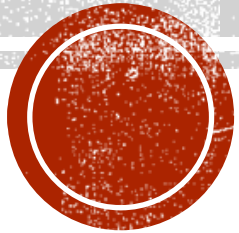
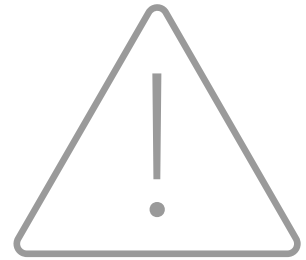
IT'S EASY TO USE RANGES, IF YOU KNOW HOW

Marco Arena



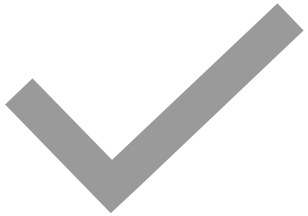
NOT IN THIS SESSION

- **Full dissertation** on ranges' internals
- **Customization**: how to build your own views and actions
- Meticulous **compliancy** with C++20 (I will use range-v3)
- **Performance** evidences and tests



IN THIS SESSION

- A gentle introduction to ranges
- Fundamental principles, patterns and tips
- Examples and game-based practice (**shirts up for grabs!**)

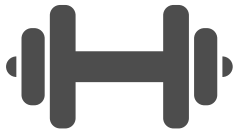


KNOWING IS NOT ENOUGH, WE MUST APPLY

25% discount:

ITALIANCPP25

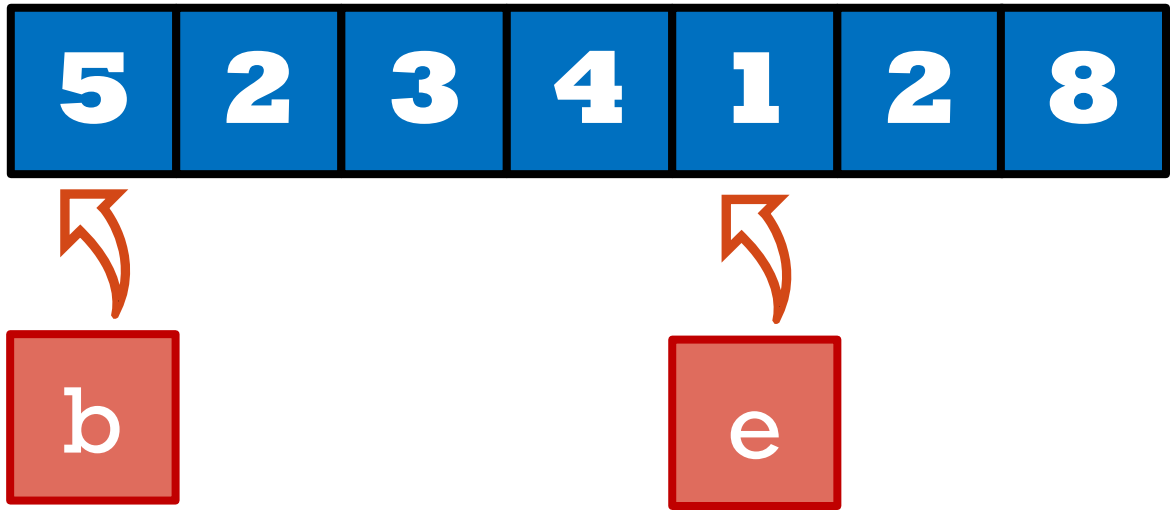
https://italiancpp.org/ranges



A GENTLE INTRODUCTION TO RANGES

A long time ago in a galaxy far, far away. . . .

THE C++ TRIAD



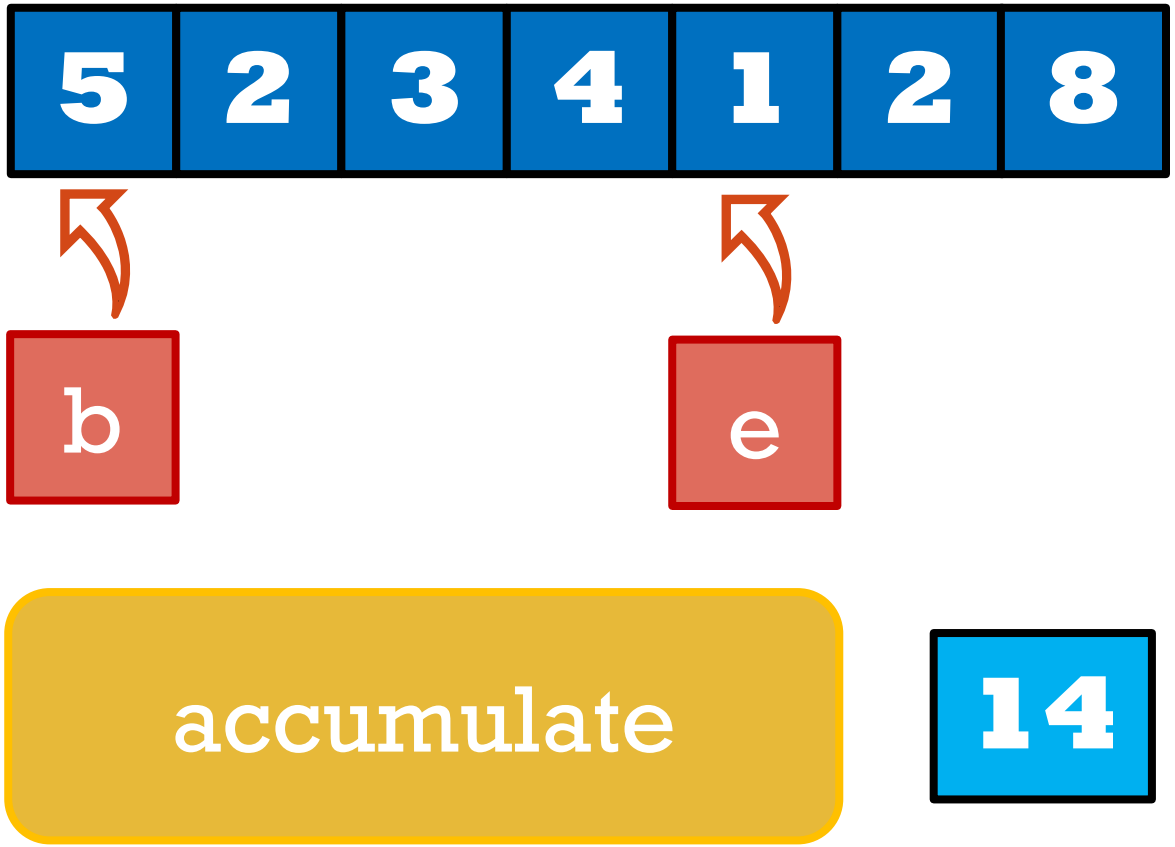
accumulate

Containers

Iterators

Algorithms

THE C++ TRIAD



Containers

Iterators

Algorithms

LIMITATIONS OF THE C++ TRIAD

#1 Verbosity / Iterators mismatch errors

Every algorithm needs both `begin` and `end` explicitly.

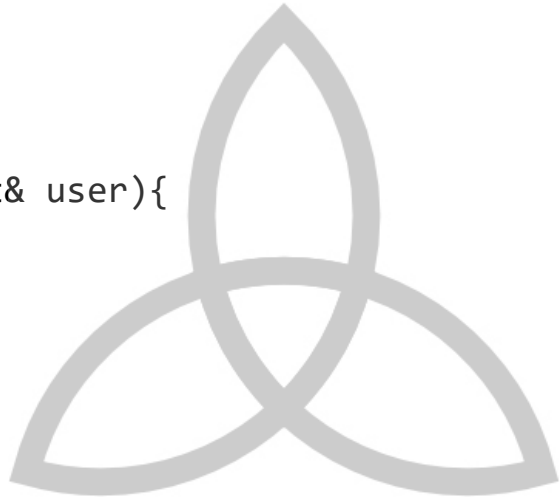
```
vector v = {...}; vector k = {...};

sort(begin(v), end(v)); // I am too lazy for this

sort(begin(v), end(k)); // ooops
```

```
sort(v); // why not just this?
```

LIMITATIONS OF THE C++ TRIAD

#2 Not easily and fluently composable

```
const vector<user> users = {{1, 10}, {2, 18}, {3, 20}, {4, 17}};

vector<user> filteredUsers;

copy_if(begin(users), end(users), back_inserter(filteredUsers), [](auto const& user) {
        return user.age >= 18;
});

vector<long> ids;

transform(begin(filteredUsers), end(filteredUsers), back_inserter(ids), [](auto const& user){
        return user.id;
});

// ids: 2, 3
```



A BIT OF HISTORY

- 2004: Boost.Range
- 2010: Boost.Range 2.0
- 2013: Eric Niebler's first commit to range-v3
- 2014: Proposal "Ranges for the Standard Library"
- 2017: Ranges TS
- 2018: Ranges merged to C++20

WHAT IS A RANGE?

