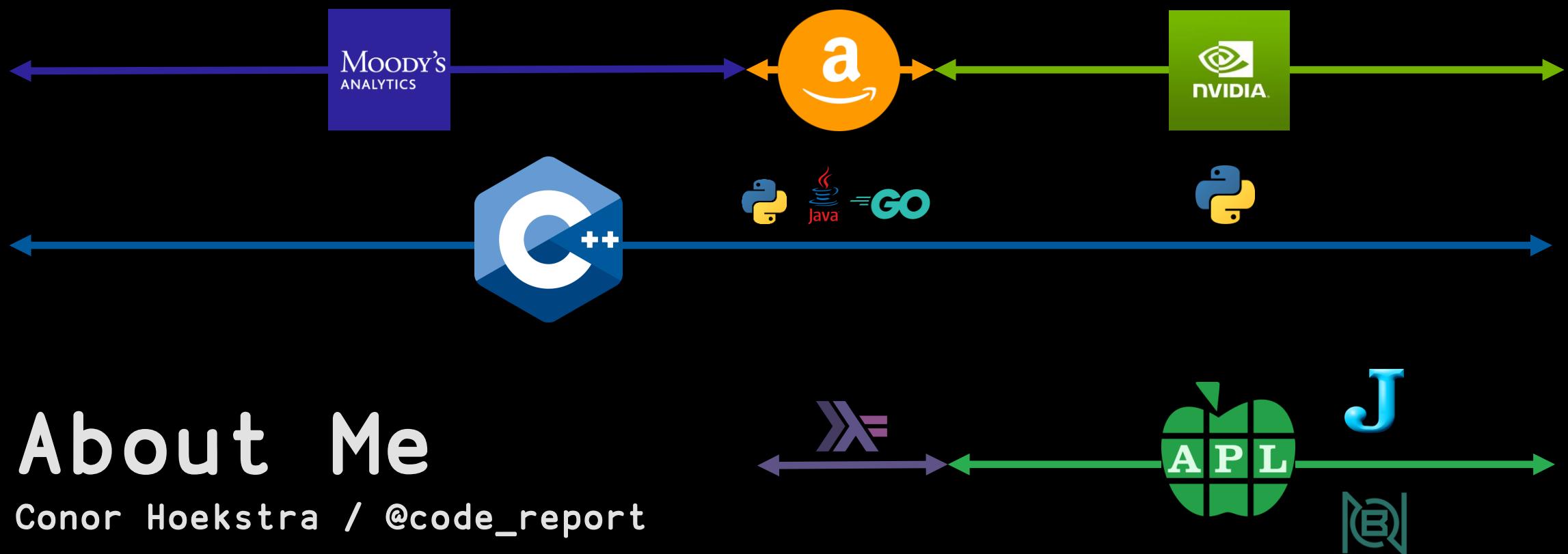
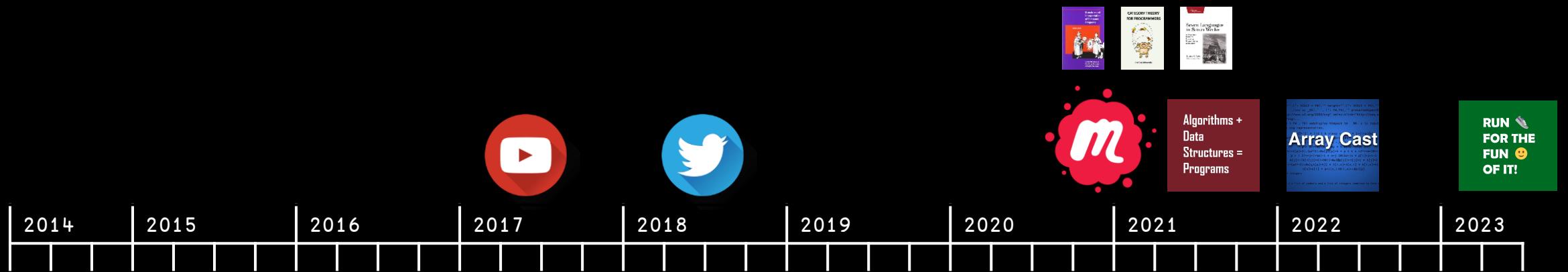




New Algorithms in C++23

Conor Hoekstra





Algorithms + Data Structures = Programs

```
":",(": SCALE * fW)," height="",(": SCALE * fH),"'
',(cnv sc _85),' ', (": fW,fH)," preserveAspectRatio="xMidYMid"
: p://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
/svg>...
0 + fW , fH) webdisplay htmpack tm N8. x is input
> svg representation.

{[]+'C' ◊ ((d t k n)exp s) ◊ I+{((<ω))[]α}
×3) ×2) ×1) ×0) ×(-1) ×(-2) ×(-3)
) ×(3) ×(2) ×(1) ×(0) ×(-1) ×(-2)
p r I×+cn[i]@~#p ◊ t k(-@i~-1)×1
4)×t[p]=3,{w/~2|i#w}×t[p]=4 ◊ p t k n r/×+cm+2@i-
p r i I×+cj+(+m)-1 ◊ n+j I@(0≤r)n ◊ p[i]+j+i-1
k[j]←(k[r[j]]=0)∨0@((>Φw)⊕p[j])×t[j]=1 ◊ t[j]+2
i←{w/~2|i#w}×t[p]=4]] ◊ t[i,x]←t[x,i] ◊ k[i,x]←k[:
n[x]+n[i] ◊ p←((x,i)@(i,x)←i#p)[p]
f integers

/* a list of symbols and a list of integers combined to form a
```

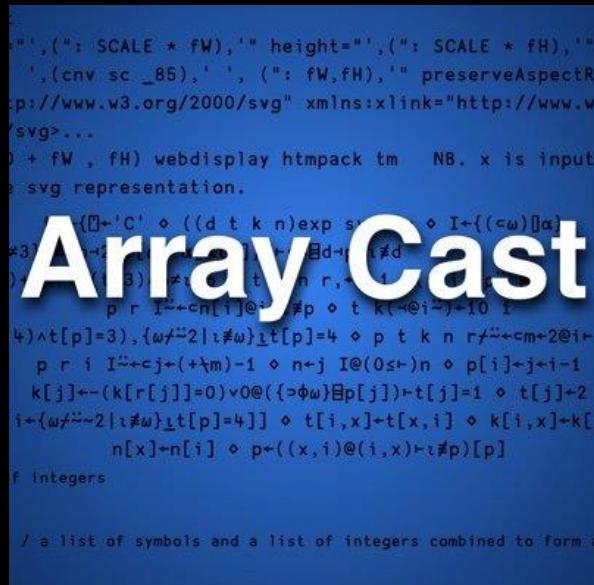


319 Videos

32 (20) Talks

Algorithms + Data Structures = Programs

136 Episodes
[@adspthepodcast](https://www.adspthepodcast.com)



56 Episodes
[@arraycast](https://www.arraycast.com)



RUN FOR THE FUN OF IT!



10 Episodes
[@conorhoekstra](https://www.conorhoekstra.com)

<https://github.com/codereport/Content>

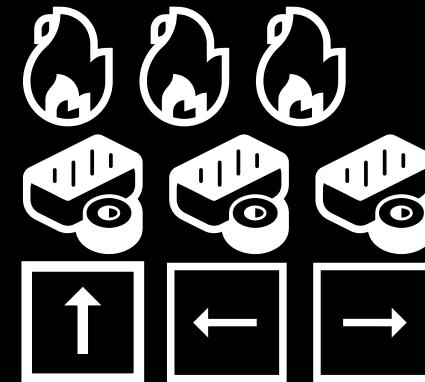
Code Links

Example	Language	GitHub Link	Godbolt Link
C++23 Ranges Status	C++	Link	GCC: https://godbolt.org/z/vsfT6e96b
Warm Up: Negatives	C++	Link	https://godbolt.org/z/8TEevabqd
Sushi for Two	C++	Link	https://godbolt.org/z/69Kh8baz3
Sushi for Two	Circle	Link	https://godbolt.org/z/P6PxMnhMz
Sushi for Two (Bad / YouTube)	C++	Link	https://godbolt.org/z/hjbKb6jWM
Max Gap	C++	Link	https://godbolt.org/z/P43EhYcsj
Max Gap	Circle	Link	https://godbolt.org/z/3K598jMMa
filter_out_html_tags	Circle	Link	https://godbolt.org/z/on5xMG5ax

Algorithm Land Overview

Problems:

- Warm Up
- Sushi for Two
- Max Gap
- filter_html_tags



C++ Algorithm Land Overview

C++98 Iterator Algorithms

```
std::find_if(a.begin(), a.end(), f)
```

```
std::count(b.begin(), b.end(), 3)
```

```
std::all_of(c.begin(), c.end(), f)
```

```
std::sort(d.begin(), d.end())
```

```
...
```

C++ Algorithm Land Overview

C++98 Iterator Algorithms	C++20 Range Algorithms
<code>std::find_if(a.begin(), a.end(), f)</code>	<code>std::ranges::find_if(a, f)</code>
<code>std::count(b.begin(), b.end(), 3)</code>	<code>std::ranges::count(b, 3)</code>
<code>std::all_of(c.begin(), c.end(), f)</code>	<code>std::ranges::all_of(c, f)</code>
<code>std::sort(d.begin(), d.end())</code>	<code>std::ranges::sort(d)</code>
...	...

