Cpp North 2025

Advanced Ranges

Writing Modular, Clean, and Efficient Code with Custom Views

Steve Sorkin

Advanced Ranges: Writing Modular, Clean, and Efficient Code with Custom Views

Bloomberg

Engineering

CppNorth 2025
July 23, 2025

Steve Sorkin
Senior Software Engineer

TechAtBloomberg.com

About Me

- At Bloomberg since 2019
- Primarily a C++ backend developer
- Previously a derivatives attorney

Bloomberg

Engineering

Outline

- Overview of ranges and views
- Custom Views
- Case Studies
- Best practices and pitfalls

Overview of Ranges

What are ranges?

➔ One of the four major additions in C++20

➔ A range can be thought of as a representation of a sequence of elements that has a begin iterator and an end sentinel

```
std::vector<int> values {5, 8, 1, 4, 9};
```

➔ This definition encompasses existing containers in the STL (vector, map, etc.) as well as UDT, as ranges, so long as they satisfy this constraint

➔ The official definition is that a range needs to comply with `std::ranges::range` concept

Why should we care about ranges?

➔ Ranges can be used in conjunction with algorithms, and ***views***, to improve code by making it more:

- Readable
- Composable
- Efficient

Views

➔ Views are lightweight, **non-owning** ranges

- ◆ Typically support lazy evaluation - allows for some interesting use cases such as infinite ranges
- ◆ Provide a “window” into an existing range via **reference semantics**
  - Memory efficient - they generally DO NOT store copies of elements
  - Modifications to the underlying range are reflected in the view and vice versa (assuming the view provides **mutable** references)

➔ Composable - multiple transformations can be chained or piped together

Bloomberg

Engineering

Range adaptors

➔ Functions that transform an input range into a view

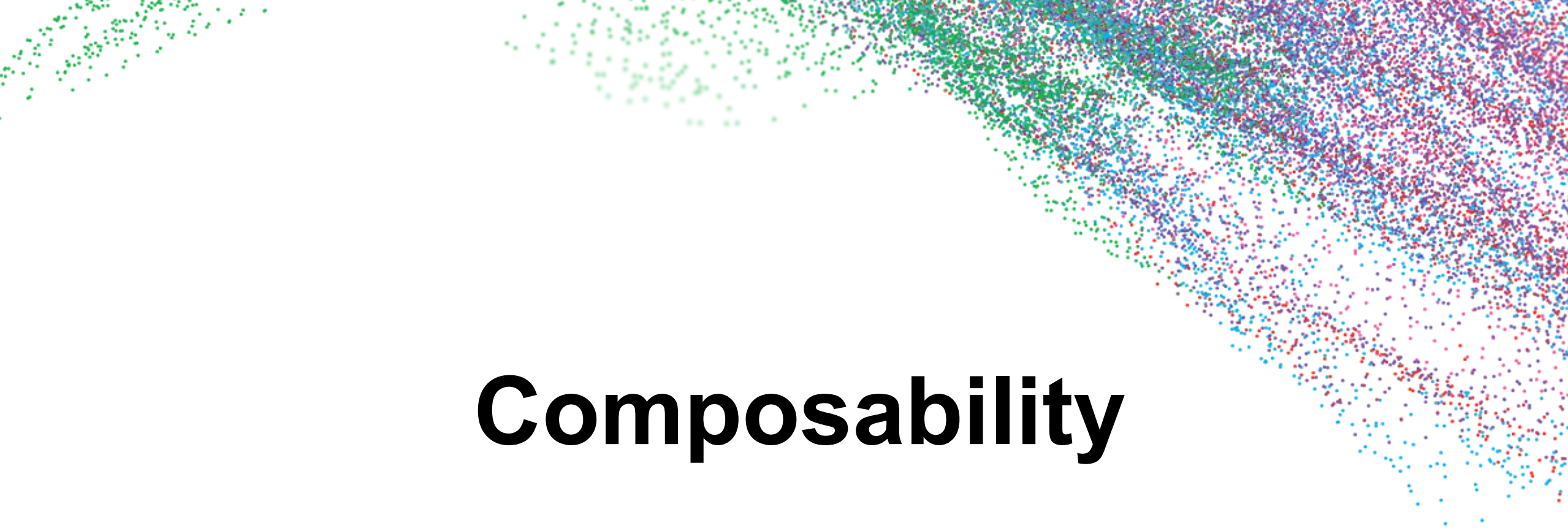
- Enable efficient composition via the **pipe** operator
- Allow for a series of transformations on an input range without the need for intermediate containers
- Enable lazy evaluation - operations are deferred until we iterate through the range

➔ Components:

- View - the result of applying the transformation or logic to an input range
- Range adaptor object - a functor that produces a closure object
- Range closure object - holds any parameters and produces the view when applied to an input range

Bloomberg

Engineering



Composability

Let’s start with a motivating example

```
struct StockData {
    std::string m_ticker;
    double m_currentPrice;
    double m_priceIncrease;
    double m_costToBuy;
};

std::vector<StockData> inputData {
        {"MSFT", 75.0, 6.3, 0},
        {"PEP", 35, 1.2, 0},
        {"COIN", 165, 7.2, 0},
        {"HOOD", 45, 15.1, 0},
        {"INTC", 18, -.3, 0},
        {"WMT", 36, 27.4, 0}
    };
```

Bloomberg

Engineering

The C++17 way

```
int main() {
    // input data already provided/exists

    std::vector<StockData> filteredData;
    auto filter_predicate = [](auto stockData) { return
stockData.m_priceIncrease > 5;};
    std::copy_if(inputData.begin(), inputData.end(),
std::back_inserter(filteredData), filter_predicate);

    // transform data

    for (auto& stockData : filteredData) {
        stockData.m_costToBuy = stockData.m_currentPrice * 100;
    }
    const int number_of_stocks_to_buy = 3;
    for (int i = 0; i < number_of_stocks_to_buy; i++) {
        std::cout << filteredData[i].m_ticker << " total cost to buy is:
" << filteredData[i].m_costToBuy << std::endl;
    }
    return 0;
}
```

- ➔ Verbose code
- ➔ Extra storage to store the results of `std::copy_if`
- ➔ Notice the potential UB if there are fewer than 3 stocks that pass the filter

Output:

MSFT total cost to buy is: 7500
COIN total cost to buy is: 16500
HOOD total cost to buy is: 4500

The C++17 way

```
int main() {
       // input data already provided/exists

    std::vector<StockData> filteredData;
    auto filter_predicate = [](auto stockData) { return stockData.m_priceIncrease > 5;};
    std::copy_if(inputData.begin(), inputData.end(), std::back_inserter(filteredData), filter_predicate);

    // transform data

    for (auto& stockData : filteredData) {
        stockData.m_costToBuy = stockData.m_currentPrice * 100;
    }
    const int number_of_stocks_to_buy = std::min(filteredData.size(), (size_t)3);
    for (int i = 0; i < number_of_stocks_to_buy; i++) {
        std::cout << filteredData[i].m_ticker << " total cost to buy is: " << filteredData[i].m_costToBuy << std::endl;
    }
    return 0;
}
```