- A range is any type providing `begin/end` iteration:

std::ranges::range

Defined in header <ranges>

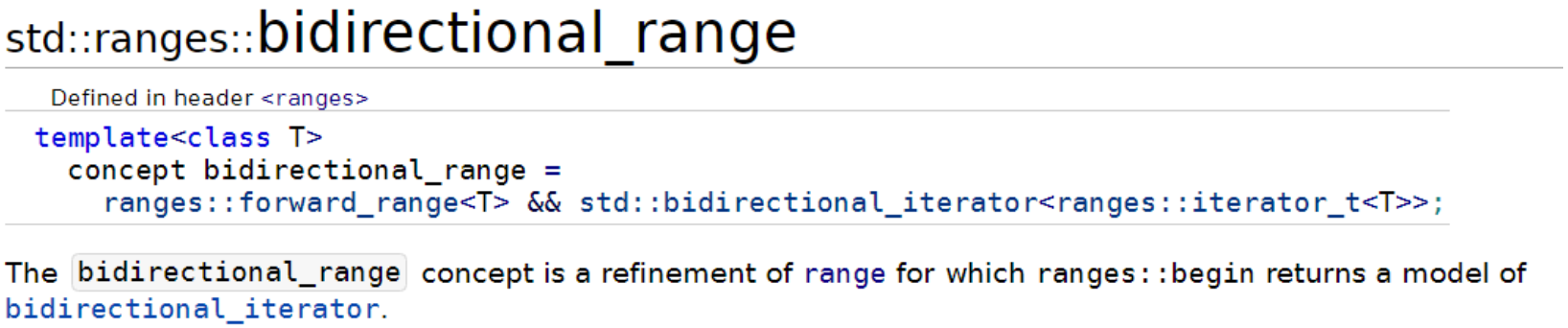
```
template< class T >
concept range = requires(T& t) {
  ranges::begin(t); // equality-preserving for forward iterators
  ranges::end  (t);
};
```

YOU ARE USING RANGES ALREADY

```
for (auto i : rng)
{
    ... do something with i...
}
```

WHAT IS A RANGE?

- Depending on the *capabilities* of the underlying iterator, a range might express additional **refinements**. For example**:**



std::ranges::bidirectional_range

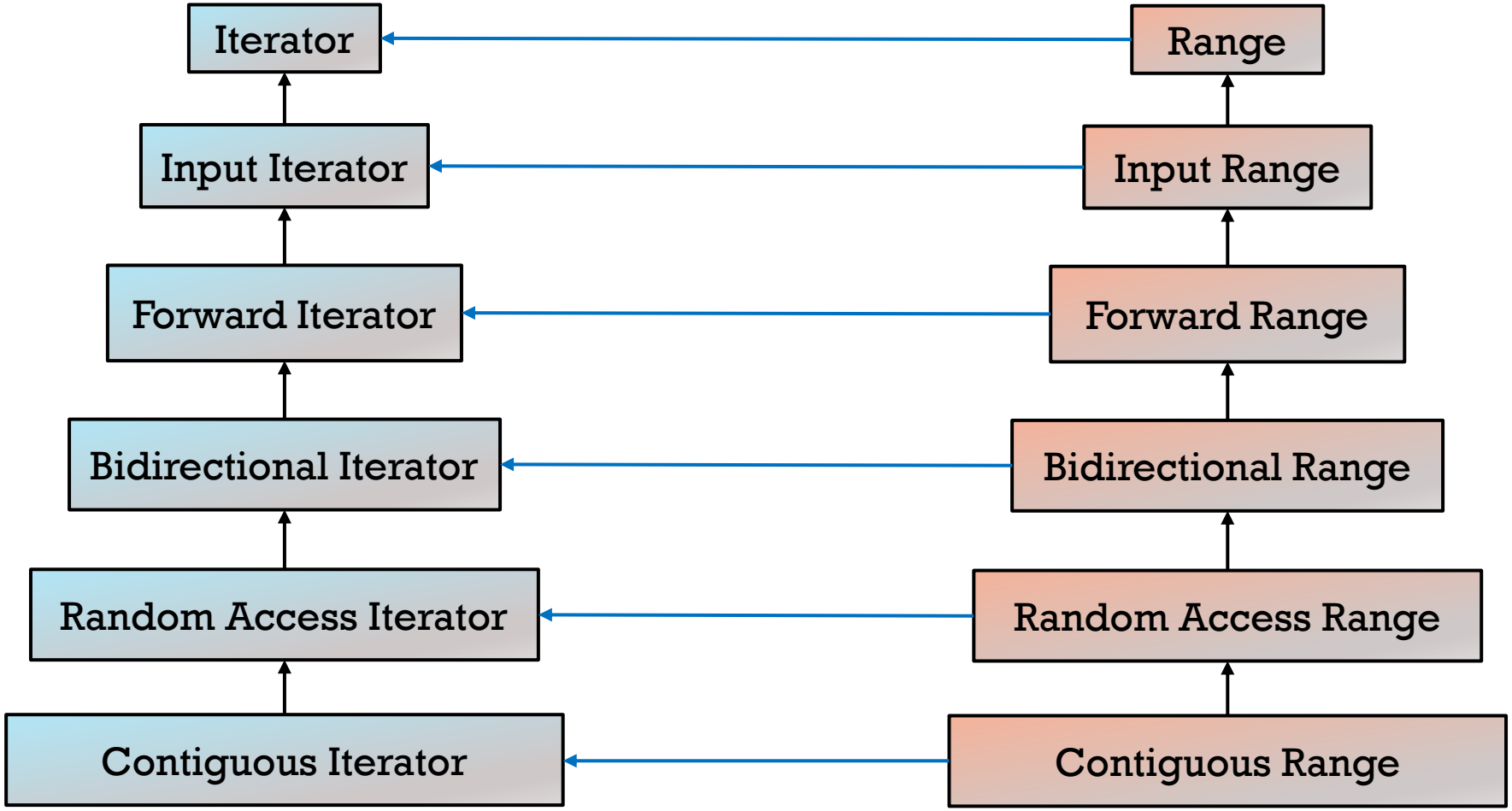
Defined in header <ranges>

```
template<class T>
  concept bidirectional_range =
    ranges::forward_range<T> && std::bidirectional_iterator<ranges::iterator_t<T>>;
```

The `bidirectional_range` concept is a refinement of range for which ranges::begin returns a model of bidirectional_iterator.

- Basically, **range concepts** go hand in hand with **iterator categories**

RANGE CONCEPTS



NEW!

RANGE-BASED OVERLOADS

- Ranges provide new **overloads** that mirror traditional STL algorithms
- These are **constrained with Concepts** and accept both **range arguments**, and also **iterator-sentinel** pair arguments (and also **projections**).

```
vector v = {25, 8, 5, 9, 1};
sort(v);

auto where = lower_bound(users, "mar", less<>{}, &user::name);
```


"That's all Folks!"

WHAT??

MY COUSIN SHOWED ME GORGEOUS PIPELINES!

DIFFERENT WAYS TO IMPLEMENT A RANGE

The "old" iterator-pair design has been relaxed to allow `ranges::end(t)` to have a different type than `ranges::begin(t)`.

This allows to capture other design models, like:

- Iterator + count
- Iterator + predicate

The end iterator is now commonly named **sentinel**.

A RANGE IS ANY "ITERABLE" TYPE

- Ranges can be **adapted** to behave differently by other range types
- For example, a string could be adapted by a special *filter range* that return only upper chars when we iterate on it:

```
std::string s = "abcReAghNGnoEp";
auto filtered = filter(s, [](auto c) { return isupper(c); });
std::cout << filtered; // RANGE
```

- Lots of these *special* range types are provided by the library

NEW!

IT'S A VIEW!

```
vector v = {1,2,3,4};

// reminder: filter = keep
auto view = views::filter(v, [](auto i) { return i%2 == 0; })

          | views::transform([](auto i) { return std::to_string(i); });


// consumes all the data
cout << view; // aka: for (auto s : view) { cout << ... }
```

WHAT IS A VIEW?

- A view is like an **expression** ready to be evaluated
- It does nothing until we iterate on it
- Anytime we "pull" one element from it, the evaluation happens **just for that element**

```
auto view = views::filter(v, [](auto i) { return i%2 == 0; })
          | views::transform([](auto i) { return std::to_string(i); });
```

*begin(view) → *begin(transform(filter(v)))

WHAT IS A VIEW?

NEW!

- A range that can be created/copied/moved/assigned in **constant time**
- Never owns nor modifies data, just **describes** the intended operation
- **Lazily evaluated** (generates its elements on demand, when it gets iterated)
- Fluently **composable** in a pipeline
- To create views, the library provides:
  - **Adaptors**: from existing ranges - e.g. `views::cycle(rng)`
  - **Factories**: from something else – e.g. `views::iota(0)`

WHAT IS AN ACTION?