C++ Algorithm Land Overview

C++98 Iterator Algorithms	C++20 Range Algorithms	C++20/23 Range Adaptors & Factories
<code>std::find_if(a.begin(), a.end(), f)</code>	<code>std::ranges::find_if(a, f)</code>	<code>std::views::take</code>
<code>std::count(b.begin(), b.end(), 3)</code>	<code>std::ranges::count(b, 3)</code>	<code>std::views::drop</code>
<code>std::all_of(c.begin(), c.end(), f)</code>	<code>std::ranges::all_of(c, f)</code>	<code>std::views::transform</code>
<code>std::sort(d.begin(), d.end())</code>	<code>std::ranges::sort(d)</code>	<code>std::views::filter</code>
<code>...</code>	<code>...</code>	<code>std::views::chunk_by</code>
		<code>...</code>

C++ Algorithm Land Overview

C++98 Iterator Algorithms	C++20 Range Algorithms	C++20/23 Range Adaptors & Factories
<code>std::find_if(a.begin(), a.end(), f)</code>	<code>std::ranges::find_if(a, f)</code>	<code>std::views::take</code>
<code>std::count(b.begin(), b.end(), 3)</code>	<code>std::ranges::count(b, 3)</code>	<code>std::views::drop</code>
<code>std::all_of(c.begin(), c.end(), f)</code>	<code>std::ranges::all_of(c, f)</code>	<code>std::views::transform</code>
<code>std::sort(d.begin(), d.end())</code>	<code>std::ranges::sort(d)</code>	<code>std::views::filter</code>
<code>...</code>	<code>...</code>	<code>std::views::chunk_by</code>
		<code>...</code>

C++20/23 Range Adaptors & Factories

`std::views::take`

`std::views::drop`

`std::views::transform`

`std::views::filter`

`std::views::chunk_by`

C++20/23 Range Adaptors & Factories

std::views::take
std::views::drop
std::views::transform
std::views::filter

std::views::chunk_by

C++20/23 Range Adaptors & Factories

drop	adjacent (pairwise)
drop_while	adjacent_transform (pairwise_transform)
elements (keys values)	cartesian_product
filter	chunk
iota	chunk_by
join	enumerate
reverse	join_with
split	slide
take	stride
take_while	zip
transform	zip_transform



```
#include <array>
#include <ranges>
#include <string>

#include <fmt/ranges.h>

using namespace std::views;
using namespace std::string_literals;

auto main() -> int {
    auto const nums      = std::array{0, 0, 1, 1, 2, 2};
    auto const animals = std::array{"cat"s, "dog"s};

    fmt::print("{}\n", iota(0, 5) | chunk(2));                                // [[0, 1], [2, 3], [4]]
    fmt::print("{}\n", nums | chunk_by(std::equal_to{}));                         // [[0, 0], [1, 1], [2, 2]]
    fmt::print("{}\n", iota(0, 5) | slide(3));                                    // [[0, 1, 2], [1, 2, 3], [2, 3, 4]]
    fmt::print("{}\n", iota(0, 10) | stride(3));                                 // [0, 3, 6, 9]
    fmt::print("{}\n", repeat(42) | take(5));                                    // [42, 42, 42, 42, 42]
    fmt::print("{}\n", zip_transform(std::plus{}, nums, nums));                  // [0, 0, 2, 2, 4, 4]
    fmt::print("{}\n", zip(iota(0, 3), iota(1, 4)));                           // [(0, 1), (1, 2), (2, 3)]
    fmt::print("{}\n", iota(0, 4) | adjacent<2>);                             // [(0, 1), (1, 2), (2, 3)]
    fmt::print("{}\n", iota(0, 4) | pairwise());                                // [(0, 1), (1, 2), (2, 3)]
    fmt::print("{}\n", iota(0, 4) | adjacent_transform<2>(std::plus{}));        // [1, 3, 5]
    fmt::print("{}\n", iota(0, 4) | pairwise_transform(std::plus{}));             // [1, 3, 5]
    fmt::print("{}\n", animals | join_with(','));                               // ['c', 'a', 't', ',', 'd', 'o', 'g']
    fmt::print("{}\n", cartesian_product(iota(0, 2), "AZ"s));                 // [(0, 'A'), (0, 'Z'), (1, 'A'), (1, 'Z')]
    fmt::print("{}\n", enumerate("APL"s));                                       // [(0, 'A'), (1, 'P'), (2, 'L')]

    return 0;
}
```















































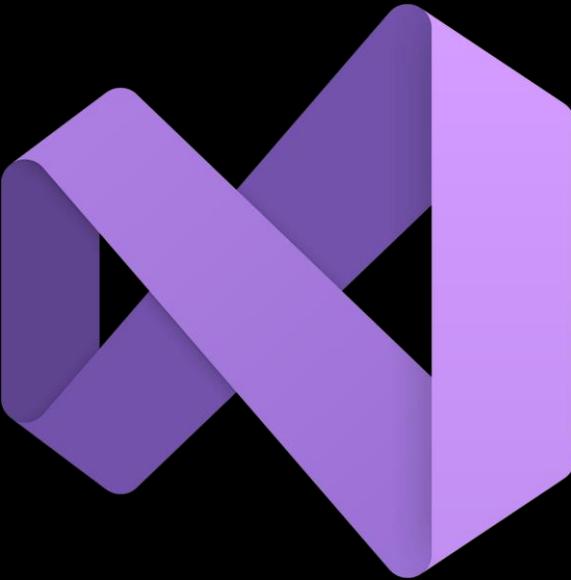


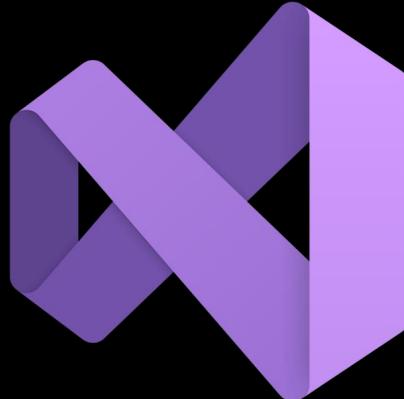




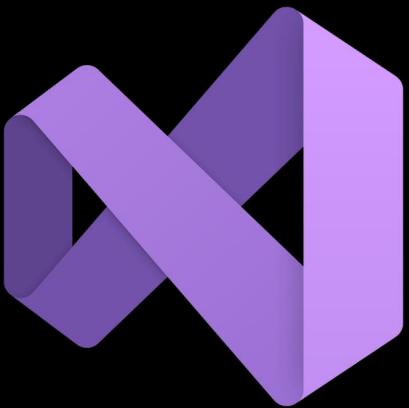








- GCC 13
- MSVC v19.36 (VS 2022 17.6)
- Clang



- GCC 13
- MSVC v19.36 (VS 2022 17.6)
- Clang

Status

View/Feature	GCC	Clang	MSVC	MSVC PR
adjacent	●	●	●	PR: microsoft/STL#3508
adjacent_transform	●	●	●	PR: microsoft/STL#3546
cartesian_product	●	●	●	PR: microsoft/STL#3561
chunk	●	●	●	PR: microsoft/STL#2685
chunk_by	●	●	●	PR: microsoft/STL#2565
enumerate	●	●	●	PR: microsoft/STL#3472
join_with	●	●	●	PR: microsoft/STL#2619
pairwise	●	●	●	PR: microsoft/STL#3508
pairwise_transform	●	●	●	PR: microsoft/STL#3546
repeat	●	●	●	PR: microsoft/STL#3142
slide	●	●	●	PR: microsoft/STL#2670
stride	●	●	●	PR: microsoft/STL#2981
zip	●	●	●	PR: microsoft/STL#3035
zip_transform	●	●	●	PR: microsoft/STL#3322
ranges::to	●	●	●	PR: microsoft/STL#2806
Formatting	●	●	●	

