Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

C++20 Ranges in Practice

Marc Paterno

21 April 2021

Section 1

My summary

What is the range library?

- The range library is an extension of the Standard Template Library that makes its iterators and algorithms more powerful by making them composable.
- Introduces *ranges* and *views*:
  - ranges encapsulate a `begin` (iterator) and an `end` (sentinel) in a single object.
  - views are "composable adaptations of ranges where the adaptation happens lazily as the view is iterated".
- Ranges do not eliminate iterators; they are an abstraction layer *over* iterators.

Several range implementations

- There is a range library that is part of C++ (the current standard, “C++20”)
- There are implementations that work with pre-20 versions of the language
  - `range v3`, by Eric Niebler, upon which the range library in the standard is based. It contains additional goodies not in the standard (mainly actions, which provide *eager* application of an algorithm that mutates a container in-place). Requires a C++14 compiler. Does not rely upon compiler support for concepts; *error messages can be horrifying*.
    - https://github.com/ericniebler/range-v3
  - `nanorange`, by our speaker Tristan Brindle; requires C++17. I have had trouble getting examples from this talk to work with the speaker’s library.
    - https://github.com/tcbrindle/NanoRange
  - Boost.range was a very early precursor, but I would not recommend it for any new use.
    - https://www.boost.org/doc/libs/1_75_0/libs/range/doc/html/index.html

Ranges offer convenience for common use cases

- How do you sort a vector of integers `v`?

```
// C++17
std::sort(begin(v), end(v));
// C++20
std::ranges::sort(v);
```

- No need to specify the start and the end when you want to sort the whole thing.
- Removes an entire class of errors: passing mismatching iterators, or iterators in the wrong order.

Avoiding dangling iterators

- Temporary variables can be dangerous because they can lead to *dangling* iterators, pointers, or references.
  - *dangling* means the iterator (or pointer, or reference) refers to an object that no longer exists.

```
#include "range/v3/all.hpp"
#include <iostream>
#include <vector>
std::vector<int> get_input() { return {1, 2, 3}; }
int main() {
  auto iter = ranges::min_element(get_input());
  std::cout << *iter << '\n';                // DOES NOT COMPILE!
}
```

Types that don't need dangling protection

- `std::ranges::enable_borrowed_range` exists to tell the compiler that things like `std::string_view` and `std::span` don't have this problem.
  - It is in the library so that you can use it to declare your own templates as borrowed ranges.
- This is because their iterators point to a controlled buffer elsewhere, and and long as that buffer exists we're OK.
  - Is this appropriate for class templates in your code base?

Borrowed range

A *borrowed range* is either:

- an lvalue (an object with a name)
- an rvalue of a type that has specialized `std::ranges::enable_borrowed_range`.

Not the same thing as a *view*.

views

- A *view* is range which:
  - is default constructible
  - has constant-time move and destruction operations (not dependent on number of elements in the view)
  - is either non-copiable or has constant-time copy operations (no accidentally expensive copy can be used)
- Views are made to be passed *by value*, keeping semantics of their use simple.
- Classes that are views have to "opt in"; specialize `std::ranges::enable_view` trait, or inherit from `std::ranges::view_base` or `std::ranges::view_interface`.
- Not all views are borrowed ranges, and not all borrowed ranges are views.
- To create a view from a borrowed range, use `std::ranges::views::all(...)`.

viewable ranges

- *views* and *borrowed ranges* are both *viewable ranges*.
- Range adaptors work only on *viewable ranges*.
- Making a mistake with this can lead to horrifying error messages.

Tristan has a blog post **Rvalue Ranges and Views in C++20** at https://tristanbrindle.com/posts/rvalue-ranges-and-views.

Algorithms on views

- The algorithms in namespace `std::views` are lazy: they produce values only as needed.
- This helps to remove the need for intermediate storage of containers of intermediate results.
- What is the efficiency?
  - Avoiding making copies is generally a benefit.
  - Implementations are not always smart enough to match the efficiency of hand-crafted code.
  - Is the improvement in *ease if reading* enough to offset *runtime speed*?
  - What about *ease of writing* and the terrible error messages?
- The answer seems to be: *use ranges when it makes the code better*.
  - We'll look at a speed comparison at the very end of this review.

