Implementing Ranges and Views

Roi Barkan
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
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 encapsulated 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
  ```cpp
  ranges::reverse(ranges::search(str, "abc"sv));
  ```
- Views as composable lazy ranges
  ```cpp
  str | views::split(' ') | views::take(2);
  ```
- Views have a value/algorithm duality
  ```cpp
  auto square_evens =
      views::filter([](auto x) { return int(x) % 2 == 0; }) |
      views::transform([](auto x) { return x * x; });
  ```
- Simple combinations can enrich our vocabulary:
  ```cpp
  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*)

- Factories: `empty, single, iota, istream, repeat*
- Rank preserving: `all, filter, transform, take{while}, drop{while}, subrange, counted, common, reverse, stride*, zip_transform*, adjacent_transform*, as_{const, rvalue}*`
- Rank decreasing - tuples: `elements, keys, values`
- Rank decreasing - ranges: `join{with*}`
- Rank increasing - tuples: `zip*, enumerate*, cartesian_product*, adjacent*`
- Rank increasing - ranges: `{lazy_}split, slide*, chunk{by}*`
- Committee plan for C++26 is in P2760
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)
    ```

- `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

```cpp
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;
}
```

Program returned: 0
[[a', 'a']]
Dealing with Dangling

- Chains involve creation (and destruction) of temporary objects
- Solution - aggregate the chain into “expression templates”:
  - \( \text{typeid}"x"s \mid \text{split}' ' \mid \text{take}(3) \approx \text{take_view<split_view<string>>} \)
- Adaptors themselves are typically small and cheap to pass as the chain grows
- Ranges can be expensive to pass → hence we use Views.
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> {
    constexpr auto operator()(forward_range auto&& rng) const { return 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 Josutis)
- `size()` - constant-time, opt-in a a **sized_range**.
- 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 \texttt{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
  - \texttt{ref\_view} - A view that points to another range (reference semantics), and cannot be constructed if the range is rvalue (about to go away)
  - \texttt{owning\_view} - A view that \textit{takes ownership} of another range (moves it inside the view), and can be constructed only from rvalues. Move-only semantics (like \texttt{unique\_ptr}).
- \texttt{views::all(rng)} will return one 3 different types of views:
  - If \texttt{rng} is a view - simply return it
  - else-if \texttt{rng} is a lvalue - return a \texttt{ref\_view} pointing to it (be careful of lifetimes
  - else return an \texttt{owning\_view} that now owns the contents of the range.
- Range adaptor views in the STL use \texttt{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](https://godbolt.org)
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>{} | all)>);
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);
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 has potential for performance gains.
See Barry About the Iterators

Implementing filter in C++

```cpp
#include <input_range V, indirect Unary_predicate<iterator_t<V>> F>
class filter_view<Iterator {
    iterator_t<V> base_ = iterator_t<V>();
    filter_view* parent_ = nullptr;

public:
    using iterator_concept = /* ... */;
    using iterator_category = /* ... */;
    using reference = range_reference_t<V>;
    using value_type = range_value_t<V>;
    using difference_type = range_difference_t<V>;

    Iterator() = default;
    Iterator(filter_view&, iterator_t<V>);  
    auto operator() const -> reference { 
        return *base_; 
    } 
    auto operator++() -> Iterator& { 
        base_ = find_if(++base_, ranges::end(parent_ -> base_), parent_ -> fum_); 
        return *this; 
    } 
    auto operator++(int) -> Iterator { 
        auto tmp = *this; 
        ++this; 
        return tmp; 
    } 
    auto operator==(Iterator const& rhs) const -> bool { 
        return base_ == rhs.base_; 
    }
};
```
Range Categories and Refinements

- Ranges are categorized by their power of iteration, similar to the C++98 iterator category model:
  - 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.
Refinement Motivation - 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
  
  ```cpp
  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 properly chunk/transform the pairs.
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](https://www.nicojosutis.com).
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 of `std::move(*iterator)`
  - `iter_swap(it1, it2)` - instead of `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  

- 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`)

```cpp
    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 algorithms - “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