



A L G O R I T H M
I N T U I T I O N

Conor Hoekstra

code_report

codereport

cppcon the c++ conference 2019

“I’m not an expert,
I’m just a dude.”

- Scott Schurr, CppCon 2015

constexpr: Applications

By Scott Schurr for Ripple Labs at CppCon September 2015



SCOTT SCHURR

constexpr:
Applications

About Me

- I love C++
- I've been coding in C++ for 5 years*
- I love **auto** (AAA)
- I prefer **east const** (2 const wests)
- I prefer “east end functions”



**Phil Nash**

We have always
been at war with
West Consta

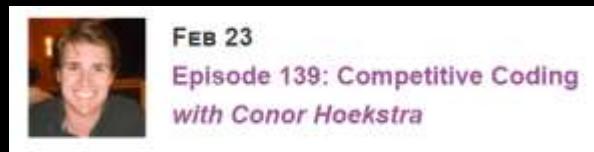
Video Sponsorship
Provided By:

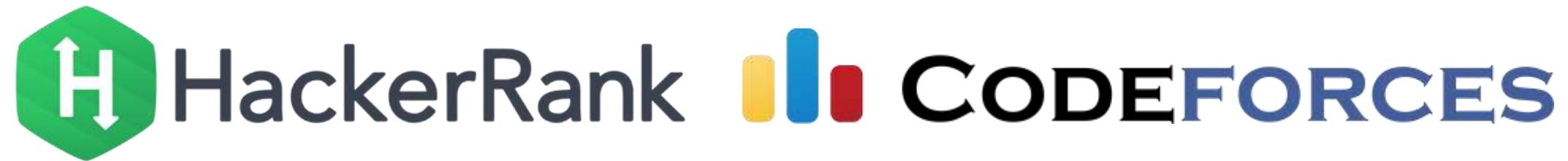


	C++	auto factorial(int n) -> int
	Rust	fn factorial(n: i32) -> i32
	Swift	func factorial(of : Int) -> Int
	Python3	def factorial(n: int) -> int
	Haskell	factorial :: (Integral a) => a -> a
	Go	func factorial(n int) int
	Maths	f(x) -> y
	Kotlin	fun factorial(a: Int): Int

About Me

- I love C++
- I've been coding in C++ for 5 years*
- I love **auto** (AAA)
- I prefer **east const**
- I prefer “east end functions”
- I love (STL) algorithms
- Very interested in programming languages
- I like to compete in competitive programming contests

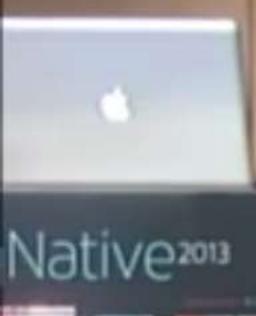




Let's go back in time...







STL Algorithms - How to use them; how to write your own

Marshall Clow
Qualcomm, Inc.

CppCon 2016

mclow@qti.qualcomm.com

@mclow

CppCon.org



MARSHALL CLOW

**STL Algorithms -
How you should use them;
how to write your own**

Simplicity is Not Just for Beginners



Idioms, Library Abstractions, Commonality

- These are old friends you can learn to recognize too
- This loop touches every element in the collection; I should use a ranged for instead of a traditional for loop
 - Or something from <algorithm>



We need to know them. All. Well. Very well.

STL algorithms can make code simpler

Sometimes, it can be spectacular.

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() || i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            }
            break;
        }
    }
}
```



```
// Next, check if the panel has moved to the left side of another panel.
auto f = begin(expanded_panels_) + fixed_index;
auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [] (const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });
// If it has, then we reorder the panels.
rotate(p, f, f + 1);
```

@JoBocca

12

@ACCUconf



accu
2018
April 11 - 14

code::dive 2018



@odinthenerd

```
auto result = std::accumulate(  
    v.begin(),  
    v.end(),  
    0,  
    better_foo);
```

Auto-Intern GmbH

48

Odin Holmes

What I Wish They Told Me - Part 2 of 2

“... and just as you can say, that would be a good use of a linked list, we don’t have that **intuition** about **algorithms** yet, and we need to.”



“... and just as you can say, that would be a good use of a linked list, we don’t have that **intuition** about **algorithms** yet, and we need to.”

- Kate Gregory



STL implementation

```
template <class I, // I models ForwardIterator
          class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);

        if (*m < v) f = next(m);
        else l = m;
    }
    return f; // Yellow arrow points here
}
```



**SEAN
PARENT**

Generic Programming

“It’s this **phenomenal** piece of work and for me,
it was like, **I want to write code like that.**”

- Sean Parent

Generic Programming, Pacific++ 2018

That is what motivated this talk ...

- wanting to develop algorithm intuition
- wanting to write “phenomenal” code
- wanting to write “beautiful code”







Goal (for you)

- Get you excited about algorithms
- Learn a new algorithm
- Start to develop some **algorithm intuition**

Interview Warm-up Question

Given an array of integers, find the difference between the minimum and maximum?

Guaranteed to have non-empty list



// Solution 1a

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto ans = numeric_limits<int>::min();
    for (int i = 0; i < v.size(); ++i) {
        for (int j = 0; j < v.size(); ++j) {
            ans = max(ans, abs(v[i] - v[j]));
        }
    }
    return ans;
}
```



```
// Solution 1b
```

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto ans = numeric_limits<int>::min();
    for (auto a : v) {
        for (auto b : v) {
            ans = max(ans, abs(a - b));
        }
    }
    return ans;
}
```



// Solution 2a

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    sort(begin(v), end(v));
    return *--end(v) - *begin(v);
}
```



// Solution 2b

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    sort(begin(v), end(v));
    return *rbegin(v) - *begin(v);
}
```



// Solution 2c

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    sort(begin(v), end(v));
    return v.back() - v.front();
}
```



// Solution 3a

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto a = numeric_limits<int>::max();
    auto b = numeric_limits<int>::min();
    for (auto e : v) {
        a = min(a, e);
        b = max(b, e);
    }
    return b - a;
}
```



// Solution 3b

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto a = *min_element(begin(v), end(v));
    auto b = *max_element(begin(v), end(v));
    return b - a;
}
```



```
// Solution 3c (C++20 Ranges)

auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto a = *min_element(v);
    auto b = *max_element(v);
    return b - a;
}
```



// Solution 4a

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto p = minmax_element(begin(v), end(v));
    return *p.second - *p.first;
}
```



// Solution 4b

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto [a, b] = minmax_element(begin(v), end(v));
    return *b - *a;
}
```



// Solution 4c (C++20)

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto [a, b] = minmax_element(v);
    return *b - *a;
}[[ Slideware Disclaimer ]]
```



- Should be using `std::` namespace
- `solve` is a terrible function name
- `a` and `b` are terrible variable names
- Failing SOLID

[[digression]]

Continuous Learning

Clint Shank

The Pragmatic Programmer

WE ALL know in the software industry that you can't afford to stand still. If you're not learning, you're falling behind.

Here's a list of ways to keep you up-to-date on the Internet for free:

- Read books, magazines, blogs, and newsgroups related to your field to go deeper into a subject, or to learn about a new one.
- If you really want to get immersed in a subject, try writing some code.
- Always try to work with a mentor or teacher, if possible. Formal education. Although you can learn a lot from books, there's nothing like learning a whole lot more from a real person. If you can't find a mentor, consider becoming one yourself.
- Use virtual mentors. Find a mentor you really like and read everything they write.
- Get to know the framework you're working with. If something works makes you happy, then it's a good source, you're really in luck. You'll be able to see what's going on under the hood and have it explained to you by some really smart people.



from just

Andrew
David

Foreword by Ward

- **Buy low, sell high.** Learning an emerging technology before it becomes popular can be just as hard as finding an undervalued stock, but the payoff can be just as rewarding. Learning Java when it first came out may have been risky, but it paid off handsomely for the early adopters who are now at the top of that field.
- **Review and rebalance.** This is a very dynamic industry. That hot technology you started investigating last month might be stone cold next year.

Goals

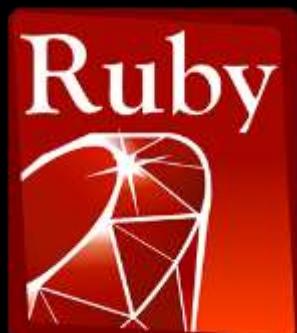
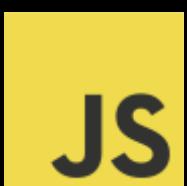
Now that you have some guidelines on what and when to add to your knowledge portfolio, what's the best way to go about acquiring intellectual capital with which to fund your portfolio? Here are a few suggestions.

- **Learn at least one new language every year.** Different languages solve the same problems in different ways. By learning several different approaches, you can help broaden your thinking and avoid getting stuck in a rut. Additionally, learning many languages is far easier now, thanks to the wealth of freely available software on the Internet (see page 267).
- **Read a technical book each quarter.** Bookstores are full of technical books on interesting topics related to your current project. Once you're in the habit, read a book a month. After you've mastered the technologies you're currently using, branch out and study some that don't relate to your project.
- **Read nontechnical books, too.** It is important to remember that computers are used by people—people whose needs you are trying to satisfy. Don't forget the human side of the equation.



// Solution 4a

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto p = minmax_element(begin(v), end(v));
    return *p.second - *p.first;
}
```



Many, but not all, methods that mutate their receiver use `!` as the last character of their name. However, this is not guaranteed to be the case. For instance, `string#concat!` is a mutating method, but it does not include a `!`.

[[digression²]]

What does “!” mean?

[[end digression²]]



```
// Solution 4a

auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto p = minmax_element(begin(v), end(v));
    return *p.second - *p.first;
}
```



```
// Solution 4a

auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto p = minmax_element(cbegin(v), cend(v));
    return *p.second - *p.first;
}
```



```
// Solution 4a

auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto p = minmax_element(cbegin(v), cend(v));
    return *p.second - *p.first;
}
```

[[end digression]]



```
def solve() -> int:  
    l = [ 2, 1, 3, 5, 4 ]  
    return max(l) - min(l)
```



```
public static int solve() {  
    List<Integer> l = Arrays.asList( 2, 1, 5, 3, 4 );  
    int a = Collections.min(l);  
    int b = Collections.max(l);  
    return b - a;  
}
```



```
solve :: [Int] -> Int  
solve xs = maximum xs - minimum xs
```

105 Algorithms?

We need to know them. All. Well. Very well.

STL algorithms can make code simpler

Sometimes, it can be spectacular.

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x < panel->cur_panel_center() || i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            }
        }
    }
}
```



