C++20 RANGES IN PRACTICE

CppNorth 🇨🇦 2022

Tristan Brindle


VULCAN
VULCAN

TODAY'S AGENDA

TODAY'S AGENDA

- Problem 1: least element of an array

TODAY'S AGENDA

- Problem 1: least element of an array
- Problem 2: sum of squares

TODAY'S AGENDA

- Problem 1: least element of an array
- Problem 2: sum of squares
- Problem 3: string trimming

TODAY'S AGENDA

- Problem 1: least element of an array
- Problem 2: sum of squares
- Problem 3: string trimming
- Questions

PROBLEM 1: MINIMUM ARRAY ELEMENT

- Given a non-empty array of integers, find the minimum value it contains
- For example, `[5, 22, 88, -12, 14] => -12`

```cpp
std::vector<int> vec = get_input();

auto iter = std::min_element(vec.begin(), vec.end());
```

```
namespace rng = std::ranges;

std::vector<int> vec = get_input();

auto iter = rng::min_element(vec.begin(), vec.end());
```

```cpp
std::vector<int> vec = get_input();

auto iter = rng::min_element(vec)
```

```
auto iter = rng::min_element(get_input());
```

```
auto iter = rng::min_element(get_input());
```

```
auto iter = rng::min_element(get_input());
```

Dangling iterator?

When you pass an rvalue range into an algorithm which normally returns an iterator, ranges will return a `std::ranges::dangling` object.

```
auto iter = rng::min_element(get_input());
```

This will compile, but `iter` will have type `ranges::dangling`.

Attempting to *use* a `dangling` object will cause a compile error

Attempting to *use* a `dangling` object will cause a compile error

```
auto iter = rng::min_element(get_input());
std::cout << *iter;
```

Attempting to *use* a `dangling` object will cause a compile error

```
auto iter = rng::min_element(get_input());
std::cout << *iter;
```

```
error: no match for 'operator*'
(operand type is 'std::ranges::dangling')
```

Ranges applies this "dangling protection" when you pass an rvalue range whose iterators can potentially dangle into an algorithm which returns an iterator.

Some types have iterators which *can* safely outlive their parent, for example `std::string_view` and `std::span`.

Some types have iterators which *can* safely outlive their parent, for example `std::string_view` and `std::span`.

```
std::string str = "Hello world";
// Weird, but okay:
auto iter = std::string_view{str}.begin();
std::cout << *iter << '\n';
```

We can disable "dangling protection" for such types by specialising the `enable_borrowed_range` trait

We can disable "dangling protection" for such types by specialising the `enable_borrowed_range` trait

```
// Outside of any namespace
template <>
inline constexpr bool
std::ranges::enable_borrowed_range<my::StringRef> = true;
```

```
auto iter = rng::find(get_string_ref(), 'x');
// iter is an iterator, *not* ranges::dangling
```

VIEWS

Informally, *views* are sometimes referred to as "ranges which do not own their elements".

The formal requirements of a `view` have changed several times over the course of Ranges standardisation.

The most recent update was P2415, "What is a View" (Barry Revzin and Tim Song)

Today, a *view* is a range such that

Today, a *view* is a range such that

- its move operations are constant-time

Today, a *view* is a range such that

- its move operations are constant-time
- (if it is copyable) its copy operations are constant-time

Today, a *view* is a range such that

- its move operations are constant-time
- (if it is copyable) its copy operations are constant-time
- destruction of a moved-from object is constant-time

Today, a *view* is a range such that

- its move operations are constant-time
- (if it is copyable) its copy operations are constant-time
- destruction of a moved-from object is constant-time
- otherwise, destruction is O(N)

Just as with borrowed ranges, a type can opt in to being a view using the `enable_view` trait, or by inheriting from `ranges::view_base`.

Today, we can turn any *movable range* into a *view* using the `std::views::all()` function

Today, we can turn any *movable range* into a *view* using the `std::views::all()` function

(Previously this only worked with borrowed ranges)

```cpp
std::vector<int> vec = get_input();

auto view1 = std::views::all(vec);
// Always okay, view1 is movable and copyable

auto view2 = std::views::all(get_input());
// Okay since P2415, view2 is move-only
```

View *adaptors* implicitly convert their arguments to views using `views::all()`.

```cpp
auto vec = get_vector();

auto v1 = vec | views::transform(func);
// Always okay: vec is an lvalue => borrowed

auto v2 = get_vector() | views::transform(func);
// Okay since P2415, v2 is move-only
```

```cpp
auto vec = get_vector();

auto v1 = vec | views::transform(func);
// Always okay: vec is an lvalue => borrowed

auto v2 = get_vector() | views::transform(func);
// Okay since P2415, v2 is move-only
```

PROBLEM 2: SUM OF SQUARES

- Given an array of integers, calculate the sum of the squares of the values it contains
- For example, `[-2, -1, 0, 1, 2] => 10`

```
auto vec = get_input();

std::transform(vec.begin(), vec.end(), vec.begin(),
               [](int i) { return i * i; });

auto sumsq = std::accumulate(vec.begin(), vec.end(), 0);
```

```
auto vec = get_input();

rng::transform(vec, vec.begin(),
               [](int i) { return i * i; });

auto sumsq = std::accumulate(vec.begin(), vec.end(), 0);
```

```
auto vec = get_input();

auto view = std::views::transform(vec,
    [](int i) { return i * i; });

auto sumsq = std::accumulate(view.begin(), view.end(), 0);
```

```
auto vec = get_input();

auto view = vec
    | std::views::transform([](int i) { return i * i; })
    | std::views::common;

auto sumsq = std::accumulate(view.begin(), view.end(), 0);
```

Unfortunately, C++20 does not have "range-ified" versions of the algorithms in the `<numeric>` header.