Bloomberg

Engineering

The Ranges' way

```cpp
int main() {
       // input data already provided/exists
    auto result = inputData | std::views::filter([](const StockData& stockData) { return stockData.m_priceIncrease > 22; })
                            | std::views::transform([](const StockData& stockData) { return StockData
                               {stockData.m_ticker, stockData.m_currentPrice,
                               stockData.m_priceIncrease, stockData.m_currentPrice*100}; })
                            | std::views::take(3);

    for (auto elem : result) {
        std::cout << elem.m_ticker << " total cost to buy is: " << elem.m_costToBuy << std::endl;
    }
    return 0;
}
```

- ➔ Cleaner code
- ➔ No intermediate container needed
- ➔ Safer - no need to worry about how many filtered elements there actually are
- ➔ Worth noting that the transform view is **not mutable**

Bloomberg

Engineering

Lazy Evaluation

Eager vs. lazy evaluation

- ➔ **Eager evaluation** immediately calculates and stores its elements when **created**
- ➔ On the other hand, a **lazy** view defers the computation until elements are needed
- ➔ Eager evaluation causes our program to pay the cost of computation up front
- ➔ Computed elements need to be stored, which affects memory usage
- ➔ There are some advantages to eager evaluation
  - ◆ Such as when iterating over the range multiple times

Measuring Computation Cost

Eager Implementation

```
int main() {
    std::list<int> initial_data (200);
    std::iota(initial_data.begin(), initial_data.end(), 3);

    std::list<int> transformed_eager_data;
    for (const auto& num : initial_data) {
        //do some math
        transformed_eager_data.push_back(num*num + 9);
    }
    std::list<int> filtered_eager_data;
    for (const auto& num : transformed_eager_data) {
        if (num % 2 == 0) {
            filtered_eager_data.push_back(num);
        }
    }

    return 0;
}
```

➔ Intermediate data structures are created as we transform and manipulate the data

- Some of this could be optimized depending on the use case (e.g., remove data elements in-place from `transformed_eager_data` during filtering)
- Regardless of optimizations this would still need more memory than a lazy implementation

➔ We also have to immediately pay the computational cost as we filter and transform data regardless of when (or if) we will need to use that data

Measuring Computational Cost

Lazy Implementation

```
int main() {
    std::list<int> initial_data (200);
    std::iota(initial_data.begin(), initial_data.end(), 3);

    auto pipeline = initial_data
        | std::views::transform([] (int x) { return x*x + 9;})
        | std::views::filter([](int x) { return x % 2 == 0; });

    // We only care about the first 10 elements
    std::list<int> result;
    std::ranges::copy_n(pipeline.begin(), 10, std::back_inserter(result));

    return 0;
}
```

- ➔ No intermediate data structure until we load the results using `std::ranges::copy_n` into a new data structure
- ➔ Since we only care about the first 10 results, we do a lot less computation than the eager implementation

Measuring Computational Cost

Putting it all together

```cpp
int main() {
    std::list<int> initial_data (200);
    std::iota(initial_data.begin(), initial_data.end(), 3);

    auto start_eager = std::chrono::high_resolution_clock::now();
    std::list<int> transformed_eager_data;
    for (const auto& num : initial_data) {
        transformed_eager_data.push_back(num*num + 9);
    }
    std::list<int> filtered_eager_data;
    for (const auto& num : transformed_eager_data) {
        if (num % 2 == 0) {
            filtered_eager_data.push_back(num);
        }
    }
    auto end_eager = std::chrono::high_resolution_clock::now();
    std::cout << "Eager evaluation time: "
              << std::chrono::duration<double, std::micro>(end_eager - start_eager).count()
              << " microseconds\n";

    auto start_lazy = std::chrono::high_resolution_clock::now();

    auto pipeline = initial_data
        | std::views::transform([] (int x) { return x*x + 9;})
        | std::views::filter([](int x) { return x % 2 == 0; });

    // We only care about the first 10 elements
    std::list<int> result;
    std::ranges::copy_n(pipeline.begin(), 10, std::back_inserter(result));

    auto end_lazy = std::chrono::high_resolution_clock::now();
    std::cout << "Lazy evaluation time: "
              << std::chrono::duration<double, std::micro>(end_lazy - start_lazy).count()
              << " microseconds\n";

    return 0;
}
```

Output:

Eager evaluation time: **51.675 microseconds**

Lazy evaluation time: **6.104 microseconds**

https://godbolt.org/z/s1EPYdTjx

Why Custom Views?

C++20 views

- `transform_view`
- `take_view`
- `take_while_view`
- `drop_view`
- `drop_while_view`
- `split_view`
- `lazy_split_view`
- `counted`
- `common_view`
- `reverse_view`
- `elements_view`
- `keys_view`
- `values_view`
- `filter_view`

C++23 views

- `join_with_view`
- `adjacent_view`
- `zip_transform_view`
- `as_const_view`
- `chunk_view`
- `stride_view`
- `enumerate_view`
- `slide_view`
- `cartesian_product_view`
- `zip_view`
- `chunk_by_view`

Why build a custom view?

➔ Can be used for memory and computational optimizations

➔ Use case that is not efficiently provided for by existing views

➔ Specific functionality which depends on prior elements or external state

TechAtBloomberg.com

Bloomberg

Engineering

Sliding Window Median Value

```
std::vector<double> input_data {100.5, 112.5, 97.35, 89.3, 79, 87.3,
                                88.35, 93.5, 95.8, 97.1, 97.95};
constexpr std::size_t rolling_window_size = 3;

for (const auto& window_data : input_data |
    std::views::slide(rolling_window_size)) {
        std::vector<double> slide_data {window_data.begin(), window_data.end()};
        std::ranges::nth_element(slide_data, slide_data.begin() + rolling_window_size/2);
        std::cout << slide_data[rolling_window_size/2] << std::endl;
}
```

- ➔ Computational complexity: **O(N*M)**
  - ◆ Each window requires copying M elements (window size) and this is done approximately N times
  - ◆ In addition there is the cost of computing the median via `nth_element`
- ➔ There is also a memory issue
  - ◆ On every iteration of the loop we are doing a heap allocation for `slide_data`
  - ◆ All window elements are copied into a new vector (`slide_data`)
- ➔ Additionally, `slide_view` avoids allocations but it still has significant performance overhead

https://godbolt.org/z/bbfs6o5n7

Sliding Window Median Value

```cpp
std::vector<double> input_data {100.5, 112.5, 97.35, 89.3, 79, 87.3,
                                88.35, 93.5, 95.8, 97.1, 97.95};
constexpr std::size_t rolling_window_size = 3;

for (const auto& window_data : input_data | std::views::slide(rolling_window_size)) {
    std::vector<double> slide_data {window_data.begin(), window_data.end()};

    const size_t mid_idx = rolling_window_size / 2;
    std::ranges::nth_element(slide_data, slide_data.begin() + mid_idx);

    if (rolling_window_size % 2 == 0) {
        auto max_it = std::max_element(slide_data.begin(), slide_data.begin() + mid_idx);
        double lower_median = *max_it;
        double higher_median = slide_data[mid_idx];
        std::cout << (lower_median + higher_median) / 2 << std::endl;
    } else {
        // Odd window size - return the middle element
        std::cout << slide_data[mid_idx] << std::endl;
    }
}
```

