Ranges

Sonntag, 8. Januar 2023 16:15

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

The similar concept already exists in other languages, or even libraries.
In Java, for instance, we have the ***streams*** - a powerful feature introduced with SDK 8, that allows any collection to be turned into stream, and chain the operation in a way that output of one operation is input into next - subsequent one, until the terminal operator is reached.
This follows functional programming principles in a great deal, having expressive syntax that is *declarative* rather than *imperative* - the implementation details of operators are hidden in the language itself.
Operators themselves are customization points: they expect the custom specific callable (like lambda, or method reference) that will be applied on each item in collection (stream).

```
List<Person> persons = Arrays.asList(...);
persons.stream().filter(person -> person.age() >= 18) // lambda
                .map(Person::getName) // class method reference
                .collect(Collectors.toList()); // terminal operator
```

The entire ***ReactiveX*** library is written fallowing the same principles, as implementation of the *Observer pattern*, with additional benefits:

- Plethora of *factory methods* so that literally everything (not only collections) can be turned into Observable: the one that publishes the items (or events)
- *Lazy by emission*: the Observable start to be active, to publish the items (events), upon the first subscription - "*Cold observables*"
- *Error handling*: exception will be propagated downstream, and will be caught by the subscribers (if such method is provided)
- Publisher and subscriber are not only decoupled by the operators in processing chain, but also subscriber can observe the emission in a different thread context

* More on that at: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/android/RxAndroid.pdf

Ranges - global overview

Ranges are first introduced in C++20, based on the **Eric Niebler's** ranges library.
Meanwhile, there is a new version available at: https://github.com/ericniebler/range-v3

Disclaimer

Ranges simplify use of the std algorithms that work with containers, by replacing the iterators range **[first, last)** with the container itself.
But this is not the main motivation for having the ranges, nor something that you couldn't accomplish on your own, with a few nifty helper functions

```
template <typename Collection>
concept iterable = requires (Collection& collection)
{
    std::begin(collection);
    std::end(collection);
};

template <typename Func, typename Iterator, typename...Args>
auto apply(Func&& func, Iterator first, Iterator last, Args&&...args)
{
    return std::invoke(std::forward<Func>(func),
                       first,
                       last,
                       std::forward<Args>(args)...);
}

/*
* This would replace using [first, last) iterators range with
* container itself, with drawback that the algorithm will be applied
* on the entire container
*/
template <typename Func, typename Collection, typename...Args>
requires iterable<Collection>
auto apply(Func&& func, Collection& collection, Args&&...args)
{
    return details::apply(std::forward<Func>(func),
                          collection.begin(),
                          collection.end(),
                          std::forward<Args>(args)...);
}
```

```
template <typename Func, typename Collection, typename...Args>
requires reverse_iterable<Collection>
auto apply_reverse(Func&& func, Collection& collection, Args&&...args)
{
    return details::apply(std::forward<Func>(func),
                          collection.rbegin(),
                          collection.rend(),
                          std::forward<Args>(args)...);
}

template <typename Collection>
void sort(Collection& collection)
{
    details::apply(std::sort<typename Collection::iterator>, collection);
}

/*
* Sort method, that takes the container as a single argument, and
* sort it in descending order
*/
template <typename Collection>
void sort_reverse(Collection& collection)
{
    details::apply_reverse(std::sort<typename Collection::reverse_iterator>, collection);
}
```

⇒ Full example: https://godbolt.org/z/Yb83bqbcq

The **lack of composability** is to be considered as the major flow in STL algorithms design, that ranges intend to overcome.
Working with iterators, rather than directly with containers, requires auxiliary memory space, for storing algorithms result (or error - exception: *std::expected*), so that it could be passed as input to the next one in processing chain.

Not only that, they ensure:

- template parameter check (*type safety*): ranges defines various **Concepts** that will be applied on template parameters to enforce the constraints that types need to satisfy in order to be a valid candidate in template parameter substitution
- **immutability**: the range adaptors, as predefined operators, that do not modify the input container, but rather create a *View* - a snapshot of the range. The resulting view of a chained operations will create its own: custom traversing rules - *Iterator*, that will apply the composed - resulting operation ***lazy*** - on each element of a View, on demand - at the point when we actually **access the element** (operator*/operator->). No auxiliary memory space is required.

```
// Output container: auxiliary memory space
std::vector<int> out;
out.reserve(in.size());

// filter: take only the even elements from input container
std::copy_if(in.begin(), in.end(), std::back_inserter(out), [](const auto& el){ return (el & 1) == 0;});

// transform: multiply each element with 2
std::transform(out.begin(), out.end(), out.begin(), [](auto& el){ return el * 2; });
```