- - Not Done
- - Started
- - Done (and available on Compiler Explorer)

[[digression]]



Parallel Block-Delayed Sequences

Sam Westrick
Carnegie Mellon University
Pittsburgh, PA, USA
swestrict@cs.cmu.edu

Daniel Anderson
Carnegie Mellon University
Pittsburgh, PA, USA
dlanders@cs.cmu.edu

Abstract

Programming languages using functions on collections of values, such as map, reduce, scan and filter, have been used for over fifty years. Such collections have proven to be particularly useful in the context of parallelism because such functions are naturally parallel. However, if implemented naively they lead to the generation of temporary intermediate collections that can significantly increase memory usage and runtime. To avoid this pitfall, many approaches use “fusion” to combine operations and avoid temporary results. However, most of these approaches involve significant changes to a compiler and are limited to a small set of functions, such as maps and reduces.

In this paper we present a library-based approach that fuses widely used operations such as scans, filters, and flattens. In conjunction with existing techniques, this covers most of the common operations on collections. Our approach is based on a novel technique which parallelizes over blocks, with streams within each block. We demonstrate the approach by implementing libraries targeting multicore parallelism in two languages: Parallel ML and C++, which have very different semantics and compilers. To help users understand when to use the approach, we define a cost semantics that indicates when fusion occurs and how it reduces memory allocations. We present experimental results for a dozen benchmarks that demonstrate significant reductions in both time and space. In most cases the approach generates code that is near optimal for the machines it is running on.

CCS Concepts: • Software and its engineering → Parallel programming languages; Functional languages; • Theory of computation → Parallel algorithms.



This work is licensed under a Creative Commons Attribution International 4.0 License.
PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9204-4/22/04.
<https://doi.org/10.1145/3503221.3508434>

Mike Rainey
Carnegie Mellon University
Pittsburgh, PA, USA
me@mike-rainey.site

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Keywords: parallel programming, fusion, collections, functional programming

1 Introduction

Collection-oriented programming is a style of programming in which programs use operations over collections of values, such as map, reduce, filter, and scan. Languages supporting this style date back to 1960s with APL (arrays) [18], SETL (sets and maps) [29], Codd’s relational algebra (relations) [11] and FP (sequences) [3]. These languages allowed a particularly simple and elegant way to work with collections. With the advent of highly parallel machines in the mid 80s, there was a significant increase in interest in this style of programming. The observation is that by raising the level of abstraction, sequential loops go away, and code often becomes inherently parallel. Furthermore, working with collections promotes a functional style of programming, and hence mostly avoids mutation and, in the context of parallelism, the potential dangerous data races they cause. Early such *data parallel* languages include CM-Lisp [19], C* [28], and Nesl [4]. Later ones used in a distributed setting include map-reduce [14] and Spark [37].

It was quickly noted, however, that collections can incur large overheads due to the generation of intermediate results. For example, a map squaring every element of a vector, followed by a reduce summing the results would naively generate an intermediate vector of the products before summing them. A loop, on the other hand, would multiply and add as it went along. The generation of this intermediate vector wastes not only space but also time, due to additional reads and writes in situations where memory bandwidth is often the bottleneck. This problem of avoiding intermediate results has been studied extensively since the 1970s [1, 35] and is often referred to as “loop-fusion”, just “fusion”, or originally “jamming”. Fusion has been applied to data-parallel languages since the start of the 90s with dozens of papers on the topic (e.g., [9, 12, 13, 17, 20–22, 24, 26, 31]). Most of these techniques rely on compiler transformations.

Interestingly, however, Keller et al. [20] were able to show that by taking advantage of standard compiler optimizations, fusion can be implemented efficiently for sequences as a

Keywords: parallel programming, fusion, collections, functional programming

1 Introduction

Collection-oriented programming is a style of programming in which programs use operations over collections of values, such as map, reduce, filter, and scan. Languages supporting this style date back to 1960s with APL (arrays) [18], SETL (sets and maps) [29], Codd’s relational algebra (relations) [11] and FP (sequences) [3]. These languages allowed a particularly simple and elegant way to work with collections. With the advent of highly parallel machines in the mid 80s, there was a significant increase in interest in this style of programming. The observation is that by raising the level of abstraction, sequential loops go away, and code often becomes inherently parallel. Furthermore, working with

or reduce can be composed. For example, a map followed by a reduce would never instantiate the intermediate list, only instantiating one element at a time, requiring $O(1)$ additional memory beyond the input to run.

Collection-oriented programming with stream fusion has gained recent popularity in the programming community, as demonstrated for example by the C++20 ranges library [27]. It promotes collection-oriented programming by supplying a variety of generic algorithms for ranges of elements in the form of so-called *view adapters*. What makes them interesting for us is that they are implemented using stream fusion.¹ The library allows operations such as maps, filters, and flattens to be composed and fused, and is entirely library-based, requiring no language extensions or specialized compiler support. However, it is designed for sequential computation, and does not support parallelism, except in easy cases. Similar tools exist in other languages, such as the `java.util.stream` library introduced in Java 8.

To make stream-fusion efficient requires the compiler to get rid of (possibly multiple) function calls on each iteration. Several special-purpose compiler techniques have been sug-

previous elements at hand, stream fusion is therefore not used if fusion supports random context. This suggests that stream fusion be powerful.

2.1 Stream-of-blocks

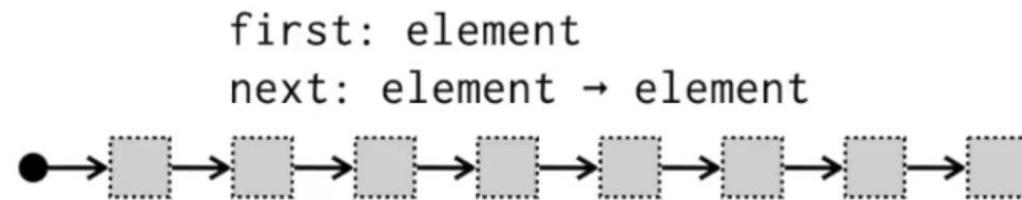
Previous work has focused on stream fusion with parallelism using a block-based approach. The idea is to partition the stream into blocks of fixed length such that the whole block can be instantiated a whole block at once. Then, the function f can be applied to each element in parallel. This approach works well when the blocks correspond to the size of memory pages, as with GPUs or very large memory blocks. It is not well suited for data structures whose sizes would have to be gigabytes or more.



Fusion Breakdown

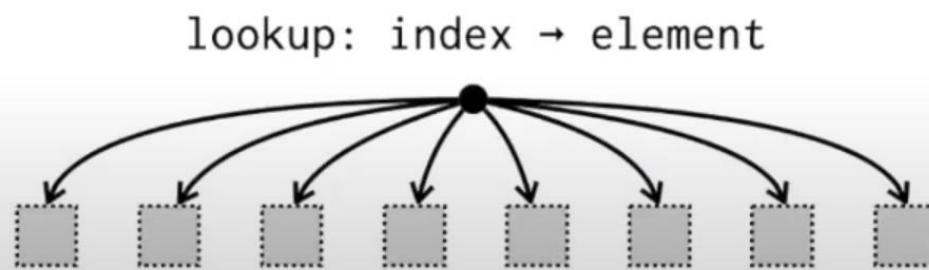
- Stream fusion

- naturally sequential
- e.g. lazy lists, Java streams, C++20 ranges/views, Rust iterators, ...



- Index fusion

- naturally parallel
- elements have to independent
- good for map/zip/reduce fusion
- e.g. Repa [1]

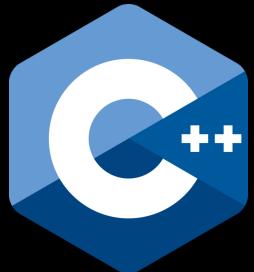


[1] Regular, Shape-polymorphic, Parallel Arrays in Haskell.

Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier.

ICFP 2010.