- ➔ Supports both odd and even window sizes
- ➔ For even-sized windows we use `max_element` to find the second middle value

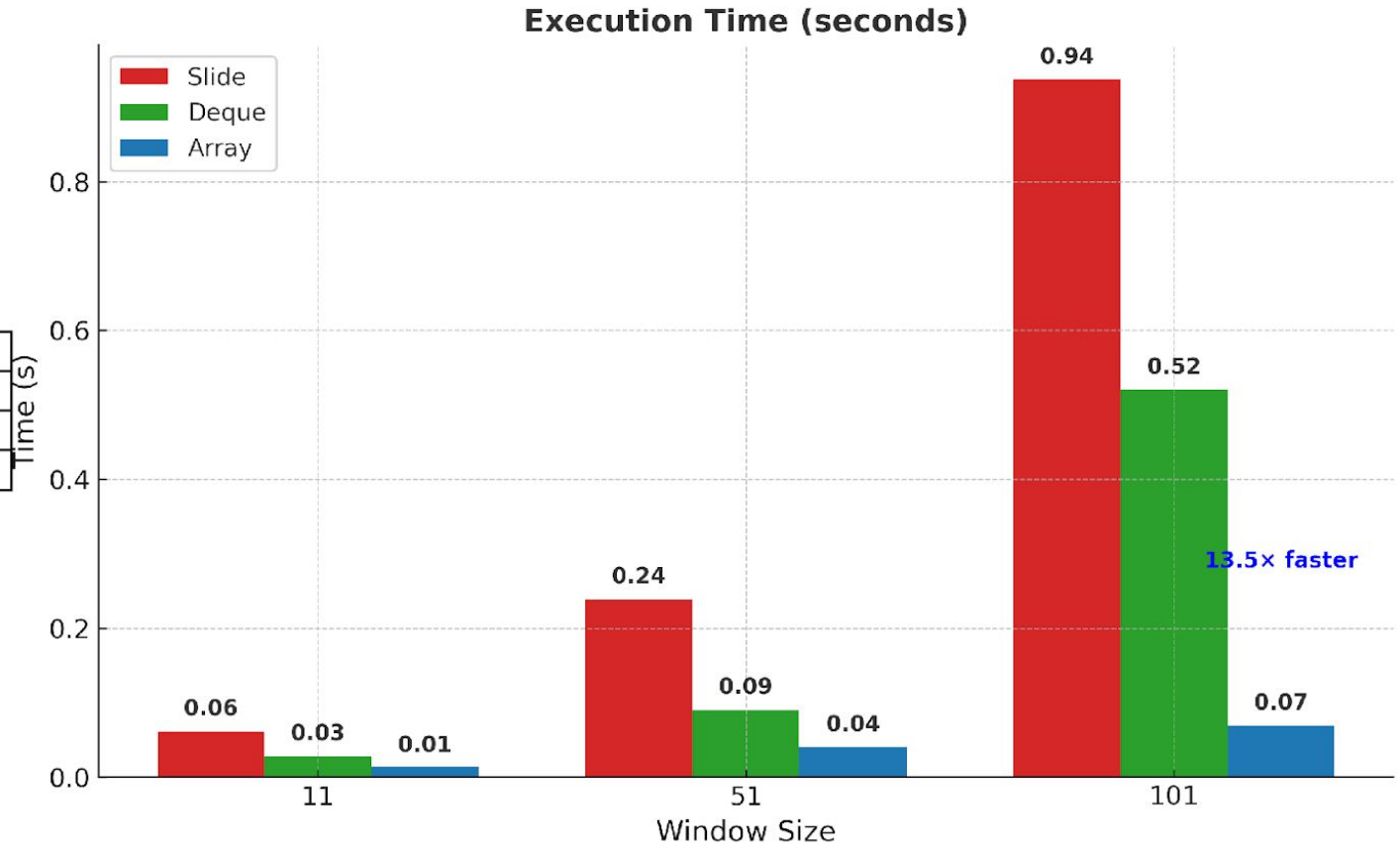
Benchmarking

- ➔ 3 implementations: `std::slide`, deque implementation (custom), circular array (custom)
- ➔ Window sizes: 11, 51, 101
- ➔ Input data size:
  - ◆ 30,000 elements
  - ◆ 100,000 elements
  - ◆ 300,000 elements

https://godbolt.org/z/9bGaz6GGz

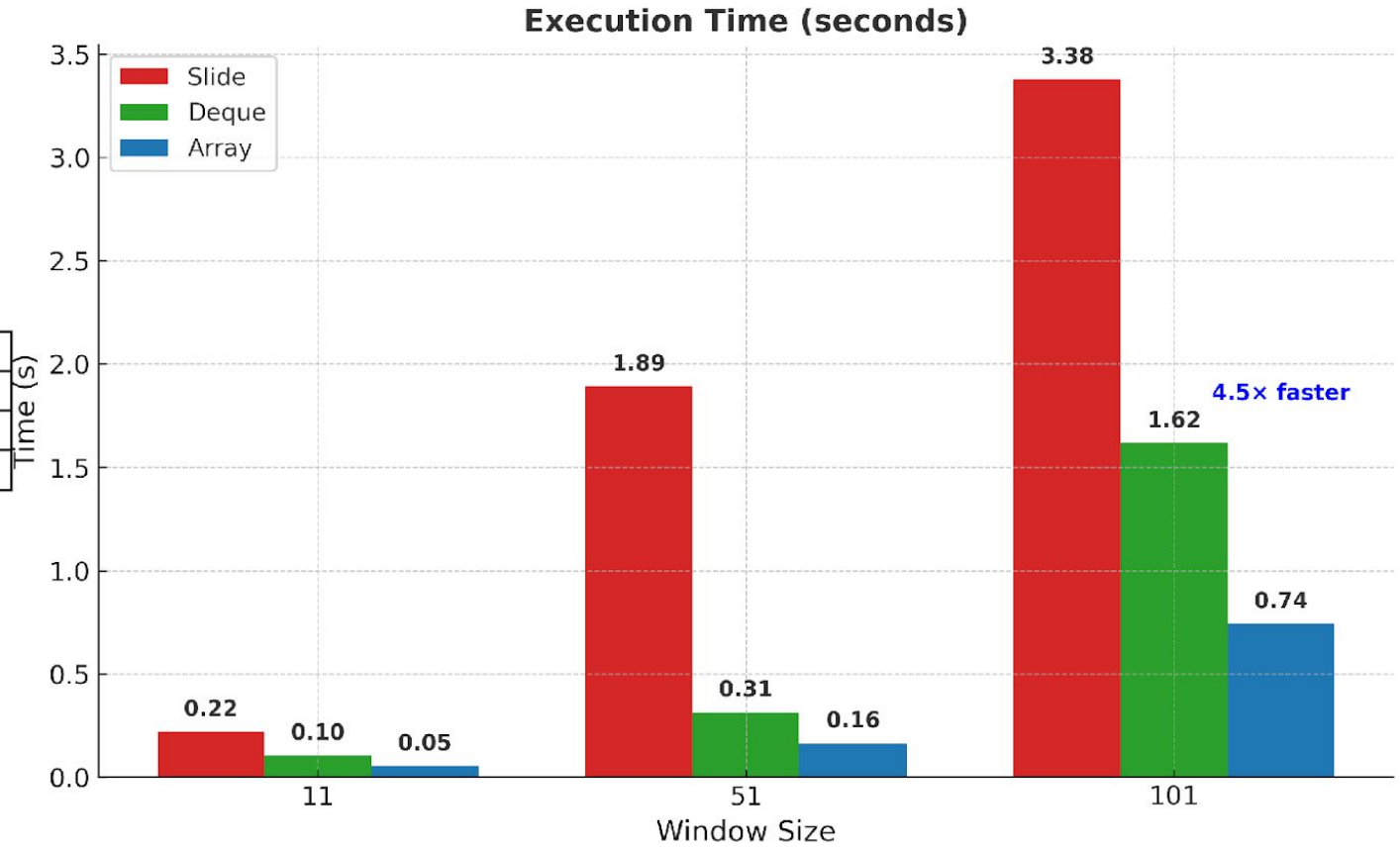
Sliding Median Benchmark – 30,000 Elements