```
// Next, check if the panel has moved to the left side of another panel.
auto f = begin(expanded_panels_) + fixed_index;
auto p = lower_bound(begin(expanded_panels_), f, center_x,
    [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });
// If it has, then we reorder the panels.
rotate(p, f, f + 1);
```

12

@JoBoccara



@ACCUconf

ACCU
2018
April 11 - 14

Hi, I'm Jonathan Boccara!

@JoBoccara

Fluent {C++}
EXPRESSIVE CODE IN C++



JONATHAN BOCCARA

105 STL Algorithms in
Less Than an Hour



71 Algorithms
+ 32 Runes

103 Algorithms





71 Algorithms
+ 32 Runes

103 Algorithms

105 Algorithms?
What are the missing 2?

clamp
destroy_at
find_if_not
iter_swap
max
min
minmax
swap





Conor Hoekstra

@code_report

@JoBoccara What are the two missing algorithms that get us to 105? clamp?
destroy_at? find_if_not? iter_swap? max? min?



Jonathan Boccara @JoBoccara · May 5

Replying to @code_report @cppnow

Hi, `find_if_not` is indeed counted as it's in the slides. Not the others you're mentioning as they doesn't take a begin and end. I've looked up your talk's abstract at C++Now. Would love to watch it when it's online (I won't be there this year)



1:30 AM - 1 May 2019

clamp

destroy_at

find_if_not

iter_swap

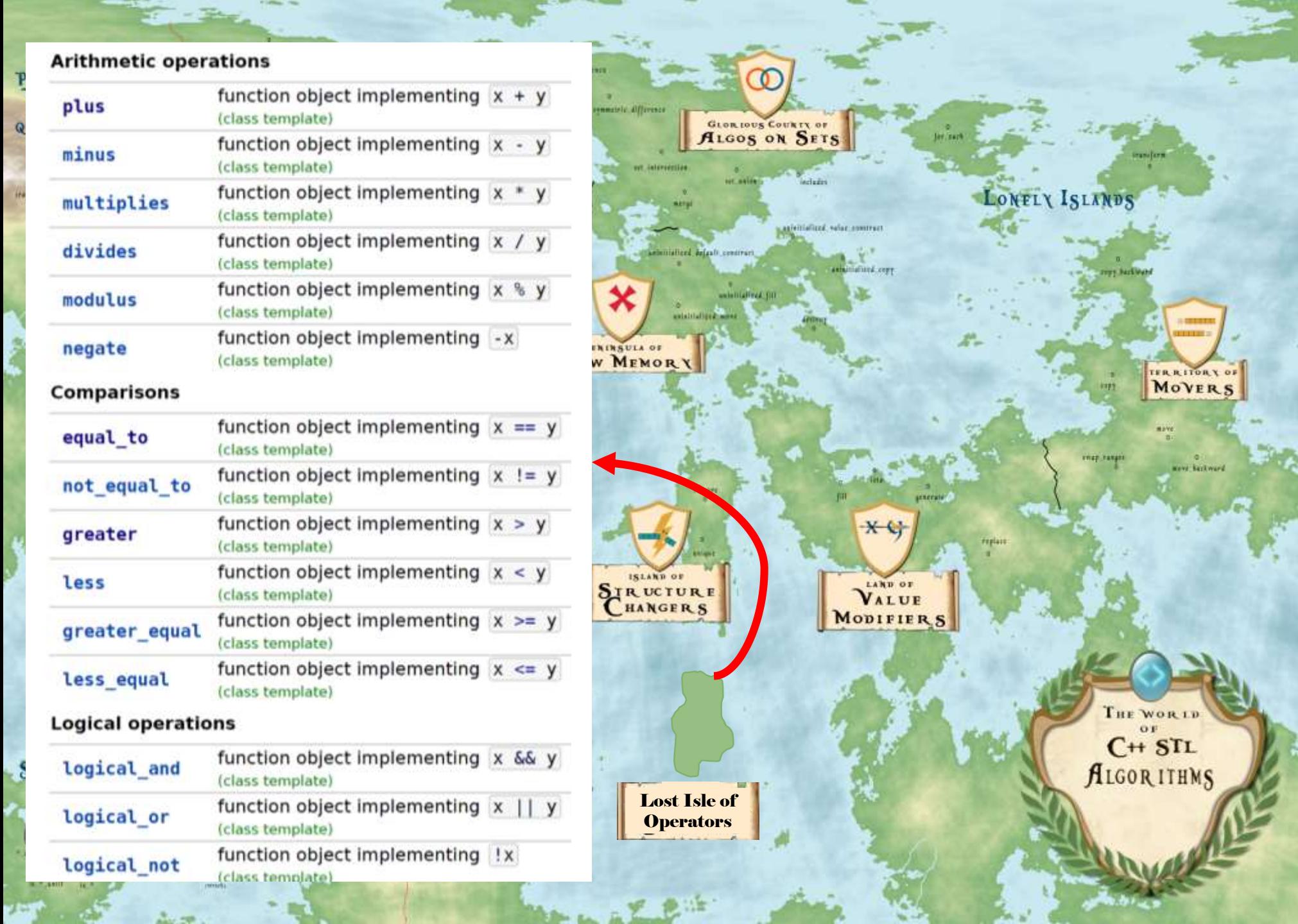
max

min

minmax

swap

clamp
destroy_at
find_if_not
iter_swap
max
min
minmax
swap



Library	Pre-C++11	C++11	C++17	Grand Total
<algorithm>	66	19	3, -1	87*
<numeric>	4	1	6	11
<memory>	3	1	9	13
Grand Total	73	21	17*	111

Can anyone name one of the **four original numeric** algorithms?

Can anyone name the **one C++11 numeric** algorithms?

Library	Pre-C++11	C++11	C++17	Grand Total
<algorithm>	66	19	3, -1	87*
<numeric>	4	1	6	11
<memory>	3	1	9	13
Grand Total	73	21	17*	111

<code>accumulate</code>	<code>partial_sum</code>
<code>adjacent_difference</code>	<code>reduce</code>
<code>exclusive_scan</code>	<code>transform_exclusive_scan</code>
<code>inclusive_scan</code>	<code>transform_inclusive_scan</code>
<code>inner_product</code>	<code>transform_reduce</code>
<code>iota</code>	

Pre-C++11

C++11

C++17



```
vector<int> v(10);
iota(begin(v), end(v), 1);

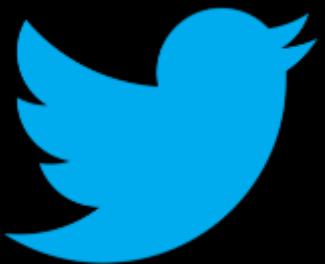
// 1 2 3 4 5 6 7 8 9 10
```



```
vector<int> v(10);
iota(rbegin(v), rend(v), 1);

// 10 9 8 7 6 5 4 3 2 1
```

[[digression]]



#iotashaming

```

// A sample standard C++20 program that prints
// the first N Pythagorean triples.

#include <iostream>
#include <optional>
#include <ranges> // New header!
using namespace std;

// maybe_view defines a view over zero or one
// objects.
template<Semiregular T>
struct maybe_view : view_interface<maybe_view<T>> {
    maybe_view() = default;
    maybe_view(T t) : data_(std::move(t)) {
    }
    T const *begin() const noexcept {
        return data_ ? &*data_ : nullptr;
    }
    T const *end() const noexcept {
        return data_ ? &*data_ + 1 : nullptr;
    }
private:
    optional<T> data_{};
};

// "for_each" creates a new view by applying a
// transformation to each element in an input
// range, and flattening the resulting range of
// ranges.
// (This uses one syntax for constrained lambdas
// in C++20.)
inline constexpr auto for_each =
    []<Range R,
     Iterator I = iterator_t<R>,
     IndirectUnaryInvocable<I> Fun>(R&& r, Fun fun)
        requires Range<indirect_result_t<Fun, I>> {
    return std::forward<R>(r)
        | view::transform(std::move(fun))
        | view::join;
};

// "yield_if" takes a bool and a value and
// returns a view of zero or one elements.
inline constexpr auto yield_if =
    []<Semiregular T>(bool b, T x) {
    return b ? maybe_view{std::move(x)}
             : maybe_view<T>{};
};

int main() {
    // Define an infinite range of all the
    // Pythagorean triples:
    using view::iota;
    auto triples =
        for_each(iota(1), [](int z) {
            return for_each(iota(1, z+1), [=](int x) {
                return for_each(iota(x, z+1), [=](int y) {
                    return yield_if(x*x + y*y == z*z,
                        make_tuple(x, y, z));
                });
            });
        });
    // Display the first 10 triples
    for(auto triple : triples | view::take(10)) {
        cout << '('
            << get<0>(triple) << ','
            << get<1>(triple) << ','
            << get<2>(triple) << ')' << '\n';
    }
}

```

```

// A sample standard C++20 program that prints
// the first N Pythagorean triples.
#include <iostream>
#include <optional>
#include <ranges> // New header!
using namespace std;

template<Semiregular T>
struct maybe_view : view_interface<maybe_view<T>> {
    maybe_view() = default;
    maybe_view(T t) : data_(std::move(t)) {
    }
    T const *begin() const noexcept {
        return data_ ? &*data_ : nullptr;
    }
    T const *end() const noexcept {
        return data_ ? &*data_ + 1 : nullptr;
    }
private:
    optional<T> data_{};
};

inline constexpr auto for_each =
    []<Range R,
     Iterator I = iterator_t<R>,
     IndirectUnaryInvocable<I> Fun>(R&& r, Fun fun)
        requires Range<indirect_result_t<Fun, I>> {
    return std::forward<R>(r)
        | view::transform(std::move(fun))
        | view::join;
};

```

```

inline constexpr auto yield_if =
    []<Semiregular T>(bool b, T x) {
        return b ? maybe_view{std::move(x)}
                  : maybe_view<T>{};
};

int main() {
    using view::iota;
    auto triples =
        for_each(iota(1), [](int z) {
            return for_each(iota(1, z+1), [=](int x) {
                return for_each(iota(x, z+1), [=](int y) {
                    return yield_if(x*x + y*y == z*z,
                                   make_tuple(x, y, z));
                });
            });
        });
    for(auto triple : triples | view::take(10)) {
        cout << '('
            << get<0>(triple) << ','
            << get<1>(triple) << ','
            << get<2>(triple) << ')' << '\n';
    }
}

```



+ / + / `` l `` 4 44



```
iota n = [1..n]
sum $ map (sum . iota) [4, 44]
```



```
auto numberOfTwitterFollowers() -> int {
    std::vector v = { 4, 44 };
    return v | rv::transform([](auto i) {
        return rv::iota(1)
            | rv::take(i)
            | hs::sum(); })
            | hs::sum();
}
```

[[end digression]]

accumulate	partial_sum
adjacent_difference	reduce
exclusive_scan	transform_exclusive_scan
inclusive_scan	transform_inclusive_scan
inner_product	transform_reduce
iota	<input checked="" type="checkbox"/>

Pre-C++11

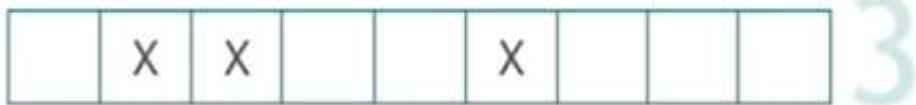
C++11

C++17

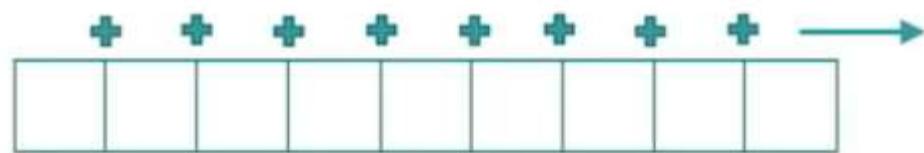
NUMERIC ALGORITHMS



count



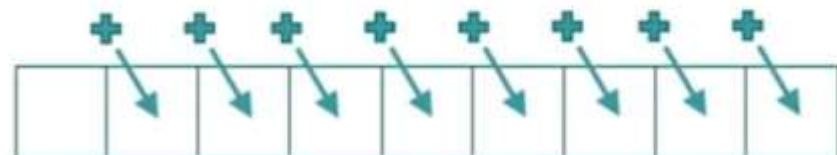
accumulate/(transform_)reduce



partial_sum

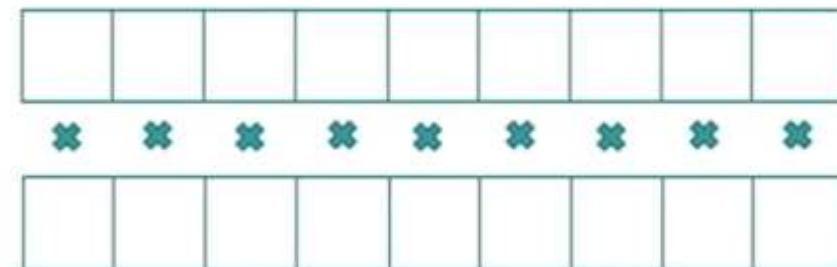
(transform_)inclusive_scan

(transform_)exclusive_scan

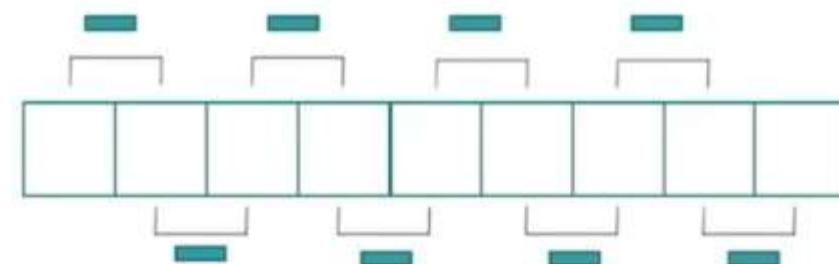


@JoBoccara

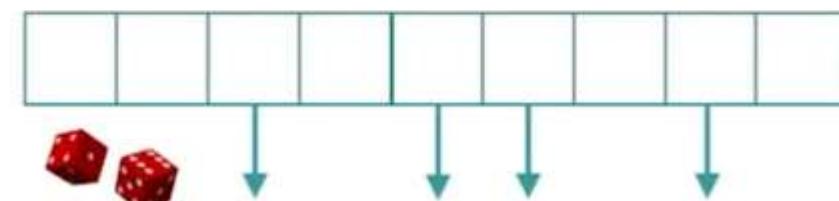
inner_product



adjacent_difference



sample





```
vector v = { 1, 2, 3 };  
auto x = accumulate(cbegin(v), cend(v), 0);
```



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

auto x = accumulate(cbegin(v), cend(v), 0);
auto y = accumulate(cbegin(v), cend(v), 0, plus{});
```



Not the best name ...