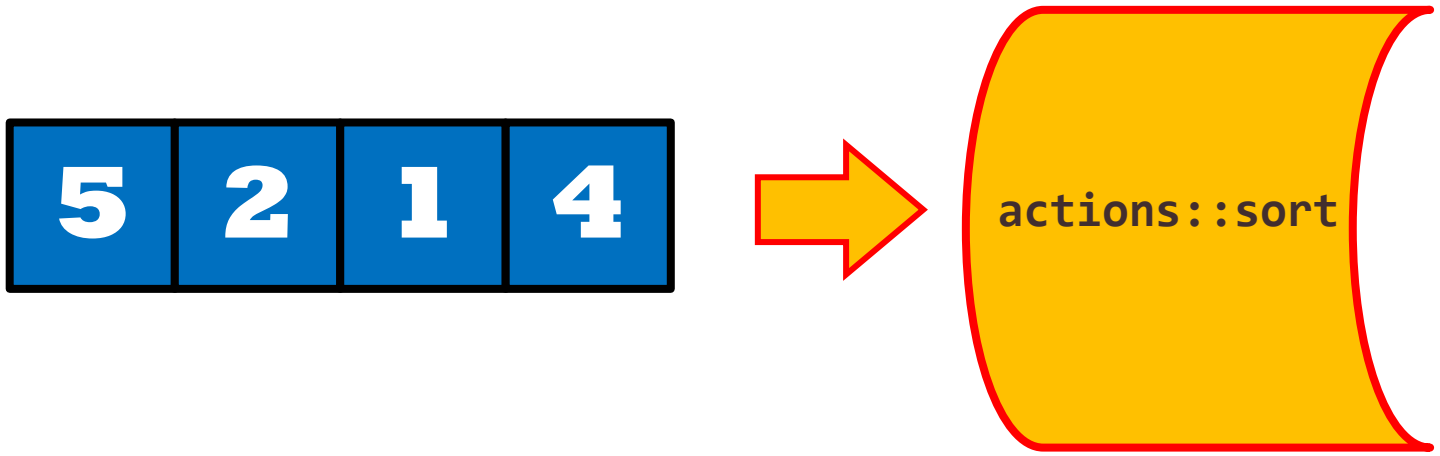
Not in C++20

- Comes into play when we need to **mutate** data
- Works on **materialized** ranges only (ultimately referring to some data)
- Still compose (**eagerly** process data in-place and then pass on to the next step)

```
std::vector<int> v = {10, 2, 10, 3, 2, 1, 1, 21, 5};

v = std::move(v) | actions::sort | actions::unique;

// or, in-place

v |= actions::sort | actions::unique;

// equivalent to

actions::unique(actions::sort(v));
```

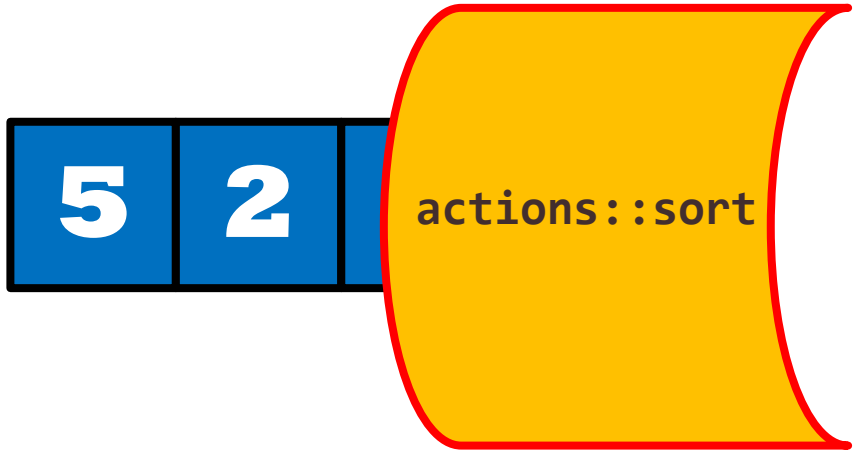
WHAT IS AN ACTION?

Not in C++20



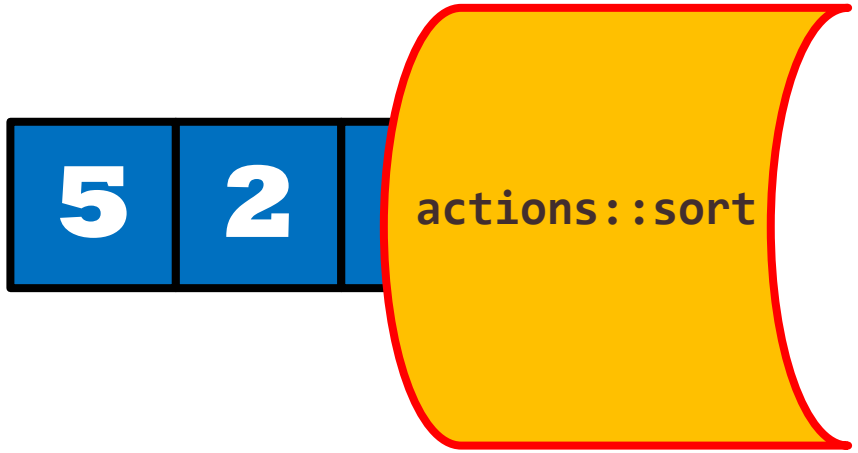
WHAT IS AN ACTION?

Not in C++20



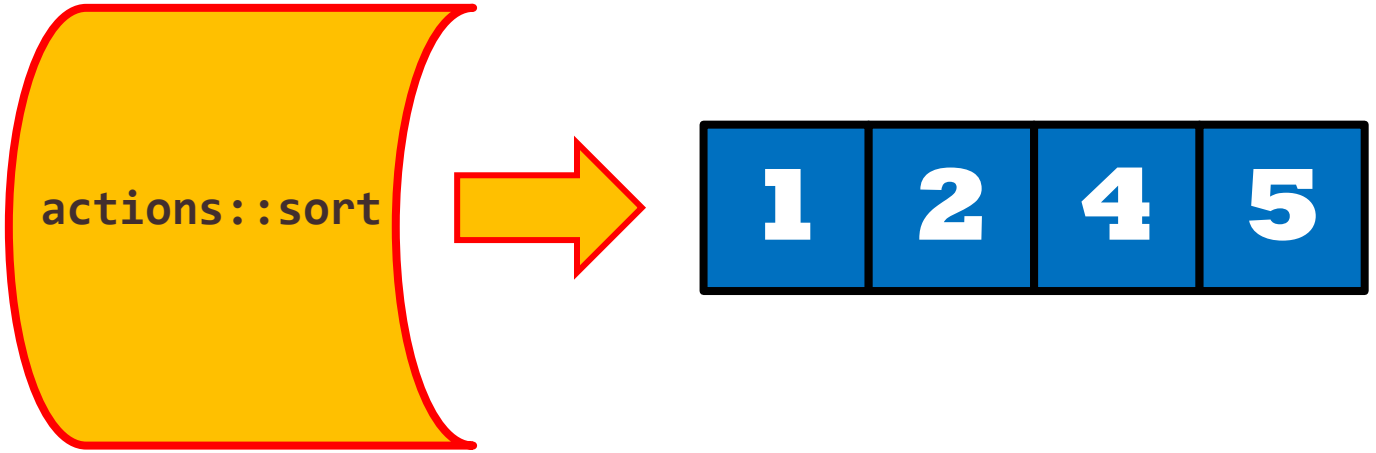
WHAT IS AN ACTION?