Window Size	Slide (ms)	Deque (ms)	Array (ms)
11.0	61.6	28.3	13.0
51.0	239.2	90.5	40.4
101.0	937.2	520.4	69.2



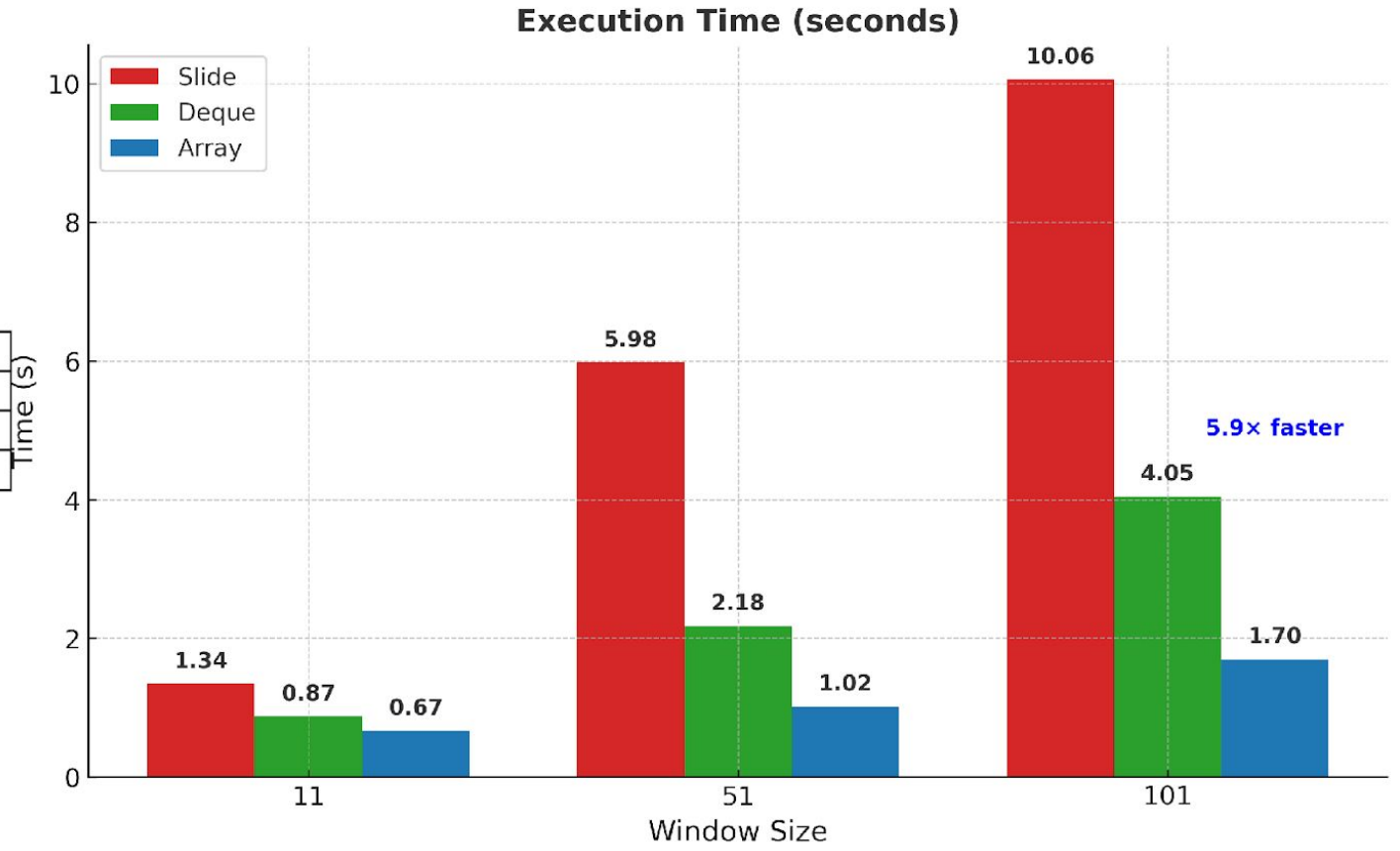
Sliding Median Benchmark – 100,000 Elements

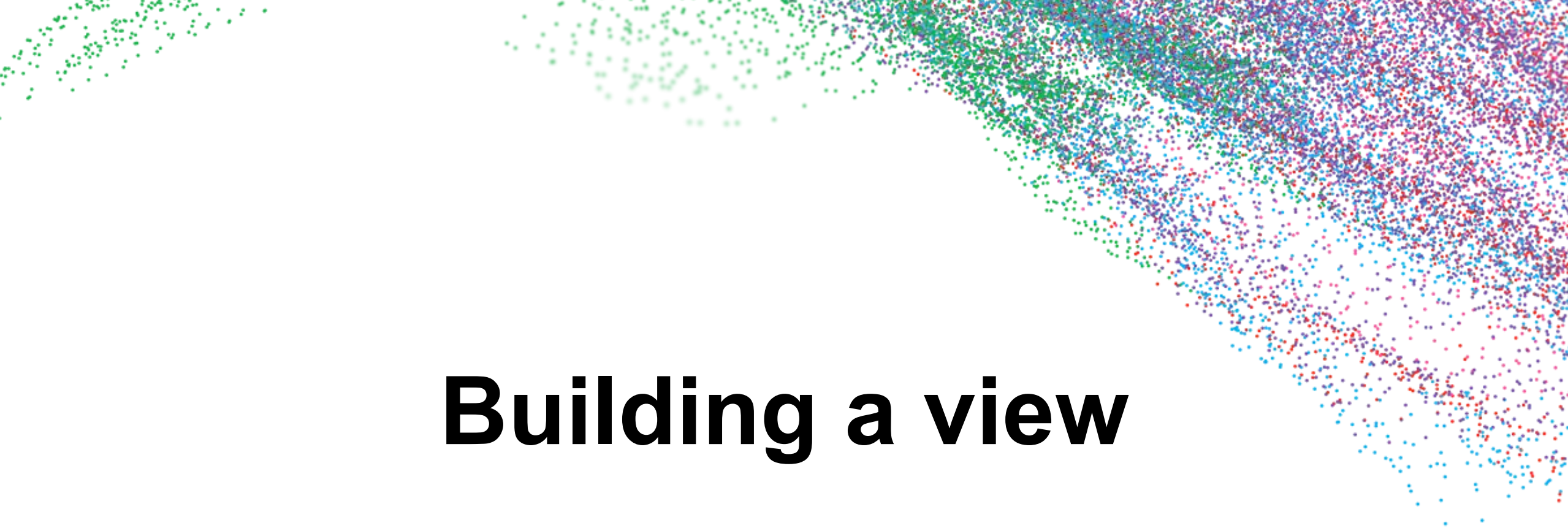
Window Size	Slide (ms)	Deque (ms)	Array (ms)
11.0	218.2	103.4	54.2
51.0	1892.9	313.0	161.8
101.0	3378.5	1617.3	743.0