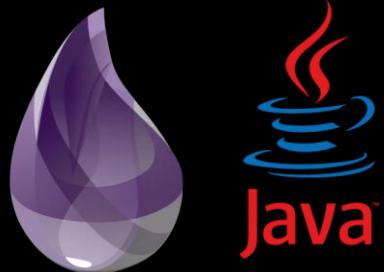
Libraries



Ranges



Iterators



Streams

Languages

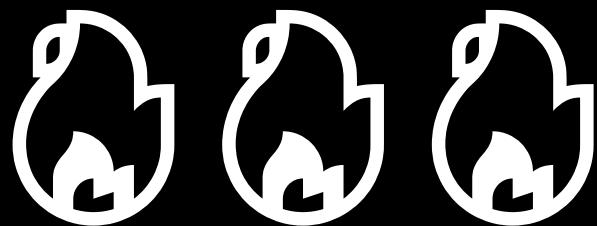


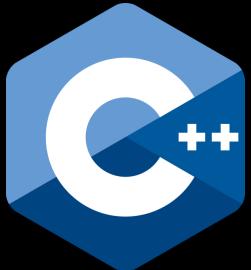
Different Programming Paradigms

- Collection Oriented Programming ☆
 - aka Functional-Style (via T. Brindle)
- Function Programming
- Object-Oriented Programming
- Imperative Programming

[[end of digression]]

Warm Up

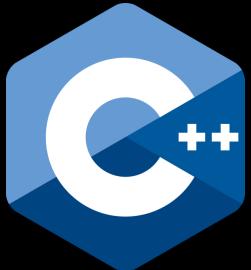




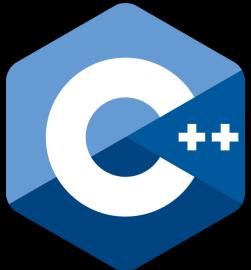
```
int num_negatives(std::vector<int> nums) {
    int count = 0;
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] < 0) ++count;
    }
    return count;
}
```



```
int num_negatives(std::vector<int> nums) {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```



```
auto num_negatives(std::vector<int> nums) {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```



```
auto num_negatives(std::vector<int> nums) -> int {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```



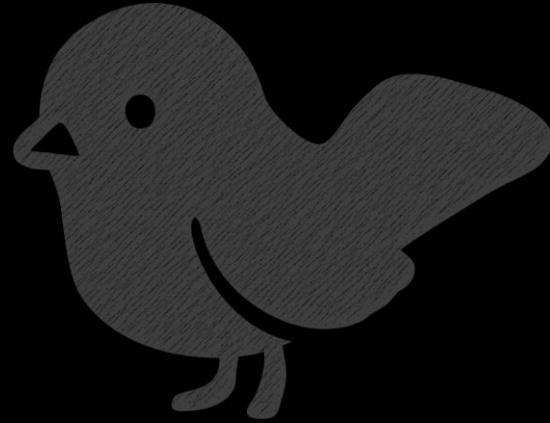
```
auto num_negatives(std::vector<int> nums) -> int {  
    return std::count_if(nums.cbegin(), nums.cend(),  
        [](auto e) { return e < 0; });  
}
```



```
auto num_negatives(std::vector<int> nums) -> int {  
    return std::ranges::count_if(nums,  
        [](auto e) { return e < 0; });  
}
```



```
auto num_negatives(std::vector<int> nums) -> int {  
    return std::ranges::count_if(nums, lt_(0));  
}
```



```
using namespace combinatorics;

auto num_negatives(std::vector<int> nums) -> int {
    return std::ranges::count_if(nums, lt_(0));
}
```

inevitably someone says...

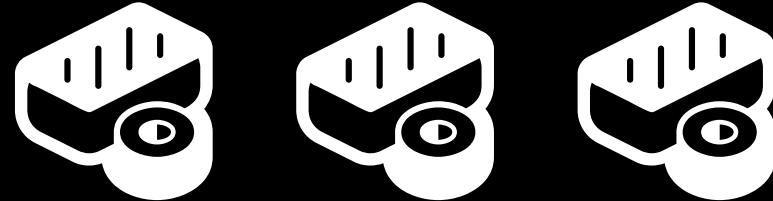


Loops are easier to read
/ understand ... everyone
knows them

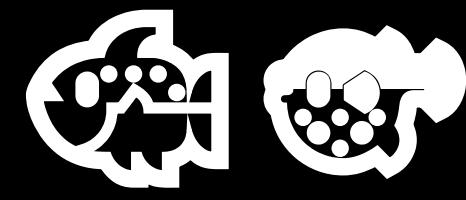
```
int num_negatives(std::vector<int> nums) {  
    int count = 0;  
    for (auto const num : nums) {  
        if (num < 0) ++count;  
    }  
    return count;  
}
```

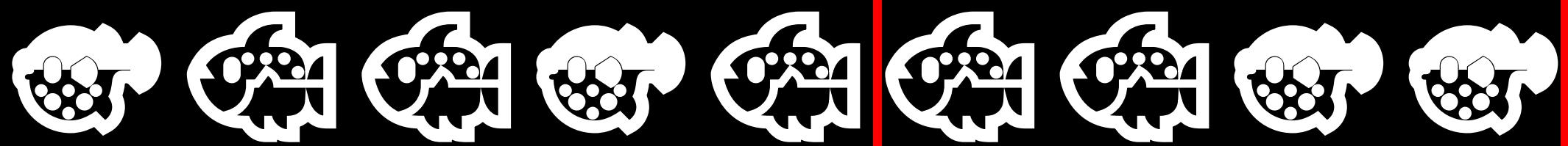
introducing one of my favorite
problems of all time...

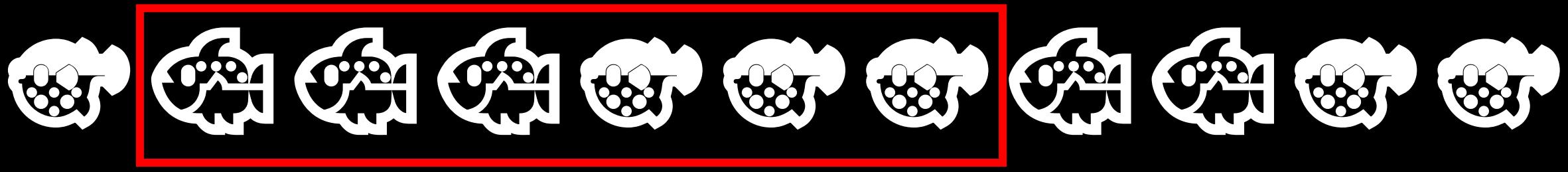
Sushi for Two



<https://codeforces.com/contest/1138/problem/A>









```
template <int N>
constexpr auto sushi_for_two(std::array<int, N> sushi) {
    int current_sushi      = 0;
    int sushi_in_a_row     = 0;
    int prev_sushi_in_a_row = 0;
    int max_of_mins        = 0;
    for (auto const s : sushi) {
        if (current_sushi != s) {
            current_sushi = s;
            if (prev_sushi_in_a_row == 0) {
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row     = 1;
            } else {
                auto const min      = std::min(sushi_in_a_row, prev_sushi_in_a_row);
                max_of_mins        = std::max(max_of_mins, min);
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row     = 1;
            }
        } else {
            sushi_in_a_row += 1;
        }
    }
    auto const min = std::min(sushi_in_a_row, prev_sushi_in_a_row);
    max_of_mins   = std::max(max_of_mins, min);
    return max_of_mins * 2;
}
```



```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(_eq_)
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_)));
}
```

q

sf2: { x }

[1, 2, 2, 1, 2, 2, 2, 1, 1]

sf2: { chunk x }

[[1], [2, 2], [1], [2, 2, 2], [1, 1]]

sf2: { count each chunk x }

[1, 2, 1, 3, 2]

sf2: { (&) prior count each chunk x }

[1, 1, 1, 1, 2]

Hooogle Translate

prior



CUDA

adjacent_difference

Thrust

[Doc](#)



C++

adjacent_difference

<numeric>

[Doc](#)



APL

/ (n-wise reduce)

-

[Doc](#)



Haskell

mapAdjacent

Data.List.HT

[Doc](#)



Kotlin

zipWithNext

collections

[Doc](#)



q

prior

-

[Doc](#)



C++

adjacent_transform

<ranges>

[Doc](#)

sf2: { (&) prior count each chunk x }

[1, 1, 1, 1, 2]

sf2: { 1 _ (&) prior count each chunk x }

[1, 1, 1, 2]

sf2: { max 1 _ (&) prior count each chunk x }

2

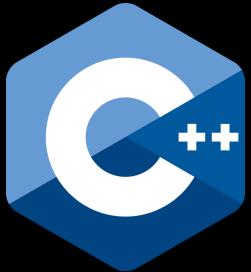
sf2: { 2 * max 1 _ (&) prior count each chunk x }

2