Not in C++20



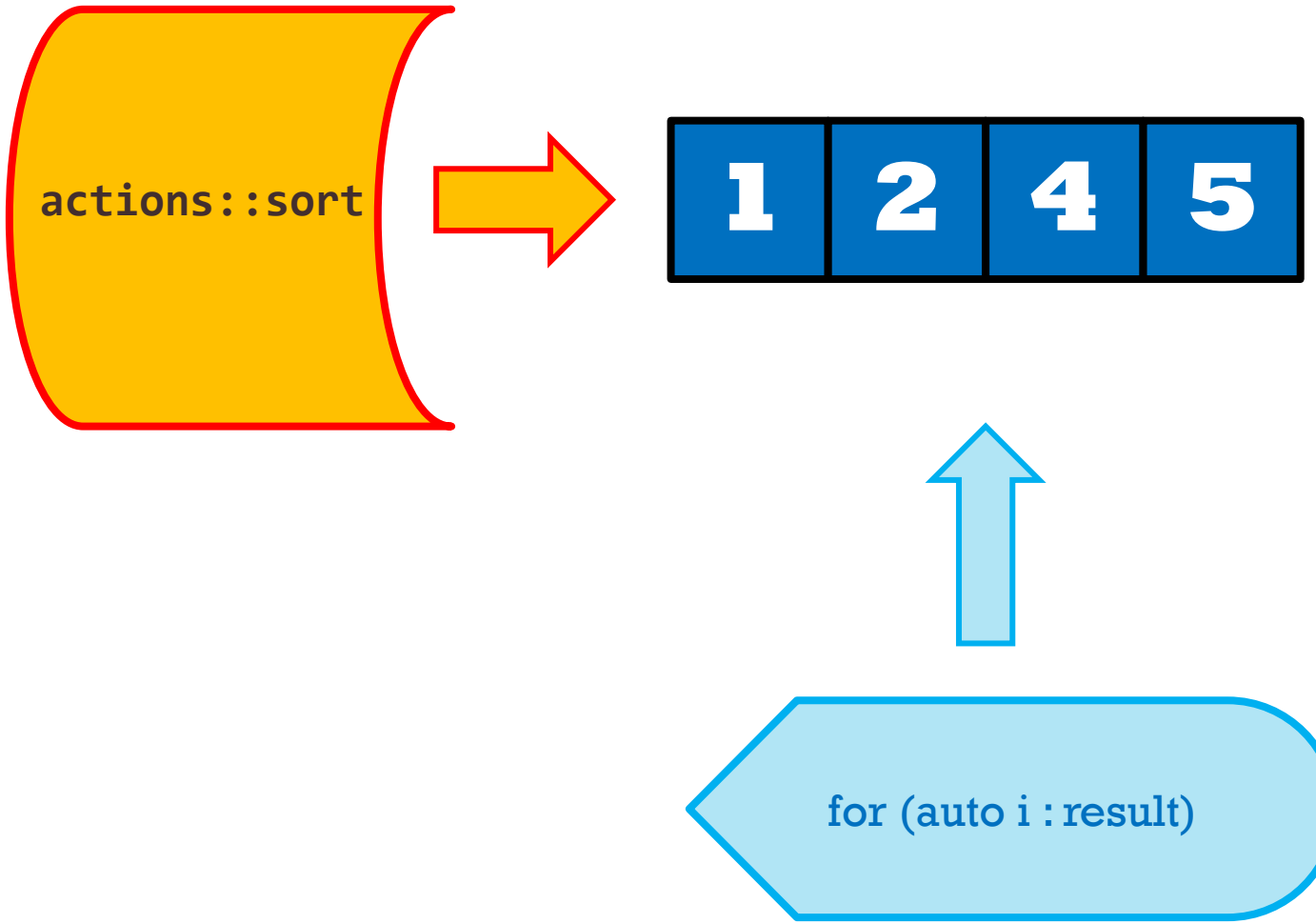
WHAT IS AN ACTION?

Not in C++20



WHAT IS AN ACTION?