Sliding Median Benchmark – 300,000 Elements

Window Size	Slide (ms)	Deque (ms)	Array (ms)
11.0	1340.6	873.0	666.2
51.0	5978.6	2176.5	1017.5
101.0	10061.9	4045.4	1699.7





Building a view

Skip every Nth Element

```cpp
int main() {
    std::vector<int> input_data {1, 4, 2, 8, 9, 11, 12, 14, 18};
    auto pipeline = input_data | std::views::filter([counter = 0](int x) mutable
                {
                    ++counter;
                    if (counter % 3 == 0) {
                        return false;
                    };
                    return true;
                });
    std::print("{}\n", pipeline); // [1, 4, 8, 9, 12, 14]
    return 0;
}
```

- ➔ Stride view won't work here
- ➔ We can do this with a filter though a bit messy

https://godbolt.org/z/E9asG1c1j

Bloomberg

Engineering

Skip every Nth Element

```cpp
int main() {
    std::vector<int> input_data {1, 4, 2, 8, 9, 11, 12, 14, 18};
    auto pipeline = input_data | std::views::filter([counter = 0](int x) mutable
                {
                    ++counter;
                    if (counter % 3 == 0) {
                        return false;
                    };
                    return true;
                });
    std::print("{}\n", pipeline); // [1, 4, 8, 9, 12, 14]
    std::print("{}\n", pipeline); // [1, 4, 2, 9, 11, 14, 18]
    return 0;
}
```

- ➔ Using a **mutable lambda** with a local counter works - but it is messy
- ➔ The counter is local and persists across iterations
- ➔ No clean way to reset the state for multiple iterations
- ➔ Issues with this approach:
  - ◆ Not reusable
  - ◆ Can lead to UB on multiple iterations
  - ◆ Hard to test

https://godbolt.org/z/Mob1cjKh5

Custom View Structure

```
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
class skip_n_view : public std::ranges::view_interface<skip_n_view<InputView>> {
    private:
        InputView input_range_;
        size_t n_;
    public:
        class iterator { /*...*/};

        skip_n_view() = default;
        constexpr skip_n_view(InputView input_range, size_t n) : input_range_(input_range), n_(n) {}

        constexpr iterator begin() const;
        constexpr iterator end() const;
};

int main() {
    std::vector<int> data {3, 4, 5};
    size_t n = 3;
    skip_n_view test{std::views::all(data), n};
    return 0;
}
```

https://godbolt.org/z/vfzq8K4qc

Template Parameters and Constraints

```
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
```

- ➔ We are building something composable so `InputView` represents the range being “piped” into this view
- ➔ The **requires** clause here says that the input needs to be a `common_range`
  - ◆ This means that the same type is used for both the begin and end iterators
- ➔ Other common constraints here might be `random_access_range` if we need indexing (`std::list` would not satisfy this) or `forward_range` which is typical for multi-pass algorithms

Inheriting from the view interface helper

```
class skip_n_view : public std::ranges::view_interface<skip_n_view<InputView>>
```

- ➔ Base class provides us with a lot of standard range behavior for free:
  - ◆ Satisfies `std::ranges::view` concept - tells the ranges library this is a view
  - ◆ Reduces boilerplate (e.g. `empty()`)
  - ◆ Provides common functionality (based on iterator category) such as `back()` (for bidirectional ranges), `[]` (for random-access ranges), etc
- ➔ Notice CRTP - common STL pattern to allow base class access to the derived class' implementation
  - ◆ This allows the base class to implement functionality (e.g., `empty()` by calling our custom functions (such as `begin()` and `end()`)

The iterator…type traits

```
class iterator {
    public:
        //iterator type traits
        using base_iterator = std::ranges::iterator_t<InputView>;
        using iterator_category = std::input_iterator_tag;
        using iterator_concept = std::input_iterator_tag;
};
```

➔ The **iterator category** gives us the traditional (pre-C++20) category of the iterator

- In this case an `input_iterator` is a simple one-pass **read-only** iterator
- Helps the compiler determine which operations are valid
- Mainly for backwards compatibility with older STL algorithms

➔ The **iterator concept** is analogous to the iterator category but meant for ranges (C++20 and beyond)

- Needed by the ranges library to determine what operations your iterator supports
- Enables the compiler to select the appropriate algorithms and prevent invalid usage at compile time

The iterator…more type traits

```
class iterator {
    public:
        //more iterator type traits
        using value_type = std::ranges::range_value_t<InputView>;
        using difference_type = std::ranges::range_difference_t<InputView>;
        using reference = std::ranges::range_reference_t<InputView>;
};
```

➔ `value_type` represents the type that the view produces when iterating

- ◆ Needed for algorithms that create or copy values

➔ `difference_type` is the type that represents differences between iterators

- ◆ Required for any algorithms that need to support concepts like advance, distance or even counting like `std::views::take`

➔ `reference` is the type that dereferencing the iterator gives you

Iterator implementation

```
class iterator {
    public:
        //iterator type traits

        iterator() = default;
        iterator(base_iterator current, base_iterator end, std::size_t n)
            : current_position_(current), end_(end), n_(n) {}

        auto operator*() const {
            return *current_position_;
        }

        iterator& operator++() {
            current_position_++;
            counter_++;
            if (current_position_ != end_ && (counter_ % n_ == 0)) {
                current_position_++;
                counter_++;
            }
            return *this;
        }

        bool operator==(const iterator& rhs) const {
            return current_position_ == rhs.current_position_;
        }

    private:
        base_iterator current_position_;
        base_iterator end_;
        std::size_t n_;
        std::size_t counter_ = 1;
};
```

- ➔ Default constructor usually not required but good practice to include
- ➔ Prefix increment mirrors our lambda logic from before
- ➔ Not equals operator for free

Bloomberg

Engineering

Begin and End

```
    constexpr iterator begin() const {
        return iterator {
            std::ranges::begin(input_range_), std::ranges::end(input_range_), n_
        };
     }
     constexpr iterator end() const {
        return iterator {
            std::ranges::end(input_range_), std::ranges::end(input_range_), n_
        };
     }
```