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

```
auto x = accumulate(cbegin(v), cend(v), 0);
```

```
auto y = accumulate(cbegin(v), cend(v), 1, multiplies{});
```



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

auto x = reduce(cbegin(v), cend(v));
auto y = reduce(cbegin(v), cend(v), 0, plus{});
auto z = reduce(cbegin(v), cend(v), 1, multiplies{});
```

Fold (higher-order function)

From Wikipedia, the free encyclopedia

In functional programming, **fold** (also termed **reduce**, **accumulate**, **aggregate**, **compress**, or **inject**) refers to a family of higher-order functions that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value. Typically, a fold is presented with a combining **function**, a top **node** of a **data structure**, and possibly some default values to be used under certain conditions. The fold then proceeds to combine elements of the data structure's **hierarchy**, using the function in a systematic way.

Folds are in a sense dual to **unfolds**, which take a **seed** value and apply a function **corecursively** to decide how to progressively construct a **corecursive** data structure, whereas a fold recursively breaks that structure down, replacing it with the results of applying a combining function at each node on its **terminal** values and the recursive results (**catamorphism**, versus **anamorphism** of unfolds).

catamorphism





`fn fold<B, F>(self, init: B, f: F) -> B`

where

`F: FnMut(B, Self::Item) -> B,`

An iterator method that applies a function, producing a single, final value.

`fold()` takes two arguments: an initial value, and a closure with two arguments: an 'accumulator', and an element. The closure returns the value that the accumulator should have for the next iteration.

The initial value is the value the accumulator will have on the first call.

After applying this closure to every element of the iterator, `fold()` returns the accumulator.

This operation is sometimes called 'reduce' or 'inject'.

Folding is useful whenever you have a collection of something, and want to produce a single value from it.

Note: `fold()`, and similar methods that traverse the entire iterator, may not terminate for infinite iterators, even on traits for which a result is determinable in finite time.



Template std.algorithm.iteration.fold

Implements the homonym function (also known as accumulate, compress, inject, or foldl) present in various programming languages of functional flavor. The call fold!(fun)(range, seed) first assigns seed to an internal variable result, also called the accumulator. Then, for each element x in range, result = fun(result, x) gets evaluated. Finally, result is returned. The one-argument version fold!(fun)(range) works similarly, but it uses the first element of the range as the seed (the range must be non-empty).

Template std.algorithm.iteration.reduce

Implements the homonym function (also known as accumulate, compress, inject, or foldl) present in various programming languages of functional flavor. There is also fold which does the same thing but with the opposite parameter order. The call reduce!(fun)(seed, range) first assigns seed to an internal variable result, also called the accumulator. Then, for each element x in range, result = fun(result, x) gets evaluated. Finally, result is returned. The one-argument version reduce!(fun)(range) works similarly, but it uses the first element of the range as the seed (the range must be non-empty).

We can use reduce to implement other algorithms...



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

auto x = my::count(cbegin(v), cend(v), 1);
auto y = std::count(cbegin(v), cend(v), 1);
```



```
namespace my {
    template<class I, class T>
    auto count(I f, I l, const T& val) -> int {
        return std::reduce(f, l, 0,
                           [val](auto a, auto b) { return a + (b == val); });
    }
}
```



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

auto x = my::any_of(cbegin(v), cend(v), [] (auto e) { return e == 3; });
auto y = std::any_of(cbegin(v), cend(v), [] (auto e) { return e == 3; });
```



```
namespace my {
    template<class I, class P>
    auto any_of(I f, I l, P p) -> bool {
        return std::reduce(f, l, false,
                           [p](auto a, auto b) { return a || p(b); });
    }
}
```





BEN DEANE

`std::accumulate:`
Exploring an
Algorithmic Empire

THE ALGORITHMS (PRE-C++17)

accumulate	adjacent_difference	adjacent_find	all_of	any_of
binary_search	copy	copy_backward	copy_if	copy_n
count	count_if	equal	equal_range	fill
fill_n	find	find_end	find_first_of	find_if
find_if_not	for_each	generate	generate_n	includes
inner_product	inplace_merge	iota	is_heap	is_heap_until
is_partitioned	is_permutation	is_sorted	is_sorted_until	
lexicographical_compare	lower_bound	make_heap	max	max_element
merge	min	min_element	minmax	minmax_element
mismatch	move	move_backward	next_permutation	none_of
nth_element	partial_sort	partial_sort_copy	partial_sum	partition
partition_copy	partition_point	pop_heap	prev_permutation	push_heap
		remove	remove_copy	remove_if
replace	replace_copy	replace_copy_if	replace_if	reverse
reverse_copy	rotate	rotate_copy	search	search_n
set_difference	set_intersection	set_symmetric_difference	set_union	shuffle
sort	sort_heap	stable_partition	stable_sort	

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

std::execution::sequenced_policy, std::execution::parallel_policy,
std::execution::parallel_unsequenced_policy

Defined in header <execution>
class sequenced_policy { /* unspecified */ }; (1) (since C++17)
class parallel_policy { /* unspecified */ }; (2) (since C++17)
class parallel_unsequenced_policy { /* unspecified */ }; (3) (since C++17)

```
auto x = reduce(cbegin(v), cend(v));  
auto y = reduce(cbegin(v), cend(v), 0, plus<>());  
auto z = reduce(cbegin(v), cend(v), 1, multiplies<>());
```

std::accumulate vs.

std::execution::seq – do not parallelise
std::execution::par – parallelise
std::execution::par_unseq – parallelise and vectorise (requires that the operation can be interleaved, so no acquiring mutexes and such)



<https://blog.tartanllama.xyz/accumulate-vs-reduce/>

<https://www.youtube.com/watch?v=FJIn1YhPJc>

accumulate	<input checked="" type="checkbox"/>	partial_sum
adjacent_difference		reduce
exclusive_scan		transform_exclusive_scan
inclusive_scan		transform_inclusive_scan
inner_product		transform_reduce
iota	<input checked="" type="checkbox"/>	

Pre-C++11

C++11

C++17



Jon Kalb is going to give Michael and Odin each one coin. Jon has N coins with different values. Jon wants the absolute difference between the value of the coins to be minimized so Michael and Odin don't fight with each other. Given an array of coin values, help Jon find this minimum.



Find the minimum difference between two values in a list.

1 4 2 -> 1

1 3 3 -> 0



```
auto min_value(vector<int>& coins) -> int {
    sort(begin(coins), end(coins));
    vector<int> diff(coins.size());
    adjacent_difference(cbegin(coins), cend(coins), begin(diff));
    return *min_element(cbegin(diff) + 1, cend(diff));
}
```



What's wrong with 2nd line?
Do we O(n) space?
Think **catamorphism**

```
auto min_value(vector<int>& c) -> int {  
    sort(begin(c), end(c));  
    vector<int> d(c.size());  
    adjacent_difference(cbegin(c), cend(c), begin(d));  
    return *min_element(cbegin(d) + 1, cend(d));  
}
```



```
auto min_value(vector<int>& c) -> int {
    sort(begin(c), end(c));
    return reduce(cbegin(c) + 1, cend(c), numeric_limits<int>::max(),
        [prev = c.front()](auto a, auto b) mutable {
            auto d = b - prev;
            prev = b;
            return min(a, d);
        });
}
```

accumulate	<input checked="" type="checkbox"/>	partial_sum
adjacent_difference	<input checked="" type="checkbox"/>	reduce
exclusive_scan		transform_exclusive_scan
inclusive_scan		transform_inclusive_scan
inner_product		transform_reduce
iota	<input checked="" type="checkbox"/>	

Pre-C++11

C++11

C++17



End of Part 1!

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

Conor Hoekstra

 code_report

 codereport



Questions?

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

Conor Hoekstra

-  code_report
-  codereport

Part 2



A L G O R I T H M
I N T U I T I O N

Conor Hoekstra

code_report

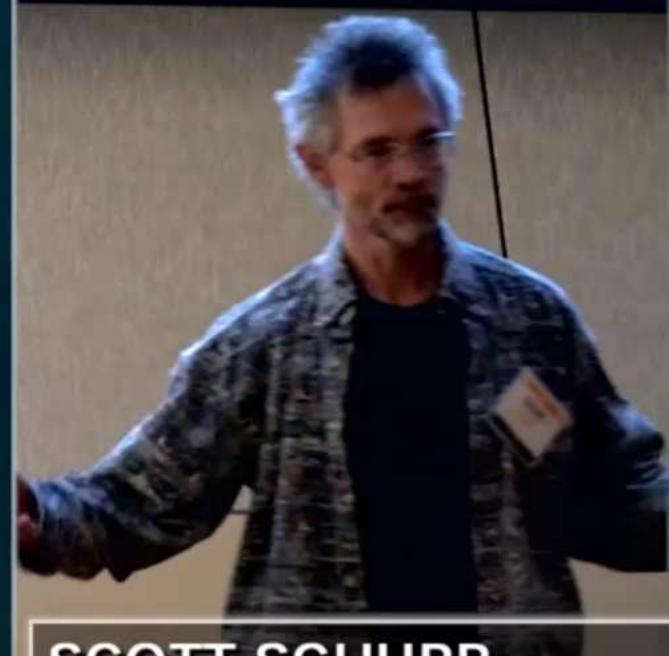
codereport

cppcon the c++ conference 2019

Part 1 Summary in
<3 Minutes

constexpr: Applications

By Scott Schurr for Ripple Labs at CppCon September 2015



SCOTT SCHURR

constexpr:
Applications





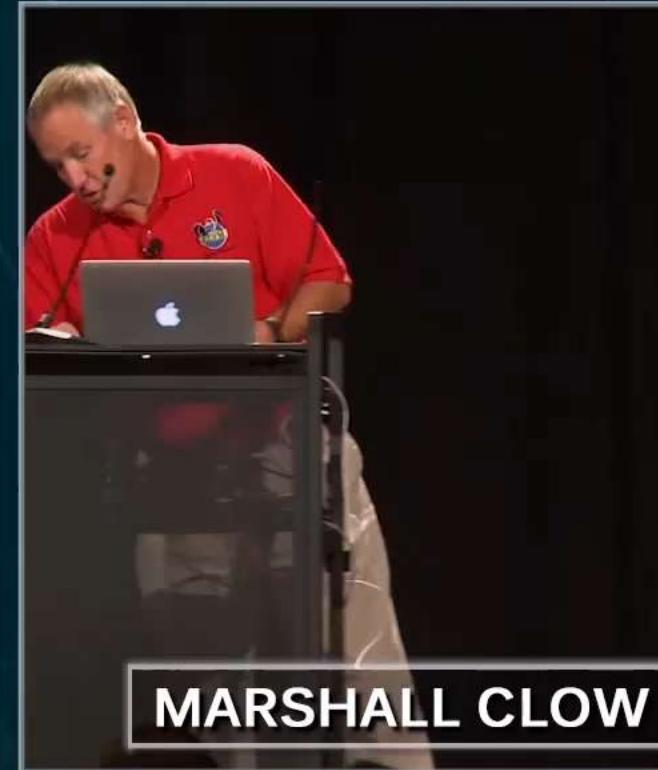
STL Algorithms - How to use them; how to write your own

Marshall Clow
Qualcomm, Inc.

CppCon 2016

mclow@qti.qualcomm.com

@mclow



MARSHALL CLOW

**STL Algorithms -
How you should use them;
how to write your own**

CppCon.org

“... and just as you can say, that would be a good use of a linked list, we don’t have that **intuition** about **algorithms** yet, and we need to.”

- Kate Gregory





Goal (for you)

- Get you excited about algorithms
- Learn a new algorithm
- Start to develop some **algorithm intuition**



// Solution 4c (C++20)

```
auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };
    auto [a, b] = minmax_element(v);
    return *b - *a;
}
```

Library	Pre-C++11	C++11	C++17	Grand Total
<algorithm>	66	19	3, -1	87*
<numeric>	4	1	6	11
<memory>	3	1	9	13
Grand Total	73	21	17*	111

accumulate	partial_sum
adjacent_difference	reduce
exclusive_scan	transform_exclusive_scan
inclusive_scan	transform_inclusive_scan
inner_product	transform_reduce
iota	<input checked="" type="checkbox"/>

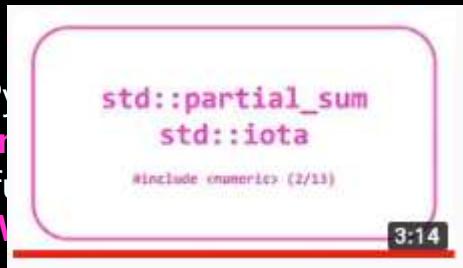
Pre-C++11

C++11

C++17



That example for Py
other features **source**
ranges can be useful
terrible example! W



+20 ranges and
I get that
etc. Still, a
code like that?!



```
▽DET[0]▽  
▽ Z←DET A;B;P;I  
[1] I←IO  
[2] Z←1  
[3] L: P←(|A[,I])↑↑/|A[,I]  
[4] →(P=I)/LL  
[5] A[I,P,]←A[P,I,]  
[6] Z←-Z  
[7] LL: Z←Z×B+A[I,I]  
[8] →(0 1 v.=Z,1↑pA)/0  
[9] A←1 1 →A-(A[,I]÷B)○.×A[I,]  
[10] →L  
[11] □EVALUATES A DETERMINANT  
▽
```



The developer
senior mem
of the histo

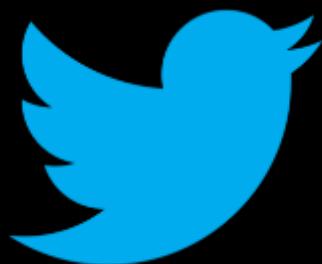
The name `iota` is taken from the programming language API. In the Turing Award lecture, Ken Thompson (inventor of API) said this:

For example, the integer function denoted by `i` produces a vector of the first `n` integers when applied to the argument `v`.

That `i` is the lower-case Greek letter iota.

In the quote above, I typed `\u03b9`, U+03b9, "GREEK SMALL LETTER IOTA", but Unicode actually has a dedicated code point for API's iota: `\u23f3` is U+23F3, "API FUNCTIONAL SYMBOL IOTA".

#iotashaming



"C++ way of creating lambdas, and the choice
of C++ standard to make things look clever
**("what's an iota? it's a Greek letter, look how
smart I am!")** are both a bit cumbersome."



on's idea
That is
sent an inclusion map in mathematics.

on to work to improve our profession. The more
the community. Giving back can take many forms;
tees and furthering open source projects. Part
ders and work through books. **Not knowing the
ud of, but an embarrassment.**

accumulate	<input checked="" type="checkbox"/>	partial_sum
adjacent_difference		reduce
exclusive_scan		transform_exclusive_scan
inclusive_scan		transform_inclusive_scan
inner_product		transform_reduce
iota	<input checked="" type="checkbox"/>	

Pre-C++11

C++11

C++17

Fold (higher-order function)

From Wikipedia, the free encyclopedia

In functional programming, **fold** (also termed **reduce**, **accumulate**, **aggregate**, **compress**, or **inject**) refers to a family of higher-order functions that analyze a recursive data structure and through use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value. Typically, a fold is presented with a combining **function**, a top **node** of a **data structure**, and possibly some default values to be used under certain conditions. The fold then proceeds to combine elements of the data structure's **hierarchy**, using the function in a systematic way.

Folds are in a sense dual to **unfolds**, which take a **seed** value and apply a function **corecursively** to decide how to progressively construct a **corecursive** data structure, whereas a fold recursively breaks that structure down, replacing it with the results of applying a combining function at each node on its **terminal** values and the recursive results (**catamorphism**, versus **anamorphism** of unfolds).

catamorphism