Not in C++20

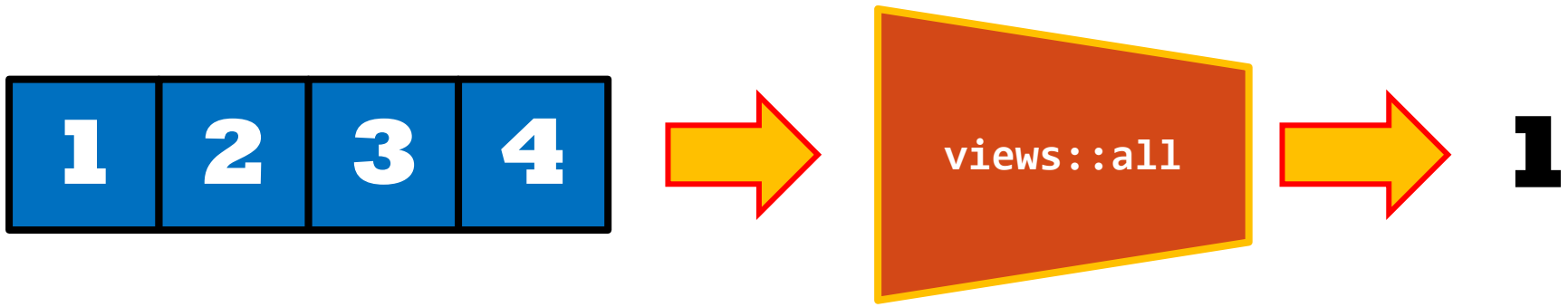


WARM-UP PRINCIPLES

Let's understand the foundations

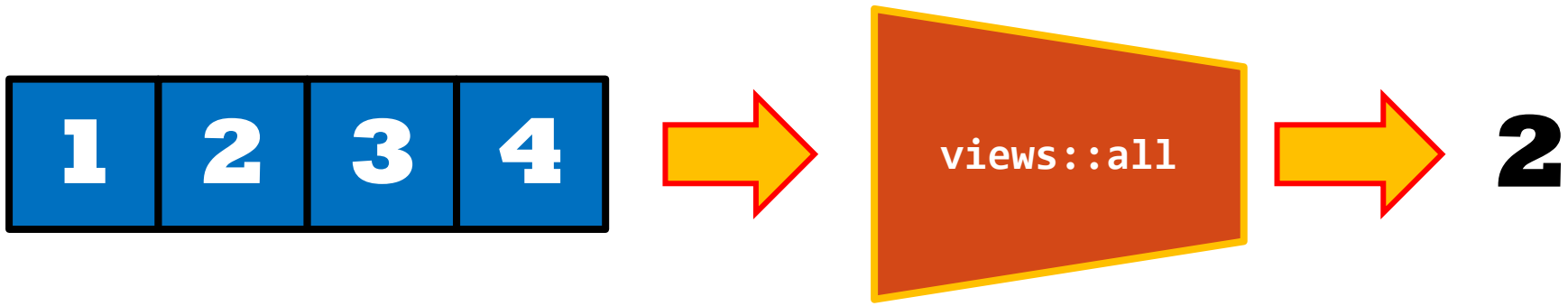
#1 UNDERSTANDING THE FLOW

- Views are evaluated on demand, **one element at a time**



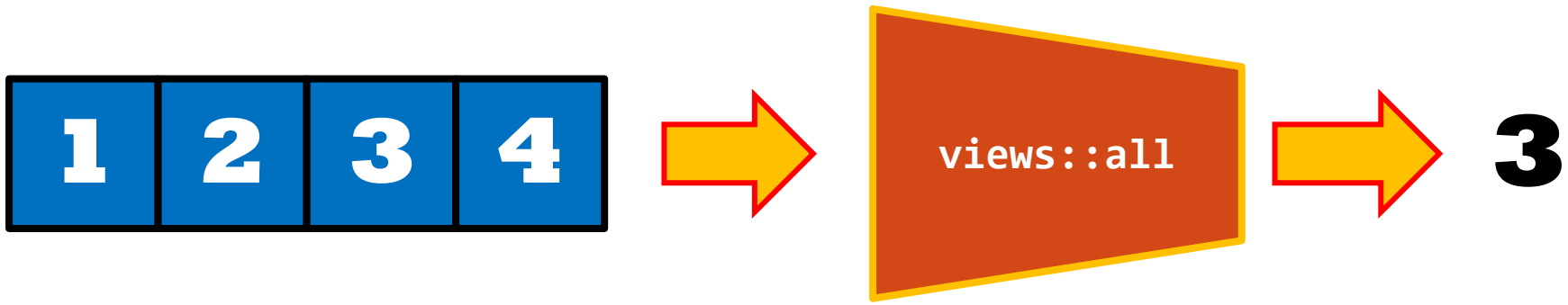
#1 UNDERSTANDING THE FLOW

- Views are evaluated on demand, **one element at a time**



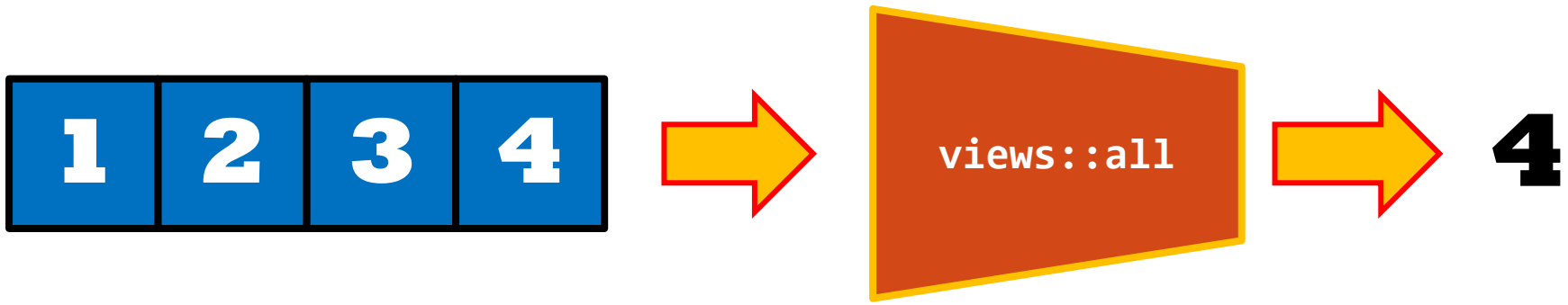
#1 UNDERSTANDING THE FLOW

- Views are evaluated on demand, **one element at a time**



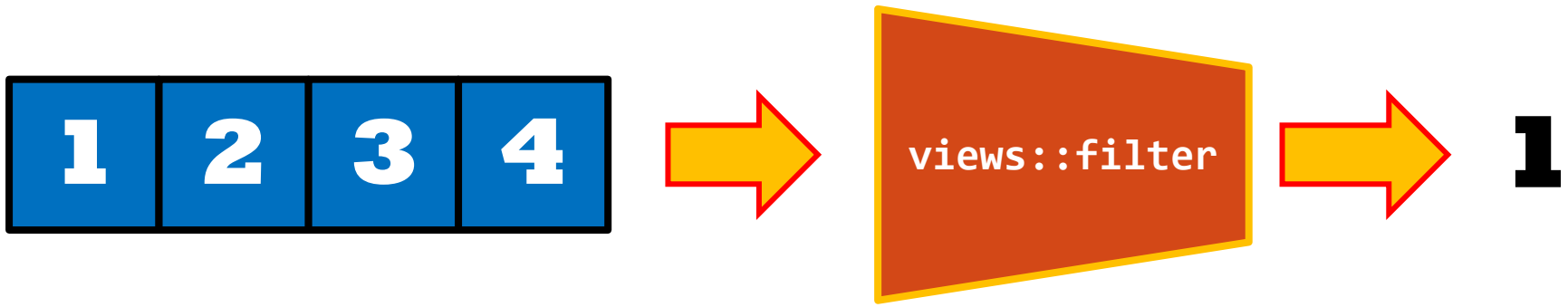
#1 UNDERSTANDING THE FLOW

- Views are evaluated on demand, **one element at a time**



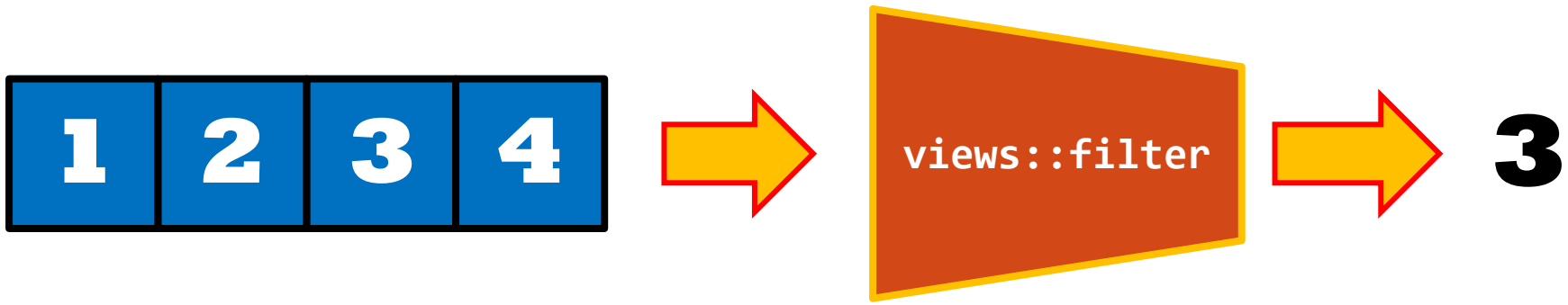
#1 UNDERSTANDING THE FLOW

- Views are evaluated on demand, **one element at a time**



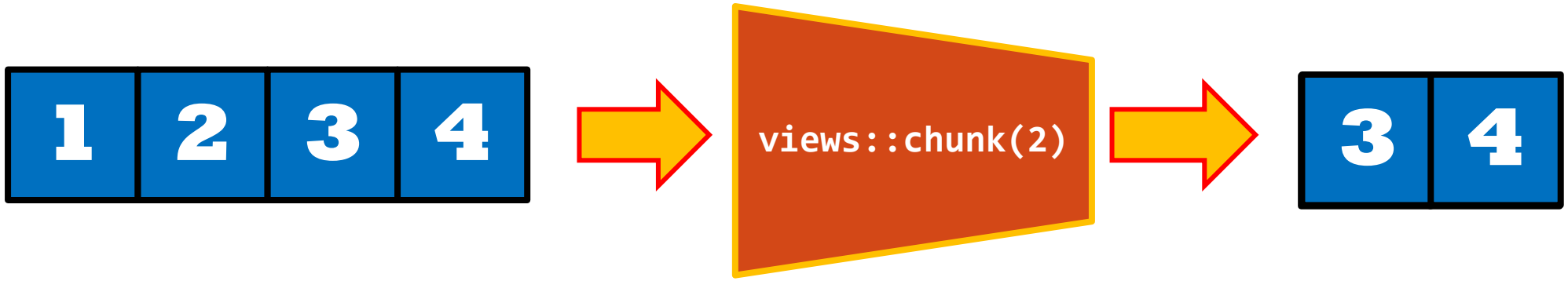
#1 UNDERSTANDING THE FLOW

- Views are evaluated on demand, **one element at a time**



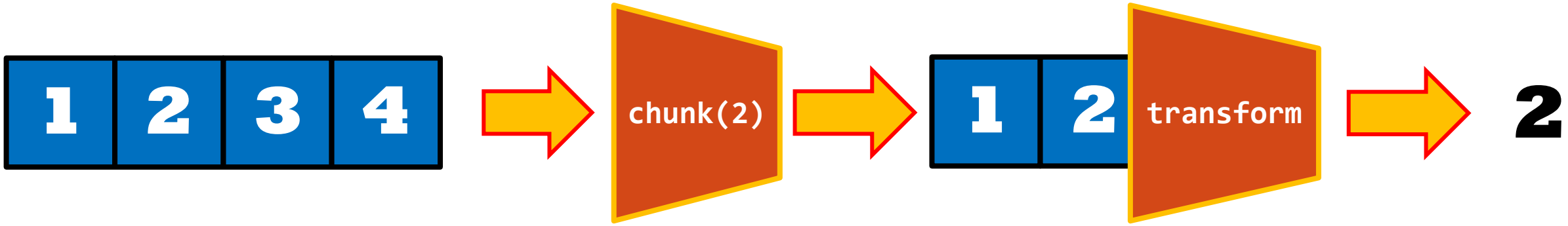
#1 UNDERSTANDING THE FLOW

- Be aware of the output **cardinality** of each view



#1 UNDERSTANDING THE FLOW

- Be aware of the output **cardinality** of each view



#1 UNDERSTANDING THE FLOW

```
std::vector input = {1, 2, 3, 4};

auto lengths = input          // [1, 2, 3, 4]

        | views::chunk(2) // [[1, 2], [3, 4]]

        | views::transform([](auto sub) { return distance(sub); }); // [2, 2]


std::cout << lengths; // [2, 2]
```

#1 UNDERSTANDING THE FLOW

```
std::vector input = {1, 2, 3, 4};

auto lengths = input          // [1, 2, 3, 4]

        | views::chunk(2) // [[1, 2], [3, 4]]

        | views::transform([](auto sub) { return distance(sub); }); // [2, 2]


std::cout << front(lengths); // 2

std::cout << *begin(lengths); // useful when...?
```

#1 UNDERSTANDING THE FLOW

```
std::vector<std::vector<int>> matrix = { {1,2,3}, {4,5,6} };

auto flatten = matrix | views::join;


std::cout << front(flatten); // 1

std::cout << flatten; // [1, 2, 3, 4, 5, 6]
```

#1 UNDERSTANDING THE FLOW

https://www.menti.com/72716ntt16

Go to **www.menti.com** and use the code **3096 7444**

#1 UNDERSTANDING THE FLOW

```
std::vector<std::vector<int>> matrix = { {1,2,3}, {4,5,6} };

auto flatten = matrix | views::join | views::chunk(2);


std::cout << front(flatten); // ?
```

#1 UNDERSTANDING THE FLOW

```
std::vector<std::vector<int>> matrix = { {1,2,3}, {4,5,6} };

auto flatten = matrix | views::join | views::chunk(2);


std::cout << front(flatten); // [1, 2]

std::cout << flatten; // [[1, 2], [3, 4], [5, 6]]
```

#2 YOU ARE USING RANGES ALREADY

- Impersonate a range-based for loop to **visualize** your pipelines

```
auto rng = v | views::A | views::B | views::C;

for (auto i : v) {
  ...ith = views::C(views::B(views::A(i)))
}
```