```
sf2: { 2 * max 1 _ (&) prior count each chunk x }
```

4

sf2: 2 * max 1 _ (&) prior count each chunk ::



```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi;
}
```

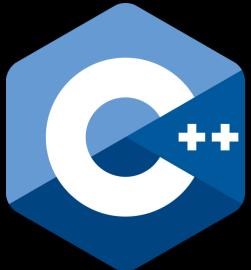
[1, 2, 2, 1, 2, 2, 2, 1, 1]



```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        | chunk_by(std::equal_to{});
}
```

```
[[1], [2, 2], [1], [2, 2, 2], [1, 1]]
```



```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance);
}
```

[1, 2, 1, 3, 2]



```
using namespace std::views;

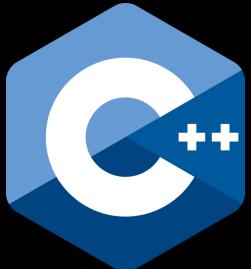
auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(
            [](auto a, auto b) { return std::min(a, b); });
}
```

[1, 1, 1, 2]



```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(
            [](auto a, auto b) { return std::min(a, b); }));
}
```



2 very irritating things about this code

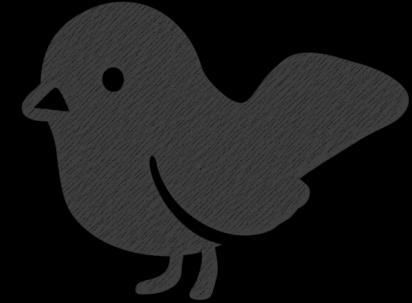
```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(
            [](auto a, auto b) { return std::min(a, b); }));
}
```



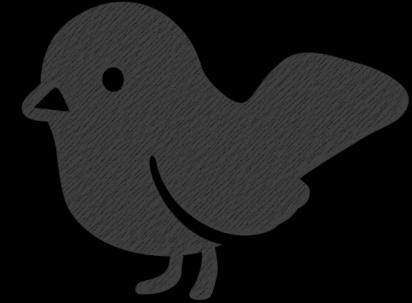
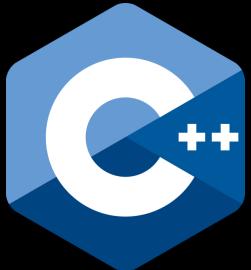
```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```



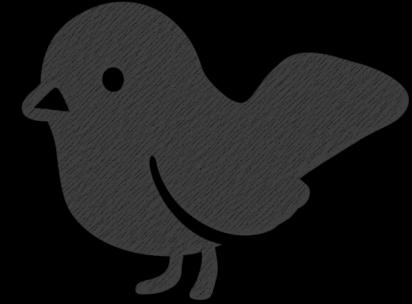
```
using namespace std::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```



```
using namespace std::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(_eq_)
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```



```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```



www.circle-lang.org/



Algorithms +
Data
Structures =
Programs

Episode 98: The Future of C++ with Sean Baxter (Part 2)

OCTOBER 7, 2022

In this episode, Bryce and Conor continue their interview with Sean Baxter about all of the C++ successor languages.

Episode 97: C++ vs Carbon vs Circle vs CppFront with Sean Baxter

SEPTEMBER 30, 2022

In this episode, Bryce and Conor talk to Sean Baxter about all of the C++ successor languages.

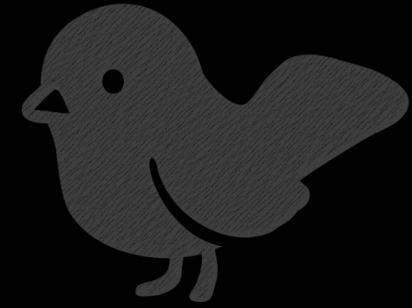


**Algorithms +
Data
Structures =
Programs**



Circle Links

Date	Type	Link
-	Website	https://www.circle-lang.org/
2019-10-03	Podcast	cpp.chat Episode 63 - All the C++30 Features - but Right Now
2020-01-23	Podcast	Circle
2022-05	Talk	Circle Metaprogramming - Sean Baxter - CppNow 2022
2022-09-30	Podcast	ADSP Episode 97: C++ vs Carbon vs Circle vs CppFront with Sean Baxter
2022-10-07	Podcast	ADSP Episode 98: The Future of C++ with Sean Baxter (Part 2)
2023-03-14	Talk	Circle Evolves C++ Sean Baxter NYC++
2023-04-03	Talk	Sean Baxter @Bloomberg: Circle Fixes Defects, Makes C++ Language Safer & More Productive

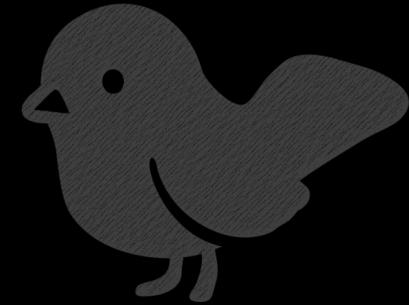


```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```



```
-std=c++2b --gcc-toolchain /opt/compiler-explorer/gcc-13.1.0/ -Wl,-rpath,/opt/compiler-explorer/gcc-13.1.0/lib64/
```



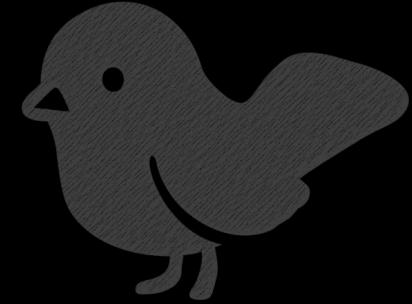
```
using namespace std::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```

Code Links

Example	Language	GitHub Link	Godbolt Link
C++23 Ranges Status	C++	Link	GCC: https://godbolt.org/z/vsfT6e96b
Warm Up: Negatives	C++	Link	https://godbolt.org/z/8TEevabqd
Sushi for Two	C++	Link	https://godbolt.org/z/69Kh8baz3
Sushi for Two	Circle	Link	https://godbolt.org/z/P6PxMnhMz
Sushi for Two (Bad / YouTube)	C++	Link	https://godbolt.org/z/hjbKb6jWM
Max Gap	C++	Link	https://godbolt.org/z/P43EhYcsj
Max Gap	Circle	Link	https://godbolt.org/z/3K598jMMa
filter_out_html_tags	Circle	Link	https://godbolt.org/z/on5xMG5ax

<https://github.com/codereport/Content/Talks>



```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```



```
auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```

```
sushiForTwo :: [Int] -> Int
sushiForTwo = (*2)
    . maximum
    . mapAdjacent min
    . map length
    . group
```



```
auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```



```
sushiForTwo :: [Int] -> Int
sushiForTwo = (*2)
    . maximum
    . mapAdjacent min
    . map length
    . group
```

q

sf2: 2 * max 1 _ (&) prior
count each chunk ::

q

sf2: 2 * max 1 _ (&) prior
count each chunk ::

sf2: { 2 * max 1 _ (&) prior count each chunk x }

```
sf2: { 2 * max 1 _ (&) prior count each chunk x }
chunk: { (where differ x) cut x }
```

sf2: { 2 * max 1 _ (&) prior deltas where differ x }

```
sf2: { 2 * max 1 _ (&) prior deltas (where differ x) , count x }
```



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi;
}
```

[1, 2, 2, 1, 2, 2, 2, 1, 1]



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1));
}
```

[1, 0, 1, 1, 0, 0, 1, 0]



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1));
}

[(1,1), (0,2), (1,3), (1,4), (0,5), (0,6), (1,7), (0,8)]
```



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst);
}
```

```
[(1,1), (1,3), (1,4), (1,7)]
```



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> transform($, _snd);
}
```

[1, 3, 4, 7]



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($);
}
```

[1, 3, 4, 7]



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size())));
}

[0, 1, 3, 4, 7, 9]
```



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $);
}
```

