from multi-input implementations
  - Heap (**`priority_queue`**) is needed for efficient **`set_union, merge, ....`**
- STL contains several algorithms for sorted ranges: **`{inplace_}merge`**, **`includes`**, **`set_{union,intersection,{symmetric_}difference}`**
  - Also search algorithms: **`{upper,lower}_bound`**, **`equal_range`**, **`(unique)`**.
- All the operations are lazy in nature
- Ranges-v3 has views for **`set_{union,intersection,{symmetric_}difference}`** with 2 input ranges
- D-lang has merge and multiWayMerge.

Implementation Approach

- Every STL algorithm with an output-iterator result can be conceptually converted to a lazy range-adaptor view.
- Basic approach - the unified iterator holds all sub-iterators, an indication of the 'current' one and a pointer to the range.
  - Key idea is that every call to `operator++()` should iteratively increment the lowest sub-iterator until a condition (based on the specific algorithm) is satisfied.
- Various details and opportunities exist for the different algorithms

Set Operation Details

- `begin()` in constant-time
  - Trivial for union, merge. Caching needed for intersection, difference.
- Iterator category
  - input iteration seems enough (single pass)
  - forward/bidirectional iteration can be preserved - bidirectional needs a second heap.
  - random-access on either input can be utilized, mostly for intersection and difference (e.g. lower_bound)
  - random-access cannot be preserved.
- `common_range` can be preserved.
- `sized_range` can be preserved for merge.

Set Operations on Multiple Inputs

- Variadic (compile time) input-count should be simple
  - Potentially use **array<variant<iterator_t<Views>…>, sizeof...(Views)>** with heap operations like **make_heap**, **pop_heap**, **push_heap**.
- Dynamic Range-of-Ranges is more tricky due to potential RAM needs. Potential approaches:
  - Take a random-access container as extra argument.
  - Take a (PMR) allocator as extra argument.
  - Expect the input range (of ranges) to be random-access and use it (like D-lang multiWayMerge)

```
auto carsByPrice =
    carsByMakerThenPrice | chunk_by([](const Car& a, const Car& b) {
                                        return a.maker == b.maker;
    }) | to<vector> |
    merge([](const Car& a, const Car& b) { return a.price < b.price; });
```

Alternative Approach - `std::generator`

- C++23's first library addition utilizing coroutines.
- A **`generator`** exposes a coroutine with **`co_yield`** calls as a **`view`**.
- Main advantage - simplicity:
  - All the intermediate state can be stored in variables
  - Procedural style instead of callback style
  - I don't think one **`generator`** can be implemented for all output-iterator range alrogrithms - "the coloring problem".
- Main disadvantages:
  - Exposes an input_range, not more
  - Performance is compiler/optimizer dependent.

Summary

- The C++ ranges library is an exemplar of composability
- Ranges were developed to be enhanced and extended
- Implementing ranges code requires know-how
  - Not rocket science
- Now it’s our turn

- Thank you !!
  - Questions and comments are welcome

Slides

Extra Slides - All Views

Factories / Generators

```
namespace stdv = std::views;



stdv::empty<char>                       //=> []
stdv::single('+')                       //=> ['+']
stdv::iota(2,5)                         //=> [2, 3, 4]
stdv::repeat(0.3,3)                     //=> [0.3, 0.3, 0.3]
```

godbolt

Rank Preserving - 1/2

```
auto not5 = [](int i){return i != 5;};
auto mult2 = [](int i){return i * 2;};
auto iota2_10 = stdv::iota(2,10);

iota2_10 | stdv::all               //=> [2, 3, 4, 5, 6, 7, 8, 9]
iota2_10 | stdv::filter(not5)      //=> [2, 3, 4, 6, 7, 8, 9]
iota2_10 | stdv::transform(mult2)  //=> [4, 6, 8, 10, 12, 14, 16, 18]
iota2_10 | stdv::take(6)           //=> [2, 3, 4, 5, 6, 7]
iota2_10 | stdv::drop(6)           //=> [8, 9]
```

godbolt

Rank Preserving - 2/2

```
auto not5 = [](int i){return i != 5;};
auto iota2_10 = stdv::iota(2,10);

iota2_10 | stdv::take_while(not5)      //=> [2, 3, 4]
iota2_10 | stdv::drop_while(not5)      //=> [5, 6, 7, 8, 9]
iota2_10 | stdv::reverse               //=> [9, 8, 7, 6, 5, 4, 3, 2]
iota2_10 | stdv::stride(3)             //=> [2, 5, 8]
iota2_10 | stdv::adjacent_transform<2>(std::plus{})
                                       //=> [5, 7, 9, 11, 13, 15, 17]
```

godbolt

Rank Preserving - Variadic ⇒ Tuples

```
auto iota2_7 = stdv::iota(2,7);
auto iota2_4 = stdv::iota(2,4);
auto iota6_9 = stdv::iota(6,9);

stdv::zip(iota2_7, iota6_9)                    //=> [(2, 6), (3, 7), (4, 8)]

stdv::cartesian_product(iota2_4, iota6_9)  //=> [(2, 6), (2, 7), (2, 8),
                                                  (3, 6), (3, 7), (3, 8)]
```

godbolt

Rank Decreasing - Tuples

```
auto the_zip = stdv::zip(iota2_7, iota6_9, "abcdef"sv);
                                   //=> [(2, 6, 'a'), (3, 7, 'b'), (4, 8, 'c')]

the_zip | stdv::keys               //=> [2, 3, 4]
the_zip | stdv::values             //=> [6, 7, 8]
the_zip | stdv::elements<2>        //=> ['a', 'b', 'c']
```

godbolt

Rank Decreasing - Variadic

```
auto iota2_7 = stdv::iota(2,7); auto iota6_9 = stdv::iota(6,9);
stdv::zip(iota2_7, iota6_9)                    //=> [(2, 6), (3, 7), (4, 8)]

stdv::zip_transform(iota2_7, iota6_9, std::multiplies{})
                                               //=> [12, 21, 32]
stdv::concat(iota2_7, iota6_9)                 //=> [2, 3, 4, 5, 6, 6, 7, 8]
```

godbolt

Rank Decreasing - Ranges

```
vector{"hey"sv, "C++"sv} | stdv::join
                                    //=> ['h', 'e', 'y', 'C', '+', '+']
(vector{"hey"sv, "C++"sv} | stdv::join_with(':'))
                                    //=> ['h', 'e', 'y', ':', 'C', '+', '+]
```

godbolt

Rank Increasing - Tuples

```
"hey"sv | stdv::enumerate
                  //=> [(0, 'h'), (1, 'e'), (2, 'y')]
"hello"sv | stdv::adjacent<3>
                  //=> [('h', 'e', 'l'), ('e', 'l', 'l'), ('l, 'l', 'o')]
```

godbolt

Rank Increasing - Ranges

```
"hey C++"sv | stdv::split(' ')  //=> [['h', 'e', 'y'], ['C', '+', '+']]
"hey C++"sv | stdv::lazy_split(' ')
                                 //=> [['h', 'e', 'y'], ['C', '+', '+']]
"hello"sv | stdv::slide(3)       //=> [['h', 'e', 'l'], ['e', 'l', 'l'],
                                       ['l, 'l', 'o']]
"hey C++"sv | stdv::chunk(3)     //=> [['h', 'e', 'y'], [' ', 'C', '+'],
                                       ['+']]
"hello C++"sv | stdv::chunk_by(equal_to{})
 //=> [['h'], ['e'], ['l', 'l'], ['o'], [' '], ['C'], ['+', '+']]
```

godbolt