#2 YOU ARE USING RANGES ALREADY

- Eventually, **break down** the pipeline into intermediate views

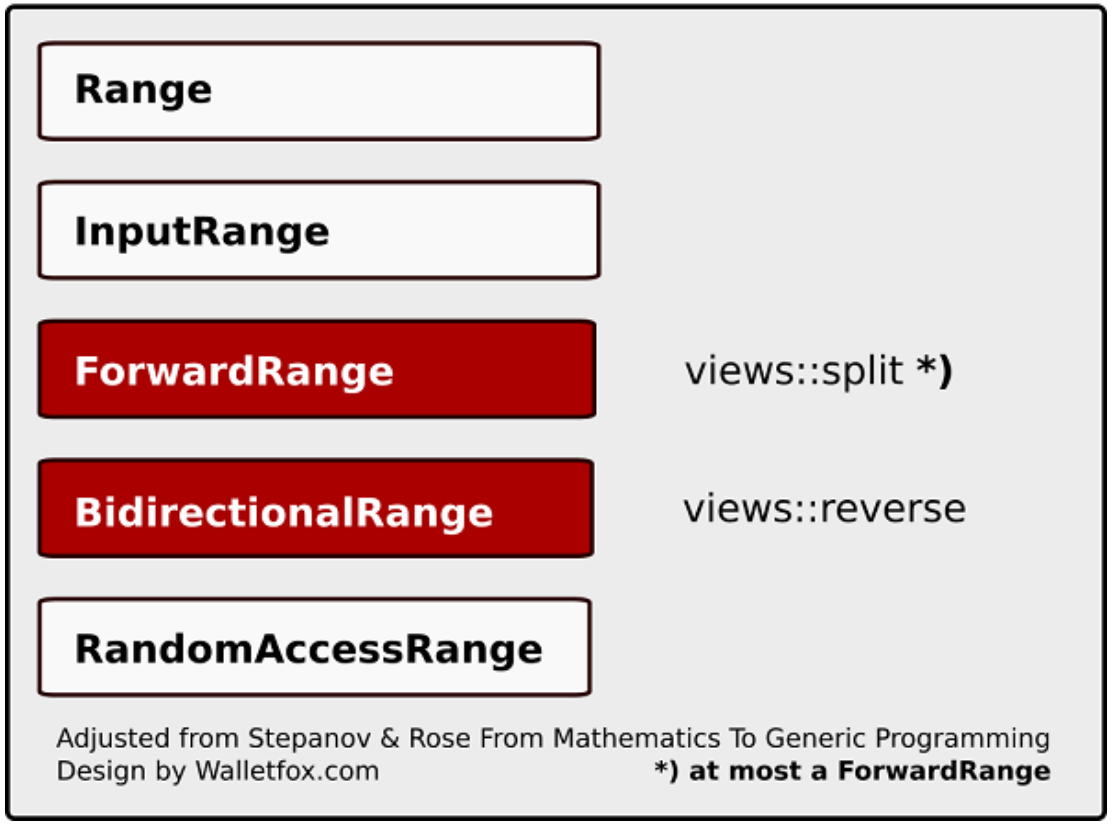
```
auto view1 = v | views::A;

std::cout << view1 << "\n";


auto view2 = view1 | views::B;

...
```

#3 BEWARE OF RANGE CONCEPTS

- Be aware of views **requirements** and **productions**

```
std::string s = "split this one";

s | views::split(' ') | views::reverse; // no way!
```

#3 BEWARE OF RANGE CONCEPTS



Courtesy of Walletfox

https://www.walletfox.com/course/ranges_views_reverse_bidirectional_concept.php

#3 BEWARE OF RANGE CONCEPTS

```
auto view1 = views::iota(1); // [1, 2, 3, 4, 5, ...]

auto view2 = view1 | views::chunk(4); // [ [1, 2, 3, 4], [5, 6, 7, 8], ... ]

auto view3 = view2 | views::take(5) | views::reverse;

std::cout << front(view3); // ?
```

https://www.menti.com/72716ntt16

Go to **www.menti.com** and use the code **3096 7444**

#3 BEWARE OF RANGE CONCEPTS

```
auto view1 = views::iota(1); // [1, 2, 3, 4, 5, ...]

auto view2 = view1 | views::chunk(4); // [ [1, 2, 3, 4], [5, 6, 7, 8], ... ]

auto view3 = view2 | views::take(5) // [ [1, 2, 3, 4], ..., [17, 18, 19, 20] ]
                   | views::reverse; // returns elements from the back...
                                     // [17, 18, 19, 20], [13, 14, 15, 16], ...

std::cout << front(view3); // [17, 18, 19, 20]
```

#3 BEWARE OF RANGE CONCEPTS

```
auto view1 = views::iota(1); // [1, 2, 3, 4, 5, ...]

auto view2 = view1 | views::chunk(4); // [ [1, 2, 3, 4], [5, 6, 7, 8], ... ]

auto view3 = view2 | views::take(5) // [ [1, 2, 3, 4], ..., [16, 17, 18, 19] ]
                   | views::transform([](auto rng) { return views::reverse(rng); });


std::cout << front(view3); // [4, 3, 2, 1]
```

#3 BEWARE OF RANGE CONCEPTS

```
auto view1 = views::iota(1); // [1, 2, 3, 4, 5, ...]

auto view2 = view1 | views::chunk(4); // [ [1, 2, 3, 4], [5, 6, 7, 8], ... ]

auto view3 = view2 | views::take(5) // [ [1, 2, 3, 4], ..., [16, 17, 18, 19] ]
                   | views::transform(views::reverse_fn{});


std::cout << front(view3); // [4, 3, 2, 1]
```

#4 MIXING VIEWS & ACTIONS

- An action can be applied to some **materialized** range (**ultimately** referring to some data)

```
vector v = {1, 2, 3, 4, 5};

actions::reverse(views::take(v, 3));

std::cout << views::all(v); // 3, 2, 1, 4, 5
```

#5 MIXING VIEWS & ALGORITHMS

- An algorithm **requires** some range **concepts** to apply its operation

```
std::cout << accumulate(views::closed_iota(1, 3), 0); // 6

sort(views::closed_iota(1, 3)); // ehm...
```

#6 DEALING WITH TEMPORARIES

- We cannot create views of temporaries

```
std::vector<T> f(T t);

auto rng = src | view::transform(f) | view::join; // nope!
```

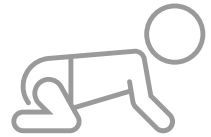
- However…

```
auto rng = src | views::transform(f) | views::cache1 | views::join;
```

https://stackoverflow.com/questions/36820639/how-do-i-write-a-range-pipeline-that-uses-temporary-containers

WARM-UP PATTERNS

Let's take the first steps in reusable solutions



#1 GOTTA PRINT 'EM ALL

- Use `views::all` on a container to (lazily) turn it into a range
- The library provides output operator for ranges*

```
vector v = {1, 2, 3, 4};

std::cout << views::all(v); // [1, 2, 3, 4]

std::cout << (v | views::all); // ditto
```

* however, printing tuples/pairs is still not supported -.-'

#2 MATERIALIZING RANGES TO CONTAINERS

- Use `ranges::to` to create a container from a range (consuming it entirely)

```
auto tens = views::closed_iota(0, 10) | to<std::vector>;

auto letters = views::zip(tens, views::iota('a')) | to<std::map>;

std::cout << letters[0]; // a

std::cout << letters[3]; // d
```

#3 CHECKING IF ALL ARE EQUAL

- Remember what classical `std::unique` does:

  it "removes" **adjacent equal elements** (except the first one)

- Can you imagine what `views::unique` does?

```
std::string letters = "aaaaa";

const auto areTheSame = distance(views::unique(letters)) == 1;
```

#3 CHECKING IF ALL ARE EQUAL

- *What about early exit?*
- Views are not for that! Use an algorithm instead:

```
std::string letters = "aaaaa";

adjacent_find(letters, std::not_equal_to<>{}) == end(letters)
```

#4 COMBINING RANGES TOGETHER

- Use `views::zip` and `views::zip_with` to combine two ranges together

```
auto letters = views::zip(views::iota(0), views::iota('a')) // [{0, 'a'}, {1, 'b'}, ... ]

auto letters = views::enumerate(views::iota('a')) // ditto

std::vector vect1 = {1, 2, 3}, vect2 = {10, 20, 30};

auto sum = views::zip_with(std::plus<>{}, vect1, vect2); // [11, 22, 33]
```

#5 GROUPING CONTIGUOUS ELEMENTS TOGETHER

- Use `views::group_by` to arrange together contiguous elements which satisfies a predicate

```
std::string pass = "ciaaaaooopass";

sort(pass); // [a, a, a, a, a, c, i, o, o, o, p, s, s]

pass | views::group_by(std::equal_to<>{}) // [[a,a,a,a,a],[c],[i],[o,o,o],[p],[s,s]]

     | views::transform([](auto rng) { return size(rng); }); // [5, 1, 1, 3, 1, 2]
```

#6 CRAFTING RANGES

- Use `views::for_each + yield_xxx` to lazily create ranges through **list comprehension**

```
views::iota(1,10) // [1, 2, 3,..., 10]
| views::for_each([](int x) {
        return yield_from(views::repeat_n(x, x)); // e.g. i=4: [4,4,4,4]
}); // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...]
```

#6 CRAFTING RANGES

- `views::for_each` transforms and additionally flattens a range of ranges:

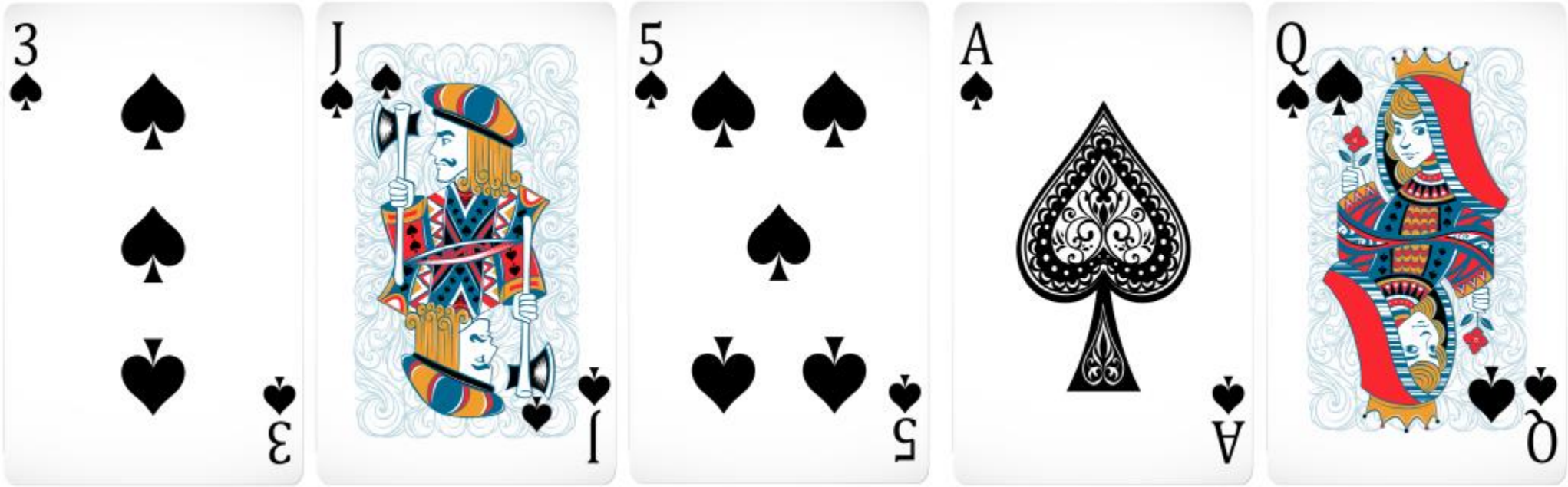
```
views::for_each = views::transform | views::join
```

LET'S PLAY!

https://www.menti.com/72716ntt16

Go to **www.menti.com** and use the code **3096 7444**

#1 HAVE YOU HIT A FLUSH?



- Can you check if any poker hand is a flush? -

https://wandbox.org/permlink/J6cVmCe5uFxZLpfw

#1 HAVE YOU HIT A FLUSH?

- Remember what classical `std::unique` does:

  it "removes" **adjacent equal elements** (except the first one)

- Can you imagine what `views::unique` does?

```
std::string letters = "aaaaa";

const auto areTheSame = distance(views::unique(letters)) == 1;
```

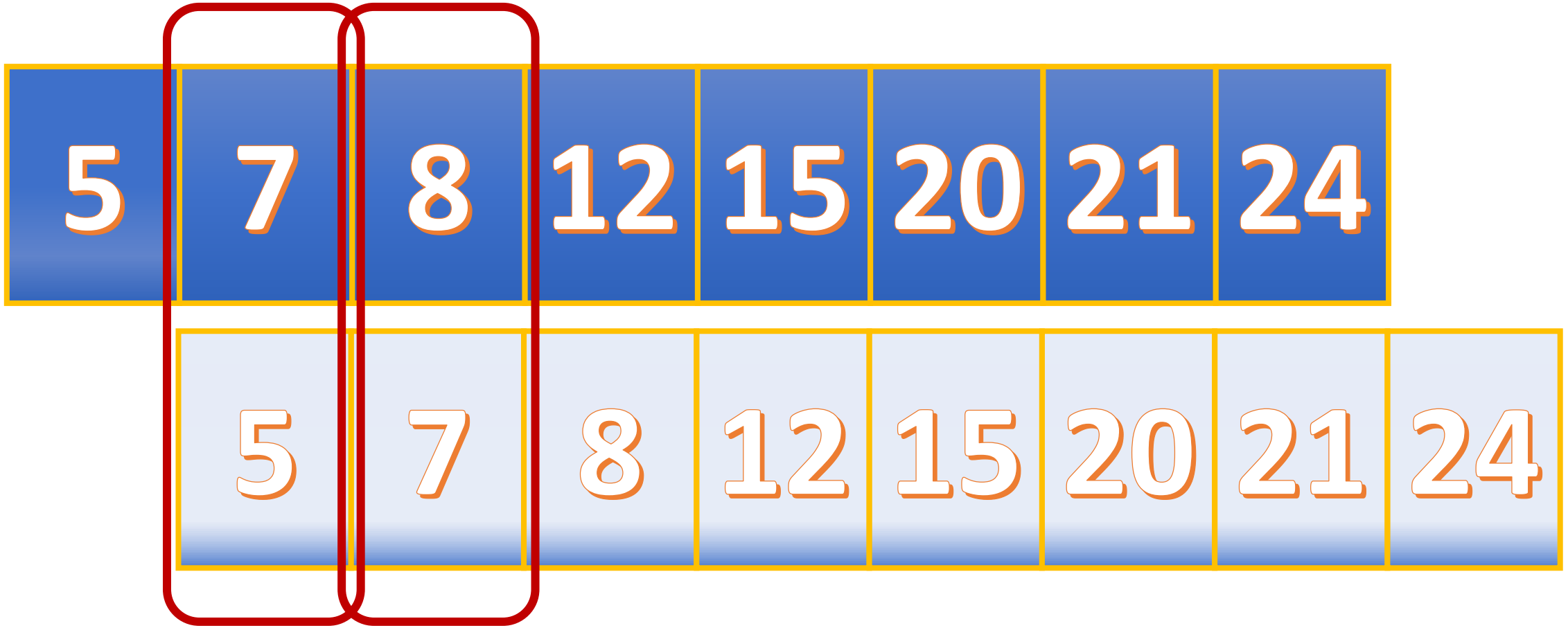
- Can you check if any poker hand is a flush? -

https://wandbox.org/permlink/J6cVmCe5uFxZLpfw

https://www.menti.com/72716ntt16

Go to **www.menti.com** and use the code **3096 7444**

#2 WHAT'S BEHIND ADJACENT_DIFFERENCE?

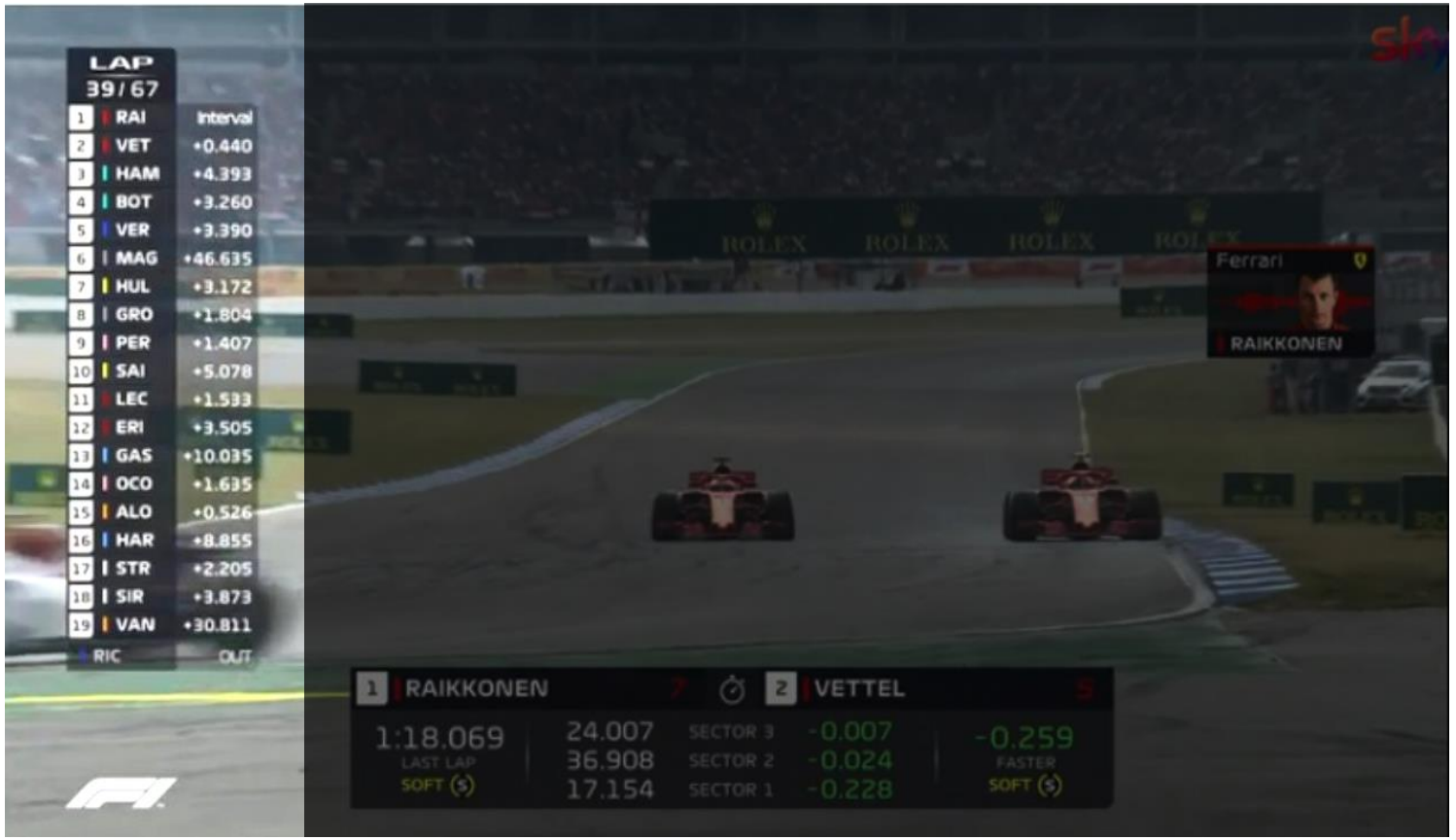


#2 WHAT'S BEHIND ADJACENT_DIFFERENCE?