TechAtBloomberg.com

Bloomberg

Engineering

Putting it all together

```cpp
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
class skip_n_view : public std::ranges::view_interface<skip_n_view<InputView>> {
    /*...*/
 };

int main() {
    std::vector<int> data {1, 4, 2, 8, 9, 11, 12, 14, 18};
    std::size_t n = 3;
    skip_n_view test{std::views::all(data), n};

    for (auto elem : test) {
        std::cout << elem << " "; // // 1 4 8 9 12 14
    }
    std::cout << std::endl;

    return 0;
}
```

https://godbolt.org/z/1je7nfn8c

Bloomberg

Engineering

Is this view composable?

```cpp
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
class skip_n_view : public std::ranges::view_interface<skip_n_view<InputView>> {
    /*...*/
 };

int main() {
    std::vector<int> data {1, 4, 2, 8, 9, 11, 12, 14, 18};
    std::size_t n = 3;
    skip_n_view test{std::views::all(data), n};

    for (auto elem : test) {
        std::cout << elem << " "; // // 1 4 8 9 12 14
    }
    std::cout << std::endl;
    auto pipeline = data | skip_n_view(3); //compiler error

    return 0;
}
```

https://godbolt.org/z/8ee4GMaj8

Bloomberg

Engineering

Inheriting from range adaptor closure

```
namespace views {
    struct skip_n_closure :
std::ranges::range_adaptor_closure<skip_n_closure> {
        std::size_t n;

        constexpr skip_n_closure(std::size_t n_val) : n(n_val) {}

        template <std::ranges::viewable_range R>
        constexpr auto operator()(R&& r) const {
            return skip_n_view{std::forward<R>(r), n};
        }
    };

    struct skip_n_t {
        constexpr skip_n_closure operator()(std::size_t n) const {
            return skip_n_closure(n);
        }
    };

    inline constexpr skip_n_t skip_n{};
}
```

- ➔ `skip_n` - the object users interact with (e.g., `views::skip_n(3)`)
- ➔ `skip_n_t` - the range adaptor object that creates closure instances
- ➔ `skip_n_closure` - the closure object itself which will hold any parameters

https://godbolt.org/z/bnbnq8rWs

Bloomberg

Engineering

Inheriting from range_adaptor_closure

- `inline constexpr skip_n_t skip_n{}` defines one global instance of the range adaptor object
- `views::skip_n(3)` returns a closure object that holds our parameter (n)
- The closure object's overloaded `operator()` takes in an input range and returns an instance of our custom view
- Because the closure object inherits from `range_adaptor_closure` we get the **pipe** operator for free, enabling pipe syntax

Final bit of scaffolding

```
template <typename R>
skip_n_view(R&&, std::size_t) -> skip_n_view<std::views::all_t<R>>;
```

Template deduction guide to help the compiler automatically determine the wrapped view type based on the input range

Is this view composable?

```cpp
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
class skip_n_view : public std::ranges::view_interface<skip_n_view<InputView>> {
    /*...*/
 };

int main() {
    std::vector<int> data {1, 4, 2, 8, 9, 11, 12, 14, 18};
    std::size_t n = 3;
    skip_n_view test{std::views::all(data), n};

    for (auto elem : test) {
        std::cout << elem << " "; // 1 4 8 9 12 14
    }
    std::cout << std::endl;
    auto pipeline = data | views::skip_n(3);
    for (auto elem : pipeline) {
        std::cout << elem << " "; // 1 4 8 9 12 14
    }
    std::cout << std::endl;
    for (auto elem : pipeline) {
        std::cout << elem << " "; // 1 4 8 9 12 14
    }
    return 0;
}
```

https://godbolt.org/z/r9P93crrs

Is this view composable?

```
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
class skip_n_view : public std::ranges::view_interface<skip_n_view<InputView>> {
    /*...*/
 };

int main() {
    std::vector<int> data {1, 4, 2, 8, 9, 11, 12, 14, 18};

    auto pipeline = data | views::skip_n(3) | std::views::take(3);
    for (auto elem : pipeline) { // compiler error
        std::cout << elem << " ";
    }

    return 0;
}
```

TechAtBloomberg.com

Bloomberg

Engineering

Back to the iterator

```
iterator operator++(int) {
    auto tmp = *this;
    ++(*this);
    return tmp;
}
```

- ➔ An input iterator requires **both** the prefix increment (++it) and the postfix increment operator (it++)
- ➔ Some STL views (and algorithms) such as `std::views::take` depend on the postfix increment operator being available
- ➔ Without it, composition with other views may fail

Is this view composable?

```
int main() {
    std::vector<int> data {1, 4, 2, 8, 9, 11, 12, 14, 18};

    auto pipeline = data | views::skip_n(3) | std::views::take(3);
    for (auto elem : pipeline) {
        std::cout << elem << " "; // 1 4 8
    }

    return 0;
}
```

https://godbolt.org/z/avf49xafP

Bloomberg

Engineering

Custom Views for Performance

Recall - Sliding window median value

```
    for (const auto& window_data : input_data |
        std::views::slide(rolling_window_size)) {
            std::vector<double> slide_data {window_data.begin(), window_data.end()};
            std::ranges::nth_element(slide_data, slide_data.begin() + rolling_window_size/2);
            std::cout << slide_data[rolling_window_size/2] << std::endl;
    }
```

➔ Heap allocation on every iteration

➔ Copying the entire window on each iteration

➔ Significant overhead from `slide_view` constructing new windows

Sliding Median - Deque-based implementation

- ➔ This implementation will have the same number of heap allocations (one per window)
- ➔ Same amount of copies but copies will come from the deque rather than a new view
- ➔ It turns out getting a new window from `std::views::slide` involves a significant amount of overhead
  - ◆ Using a deque greatly improves performance since fetching a new window just involves removing one element and adding another
  - ◆ Both are O(1) operations

Deque Implementation

```
template <typename InputView>
requires std::ranges::common_range<InputView>
class sliding_median_view_deque : public
std::ranges::view_interface<sliding_median_view_deque<InputView>> {
private:
    InputView input_range_;
    std::size_t window_size_;

public:
    class iterator {
    public:
        // Iterator type traits
        using base_iterator = std::ranges::iterator_t<InputView>;
        using iterator_category = std::input_iterator_tag;
        using iterator_concept = std::input_iterator_tag;

        using value_type = std::ranges::range_value_t<InputView>;
        using difference_type = std::ranges::range_difference_t<InputView>;

        using reference = value_type; // Return by value for the median

        /*...*/
```

```
private:
        base_iterator current_;
        base_iterator end_;
        std::size_t window_size_;
        std::deque<value_type> window_;
```

- ➔ Notice how the reference here now needs to be passed by value
- ➔ In prior custom view implementations, dereferencing the iterator gave you a reference to an existing element in the **underlying range**
- ➔ In this view when dereferencing the iterator we get a **new value** that does not exist in the underlying range

Deque Implementation

Constructor

```
    iterator(base_iterator begin, base_iterator end, std::size_t window_size)
        : current_(begin), end_(end), window_size_(window_size) {

        if (begin == end) {
            return;
           }

        auto count = 0;
        auto it = begin;
        while (it != end && count < window_size) {
            window_.push_back(*it);
            ++it;
            ++count;
        }

        if (count < window_size) {
            current_ = end_;
        }
    }
```