All being well, C++23 will provide a family of range-based `fold()` functions which we can use instead

In the mean time, let's write our own range-based `accumulate()`!

```
template <typename I, typename Init>
Init accumulate(I first, I last, Init init)
{
    while (first != last) {
        init = std::move(init) + *first;
        ++first;
    }
    return init;
}
```

```
template <typename I, typename S, typename Init>
Init accumulate(I first, S last, Init init)
{
    while (first != last) {
        init = std::move(init) + *first;
        ++first;
    }
    return init;
}
```

```
template <typename I, typename S, typename Init,
          typename Op = std::plus<>>
Init accumulate(I first, S last, Init init, Op op = Op{})
{
    while (first != last) {
        init = op(std::move(init), *first);
        ++first;
    }
    return init;
}
```

```
template <typename I, typename S, typename Init,
          typename Op = std::plus<>>
    requires
        std::input_iterator<I> &&
        std::sentinel_for<S, I>
Init accumulate(I first, S last, Init init, Op op = Op{})
{
    while (first != last) {
        init = op(std::move(init), *first);
        ++first;
    }
    return init;
}
```

```
template <std::input_iterator I, std::sentinel_for<I> S,
          typename Init, typename Op = std::plus<>>
Init accumulate(I first, S last, Init init, Op op = Op{})
{
    while (first != last) {
        init = op(std::move(init), *first);
        ++first;
    }
    return init;
}
```

```cpp
template <std::input_iterator I, std::sentinel_for<I> S,
          typename Init = std::iter_value_t<I>,
          typename Op = std::plus<>>
Init accumulate(I first, S last, Init init = Init{},
                Op op = Op{})
{
    while (first != last) {
        init = std::invoke(op, std::move(init), *first);
        ++first;
    }
    return init;
}
```

```cpp
auto vec = get_input();

auto view = vec
    | views::transform([](int i) { return i * i; })
    | views::common;

auto sumsq = std::accumulate(view.begin(), view.end(), 0);
```

```
auto vec = get_input();

auto view = vec
    | views::transform([](int i) { return i * i; });

auto sumsq = accumulate(view.begin(), view.end());
```

```
// Range overload
template <rng::input_range R,
          typename Init = rng::range_value_t<R>,
          typename Op = std::plus<>>
Init accumulate(R&& rng, Init init = Init{}, Op op = Op{})
{
    return accumulate(rng::begin(rng), rng::end(rng),
                      std::move(init), std::move(op));
}
```

```
auto vec = get_input();

auto view = vec
    | views::transform([](int i) { return i * i; });

auto sumsq = accumulate(view.begin(), view.end());
```

```
auto vec = get_input();

auto view = vec
    | views::transform([](int i) { return i * i; });

auto sumsq = accumulate(view);
```

```
auto vec = get_input();

auto sumsq = accumulate(
    vec | views::transform([](int i) { return i * i; }));
);
```

```
template <std::input_iterator I, std::sentinel_for<I> S,
          typename Init = std::iter_value_t<I>,
          typename Op = std::plus<>,
          typename Proj = std::identity>
Init accumulate(I first, S last, Init init = Init{},
                Op op = Op{}, Proj proj = Proj{})
{
    while (first != last) {
        init = std::invoke(op, std::move(init),
                           std::invoke(proj, *first));
        ++first;
    }
    return init;
}
```

```
template <rng::input_range R,
          typename Init = rng::range_value_t<R>,
          typename Op = std::plus<>,
          typename Proj = std::identity>
Init accumulate(R&& rng, Init init = Init{},
                Op op = Op{}, Proj proj = Proj{})
{
    return accumulate(rng::begin(rng), rng::end(rng),
                      std::move(init), std::move(op),
                      std::move(proj));
}
```

```
auto vec = get_input();

auto sumsq = accumulate(
    vec | views::transform([](int i) { return i * i; }));
);
```

```
auto vec = get_input();

auto sumsq = accumulate(vec, {}, {},
    [](int i) { return i * i; }));
);
```

```
auto sumsq = accumulate(get_input(), {}, {},
    [] (int i) { return i * i; });
```

PROBLEM 3: STRING TRIMMING

- Given an input string, construct a new string with leading and trailing whitespace removed
- For example, `" \n\t\r Hello World! \n\n" => "Hello World!"`

```
std::string trim_str(const std::string& str)
{

}
```

```
std::string trim_str(const std::string& str)
{
    auto view = trim(str);
    return to_string(view);
}
```

```
template <typename R>
std::string to_string(R&& rng)
{

}
```

```
template <typename R>
std::string to_string(R&& rng)
{
    auto v = views::common(forward<R>(rng));
    return std::string(v.begin(), v.end());
}
```

In Range-V3, we could use `ranges::to` to perform the conversion:

```
std::string str =
    input
    | views::transform(...)
    | views::filter(...)
    | ...
    | ranges::to<std::string>;
```

All being well, `std::ranges::to` will be available in C++23!