`views::common`

- Used to transform a range to something that has common types for `begin` and `end`. Good practice when passing a range into a C++17 algorithm.
- In C++20, this is still important because there isn't yet a "range-ified" `<numeric>`, where `std::accumulate`, `std::reduce` and `std::transform_reduce` live.

Projections

- A *projection* is a transformation built into the algorithm itself.
- By default, range algorithms use `std::identity`, which just returns its argument.
- Can help simplify even already simple code:

```
std::vector<Employee> scd = get_scd_sorted();
// Predicate supplied as a lambda
auto not_me =
    ranges::find_if(scd,
                    [](auto const& p){return p.first_name() == "Marc";});

// Using a projection rather than a lambda
auto also_not_me       =
    ranges::find(scd, "Marc", &Employee::first_name);
```

Numeric algorithms

Author's code at github.com/tcbrindle/numeric_ranges.

- Requires C++20 ranges or the use of `nanorange`.
- Only an approximation for what has been proposed for C++23: *no constraints* on templates

I don't often see `std::accumulate` and `std::inner_product` (or the better-named `std::reduce` and `std::transform_reduce`) used.

- `std::accumulate` and `std::inner_product` are *not* new.
- Is this for good reason? (lack of flexibility? performance?)
- Or is it for less good reason? (unfamiliarity? FUD?)

Section 2

Trimming strings

Trimming strings

It may be in *less* numeric algorithms like this that we would see most common use of range algorithms.

Divide and conquer

Tristan describes a general technique for applying composable algorithms like those of the range library: *break up the problem into pieces*.

- What are the pieces?
- Is each step general?
- Can each step be generalized?
- Example:
  - generate trimmed view:
    - trim from the front;
    - trim from the back;
  - turn the view into a container (here, a `std::string`).

Second part first: turn the view into a string

- Generalized problem: turn view into some realized container.
- `std::ranges::to<C>` to turn a range into a container.
- What containers? Sequences, not mappings.
- Tristan's example shows `std::string`.
- github.com/cor3ntin/rangesnext contains several items proposed for C++23. What does it depend upon?

First part second: more breaking up the problem

- `trim` is made from `trim_front` and `trim_back`.
- Each one is independently meaningful and useful (and *testable*)!
- No loss of efficiency? (Because we're not copying containers, but working with views.)

Function templates declared with return type `auto` all over the place

- Are they needed for correctness? Or just simpler (and less ugly) to write?
- Are they damaging to readability? How do we combat this?

`trim_back` implemented using *two* reverses of the view. What is the cost of this?

- Quick measurements show `trim_front` is ~ 1/3 the cost of `trim_back`.

“Simplification” of the code?

First version of `trim_front` was a function that took a range, and return an adapted range.

“Simplification” of it is a function that just returns the adaptor

Result is *composed functions*, in the style of many functional programming languages.

- Does this actually help with composition, or with clarity?
- Does it do damage to efficiency?

Final tweaks

- the functions `trim()`, `trim_back()`, `trim_front()` each always returns the same value
- turn that kind of function into an `inline constexpr` variable.

Final version

```
namespace rv = ranges::view
inline constexpr auto trim_front = rv::drop_while(::isspace);
inline constexpr auto trim_back = rv::reverse | trim_front | rv::reverse;
inline constexpr auto trim = trim_front | trim_back;
inline std::string
trim_str(std::string const& s) {return s | trim | ranges::to<std::string>;};
```

ns/op	op/s	err%	total	benchmark
105.09	9,515,828.68	0.2%	0.01	`cetlib::trim`
162.68	6,147,032.17	4.4%	1.00	`range trim`

- Compiled with `g++-10 -fconcepts -O3 -std=c++2a` on macOS Catalina Intel Core i9 @2.4 GHz.