```
fn fold<B, F>(self, init: B, f: F) -> B
```

where

```
F: FnMut(B, Self::Item) -> B,
```

An iterator method that applies a function, producing a single, final value.

`fold()` takes two arguments: an initial value, and a closure with two arguments: an 'accumulator', and an element. The closure returns the value that the accumulator should have for the next iteration.

The initial value is the value the accumulator will have on the first call.

After applying this closure to every element of the iterator, `fold()` returns the accumulator.

This operation is sometimes called 'reduce' or 'inject'.

Folding is useful whenever you have a collection of something, and want to produce a single value from it.

Note: `fold()`, and similar methods that traverse the entire iterator, may not terminate for infinite iterators, even on traits for which a result is determinable in finite time.



Template std.algorithm.iteration.fold

Implements the homonym function (also known as accumulate, compress, inject, or foldl) present in various programming languages of functional flavor. The call fold!(fun)(range, seed) first assigns seed to an internal variable result, also called the accumulator. Then, for each element x in range, result = fun(result, x) gets evaluated. Finally, result is returned. The one-argument version fold!(fun)(range) works similarly, but it uses the first element of the range as the seed (the range must be non-empty).

Template std.algorithm.iteration.reduce

Implements the homonym function (also known as accumulate, compress, inject, or foldl) present in various programming languages of functional flavor. There is also fold which does the same thing but with the opposite parameter order. The call reduce!(fun)(seed, range) first assigns seed to an internal variable result, also called the accumulator. Then, for each element x in range, result = fun(result, x) gets evaluated. Finally, result is returned. The one-argument version reduce!(fun)(range) works similarly, but it uses the first element of the range as the seed (the range must be non-empty).





std::accumulate: Exploring an Algorithmic Empire

THE ALGORITHMS (PRE-C++17)

accumulate	adjacent_difference	adjacent_find	all_of	any_of
binary_search	copy	copy_backward	copy_if	copy_n
count	count_if	equal	equal_range	fill
fill_n	find	find_end	find_first_of	find_if
find_if_not	for_each	generate	generate_n	includes
inner_product	inplace_merge	iota	is_heap	is_heap_until
is_partitioned	is_permutation	is_sorted	is_sorted_until	
lexicographical_compare	lower_bound	make_heap	max	max_element
merge	min	min_element	minmax	minmax_element
mismatch	move	move_backward	next_permutation	none_of
nth_element	partial_sort	partial_sort_copy	partial_sum	partition
partition_copy	partition_point	pop_heap	prev_permutation	push_heap
		remove	remove_copy	remove_if
replace	replace_copy	replace_copy_if	replace_if	reverse
reverse_copy	rotate	rotate_copy	search	search_n
set_difference	set_intersection	set_symmetric_difference	set_union	shuffle
sort	sort_heap	stable_partition	stable_sort	

Of 90 total, 77 are basically "plain loops"

accumulate	<input checked="" type="checkbox"/>	partial_sum
adjacent_difference	<input checked="" type="checkbox"/>	reduce
exclusive_scan		transform_exclusive_scan
inclusive_scan		transform_inclusive_scan
inner_product		transform_reduce
iota	<input checked="" type="checkbox"/>	

Pre-C++11

C++11

C++17



Jon Kalb is going to give Michael and Odin each one coin. Jon has N coins with different values. Jon wants the absolute difference between the value of the coins to be minimized so Michael and Odin don't fight with each other. Given an array of coin values, help Jon find this minimum.

1 4 2 -> 1

1 3 3 -> 0



Find the minimum difference between two values in a list.

$$1 \ 4 \ 2 \quad -> \quad 1$$

$$1 \ 3 \ 3 \quad -> \quad 0$$



```
auto min_value(vector<int>& coins) -> int {
    sort(begin(coins), end(coins));
    vector<int> diff(coins.size());
    adjacent_difference(cbegin(coins), cend(coins), begin(diff));
    return *min_element(cbegin(diff) + 1, cend(diff));
}
```



```
auto min_value(vector<int>& c) -> int {
    sort(begin(c), end(c));
    return reduce(cbegin(c) + 1, cend(c), numeric_limits<int>::max(),
        [prev = c.front()](auto a, auto b) mutable {
            auto d = b - prev;
            prev = b;
            return min(a, d);
        });
}
```

accumulate	<input checked="" type="checkbox"/>	partial_sum
adjacent_difference	<input checked="" type="checkbox"/>	reduce
exclusive_scan		transform_exclusive_scan
inclusive_scan		transform_inclusive_scan
inner_product		transform_reduce
iota	<input checked="" type="checkbox"/>	

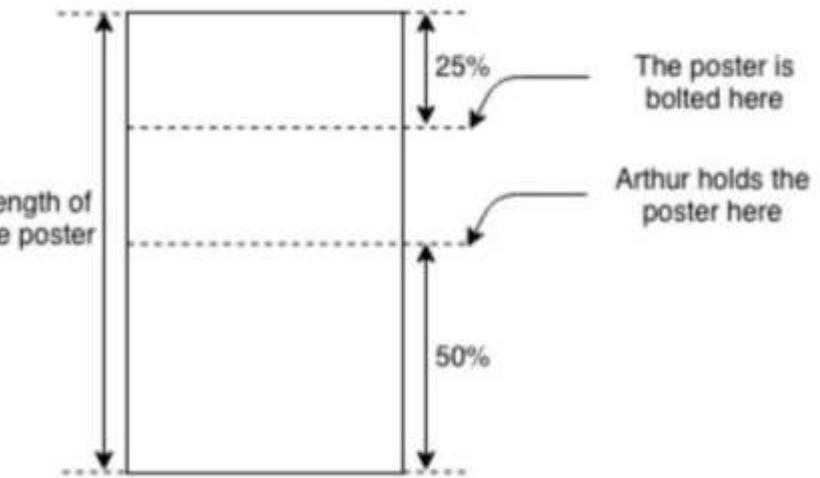
Pre-C++11

C++11

C++17



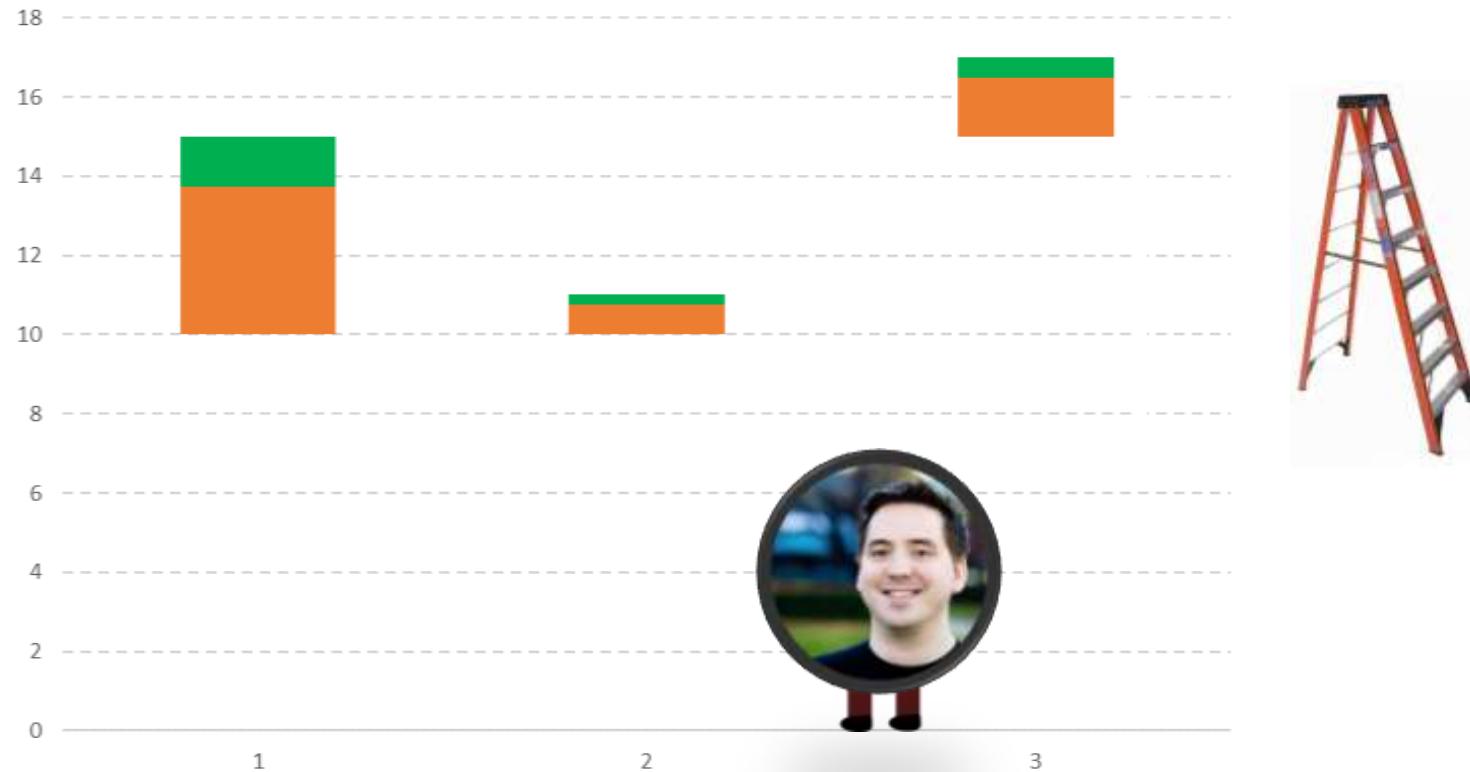
You are given the length (l) of N posters, and the wall heights (w) at which they will be hung. They are hung at the 75% mark of the poster. Given Chandler has height h , how tall a ladder does he need?



HourRank 31: Problem 1 – Hanging Posters

3 5
15 11 17
5 1 2

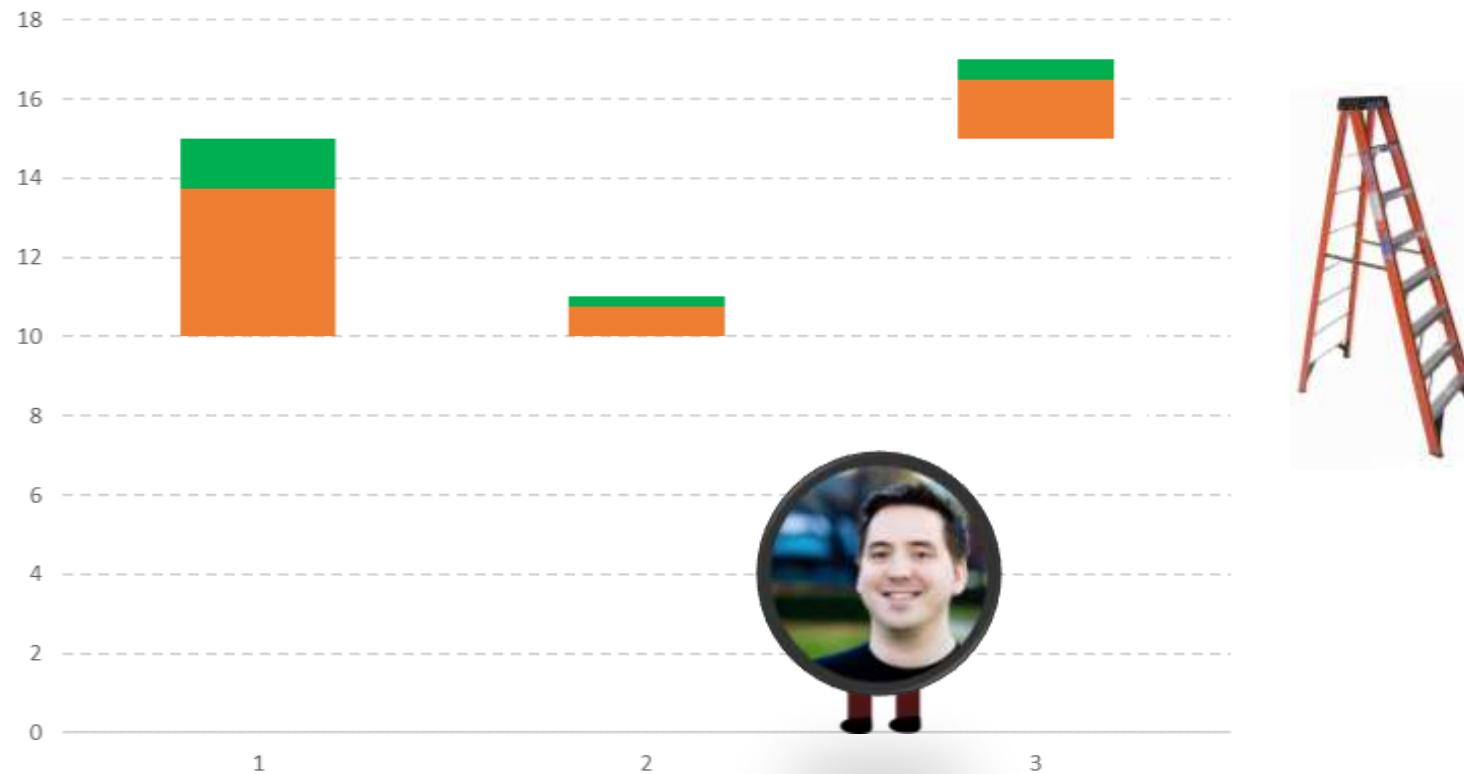
$$\left\lceil w - \frac{l}{4} \right\rceil - h = 12$$



HourRank 31: Problem 1 – Hanging Posters

3 5
15 11 17
5 1 2

$$\text{ceil}\left(w - \frac{l}{4}\right) - h = 12$$





```
int solve(int h, vector<int> w, vector<int> l) {
    int p = 0;
    for (int i = 0; i < w.size(); ++i)
        p = max(p, w[i] - l[i] / 4);
    return max(0, p - h);
}
```



```
public static int solve(int h, List<Integer> w, List<Integer> l) {  
    int p = 0;  
    for (int i = 0; i < w.size(); ++i)  
        p = Math.max(p, w.get(i) - l.get(i)/4);  
    return Math.max(0, p - h);  
}
```



Then Simon Brand tweeted...