- `adjacent_difference` can be seen as an application of `views::zip_with`

```
// we assume the range is non-empty

views::zip_with(std::minus<>{}, views::drop(vec, 1), vec);

// strictly speaking, the first element is included as-is

views::concat(views::single(front(vec)),

              views::zip_with(std::minus<>{}, views::drop(vec, 1), vec));
```

#2 WHAT'S BEHIND ADJACENT_DIFFERENCE?



- Can you turn "leader" into "interval"? -

https://wandbox.org/permlink/aW2iA9aSlSe1Br7A

https://www.menti.com/1isqpe4ipu

Go to **www.menti.com** and use the code **3408 8068**

#3 DECODING MESSAGES

```
auto uniques = input
        | views::group_by([](auto a, auto b) { return isalpha(a) == isalpha(b); })
        | views::filter([](auto rng) { return isdigit(at(rng, 0)); })
        | views::transform([](auto rng) { return std::stoi(to<std::string>(rng)); })
        | to<std::set>;
```

https://wandbox.org/permlink/SPAsgYK2Yoarsh4l

#4 REVISITING FIZZ-BUZZ

```
std::array<std::string,3> fizz {"","","Fizz"};

std::array<std::string,5> buzz {"","","","","Buzz"};

auto r_fizzes = fizz | views::cycle; // [ , ,Fizz, , ,Fizz...]

auto r_buzzes = buzz | views::cycle; // [ , , , ,Buzz, , , , ,Buzz...]

auto r_fizzbuzz = views::zip_with(std::plus{}, r_fizzes, r_buzzes); // [ , ,Fizz, ,Buzz,Fizz, , ,Fizz,...]

auto r_int_str = views::iota(1) | views::transform([](int x){ return std::to_string(x);}); // [1,2,3...]

auto rng = views::zip_with([](auto a, auto b){return std::max(a,b);}, r_fizzbuzz, r_int_str);

std::cout << (rng | views::take(20)) << "\n";
```

Courtesy of Walletfox

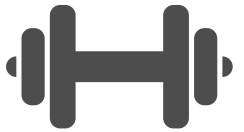
https://www.walletfox.com/publications_ranges.php

KNOWING IS NOT ENOUGH, WE MUST APPLY

25% discount:

ITALIANCPP25

https://italiancpp.org/ranges



ADDITIONAL RESOURCES

- Official Documentation
- Fully Functional C++ with Range-v3 (book – discount code: **ITALIANCPP25**)
- An Introduction to the Range-v3 Library (article)
- Some videos from conferences (videos)

SOME IDEAS FOR PRACTICING

- Fully Functional C++ with Range-v3 (book – discount code: **ITALIANCPP25**)
- HackerRank, LeetCode, CodingGame, etc (competitive programming)
- Advent of Code (I have applied ranges on many challenges of 2020 contest)
- Your real-world code (open your mind, think differently)

- If you like, **share your challenges/solutions** with me (**marco@italiancpp.org**), on our Slack channel **#learn,** or just tag me on twitter **@ilpropheta**


"That's all Folks!"

BONUS CONTENT

Intrigued? Let me share something more

SIZE VS DISTANCE

- `ranges::size` calculates the number of elements in a range **in constant time** (requires a range *capable of doing that*)
- `ranges::distance` calculates the number of **hops** needed for iterating over a range (when possible, in constant time, otherwise it "iterated" the range up to the end)

```
std::string s = "h2o";

std::cout << (distance(s) == size(s)); // 1

std::cout << size(s | views::filter(isalpha)); // does not compile!

std::cout << distance(s | views::filter(isalpha)); // 2
```

REPEATING TO INFINITY AND BEYOND

- `views::cycle` is used to repeat a *range* to infinity

```
views::single(1); // [1]

views::cycle(views::single(1)); // [1, 1, 1, ...]

std::vector v = {1, 2, 3};

views::cycle(v); // [1, 2, 3, 1, 2, 3, 1, 2 ...]

views::cycle(views::single(v)); // [v, v, v, ...]

views::cycle(views::single(views::all(v))); // [ [1,2,3], [1,2,3], ... ]
```

DIRECTORY CONTENT CYCLING

```
auto files = subrange(std::filesystem::directory_iterator{"/home"}, std::filesystem::directory_iterator{})
           | views::filter([](const auto& de){
                 return de.is_regular_file() && de.path().extension() == ".txt"; })
           | views::transform(&std::filesystem::directory_entry::path)
           | to<std::vector>;

std::cout << views::all(files) << "\n";

auto cycled = files | views::cycle;

auto it = begin(cycled);

std::cout << *it++ << "\n";
std::cout << *it++ << "\n";
std::cout << *it++ << "\n";
std::cout << *it++ << "\n";
std::cout << *it++ << "\n";
// ...
```

ALL STRING ROTATIONS

- Generate the range of the **rotations** of a string. E.g. abc → [abc], [bca], [cab]

```
std::string rotateThis = "abc";

const auto len = distance(rotateThis);

auto cycled  = rotateThis | views::cycle;

auto rotations = views::iota(0, len) | views::transform([=](auto i) {

   return cycled | views::drop(i) | views::take(len);

});
```

https://wandbox.org/permlink/giHjuzDyggbKxRqK

SIMPLE CHARS COMPRESSION

- Compress a string like `"aaaabbbcca"`
  to something like `[("a", 4), ("b", 3), ("c", 2), ("a", 1)]`

```
std::string input = "aaaabbbcca";
auto output = input | views::group_by(std::equal_to<>{}) // [a,a,a,a],[b,b,b],[c,c],[a]
                    | views::transform([](auto subr) {
                          return std::make_pair(front(subr), size(subr));
                      });
for (auto [letter, size] : output)
    std::cout << letter << "," << size << "\n";
```

MATRIX ACCESS

```
std::vector<std::vector<int>> m = { {1,2,3}, {4,5,6}, {7,8,9}  };
auto nr = distance(m), nc = distance(front(m)); // rows=3, cols=3

auto allRows = m  | views::join; // [1,2,3,4,5..]
auto c0 = allRows | views::drop(0) | views::stride(nc); // [1,4,7]
auto c2 = allRows | views::drop(2) | views::stride(nc); // [3,6,9]
auto diagonal = allRows | views::stride(nc + 1); // [1,5,9]

std::cout << "c0: " << c0 << "\n";
std::cout << "c2: " << c2 << "\n";
std::cout << "diagonal: " << diagonal << "\n";
```

Courtesy of Walletfox

https://www.walletfox.com/publications_ranges.php

CSV READING

```
std::ifstream file("file");
auto lines = getlines(file) | views::for_each([](auto line) {
        return yield(line | views::split(',') | to<std::vector<std::string>>);
});

// lines is a lazy range of vector<std::string>
for (auto line : lines | views::take(5)) {
        std::cout << views::all(line) << "\n";
}
```

https://wandbox.org/permlink/DORWPA6kACegAX9y

REVERSE THE WORDS OF A STRING

- For example: "reverse words in this string" becomes "string this in words reverse"
- A possible approach is to reverse the string blindly and then reverse the individual words
- We have learnt that `views::split | views::reverse` does not work. However…

```
std::string input = "reverse words in this string";
auto rev = views::reverse(input) | views::group_by([](auto c1, auto c2) {
    return isalpha(c1) == isalpha(c2);
}) | views::for_each(views::reverse_fn{});
```

https://wandbox.org/permlink/QOzBX2TzaWYgIwBi

REVERSE THE WORDS OF A STRING

- Let's figure out what the code in the previous slide does for the first iteration
  - `reverse(input)` virtually swaps the first 'r' with the last 'g'
  - so, it outputs 'g' to the next combinator that is `group_by`
  - `group_by` "accumulates" chars until two non-alpha adjacents are found (e.g. 's' and whitespace)
  - when such a pair of letters is found, it produces a subrange including all the "good" ones
  - however, due to `reverse`, remember that letters are accumulated from the back (e.g. `[g, n, i, r, t, s]`)
  - afterwards, `for_each` takes this subrange and applies `reverse_fn` (that is just reverse)
  - so, `[g, n, i, r, t, s]` is turned into `[s, t, r, i, n, g]`
  - That's it!