[1, 2, 1, 3, 2]



```
using namespace ranges::views;
using namespace combinator;

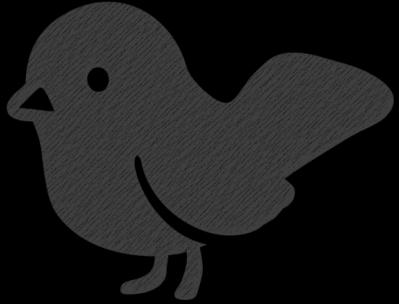
auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1));
}
```

[1, 1, 1, 2]



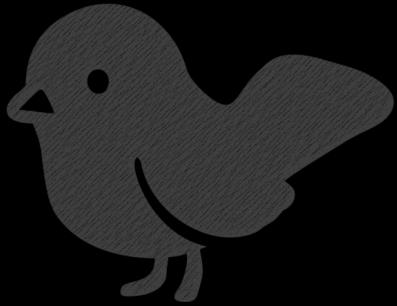
```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```



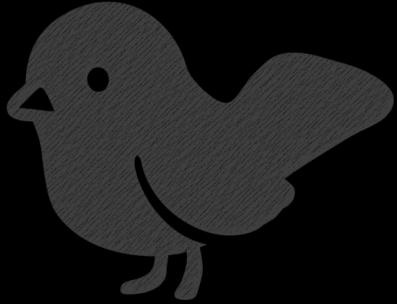
```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```



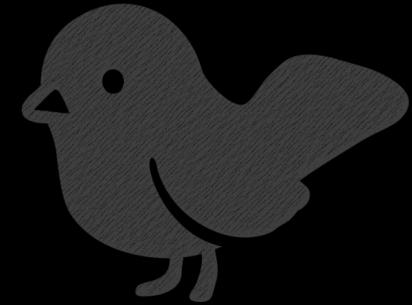
```
auto adjacent_transform_2(auto&& rng, auto op) {
    return rng |> zip_with(op, $, $ | drop(1));
}
auto differ(auto&& rng) { return adjacent_transform_2(rng, _neq_); }
auto deltas(auto&& rng) { return adjacent_transform_2(rng, _c(_sub_)); }

auto indices(auto&& rng) {
    return rng
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $);
}
```



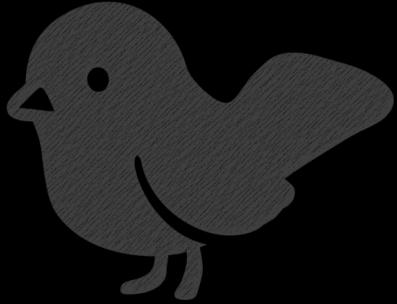
```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```



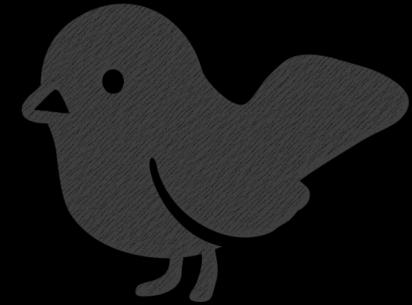
```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> differ($)
        |> indices($)
        |> concat($, single(sushi.size()))
        |> deltas($)
        |> adjacent_transform_2($, _min_)
        |> std::ranges::max($) * 2;
}
```



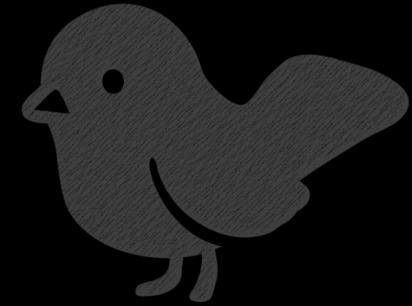
```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```



```
using namespace ranges::views;
using namespace combinator;

auto sushi_for_two(std::vector<int> sushi) {
    auto indices = concat(
        concat(single(0),
            zip(zip_with(_neq_, sushi, sushi | drop(1)),
                iota(1))
            | filter(_fst)
            | values),
        single(sushi.size())));
    auto deltas = zip_with(_sub_, indices | drop(1), indices);
    return 2 * ranges::max(zip_with(_min_, deltas, deltas | drop(1)));
}
```



```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```

[[digression]]

Loops are easier to read
/ understand ... everyone
knows them

0

@OMGclueless 5 days ago

Why does he show such a crazy solution for the for-loop-based solution? If you're allowed to be "clever" in the range-based solution you're allowed to be "clever" in the loop-based solution too. Here's a solution:

```
std::vector<int> runs;
int current_run = 1;
for (int i = 1; i < N; ++i) {
    if (sushi[i] == sushi[i - 1]) ++current_run;
    else { runs.push_back(current_run); current_run = 1; }
}
runs.push_back(current_run);

int max_run = runs[0];
for (int i = 1; i < runs.size(); ++i) {
    max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
}
return 2 * max_run;
```

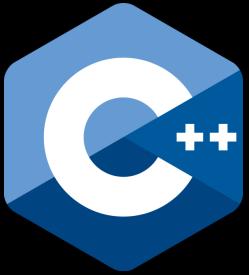
Show less



2



Reply



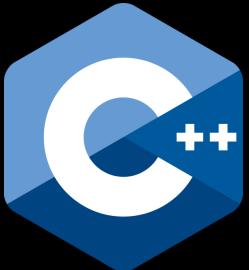
```
// original youtube comment from OMGcLueless
auto sushi_for_two(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < N; ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else { runs.push_back(current_run); current_run = 1; }
    }
    runs.push_back(current_run);

    int max_run = runs[0];
    for (int i = 1; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```



```
// fix compilation error (N should be sushi.size())
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else { runs.push_back(current_run); current_run = 1; }
    }
    runs.push_back(current_run);

    int max_run = runs[0];
    for (int i = 1; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```



```
sushi_for_two ({2, 2, 2, 1, 1, 2, 2}); // 4 (but is 6)
```

```
// fix compilation error (N should be sushi.size())
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else { runs.push_back(current_run); current_run = 1; }
    }
    runs.push_back(current_run);

    int max_run = runs[0];
    for (int i = 1; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```



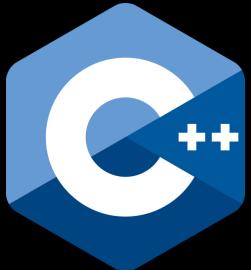
```
// fix compilation error (N should be sushi.size())
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else { runs.push_back(current_run); current_run = 1; }
    }
    runs.push_back(current_run);

    int max_run = runs[0];
    for (int i = 1; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```



```
// working
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else { runs.push_back(current_run); current_run = 1; }
    }
    runs.push_back(current_run);

    int max_run = std::min(runs[0], runs[1]);
    for (int i = 2; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```

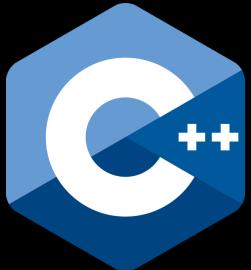


```
// working and formatted
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else {
            runs.push_back(current_run);
            current_run = 1;
        }
    }
    runs.push_back(current_run);

    int max_run = std::min(runs[0], runs[1]);
    for (int i = 2; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```



```
template <int N>
constexpr auto sushi_for_two(std::array<int, N> sushi) {
    int current_sushi      = 0;
    int sushi_in_a_row     = 0;
    int prev_sushi_in_a_row = 0;
    int max_of_mins        = 0;
    for (auto const s : sushi) {
        if (current_sushi != s) {
            current_sushi = s;
            if (prev_sushi_in_a_row == 0) {
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row     = 1;
            } else {
                auto const min      = std::min(sushi_in_a_row, prev_sushi_in_a_row);
                max_of_mins        = std::max(max_of_mins, min);
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row     = 1;
            }
        } else {
            sushi_in_a_row += 1;
        }
    }
    auto const min = std::min(sushi_in_a_row, prev_sushi_in_a_row);
    max_of_mins   = std::max(max_of_mins, min);
    return max_of_mins * 2;
}
```



```
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else {
            runs.push_back(current_run);
            current_run = 1;
        }
    }
    runs.push_back(current_run);

    int max_run = std::min(runs[0], runs[1]);
    for (int i = 2; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```



O(n) Memory

```
auto sushi_for_two_wrong(std::vector<int> sushi) {
    std::vector<int> runs;
    int current_run = 1;
    for (int i = 1; i < sushi.size(); ++i) {
        if (sushi[i] == sushi[i - 1]) ++current_run;
        else {
            runs.push_back(current_run);
            current_run = 1;
        }
    }
    runs.push_back(current_run);

    int max_run = std::min(runs[0], runs[1]);
    for (int i = 2; i < runs.size(); ++i) {
        max_run = std::max(max_run, std::min(runs[i], runs[i - 1]));
    }
    return 2 * max_run;
}
```