```
def solve(h, w, l):
    p = max(a - b//4 for a, b in zip(w, l))
    return max(0, p - h)
```

```
int main() {
    auto hamming_distance = []([auto&& r1, auto&& r2) {
        return accumulate(view::zip(r1, r2), 0, ranges::plus{},
            []([auto&& x) { return x.first != x.second; });
    };

    auto ns = ranges::istream_range<std::string>(std::cin) | to_vector;
    auto found = view::cartesian_product(ns, ns)
        | view::filter([&]([auto&& p) {
            return hamming_distance(get<0>(p), get<1>(p)) == 1;
        });
    for (auto [s1,s2] : found | view::take(1)) {
        for (auto[c1, c2] : view::zip(s1, s2)) {
            if (c1 == c2) std::cout << c1;
        }
    }
}
```



Conor Hoekstra @code_report · 3 Dec 2018



Replying to [@TartanLlama](#)

Omgoodness! Ranges comes with zip??? Please tell me this is coming with C++20.



Simon Brand @TartanLlama · 3 Dec 2018



Parts of range-v3 are coming in 20, I don't believe zip is though. [@cjdb_ns](#) ?





Conor Hoekstra @code_report · 3 Dec 2018

This makes me so incredibly happy! I literally just yesterday googled, C++17 / C++20 zip to see if they had anything, because I wrote some code in both C++ and #Python and Python was so much more beautiful.

```
int solve(int h, vector<int> w, vector<int> l) {
    int p = 0;
    for (int i = 0; i < w.size(); ++i)
        p = max(p, w[i] - l[i] / 4);
    return max(0, p - h);
}

def solve(h, w, l):
    p = max(a - b//4 for a, b in zip(w, l))
    return max(0, p - h)
```

2

1

2

1



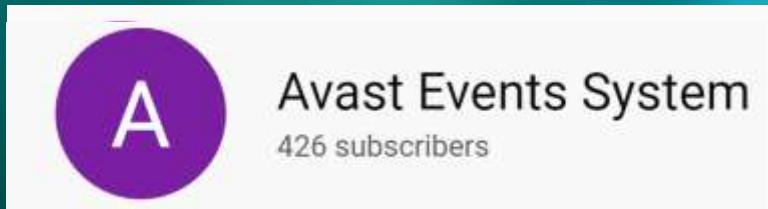
Conor Hoekstra @code_report · 16 Dec 2018



Also, I just discovered std::inner_product - a beautiful temporary solution to a lack of zip. #cpp #inner_product

```
int solve(int h, vector<int> w, vector<int> l) {
    return max(0, inner_product(begin(w), end(w), begin(l), 0,
        [] (auto a, auto b) { return max(a, b); },
        [] (auto a, auto b) { return a - b / 4; }) - h);
}
```





```
double rms(const vector<double> &v) {  
    double sum = 0.0;  
    for (vector<double>::const_iterator i = v.begin();  
         i != v.end();  
         i++) {  
        sum += *i * *i;  
    }  
    return sqrt(sum);  
}
```

```
e rms(const vector<double> &v) {  
    double sum = 0.0;  
    for (vector<double>::const_iterator i = v.begin();  
         i != v.end();  
         i++) {  
        sum += *i * *i;  
    }  
    return sqrt(sum);  
}
```



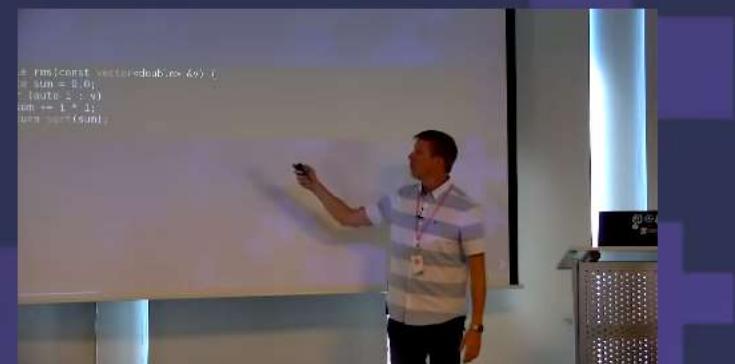
```
double rms(const vector<double> &v) {
    auto sum = 0.0;
    for (auto i = v.begin(); i != v.end(); i++) {
        sum += *i * *i;
    }
    return sqrt(sum);
}
```

```
# rms(const vector<double> &v)
# sum = 0.0;
# (auto i = v.begin(); i != v.end(); i++)
# sum += *i * *i;

sum = 0.0;
```



```
double rms(const vector<double> &v) {  
    auto sum = 0.0;  
    for (auto i : v)  
        sum += i * i;  
    return sqrt(sum);  
}
```



```
static double accum(double partialSum, double elem) {  
    return partialSum + elem * elem;  
}  
double rms(const vector<double> &v) {  
    return sqrt(accumulate(v.begin(), v.end(), 0.0, accum));  
}
```



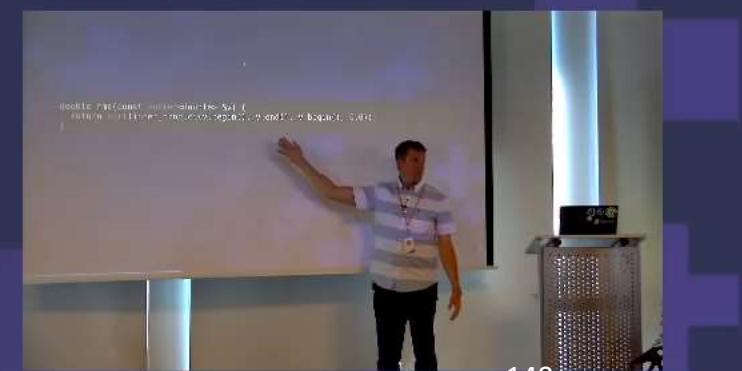
```
double rms(const vector<double> &v) {
    return sqrt(accumulate(
        v.begin(), v.end(), 0.0,
        [](double partialSum, double elem) {
            return partialSum + elem * elem;
        }
    ));
}
```



```
double rms(const vector<double> &v) {  
    return sqrt(inner_product(v.begin(), v.end(), v.begin(), 0.0));  
}
```



```
double rms(const vector<double> &v) {  
    return sqrt(inner_product(v.begin(), v.end(), v.begin(), 0.0));  
}
```



Not the best name ...



Conor Hoekstra @code_report · 16 Dec 2018

Also, I just discovered std::inner_product - a beautiful temporary solution to a lack of zip. #cpp #inner_product

```
int solve(int h, vector<int> w, vector<int> l) {
    return max(0, inner_product(begin(w), end(w), begin(l), 0,
        [] (auto a, auto b) { return max(a, b); },
        [] (auto a, auto b) { return a - b / 4; }) - h);
}
```



2



And then ...



transform_reduce





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

auto x = transform_reduce(cbegin(v), cend(v), cbegin(u), 0);
```



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

auto x = transform_reduce(cbegin(v), cend(v), cbegin(u), 0);
auto y = transform_reduce(cbegin(v), cend(v), cbegin(u), 0, plus{}, multiplies{});
```



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

auto x = transform_reduce(cbegin(v), cend(v), cbegin(u), 0);
auto y = transform_reduce(cbegin(v), cend(v), cbegin(u), 0, plus{}, multiplies{});
auto z = transform_reduce(cbegin(v), cend(v), cbegin(u), 0,
    plus{},
    multiplies{});
```



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

auto x = transform_reduce(cbegin(v), cend(v), cbegin(u), 0);
auto y = transform_reduce(cbegin(v), cend(v), cbegin(u), 0, plus{}, multiplies{});
auto z = transform_reduce(cbegin(v), cend(v), cbegin(u), 0,
    [](auto a, auto b) { return max(a, b); },
    multiplies{});
```



Why not call it `zip_reduce`? Doesn't need to `zip`!!

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

auto x = transform_reduce(cbegin(v), cend(v), cbegin(u), 0);
auto y = transform_reduce(cbegin(v), cend(v), cbegin(u), 0, plus{}, multiplies{});
auto z = transform_reduce(cbegin(v), cend(v), cbegin(u), 0,
    [](auto a, auto b) { return max(a, b); },
    [](auto a, auto b) { return a + b * b; }));
```

std::transform_reduce

Defined in header `<numeric>`

```
template<class InputIt1, class InputIt2, class T>
T transform_reduce(InputIt1 first1, InputIt1 last1, InputIt2 first2, T init); (1) (since C++17)

template <class InputIt1, class InputIt2, class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                  T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2); (2) (since C++17)

template<class InputIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(InputIt first, InputIt last,
                  T init, BinaryOp binop, UnaryOp unary_op); (3) (since C++17)

template<class ExecutionPolicy,
          class ForwardIt1, class ForwardIt2, class T>
T transform_reduce(ExecutionPolicy&& policy,
                  ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, T init); (4) (since C++17)

template<class ExecutionPolicy,
          class ForwardIt1, class ForwardIt2, class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy,
                  ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2,
                  T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2); (5) (since C++17)

template<class ExecutionPolicy,
          class ForwardIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
                  ForwardIt first, ForwardIt last,
                  T init, BinaryOp binary_op, UnaryOp unary_op); (6) (since C++17)
```



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



```
auto x = reduce(cbegin(v), cend(v), 0,
    [](auto a, auto b) { return a + b * b; });
```



```
auto y = transform_reduce(cbegin(v), cend(v), 0,
    std::plus{},
    [](auto e) { return e * e; });
```

$$\sqrt{\frac{\sum x_i^2}{n}}$$



```
double rms(const vector<double> &v) {  
    return sqrt(inner_product(v.begin(), v.end(), v.begin(), 0.0));  
}
```



Let's revisit our adjacent_difference
/ reduce question



```
auto min_value(vector<int>& c) {
    sort(begin(c), end(c));
    return reduce(cbegin(c) + 1, cend(c), numeric_limits<int>::max(),
        [prev = c.front()](auto a, auto b) mutable {
            auto d = abs(b - prev);
            prev = b;
            return min(a, d);
        });
}
```



Can anyone see how to improve this?
Hint: make use of a STL function object

```
auto min_value(vector<int>& c) {
    sort(begin(c), end(c));
    return transform_reduce(cbegin(c), --cend(c), ++cbegin(c),
                           numeric_limits<int>::max(),
                           [](auto a, auto b) { return min(a, b); },
                           [](auto a, auto b) { return abs(a - b); });
}
```



```
auto min_value(vector<int>& c) {
    sort(begin(c), end(c));
    return transform_reduce(++cbegin(c), cend(c), cbegin(c),
        numeric_limits<int>::max(),
        [] (auto a, auto b) { return min(a, b); },
        std::minus{});
}
```



```
min_value :: [Int] -> Int
min_value = minimum . mapAdjacent (flip (-)) . sort
```



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

```
min_value :: [Int] -> Int
min_value = minimum
    . mapAdjacent (flip (-))
    . sort
```



20
Ranges



1. filter
 2. transform
 3. take
 4. take_while
 5. drop
 6. drop_while
 7. join
 8. split
 9. counted
 10. common
 11. reverse
 12. istream
1. zip
 2. group_by
 3. tail





20
Ranges

```
auto solve(vector<int> v) -> int {
    return v | rv::all
              | ra::sort
              | hs::adjacent_transform(minus{})
              | hs::minimum();
}
```



20
Ranges

```
auto minimum() {
    return ranges::make_pipeable([](auto rng) {
        return ranges::min(rng);
    });
}
```



20
Ranges

```
template <typename BinOp>
auto adjacent_transform(BinOp binop) {
    return ranges::make_pipeable([&](auto&& rng) {
        return rv::zip(rng, rv::tail(rng))
            | rv::transform([&](auto p) {
                auto [a, b] = p;
                return binop(b, a);
            });
    });
}
```



20 Ranges

```
template <typename BinOp>
auto adjacent_transform(BinOp binop) {
    return ranges::make_pipeable([&](auto&& rng) {
        return rv::zip(rng, rv::tail(rng))
            | rv::transform([&](auto p) {
                auto [a, b] = p;
                return binop(b, a);
            });
    });
}
```

[[digression]]