Deque Implementation

Dereference Operator

```
    auto operator*() const {
        std::vector<value_type> sorted(window_.begin(), window_.end());
        const size_t mid_idx = sorted.size() / 2;

        std::nth_element(sorted.begin(), sorted.begin() + mid_idx, sorted.end());

        if (sorted.size() % 2 == 0) {
            // Even number of elements - find the other middle element
            // Find the max of the lower half
            auto max_it = std::max_element(sorted.begin(), sorted.begin() + mid_idx);
            auto lower_median = *max_it;
            auto higher_median = sorted[mid_idx];
            return (lower_median + higher_median) / 2;
        } else {
            // Odd number of elements - return the middle one
            return sorted[mid_idx];
        }
    }
```

Deque Implementation

Prefix Increment Operator

```
        iterator& operator++() {
            ++current_;

            if (current_ + window_size_ > end_) {
                current_ = end_;
                window_.clear();
                return *this;
            }

            window_.pop_front();
            window_.push_back(*(current_ + window_size_ - 1));

            return *this;
        }
```

Sliding Median - Circular Array implementation

- ➔ Eliminates heap allocations by using fixed-size arrays allocated on the **stack**
- ➔ Improves cache locality compared to a deque
- ➔ Similar to a deque avoids creating subranges on every iteration
- ➔ Unlike a deque, a `std::array` would require shifting elements on every window slide
  - ◆ An O(M) operation
  - ◆ Avoided by using a **circular array**
- ➔ **<u>Downsides</u>**:
  - ◆ Code bloat if this is done for many window sizes
  - ◆ Need to know size at compile time
  - ◆ Does not avoid copy costs

Circular Array Implementation

```
template <typename InputView, std::size_t WindowSize>
requires std::ranges::common_range<InputView>
class sliding_median_view_array : public std::ranges::view_interface<sliding_median_view_array<InputView,
WindowSize>> {
private:
    InputView input_range_;

public:
    class iterator {
    public:
        using base_iterator = std::ranges::iterator_t<InputView>;
        using iterator_category = std::input_iterator_tag;
        using iterator_concept = std::input_iterator_tag;

        using value_type = std::ranges::range_value_t<InputView>;
        using difference_type = std::ranges::range_difference_t<InputView>;

        using reference = value_type;
        /*...*/
    private:
        base_iterator current_;
        base_iterator end_;
        std::array<value_type, WindowSize> window_{};
        std::size_t window_size_ = 0;
        std::size_t oldest_idx_ = 0; // Track the position of the oldest element
    };
```

Circular Array Implementation

Constructor

```
iterator(base_iterator begin, base_iterator end)
            : current_(begin), end_(end) {

            if (begin == end) {
                return;
            }

            auto count = 0;
            auto it = begin;
            while (it != end && count < WindowSize) {
                window_[count] = *it;
                ++it;
                ++count;
            }

            if (count < WindowSize) {
                current_ = end_;
            }

            window_size_ = count;
            oldest_idx_ = 0;
        }
```

Circular Array Implementation

Dereference Operator

```
auto operator*() const {

    std::array<value_type, WindowSize> sorted;
    std::copy_n(window_.begin(), window_size_, sorted.begin());

    const size_t mid_idx = window_size_ / 2;

    std::nth_element(sorted.begin(), sorted.begin() + mid_idx, sorted.begin() + window_size_);

    if (window_size_ % 2 == 0) {
        auto max_it = std::max_element(sorted.begin(), sorted.begin() + mid_idx);
        auto lower_median = *max_it;
        auto higher_median = sorted[mid_idx];
        return (lower_median + higher_median) / 2;
    } else {
        return sorted[mid_idx];
    }
}
```

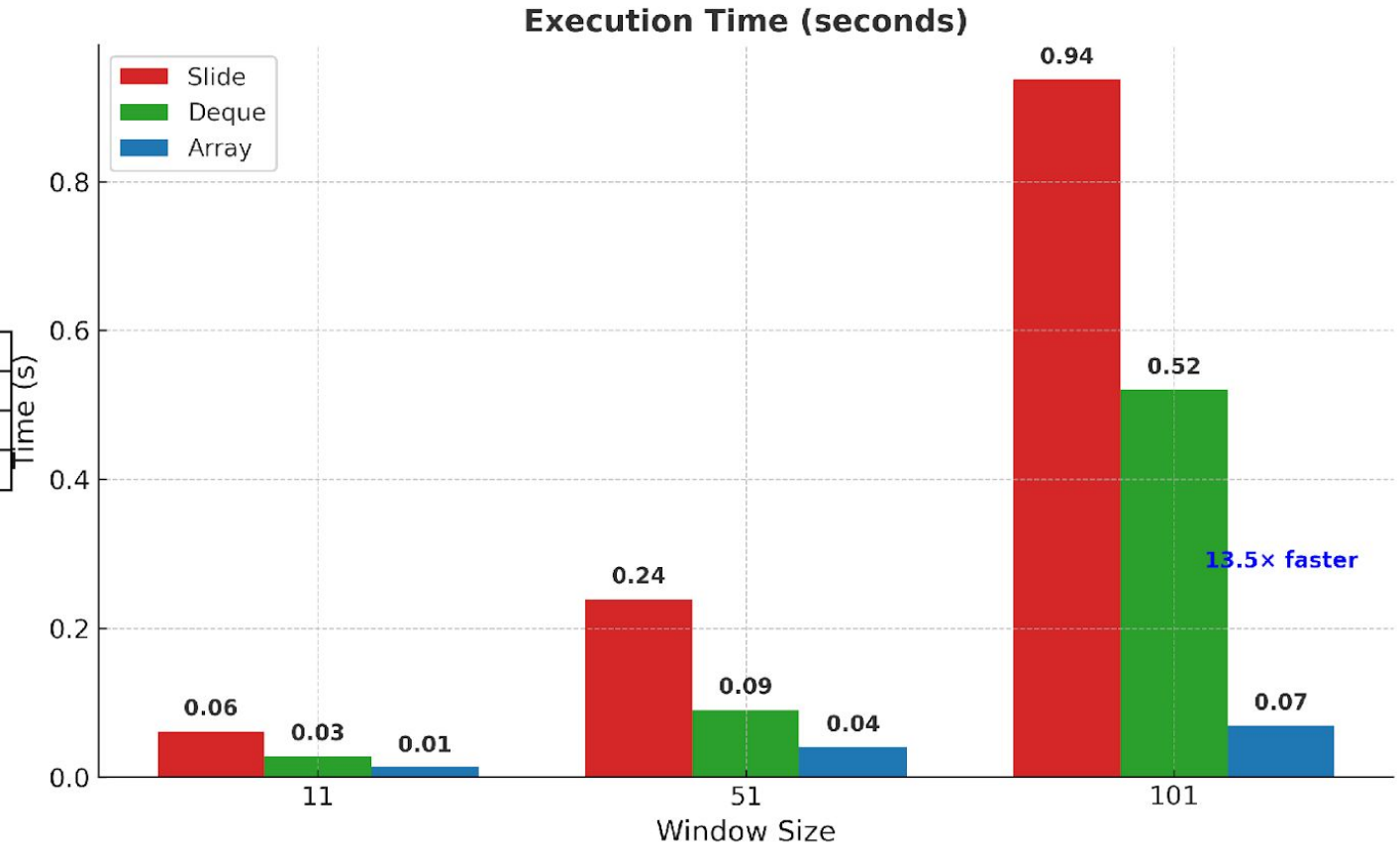
Array Implementation

Prefix Increment Operator

```
        iterator& operator++() {
            ++current_;

            if (current_ + WindowSize - 1 >= end_) {
                current_ = end_;
                window_size_ = 0;
                return *this;
            }

            // Replace oldest element with the new element
            window_[oldest_idx_] = *(current_ + WindowSize - 1);
            // Update oldest index
            oldest_idx_ = (oldest_idx_ + 1) % WindowSize;

            return *this;
        }
```