	OMGclueless's For Loops	Conor's For Loop	Ranges
--	--------------------------------	-------------------------	---------------

	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$

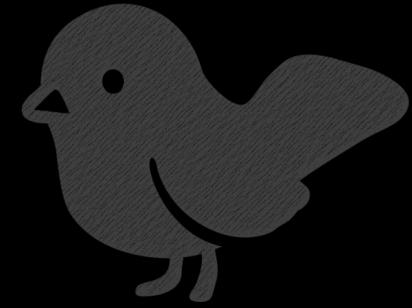
	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Memory Complexity	$O(n)$	$O(1)$	$O(1)$

	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Memory Complexity	$O(n)$	$O(1)$	$O(1)$
Bug Prone	Yes	Yes	No

	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Memory Complexity	$O(n)$	$O(1)$	$O(1)$
Bug Prone	Yes	Yes	No
Readability	Bad	Bad	Amazing

	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Memory Complexity	$O(n)$	$O(1)$	$O(1)$
Bug Prone	Yes	Yes	No
Readability	Bad	Bad	Amazing
Declarative	No (1 vec, 2 int)	No (4 int)	Yes

	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Memory Complexity	$O(n)$	$O(1)$	$O(1)$
Bug Prone	Yes	Yes	No
Readability	Bad	Bad	Amazing
Declarative	No (1 vec, 2 int)	No (4 int)	Yes
Easily Parallelizable	No	No	Yes

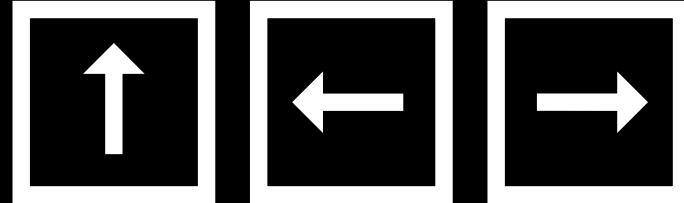


```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```

[[end of digression]]

Max Gap



<https://leetcode.com/problems/maximum-gap/>

[8, 4, 1, 3, 10]

[1, 3, 4, 8, 10]

[1, 3, 4, 8, 10]

[2, 1, 4, 2]

[1, 3, 4, 8, 10]

[2, 1, **4**, 2]



```
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    return nums;
}
```

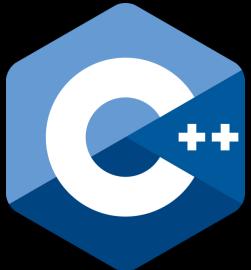
[8, 4, 1, 3, 10]



```
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return nums;
}
```

[1, 3, 4, 8, 10]



```
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return nums
        | adjacent_transform<2>(std::minus{});
}
```

[2, 1, 4, 2]



anyone notice the bug?

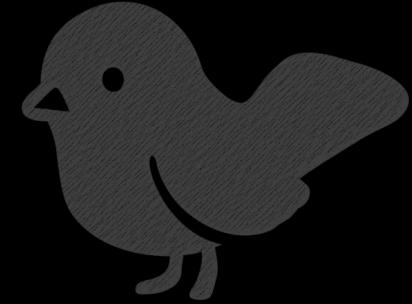
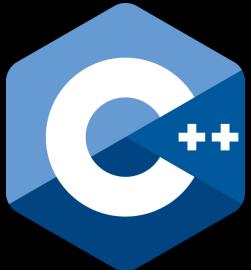
```
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return std::ranges::max(
        | adjacent_transform<2>(std::minus{})));
}
```



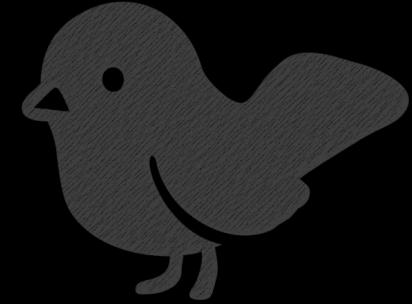
```
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return std::ranges::max(
        | reverse
        | adjacent_transform<2>(std::minus{}));
}
```



```
using namespace std::views;
using namespace combinator;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return std::ranges::max(nums
        | adjacent_transform<2>(_c(_sub_)));
}
```



```
using namespace std::views;
using namespace combinator;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return nums
        |> adjacent_transform<2>($, _c(_sub_))
        |> std::ranges::max($);
}
```

filter_out_html_tags

around first-parameter-passing. It would be nice to not have to pay more syntax when we don't need to.

On the whole though the argument seems to strongly favor placeholders, and if anything exploring a special case of the pipeline operator that does left-threading only and has a more restrictive right-hand side to avoid potential bugs. That might still allow the best of both worlds.

A recent [Conor Hoekstra talk](#) has a nice example that I'll present multiple different ways (in all cases, I will not use the `|` from Ranges).

With left-threading, the example looks like:

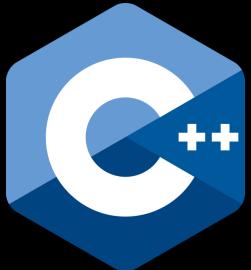
```
auto filter_out_html_tags(std::string_view sv) -> std::string {
    auto angle_bracket_mask =
        sv |> std::views::transform([](char c){ return c == '<' or c == '>'; });

    return std::views::zip_transform(
        std::logical_or(),
        angle_bracket_mask,
        angle_bracket_mask |> rv::scan_left(std::not_equal_to{})
    ) |> std::views::zip(sv)
    |> std::views::filter([](auto t){ return not std::get<0>(t); })
    |> std::views::values()
    |> std::ranges::to<std::string>();
}
```

Notably here we run into both of the limitations of left-threading: we need to pipe into a parameter other than the first and we need to pipe more than once. That requires introducing a new named variable, which is part of what this facility is trying to avoid the need for. This is not a problem for either of the placeholder-using models, as we'll see shortly.

With the placeholder-mandatory model, we don't need that temporary, since we can select which parameters of `zip_transform` to pipe into, and indeed we can pipe twice (I'll have more to say about nested placeholders later):

```
auto filter_out_html_tags(std::string_view sv) -> std::string {
    return sv
    |> std::views::transform(%, [](char c){ return c == '<' or c == '>'; })
    |> std::views::zip_transform(std::logical_or{}, %, % |> rv::scan_left(%,
        std::not_equal_to{}))
    |> std::views::zip(%, sv)
    |> std::views::filter(%, [](auto t){ return not std::get<0>(t); })
    |> std::views::values(%)
    |> std::ranges::to<std::string>(%);
```

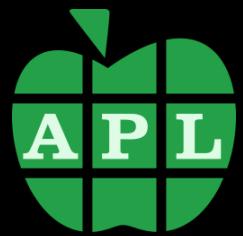


```
auto filter_out_html_tags(std::string_view sv) -> std::string {
    auto angle_bracket_mask =
        sv |> std::views::transform([](char c){ return c == '<' or c == '>'; });

    return std::views::zip_transform(
        std::logical_or(),
        angle_bracket_mask,
        angle_bracket_mask |> rv::scan_left(std::not_equal_to{})
    ) |> std::views::zip(sv)
    |> std::views::filter([](auto t){ return not std::get<0>(t); })
    |> std::views::values()
    |> std::ranges::to<std::string>();
}
```



```
'<div>Hello <b>C++North!</b></div>'
```

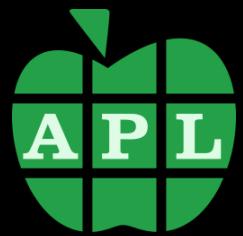


```
'<'='<div>Hello <b>C++North!</b></div>'
```

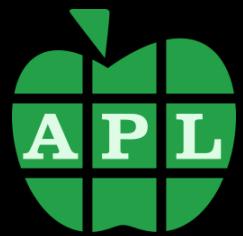


'<'='<div>Hello C++North!</div>'

1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0

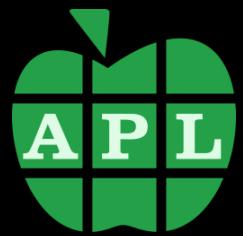