With ranges, it becomes much more concise - easy to read (from left to right)

```
// transform(filter(in))
auto view = in | std::views::filter([](const auto& el) { return (el & 1) == 0; })
               | std::views::transform([](auto& el) { return el * 2; });
```

- Full example: https://godbolt.org/z/MzjY9qTTd

This is due to Unix-like **"pipe" operator**, that can be generically implemented as

```
template <
          std::ranges::viewable_range Range, // constraint: works only with viewable_range
          std::invocable<Range> Adaptor // constraint: RangeAdaptorObject accept the viewable_range, and convert it to view
         >
constexpr std::ranges::view auto operator | (Range&& range, const Adaptor& adaptor)
{
    return std::invoke(adaptor, std::forward<Range>(range));
}
```

@note The pipe operator works with *in* - std::vector<int>, because the collections are ranges, since they do satisfy the range concept (being *iterable*)

```
template< class T >
concept range = requires( T& t ) {
    ranges::begin(t); // equality-preserving for forward iterators
    ranges::end(t);
};
```

Ranges - details

The mental model of the ranges is **[first, last)** iterator pair.
This is true for the *finite* ranges, where first and last are the same iterator types.
The library provides additionally the way to specify the "*infinite*" ranges **[first, …)** for which the end sentinel
may have a different type than *first* iterator.
This is a special *tag* type: **std::default_sentinel_t**, that enables sentinel to have a valid type and catch the matching *operator==* from overloaded set,
later on.

@note Including the <compare> header file, the compiler will auto-generate the non-equal operators, that are required for using sentinel with rangified algorithms such as *std::ranges::transform*, or *std::ranges::for_each*

⇒ Example: https://godbolt.org/z/ojGs5zGxz

To work with the STL algorithms (like *std::accumulate* in <numeric> header) which require that range iterators are of the same type,
std::ranges::common_range converter should be used for the cases where this requirement is not satisfied - i.e. sentinel is of a different type
(like with *std::views::take_while*)

```
auto out = std::views::iota(1) |
           std::views::take_while([](const auto& el) { return el <= 5;}) |
           std::views::common;

const auto sum = std::accumulate(out.begin(), out.end(), 0);
```

@note The library implementation is logically grouped into following namespaces

```
namespace std {
    namespace ranges {
        namespace views {
            inline std::ranges::common_range common;
        }
    }
    namespace views = std::ranges::views; // shortcut for range adaptors
}
```
so that `std::ranges::common_range` becomes `std::views::common`,

or for the range adaptors

```
namespace views {
    inline std::ranges::<adaptor>_view adaptor;
}
```
so that `std::ranges::take_while_view` becomes `std::views::take_while`.

The interesting adaptor *std::views::iota* is a factory method (rangified std::iota) that generates the increasing sequence
of elements, where the final range boundary may be omitted (as in this case).
This will result in generating hypothetically an *infinite* sequence of subsequent elements - but "*lazy*", at the point when the expression
is evaluated (in this example, as part of the *out* direct initialization).
With *std::views::take_while* - we will actually stop further generation of elements, when the boundary value is reached.
The meaningful replacement for the exemplary code would be, of course, the finite version

```
std::views::iota(1, 5)
```

Range adaptors

Range adaptors are operators ("combinators") that are used to transform the ranges into the views.
The library provides a set of ready-to-use adaptors, to make our tasks more confortable, like already demonstrated
std::views::filter and *std::views::transform*.
Apart of the "pipe" syntax, which is most widely used

```
range | adaptor(args…)
```
we can write the same as
```
adaptor(range, args…)
```
or even
```
adaptor(args…)(range)
```

One can create the custom adaptors, either as a wrapper around the existing one,

```
template <typename Func>
auto customTransformAdaptor(Func&& func)
{
    //invokable that can be "piped" - a curry function
    return [f = std::forward<Func>(func)] (auto&&...args) mutable
           {
               // you would do something meaningful on top of the std::views::transform
               auto fun = std::bind(f, std::placeholders::_1, std::forward<decltype(args)>(args)...);
               return std::views::transform(fun);
           };
```

```
}
```

where the *std::placeholders::_1* is reserved for the resulting view of the previous operation in chain, so that

```
std::vector<int> in {1, 5, -12, 7, 24, 2};

auto multiply_transform = customTransformAdaptor([](int& el, int n) { return n * el; });
auto view = in | std::views::filter([](const auto& el) { return (el & 1) == 0; })
               | multiply_transform(3);
```

⇒ Full example: https://godbolt.org/z/z4WGz5bj7

or it can be tailored for the custom specific view: to convert the given range in that particular view.
More on that in Views section.