Sliding Median Benchmark – 30,000 Elements

Window Size	Slide (ms)	Deque (ms)	Array (ms)
11.0	61.6	28.3	13.0
51.0	239.2	90.5	40.4
101.0	937.2	520.4	69.2



Filtering Based on External State

External Data

```
// List of stocks we are monitoring
std::vector<std::string> stocks = {"MSFT", "TSLA", "IBM", "NVDA", "PEP", "AML", "AAPL", "APP"};

// External data: stream of data
// Stocks mapped to some threshold (e.g., Sharpe Ratio)
std::unordered_map<std::string, double> stock_data = {
    {"MSFT", 1.1},
    {"TSLA", 1.0},
    {"NVDA", 1.6},
    {"PEP", 1.8},
    {"AAPL", 2.1},
    {"APP", 2.3}
};
```


Filtering with external state

```
double threshold = 1.5;

auto pipeline = stocks | std::views::filter([&stock_data, threshold](const std::string& stock) {
    if (stock_data.contains(stock) && stock_data[stock] >= threshold) return true;
    return false;
});
for (const auto& stock : pipeline) {
    std::cout << stock << " ,"; // NVDA, PEP, AAPL, APP
}
```

- ➔ Filtering condition is now tied to **external state**
- ➔ Encapsulation is broken - filter logic depends on variables outside of the pipeline
  - ◆ This creates hidden dependencies
- ➔ Testing is challenging - requires setup of both shared state and the pipeline

https://godbolt.org/z/1Eez5zbn1

Bloomberg

Engineering

Filtering with external state

```
double threshold = 1.5;

auto pipeline = stocks | std::views::filter([&stock_data, threshold](const std::string& stock) {
    if (stock_data.contains(stock) && stock_data[stock] >= threshold) return true;
    return false;
});

stock_data["AAPL"] = 0.5;
for (const auto& stock : pipeline) {
    std::cout << stock << " ,"; // NVDA, PEP, APP
}
```

- ➔ Views are **lazy** - execution happens at traversal, not creation
- ➔ External state may change between creation and use
- ➔ Behavior becomes **non-deterministic** and coupled to external timing
- ➔ Custom views do not resolve this, but they make these **dependencies explicit**

https://godbolt.org/z/bhPYPPvPM

Custom views for external state

```
template <std::ranges::view InputView> requires std::ranges::common_range<InputView>
class stock_threshold_view : public std::ranges::view_interface<stock_threshold_view<InputView>> {
private:
    InputView base_;
    std::unordered_map<std::string, double> const& stock_data_;
    double threshold_;

    // Iterator class that handles filtering
    class iterator {
        /*...*/
public:
    stock_threshold_view(InputView base,
                         std::unordered_map<std::string, double> const& stock_data,
                         double threshold)
        : base_(std::move(base)), stock_data_(stock_data), threshold_(threshold) {}

    auto begin() { /*...*/}

    auto end() { /*...*/}
};
```

Custom views for external state

```
void find_next_valid() {
    while (current_ != end_) {
        const auto& stock = *current_;
        if (stock_data_->contains(stock) && (*stock_data_).at(stock) >= threshold_) {
            return;
        }
        ++current_;
    }
}
```

➔ Used by constructor and increment operator to skip invalid elements

➔ Filters out stocks (i) with no data or (ii) that don’t meet the threshold

➔ Ensures iterator **always** points to the next valid element

Bloomberg

Engineering

Iterator constructor and prefix increment

```
iterator(base_iterator current, base_iterator end,
         std::unordered_map<std::string, double> const* stock_data,
         double threshold)
    : current_(current), end_(end),
      stock_data_(stock_data), threshold_(threshold) {
    // Find the first valid element
    find_next_valid();
}
```

```
iterator& operator++() {
    if (current_ != end_) {
        ++current_;
        find_next_valid();
    }
    return *this;
}
```

Bloomberg

Engineering

Using the view

```
double threshold = 1.5;

auto pipeline = stocks | views::stock_threshold(stock_data, threshold);

std::cout << "Stocks above threshold: ";
for (const auto& stock : pipeline) {
    std::cout << stock << ", "; // NVDA, PEP, AAPL, APP
}
std::cout << std::endl;

// Test reference semantics
stock_data["AAPL"] = 0.5;

std::cout << "After changing AAPL value: ";
for (const auto& stock : pipeline) {
    std::cout << stock << ", "; // NVDA, PEP, APP
}
std::cout << std::endl;
```

- ➔ State is now **explicitly encapsulated** in the view
- ➔ The pipeline is easier to:
  - ◆ Test
  - ◆ Debug
  - ◆ Reuse

https://godbolt.org/z/qnTdozoo6

Comparing the two approaches

	Lambda	Custom View
Encapsulation	Logic and state are separate	Logic and state are encapsulated in the view
Reusability	Limited; requires writing new lambdas	Easily reusable with different ranges and data
Testing	External state and lambda must be setup correctly	View can be tested in isolation
Debugging	Requires inspecting lambda closures	Debuggable with standard tools

Best Practices

When does it make sense to build custom views?

- ➔ Standard views do not cleanly support your use case
- ➔ Implementing domain specific logic
- ➔ To create **reusable**, composable transformations of custom logic
- ➔ Encapsulate external state
- ➔ Optimizing performance-critical code paths where custom logic can outperform STL solutions (e.g., sliding median)

Best Practices

What should we keep in mind when building custom views?

- ➔ Be explicit about writing out iterator traits
- ➔ Ensure the iterator satisfies all requirements for its category
- ➔ Missing traits or requirements can cause subtle bugs, especially in regards to composition
- ➔ If your view is meant to be composable, prefer capturing any external state by value
- ➔ The STL assumes views are self-contained and copyable

Best Practices

- ➔ If your view depends on external state, make that dependency explicit in the API
- ➔ Use `std::ranges::range_adaptor_closure` to support composition
- ➔ Keep your implementation as simple as possible
  - ◆ Views are complicated enough!

Pitfalls

- ➔ Views are non-owning - the underlying range should outlive the view
- ➔ Otherwise this could lead to **dangling references**
- ➔ Use `std::ranges::to` (or `std::copy_n`) to materialize views into owning containers when needed
- ➔ Pipelines are re-evaluated every time they are traversed
  - ◆ This can end up being significantly less efficient than eager evaluation

Pitfalls

- ➔ Custom views may not compose as effortlessly as standard range adaptors
- ➔ Capturing external state by reference can cause composition failures
- ➔ Inline pipelines may fail if your view is not copyable or self-contained

Questions?