For a C++20 compatible implementation, see github.com/cor3ntin/rangesnext;

```
std::string trim_str(const std::string& str)
{
    auto view = trim(str);
    return view | rangesnext::to<std::string>;
}
```

```
std::string trim_str(const std::string& str)
{
    return trim(str)
         | rangesnext::to<std::string>;
}
```

```
template <typename R>
auto trim(R&& rng)
{

}
```

```
template <typename R>
auto trim(R&& rng)
{
    return trim_back(trim_front(forward<R>(rng)));
}
```

```
template <typename R>
auto trim_front(R&& rng)
{

}
```

```
template <typename R>
auto trim_front(R&& rng)
{
    return views::drop_while(forward<R>(rng), ::isspace);
}
```

```
template <typename R>
auto trim_front(R&& rng)
{
    return forward<R>(rng) | views::drop_while(::isspace);
}
```

```
template <typename R>
auto trim_back(R&& rng)
{

}
```

```
template <typename R>
auto trim_back(R&& rng)
{
    return forward<R>(rng)
        | views::drop_last_while(::isspace); // :(
}
```

```
template <typename R>
auto trim_back(R&& rng)
{
    return forward<R>(rng)
        | views::reverse
        | views::drop_while(::isspace)
        | views::reverse;
}
```

```
template <typename R>
auto trim_front(R&& rng) {
    return forward<R>(rng) | views::drop_while(::isspace);
}

template <typename R>
auto trim_back(R&& rng) {
    return forward<R>(rng)
         | views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}

template <typename R>
auto trim(R&& rng) {
    return trim_back(trim_front(forward<R>(rng)));
}

std::string trim_str(const std::string& str) {
    return trim(str)
        | rangesnext::to<std::string>;
}
```

```
template <typename R>
auto trim_front(R&& rng)
{
    return forward<R>(rng) | views::drop_while(::isspace);
}
```

```
auto trim_front()
{
    return views::drop_while(::isspace);
}
```

```
template <typename R>
auto trim_back(R&& rng)
{
    return forward<R>(rng)
         | views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}
```

```
auto trim_back()
{
    return views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}
```

```
auto trim_front() {
    return views::drop_while(::isspace);
}

auto trim_back() {
    return views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}

template <typename R>
auto trim(R&& rng) {
    return
        trim_back()(
            trim_front()(
                forward<R>(rng)));
}

std::string trim_str(const std::string& str) {
    return trim(str)
         | rangesnext::to<std::string>;
}
```

```
auto trim_front() {
    return views::drop_while(::isspace);
}

auto trim_back() {
    return views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}

template <typename R>
auto trim(R&& rng) {
    return forward<R>(rng)
         | trim_front()
         | trim_back();
}

std::string trim_str(const std::string& str) {
    return trim(str)
         | rangesnext::to<std::string>;
}
```

```
auto trim_front() {
    return views::drop_while(::isspace);
}

auto trim_back()
{
    return views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}

auto trim() {
    return trim_front() | trim_back();
}

std::string trim_str(const std::string& str) {
    return trim()(str)
         | rangesnext::to<std::string>;
}
```

```
auto trim_front() {
    return views::drop_while(::isspace);
}

auto trim_back()
{
    return views::reverse
         | views::drop_while(::isspace)
         | views::reverse;
}

auto trim() {
    return trim_front() | trim_back();
}

std::string trim_str(const std::string& str) {
    return str
         | trim()
         | rangesnext::to<std::string>;
}
```

```
auto trim_front() {
    return views::drop_while(::isspace);
}
```

```
inline constexpr auto trim_front =
    views::drop_while(::isspace);
```

```
inline constexpr auto trim_back =
      views::reverse
    | views::drop_while(::isspace)
    | views::reverse;
```

```
inline constexpr auto trim_front = views::drop_while(::isspace);

inline constexpr auto trim_back =
      views::reverse
    | views::drop_while(::isspace)
    | views::reverse;

inline constexpr auto trim = trim_front | trim_back;

std::string trim_str(const std::string& str) {
    return str | trim | rangesnext::to<std::string>;
}
```

```
inline constexpr auto trim_front = views::drop_while(::isspace);

inline constexpr auto trim_back =
      views::reverse
    | trim_front
    | views::reverse;

inline constexpr auto trim = trim_front | trim_back;

std::string trim_str(const std::string& str) {
    return str | trim | rangesnext::to<std::string>;
}
```

THANK YOU VERY MUCH!

@tristanbrindle

github.com/tcbrindle

"An Overview of Standard Ranges"

CppCon 2019

https://youtu.be/SYLgG7Q5Zws

"Conquering C++20 Ranges"

CppCon 2021

https://youtu.be/3MBtLeyJKg0