20
Ranges

```
template <typename BinOp>
auto adjacent_transform(BinOp binop) {
    return ranges::make_pipeable([&](auto&& rng) {
        return rv::zip(rng, rv::tail(rng))
            | rv::transform([&](auto p) {
                auto [a, b] = p;
                return binop(b, a);
            });
    });
}
```

Structured bindings with polymorphic lambdas

Aaryaman Sagar (aary800@gmail.com)

August 14, 2017

1 Introduction

This paper proposes usage of structured bindings with polymorphic lambdas, adding them to another place where `auto` can be used as a declarator

```
std::for_each(map, [](auto [key, value]) {
    cout << key << " " << value << endl;
});
```

This would make for nice syntactic sugar to situations such as the above without having to decompose the tuple-like object manually, similar to how structured bindings are used in range based for loops

```
for (auto [key, value] : map) {
    cout << key << " " << value << endl;
}
```

```
int main() {
    auto hamming_distance = [] (auto&& r1, auto&& r2) {
        return accumulate(view::zip(r1, r2), 0, ranges::plus{},
            [] (auto&& x) { return x.first != x.second; });
    };

    auto ns = ranges::istream_range<std::string>(std::cin) | to_vector;
    auto found = view::cartesian_product(ns, ns)
        | view::filter([&] (auto&& p) {
            return hamming_distance(get<0>(p), get<1>(p)) == 1;
        });
    for (auto [s1,s2] : found | view::take(1)) {
        for (auto[c1, c2] : view::zip(s1, s2)) {
            if (c1 == c2) std::cout << c1;
        }
    }
}
```

[[end digression]]

[[digression]]

Introduction
ooooooooooooooPush
ooooooooooooooPipelines
ooooooooooooooGoing postal
ooooooImplementation
ooooooooooooooooooooThe EU
oo 11

Composition

Task: Count repeated values



© Ivan Ćukić, 2019

9

Ivan says three really important things:

- “**inner_product** is a really cool example that **you should never use**”
- “**transform_reduce sounds much nicer** ... but the problem is a little bit bigger than just the name”
- “we have three different operations [zip, transform, reduce] but since we have the iterators and **iterators are not easily composable** this could not have just been three different algorithms – this needs to be a single algorithm that is a composition of all the previous three”



```
auto count_adj_equals(vector<int> const& v) -> int {  
    vector d(v.size(), 0);  
    adjacent_difference(cbegin(v), cend(v), begin(d));  
    return count_if(cbegin(d) + 1, cend(d),  
                    [](auto e) { return e == 0; });  
}
```



[1,1,0,0,2,2,3]
[0,1,0,-2,0,-1]
3

```
count_if f = length . filter f
adj_diff    = mapAdjacent (-)

count_adj_equals :: [Int] -> Int
count_adj_equals = count_if (==0) . adj_diff
```



20
Ranges

```
template <typename T>
auto count_adj_equals(T const& v) -> int {
    return v | hs::adjacent_transform(equal_to{})
               | hs::count_if([](auto e) { return e; });
}
```



20 Ranges

```
template <typename Op>
auto count_if(Op op) {
    return ranges::make_pipeable([&](auto rng) {
        return ranges::count_if(rng, op);
    });
}
```

[[end digression]]



transform_reduce



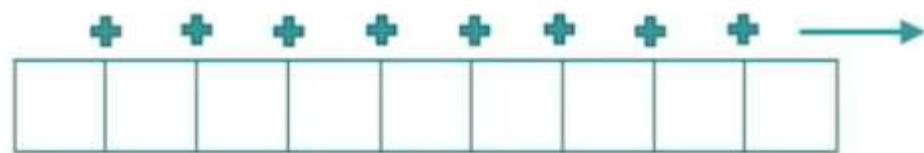
NUMERIC ALGORITHMS



count



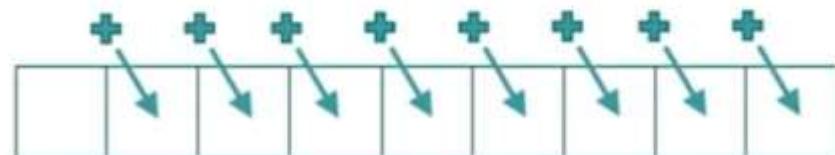
accumulate/(transform_)reduce



partial_sum

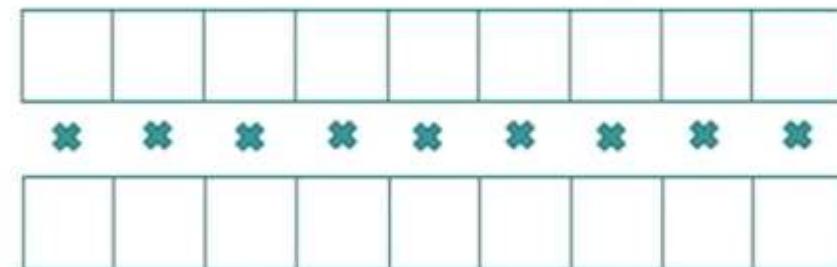
(transform_)inclusive_scan

(transform_)exclusive_scan

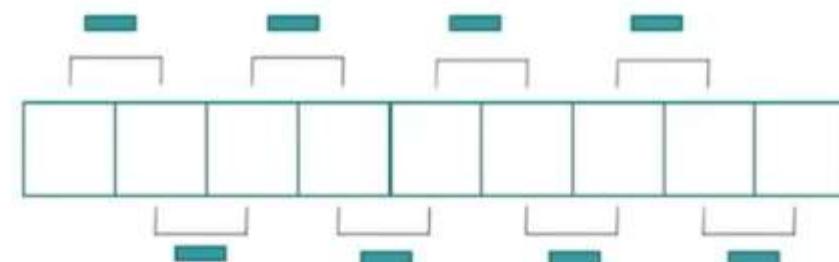


@JoBoccara

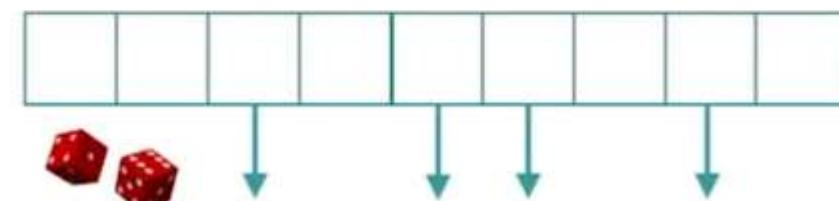
inner_product



adjacent_difference



sample



accumulate	<input checked="" type="checkbox"/>	partial_sum
adjacent_difference	<input checked="" type="checkbox"/>	reduce
exclusive_scan		transform_exclusive_scan
inclusive_scan		transform_inclusive_scan
inner_product	<input checked="" type="checkbox"/>	transform_reduce
iota	<input checked="" type="checkbox"/>	

Pre-C++11

C++11

C++17

partial_sum applications

- Subarray sums
- Use with `binary_search`
- Lambda+



```
vector v = { 1, 1, 1, 1, 1 };
vector u(5, 0);
partial_sum(cbegin(v), cend(v), begin(u));

// 1 2 3 4 5

vector v = { 1, 2, 3, 4, 5 };
vector u(5, 0);
partial_sum(cbegin(v), cend(v), begin(u));

// 1 3 6 10 15
```



```
// map<int, int> queries;

vector v = { 1, 2, ... , 100 };
vector u(v.size(), 0);
partial_sum(cbegin(v), cend(v), begin(u));

for (auto [a, b] : queries) {
    auto sum = u[b] - u[a];
}
```