```
'<'='<div>Hello <b>C++North!</b></div>'  
10000000001000000000001000100000
```



'<'='<div>Hello C++North!</div>'

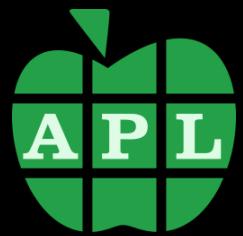
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0



'<' =

'<div>Hello C++North!</div>'

1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0



'<' =
'<div>Hello C++North!</div>'

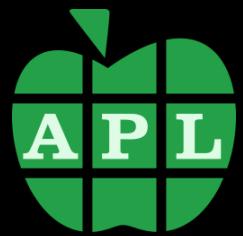
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0



{ ' < ' = ω }

' <div>Hello C++North!</div> '

1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0



{ $\omega \in ' < > '$ }

'<div>Hello C++North!</div>'

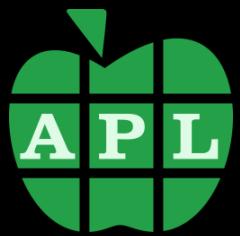
1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1



{≠\w€ ' <> ' }

'<div>Hello C++North!</div>'

1 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0



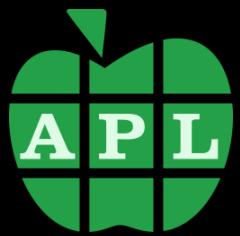
{~≠\ω∈'<>' }
'<div>Hello C++North!</div>'

0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 1



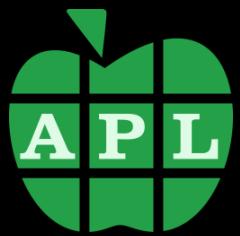
{ $\omega / \sim \neq \backslash \omega \in ' < > '$ }
'<div>Hello C++North!</div>'

0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1

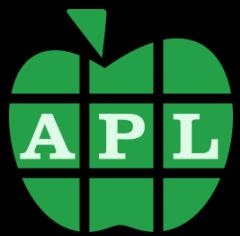


{ $\omega / \sim \neq \backslash \omega \in ' < > '$ }
'<div>Hello C++North!</div>'

< div > Hello < b > C + + N o r t h ! < / b > < / div >
0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1



```
{\omega/\sim\neq\wedge\in'<>' }  
'<div>Hello <b>C++North!</b></div>'  
>Hello >C++North!>>
```



```
{ω/⍨(~↑∨≠＼)ω∊'<>' }  
'<div>Hello <b>C++North!</b></div>'
```

Hello C++North!



```
{ω/⍨(⊤~≠＼)ω∈'<>' }  
'<div>Hello <b>C++North!</b></div>'
```

Hello C++North!



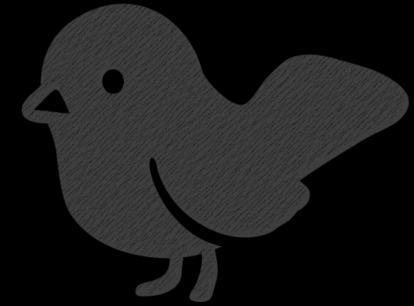
```
auto filter_out_html_tags(std::string_view sv) -> std::string {
    auto angle_bracket_mask =
        sv |> std::views::transform([](char c){ return c == '<' or c == '>'; });

    return std::views::zip_transform(
        std::logical_or(),
        angle_bracket_mask,
        angle_bracket_mask |> rv::scan_left(std::not_equal_to{}))
        |> std::views::zip(sv)
        |> std::views::filter([](auto t){ return not std::get<0>(t); })
        |> std::views::values()
        |> std::ranges::to<std::string>();
}
```



```
using namespace std::views;

auto filter_out_html_tags(std::string_view sv) {
    return sv
        |> transform($, [](auto e) { return e == '<' or e == '>'; })
        |> zip_transform(std::logical_or{}, $, scan_left($, true, std::not_equal_to{}))
        |> zip($, sv)
        |> filter($, [](auto t) { return not std::get<0>(t); })
        |> values($)
        |> ranges::to<std::string>($);
}
```



```
using namespace std::views;
using namespace combinators;

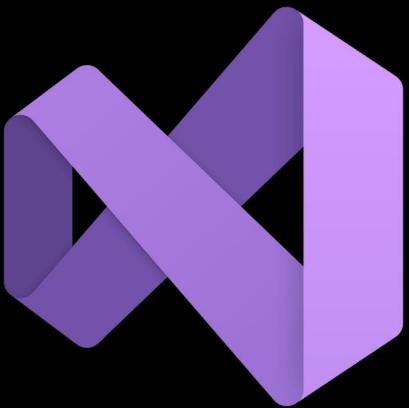
auto filter_out_html_tags(std::string_view sv) {
    return sv
        |> transform($, _phi(_eq('<'), _or_, _eq('>')))
        |> zip_transform(_or_, $, scan_left($, true, _neq_))
        |> zip($, sv)
        |> filter($, _b(_not, _fst))
        |> values($)
        |> ranges::to<std::string>($);
}
```

In Summary

C++20/23 Range Adaptors & Factories

drop	adjacent (pairwise)
drop_while	adjacent_transform (pairwise_transform)
elements (keys values)	cartesian_product
filter	chunk
iota	chunk_by
join	enumerate
reverse	join_with
split	slide
take	stride
take_while	zip
transform	zip_transform





- GCC 13
- MSVC v19.36 (VS 2022 17.6)
- Clang

Status

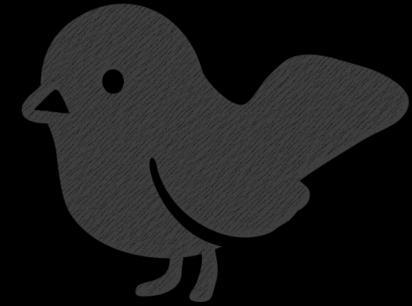
View/Feature	GCC	Clang	MSVC	MSVC PR
adjacent	●	●	●	PR: microsoft/STL#3508
adjacent_transform	●	●	●	PR: microsoft/STL#3546
cartesian_product	●	●	●	PR: microsoft/STL#3561
chunk	●	●	●	PR: microsoft/STL#2685
chunk_by	●	●	●	PR: microsoft/STL#2565
enumerate	●	●	●	PR: microsoft/STL#3472
join_with	●	●	●	PR: microsoft/STL#2619
pairwise	●	●	●	PR: microsoft/STL#3508
pairwise_transform	●	●	●	PR: microsoft/STL#3546
repeat	●	●	●	PR: microsoft/STL#3142
slide	●	●	●	PR: microsoft/STL#2670
stride	●	●	●	PR: microsoft/STL#2981
zip	●	●	●	PR: microsoft/STL#3035
zip_transform	●	●	●	PR: microsoft/STL#3322
ranges::to	●	●	●	PR: microsoft/STL#2806
Formatting	●	●	●	

- - Not Done
- - Started
- - Done (and available on Compiler Explorer)

	OMGclueless's For Loops	Conor's For Loop	Ranges
Time Complexity	$O(n)$	$O(n)$	$O(n)$
Memory Complexity	$O(n)$	$O(1)$	$O(1)$
Bug Prone	Yes	Yes	No
Readability	Bad	Bad	Amazing
Declarative	No (1 vec, 2 int)	No (4 int)	Yes
Easily Parallelizable	No	No	Yes*

Code Links

Example	Language	GitHub Link	Godbolt Link
C++23 Ranges Status	C++	Link	GCC: https://godbolt.org/z/vsfT6e96b
Warm Up: Negatives	C++	Link	https://godbolt.org/z/8TEevabqd
Sushi for Two	C++	Link	https://godbolt.org/z/69Kh8baz3
Sushi for Two	Circle	Link	https://godbolt.org/z/P6PxMnhMz
Sushi for Two (Bad / YouTube)	C++	Link	https://godbolt.org/z/hjbKb6jWM
Max Gap	C++	Link	https://godbolt.org/z/P43EhYcsj
Max Gap	Circle	Link	https://godbolt.org/z/3K598jMMa
filter_out_html_tags	Circle	Link	https://godbolt.org/z/on5xMG5ax



```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> chunk_by($, _eq_)
        |> transform($, std::ranges::distance)
        |> adjacent_transform<2>($, _min_)
        |> std::ranges::max($) * 2;
}
```



Thank You

<https://github.com/codereport/Content/Talks>

Conor Hoekstra

 code_report

 codereport



Questions?

<https://github.com/codereport/Content/Talks>

Conor Hoekstra

 code_report

 codereport