Projection

Rangified algorithms (like std::ranges::sort) are enriched with additional - *Projection* argument.
It's transformation that will be applied on each element in a range, before any element being actually inspect.
Under the hood, it uses *std::invoke*, a universal callable mechanism that works even with pointers/references on non-static member variables

*More on ***std::invoke***, and how this could be implemented for pre-C++17 compilers: https://godbolt.org/z/rGdEdcf6a

If we go back to our initial example with Java streams, the corresponding C++ representation could be

```
auto adults = persons | std::views::filter([](const auto& person){ return person.age >= 18; })
                      | std::views::transform(&Person::name)
                      | std::ranges::to<std::vector>();
```

Well, almost, since the converter *std::ranges::to* is available starting with C++23.
Therefore, you need to do it manually

```
std::vector<std::string> adultPersonsByName;
std::ranges::copy(adults, std::back_inserter(adultPersonsByName));
```

Let's go back to *Projection*, and sort example.
We can now write, along with comparator argument (default *std::ranges::less*), more powerful sort criteria, with less effort - verbosity that comes with lambdas, simple by refering the non-static member variable

```
std::ranges::sort(persons, {}, &Person::name);
std::ranges::sort(persons, std::ranges::greater{}, &Person::age);
```

⇒ Full example: https://godbolt.org/z/sn6fsea71

Views

Views are *snapshot*: subset of the range, that are cheap to move - for which
the move operations and destructor have constant time complexity - O(1).
@note This is true for *non-owning* views. Otherwise, destructor has O(N) time complexity.

Or, expressing these properties - requirements through the concepts

```
template <typename T>
concept view = ranges::range<T> && std::movable<T> && ranges::enable_view<T>;
```

The all views originate from the base class - *std::ranges::view_base*

```
template <typename T>
inline constexpr bool enable_view = std::derived_from<T, view_base>
```

Actually, they are implemented using ***std::ranges::view_interface***
which is *CRTP wrapper* around the *Derived* implementation, with quite simple - helper interface

```
template <std::ranges::view R>
class MyView : public std::ranges::view_interface<MyView<R>>
{
    public:

        MyView() = default; // default constructible
        constexpr MyView(R&& range) noexcept: m_base(std::move(range)) {}
        // Other c-tors in overloading set, if any

        constexpr R base() const & { return m_base; }
```

```
        constexpr R base() && { return std::move(m_base); }

        // std::ranges::view_interface interface
        constexpr auto begin() const {/*...*/};
        constexpr auto end() const {/*...*/};
         …

    private:
        R m_base = {};
};
```

As shown, they are desinged primarly to support the *composability*, along with range adaptors.
The important feature of views is that they are "***lazy***" by evaluation - which is one of the most significant asset of functional - declarative programming.
They pull the data on demand - in the processing chain of operators.
That is why it's also known as "*pull model*".

As a side note, with the latest library version, we can turn any movable range into view with ***std::views::all***() function. Previously, it worked only with *borrowed* ranges (for which the iterator can outlive the parent object, like with std::string_view).
Rvalue Ranges and Views in C++20 (tristanbrindle.com)

⇒ Fibonacci with views: https://godbolt.org/z/beEGanzqW

Useful links:

Official std page
Ranges library (C++20) - cppreference.com

Andreas Fertig: Programming with C++20
https://andreasfertig.com/books/programming-with-cpp20/

Eric Niebler (2017)
https://www.youtube.com/watch?v=LNXkPh3Z418

Eric Niebler blog
https://ericniebler.com/2018/12/05/standard-ranges/

Think-Cell range library
https://github.com/think-cell/range

https://hannes.hauswedell.net/post/2019/11/30/range_intro/

Projection by Bartlomiej Filipek
https://www.cppstories.com/2023/projections-examples-ranges/
https://www.cppstories.com/2020/10/understanding-invoke.html/

Rainer Grimm
https://www.modernescpp.com/index.php/c-20-the-ranges-library

Tristan Brindle, CppNorth 2022
https://m.youtube.com/watch?v=L0bhZp6HMDM

https://youtu.be/O8HndvYNvQ4