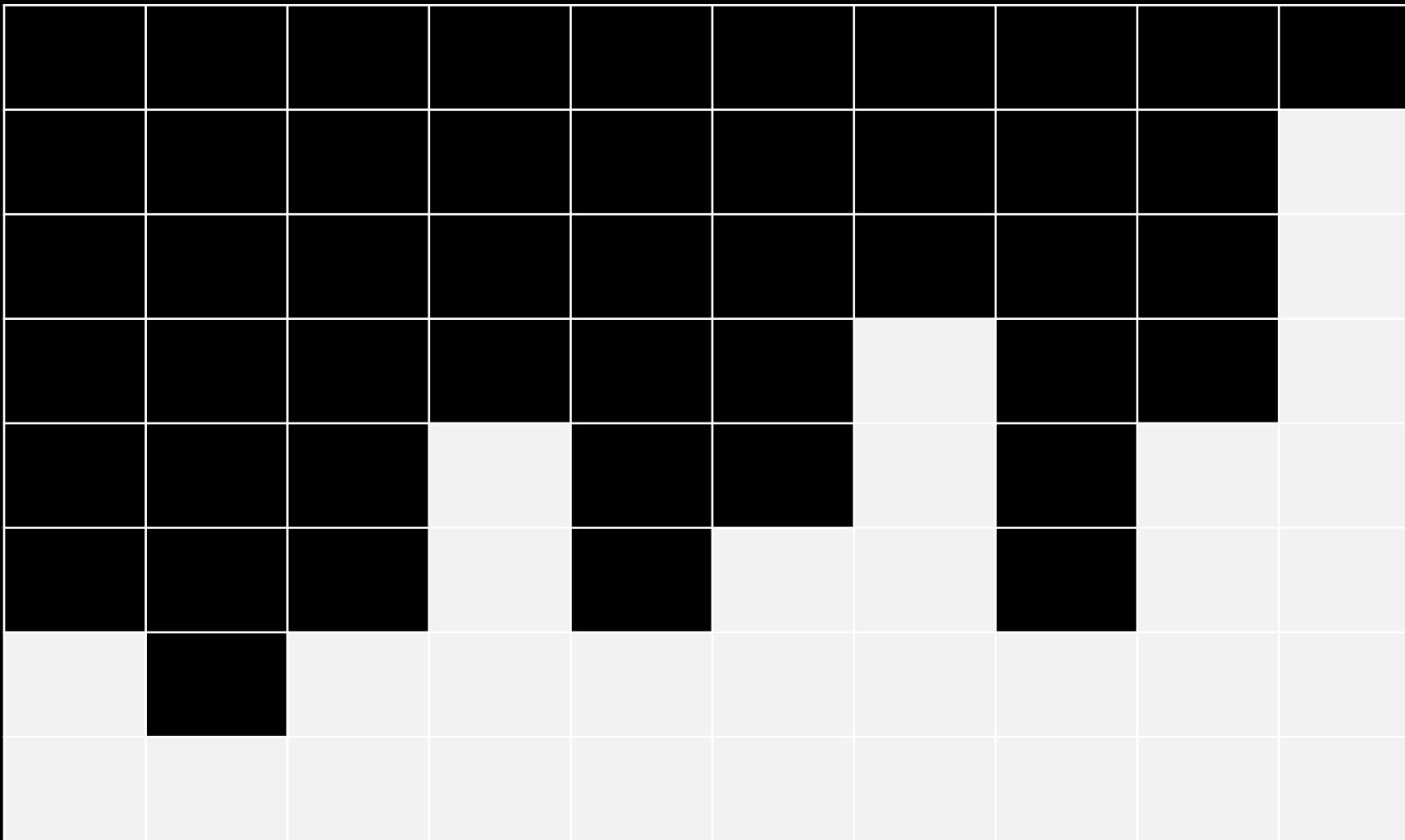
42. Trapping Rain Water

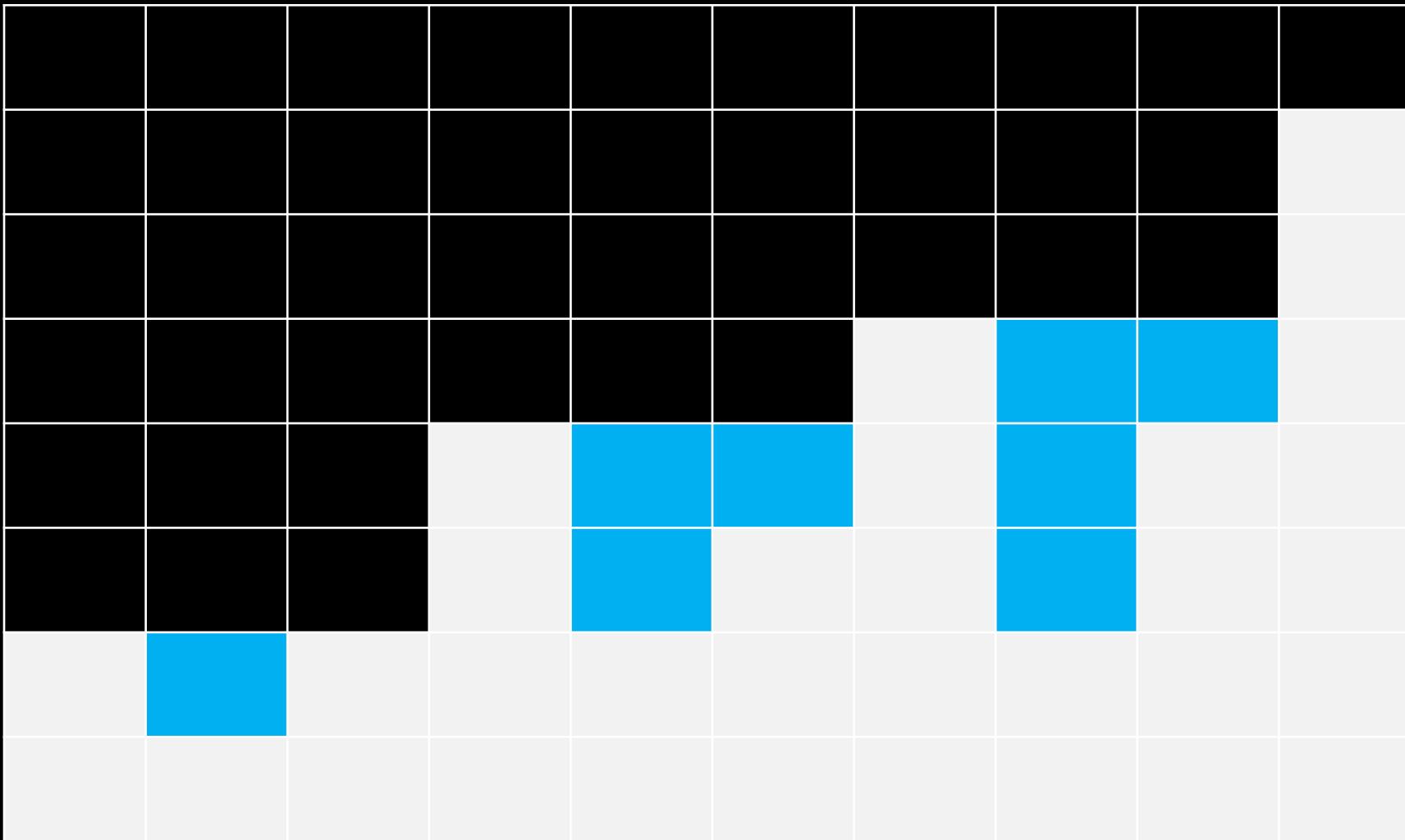
Hard 3387 61 Favorite Share

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!







```
auto solve() {
    vector v = { 2, 1, 2, 4, 2, 3, 5, 2, 4, 7 };
    auto m = v.front(); // max so far
    auto ans = 0;
    for (auto e : v) {
        m = max(m, e);
        ans += m - e;
    }
    return ans;
}
```



```
auto solve() {
    vector v = { 2, 1, 2, 4, 2, 3, 5, 2, 4, 7 };
    vector u(v.size(), 0);
    partial_sum(cbegin(v), cend(v), begin(u),
                [] (auto a, auto b) { return max(a, b); });
    return transform_reduce(cbegin(v), cend(v), cbegin(u), 0,
                           std::plus{},
                           [] (auto a, auto b) { return abs(a - b); });
}
```



Not the best name ...

```
auto solve() {
    vector v = { 2, 1, 2, 4, 2, 3, 5, 2, 4, 7 };
    vector u(v.size(), 0);
    partial_sum(cbegin(v), cend(v), begin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus{},
                           std::minus{});
}
```



```
auto solve() {
    vector v = { 2, 1, 2, 4, 2, 3, 5, 2, 4, 7 };
    vector u(v.size(), 0);
    inclusive_scan(cbegin(v), cend(v), begin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus{},
                           std::minus{});
}
```



42. Trapping Rain Water

Hard 3387 61 Favorite Share

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!



```
int trap(vector<int>& v) {
    vector u(v.size(), 0);
    inclusive_scan(begin(v), end(v), begin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus<>(),
                           std::minus<>());
}
```



```
int trap(vector<int>& v) {
    vector u(v.size(), 0);
    auto it = max_element(begin(v), end(v));
    inclusive_scan(begin(v), end(v), begin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus<>(),
                           std::minus<>()));
}
```



```
int trap(vector<int>& v) {
    vector u(v.size(), 0);
    auto it = max_element(begin(v), end(v));
    inclusive_scan(begin(v), next(it), begin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus<>(),
                           std::minus<>());
}
```



```
int trap(vector<int>& v) {
    vector u(v.size(), 0);
    auto it = max_element(begin(v), end(v));
    inclusive_scan(begin(v), next(it), begin(u), ufo::max{});
    inclusive_scan(rbegin(v), rev(it), rbegin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus<>(),
                           std::minus<>()));
}
```



```
template<class T>
using rev = reverse_iterator<T>;\n\nint trap(vector<int>& v) {
    vector u(v.size(), 0);
    auto it = max_element(begin(v), end(v));
    inclusive_scan(begin(v), next(it), begin(u), ufo::max{});
    inclusive_scan(rbegin(v), rev(it), rbegin(u), ufo::max{});
    return transform_reduce(cbegin(u), cend(u), cbegin(v), 0,
                           std::plus<>(),
                           std::minus<>()));
}
```

accumulate	<input checked="" type="checkbox"/>	partial_sum	<input checked="" type="checkbox"/>
adjacent_difference	<input checked="" type="checkbox"/>	reduce	<input checked="" type="checkbox"/>
exclusive_scan	<input checked="" type="checkbox"/>	transform_exclusive_scan	
inclusive_scan	<input checked="" type="checkbox"/>	transform_inclusive_scan	
inner_product	<input checked="" type="checkbox"/>	transform_reduce	<input checked="" type="checkbox"/>
iota	<input checked="" type="checkbox"/>		

Pre-C++11

C++11

C++17

The Algorithm Intuition Table

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
iota	N / A	N / A	Transform



```
template <class I, class T, class R, class M>
auto adjacent_reduce(I f, I l, T init, R r, M m) {
    return inner_product(f, --l, ++f, init, r, m);
}
```

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
iota	N / A	N / A	Transform

Solving a Problem with The Algorithm Intuition Table



Round 77: Problem A – Football

Petya loves football very much. One day, as he was watching a football match, he was writing the players' current positions on a piece of paper. To simplify the situation he depicted it as a string consisting of zeroes and ones. A zero corresponds to players of one team; a one corresponds to players of another team. If there are at least 7 players of some team standing one after another, then the situation is considered dangerous. For example, the situation 0010011011111101 is dangerous and 11110111011101 is not. You are given the current situation. Determine whether it is dangerous or not.



Round 77: Problem A – Football

Given a string of 1s and 0s, return true if the longest (contiguous) substring of *equal* elements is greater than or equal to 7.

Examples

input
output
001001
NO

input
output
1000000001
YES



```
auto dangerous_team(string const& s) -> bool {
    auto max_players = 1, curr_players = 1;
    for (int i = 1; i < s.size(); ++i) {
        curr_players = s[i] == s[i - 1] ? curr_players + 1 : 1;
        max_players = max(max_players, curr_players);
    }
    return max_players >= 7;
}
```

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
iota	N / A	N / A	Transform



```
auto dangerous_team(const string& s) -> bool {
    return adjacent_reduce(cbegin(s), cend(s),
        make_pair(1, 1),
        [] (pair<int, int> acc, bool equal) {
            auto [max_player, curr_player] = acc;
            curr_player = equal ? curr_player + 1 : 1;
            return make_pair(max(max_player, curr_player), curr_player);
        },
        std::equal_to{}).first >= 7;
}
```



```
auto dangerous_team(string const& s) -> bool {
    return adjacent_reduce(cbegin(s), cend(s),
        make_pair(1, 1),
        [] (pair<int, int> acc, bool equal) {
            auto [mp, cp] = acc;
            cp = equal ? cp + 1 : 1;
            return make_pair(max(mp, cp), cp);
        },
        std::equal_to{}).first >= 7;
}
```



"001001"
["00","1","00","1"]
[2,1,2,1]
2
False

```
dangerous_teams :: String -> Bool
dangerous_teams = (>=7)
    . maximum
    . map length
    . group
```



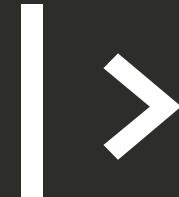
20 Ranges

```
auto dangerous_teams(string const& s) -> bool {
    return s | rv::group_by(equal_to{})
              | rv::transform(ranges::distance)
              | hs::maximum()
              | hs::gte(7);
}
```

[[digression]]



```
auto dangerous_teams(std::string const& s) -> bool {  
    return s  
        | views::group_by(std::equal_to{})  
        | views::transform(ranges::distance)  
        | ranges::any_of([](std::size_t s){  
            return s >= 7;  
        });  
}
```



<https://brevzin.github.io/c++/2019/08/22/ufcs-custom-extension/>

[[end digression]]

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
😊	2	No	Reduce*
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
😊	1*	No	Transform
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
transform	1*	No	Transform
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
transform	1*	No	Transform
😊	1	No	Reduce*
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
transform	1*	No	Transform
find*	1	No	Reduce*
iota	N / A	N / A	Transform



```
auto generate_n_fibonacci(int n) {
    vector fib(n, 1);
    transform(cbegin(fib),
              cend (fib) - 2,
              cbegin(fib) + 1,
              begin (fib) + 2,
              std::plus{});
    return fib;
}
```



```
auto generate_n_fibonacci(int n) {
    vector fib(n, 1);
    adjacent_difference(cbegin(fib),
                        cend (fib) - 1,
                        begin (fib) + 1,
                        std::plus{}));
    return fib;
}
```

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_difference	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
transform	1*	No	Transform
find*	1	No	Reduce*
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform
accumulate / reduce	1	Yes Init = Specified	Reduce
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform
exclusive_scan	1	Yes Init = Specified	Transform
adjacent_transform	2	No	Transform
adjacent_reduce	2	Yes	Reduce
adjacent_find*	2	No	Reduce*
transform	1*	No	Transform
find*	1	No	Reduce*
iota	N / A	N / A	Transform

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}
exclusive_scan	1	Yes Init = Specified	Transform	plus{}
adjacent_transform adjacent_difference	2	No	Transform	minus{}
adjacent_reduce	2	Yes	Reduce	-
adjacent_find*	2	No	Reduce*	equal_to{}
transform	1*	No	Transform	-
find*	1	No	Reduce*	-
iota	N / A	N / A	Transform	

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}
exclusive_scan	1	Yes Init = Specified	Transform	plus{}
adjacent_transform adjacent_difference	2	No	Transform	minus{}
adjacent_reduce	2	Yes	Reduce	-
adjacent_find*	2	No	Reduce*	equal_to{}
transform	1*	No	Transform	-
find*	1	No	Reduce*	-
iota	N / A	N / A	Transform	

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}
exclusive_scan	1	Yes Init = Specified	Transform	plus{}
adjacent_transform adjacent_difference	2	No	Transform	minus{}
	2	Yes	Transform	
adjacent_reduce	2	Yes	Reduce	-
adjacent_find*	2	No	Reduce*	equal_to{}
transform	1*	No	Transform	-
find*	1	No	Reduce*	-
iota	N / A	N / A	Transform	

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}
exclusive_scan	1	Yes Init = Specified	Transform	plus{}
adjacent_transform adjacent_difference	2	No	Transform	minus{}
adjacent_inclusive_scan	2	Yes	Transform	
adjacent_reduce	2	Yes	Reduce	-
adjacent_find*	2	No	Reduce*	equal_to{}
transform	1*	No	Transform	-
find*	1	No	Reduce*	-
iota	N / A	N / A	Transform	



```
template<class I, class O, class BinOp1, class BinOp2>
auto adjacent_inclusive_scan(I f, I l, O o, BinOp1 binop1,
                             BinOp2 binop2) {
    if (f != l) {
        auto prev = *f; // previous element
        auto acc = *f; // accumulator
        *o = prev; // first element special treatment
        while (++f != l) {
            acc = binop1(acc, binop2(*f, prev));
            *++o = acc;
            prev = std::move(*f);
        }
        ++o;
    }
    return o;
}
```



```
adjacent_inclusive_scan(cbegin(v),
                        cend(v),
                        begin(u),
                        std::plus(),
                        [](auto a, auto b) {
                            return a == b ? b : 0;
                       });
```

accumulate	<input checked="" type="checkbox"/>	partial_sum	<input checked="" type="checkbox"/>
adjacent_difference	<input checked="" type="checkbox"/>	reduce	<input checked="" type="checkbox"/>
exclusive_scan	<input checked="" type="checkbox"/>	transform_exclusive_scan	
inclusive_scan	<input checked="" type="checkbox"/>	transform_inclusive_scan	
inner_product	<input checked="" type="checkbox"/>	transform_reduce	<input checked="" type="checkbox"/>
iota	<input checked="" type="checkbox"/>		

Pre-C++11

C++11

C++17

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}
exclusive_scan	1	Yes Init = Specified	Transform	plus{}
adjacent_transform adjacent_difference	2	No	Transform	minus{}
transform_inclusive_scan	2	Yes	Transform	-
adjacent_reduce	2	Yes	Reduce	-
adjacent_find*	2	No	Reduce*	equal_to{}
transform	1*	No	Transform	-
find*	1	No	Reduce*	-
iota	N / A	N / A	Transform	

std::transform_inclusive_scan

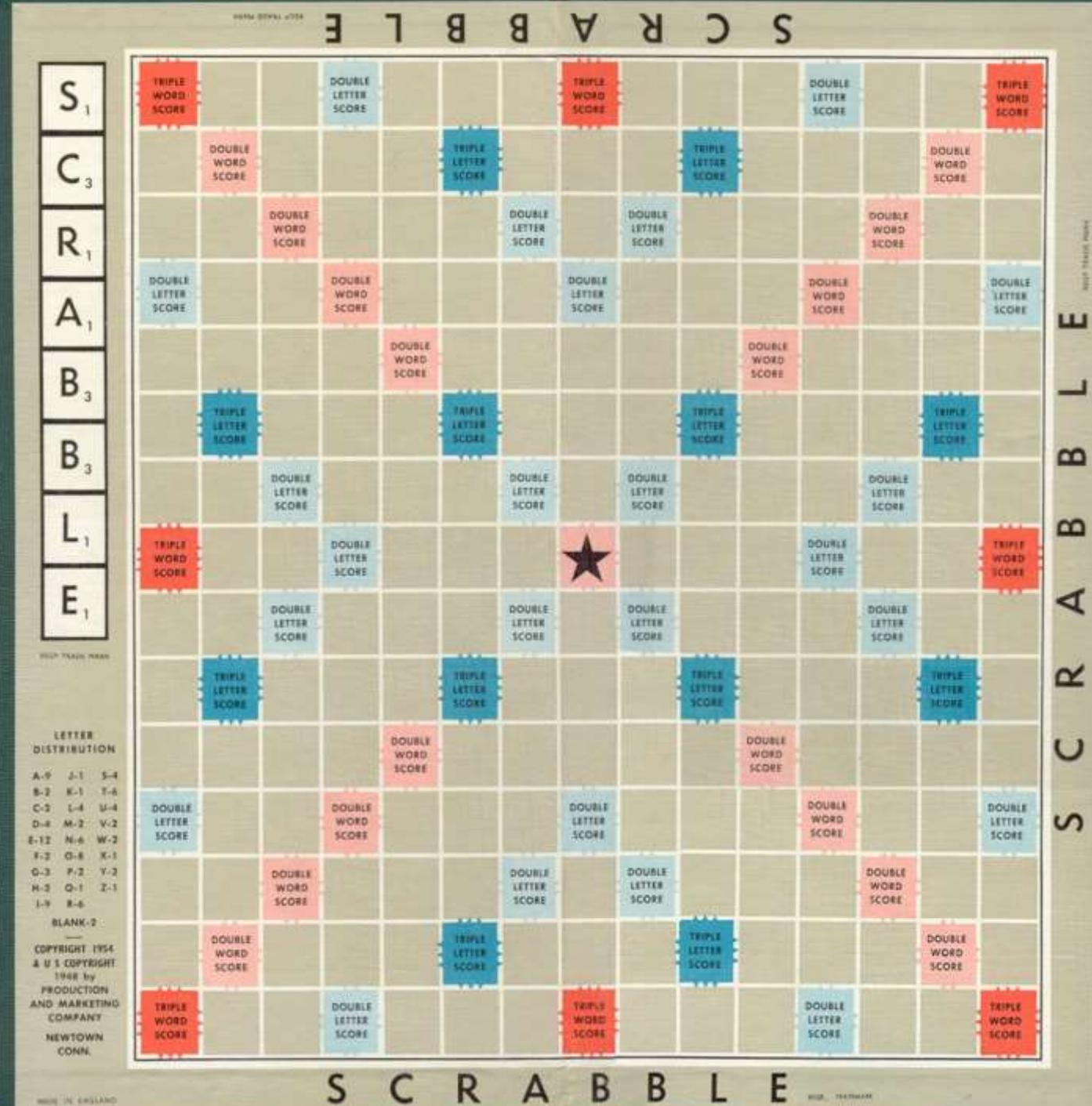
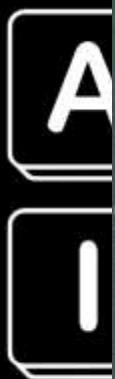
Defined in header `<numeric>`

```
template< class InputIt, class OutputIt,
          class BinaryOperation, class UnaryOperation >
OutputIt transform_inclusive_scan( InputIt first, InputIt last, OutputIt d_first,
                                   BinaryOperation binary_op, UnaryOperation unary_op );(1) (since C++17)
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class BinaryOperation, class UnaryOperation >
ForwardIt2 transform_inclusive_scan( ExecutionPolicy&& policy,
                                     ForwardIt1 first, ForwardIt1 last, ForwardIt2 d_first,
                                     BinaryOperation binary_op, UnaryOperation unary_op );(2) (since C++17)
template< class InputIt, class OutputIt,
          class BinaryOperation, class UnaryOperation, class T >
OutputIt transform_inclusive_scan( InputIt first, InputIt last, OutputIt d_first,
                                   BinaryOperation binary_op, UnaryOperation unary_op,
                                   T init );(3) (since C++17)
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class BinaryOperation, class UnaryOperation, class T >
ForwardIt2 transform_inclusive_scan( ExecutionPolicy&& policy,
                                     ForwardIt1 first, ForwardIt1 last, ForwardIt2 d_first,
                                     BinaryOperation binary_op, UnaryOperation unary_op,
                                     T init );(4) (since C++17)
```

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}
exclusive_scan	1	Yes Init = Specified	Transform	plus{}
adjacent_transform adjacent_difference	2	No	Transform	minus{}
adjacent_inclusive_scan	2	Yes	Transform	
adjacent_reduce	2	Yes	Reduce	-
adjacent_find*	2	No	Reduce*	equal_to{}
transform	1*	No	Transform	-
find*	1	No	Reduce*	-
iota	N / A	N / A	Transform	

Algorithm	Indexes Viewed	Accumulator	Reduce / Transform	Default Op	Filter aka _if
accumulate / reduce	1	Yes Init = Specified	Reduce	plus{}	
inner_product / transform_reduce	1*	Yes Init = Specified	Reduce	plus{}, mult{}	
partial_sum / inclusive_scan	1	Yes Init = First elem	Transform	plus{}	
exclusive_scan	1	Yes Init = Specified	Transform	plus{}	
adjacent_transform adjacent_difference	2	No	Transform	minus{}	
transform_inclusive_scan	2	Yes	Transform	-	
adjacent_reduce	2	Yes	Reduce	-	
adjacent_find*	2	No	Reduce*	equal_to{}	
transform	1*	No	Transform	-	
find*	1	No	Reduce*	-	
iota	N / A	N / A	Transform		

One last



S C R A B B L E



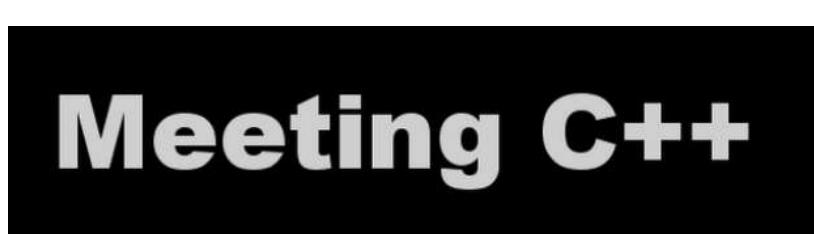
One last problem...

```
vector w = {1, 1, 2, 1, 1, 1, 1, 1, 4, 3};  
vector t = {1, 1, 3, 1, 1, 1, 3, 1, 1};
```

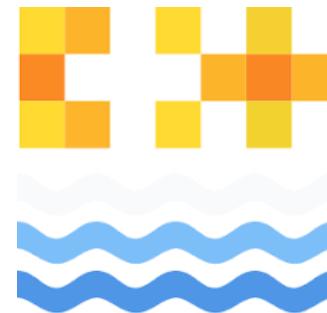


```
auto calc_word_score(vector<int> const& word,  
                     vector<int> const& tile) -> int {  
    return transform_reduce(cbegin(word),  
                           cend(word),  
                           cbegin(tile),  
                           0);  
}
```

Before we conclude ...

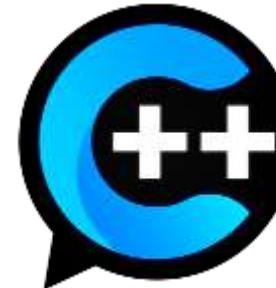


code::dive





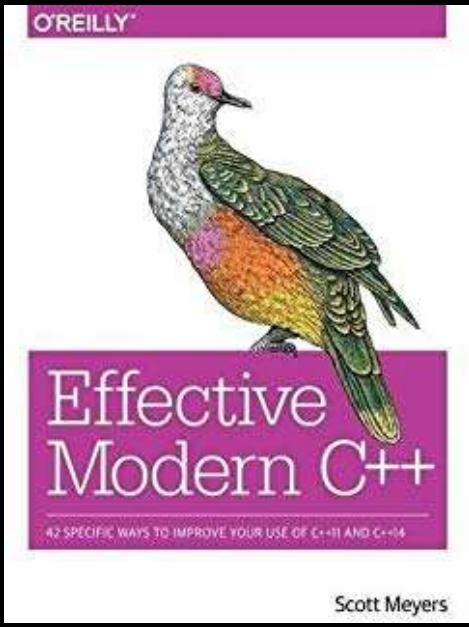
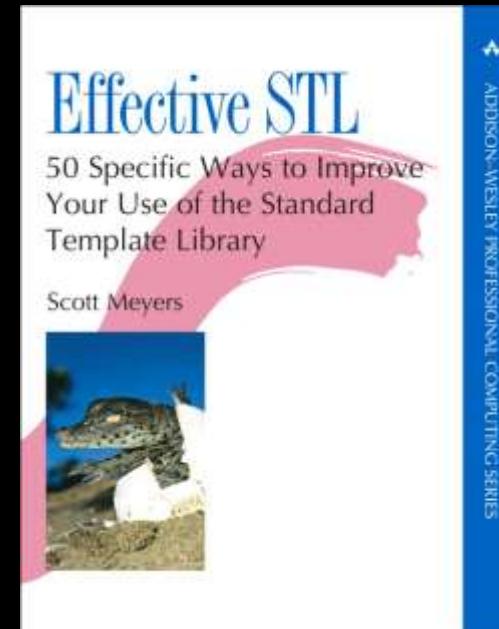
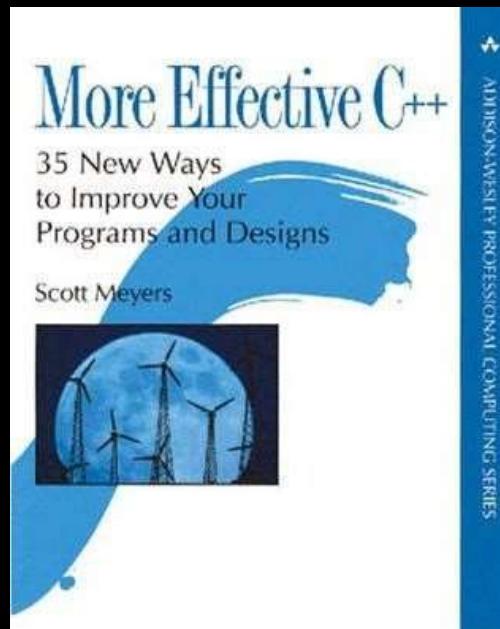
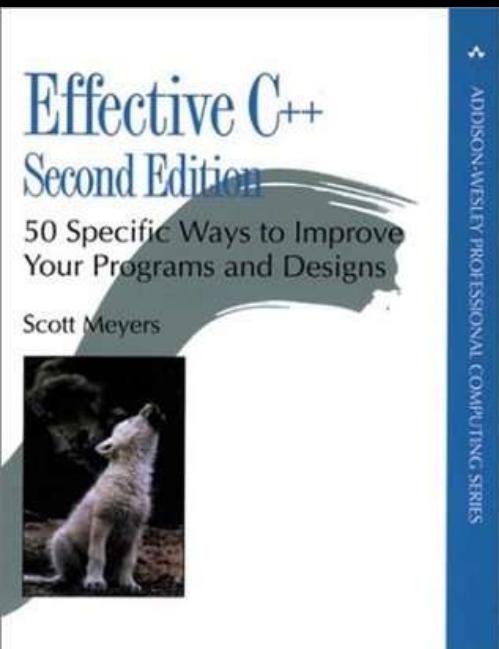
auto CppCast = pod_cast<C++>("http://cppcast.com");



Cpp.chat



Read these





code_report



codereport

The screenshot shows a video player interface. On the left, there are two video thumbnails. The top one is titled "STL Algorithms 13: std::unique & std::unique_copy" and the bottom one is titled "STL Algorithms 12: min, max, minmax, min_element, max_element & std::minmax, std::min_element, std::max_element, std::minmax_element". Both thumbnails show a duration of "duration: 00:00:00 / 00:00:00". To the right of the thumbnails is a large image of a smiling woman with short hair, wearing a dark t-shirt. She is positioned in front of a background filled with logos of various coding platforms: HackerRank, topcoder, hackerearth, CODECHEF, LeetCode, and CODEFORCES. At the bottom of the screen, there are two sets of social media links. The left set is for "/codereport" and includes a YouTube icon, a GitHub icon, and a Twitter icon. The right set is for "/code_report" and also includes a Twitter icon.

HackerRank topcoder hackerearth CODECHEF LeetCode CODEFORCES

STL Algorithms 13: std::unique & std::unique_copy
std::unique
std::unique_copy
duration: 00:00:00 / 00:00:00 13:57

STL Algorithms 12: min, max, minmax, min_element, max_element & std::minmax, std::min_element, std::max_element, std::minmax_element
duration: 00:00:00 / 00:00:00 6:10

/codereport

/code_report

Conclusion

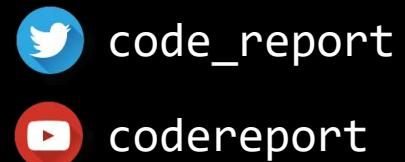
1. Algorithms are awesome! And fun!
2. Especially `transform_reduce`
 1. `minmax_element`, `min_element`, `max_element`, `sort`, `iota`, `count_if`,
`inner_product`, `adjacent_difference`, `partial_sum`, `accumulate`,
`reduce`, `inclusive_scan`, `exclusive_scan`
3. Know the default operations that algorithms come with
4. Leverage algorithms with function objects / lambdas
5. C++20 ranges and composable algorithms are coming



Thank you!

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

Conor Hoekstra





Questions?

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

Conor Hoekstra



code_report



codereport