FLATIRON INSTITUTE
Center for Computational Quantum Physics

The Standard Template Library

Nils Wentzell
Oct 24, 2019

Introduction

- Library of generic Containers & Algorithms
- Generic Iterator Interface allows for interoperability

Containers

```
std::vector<T>
std::array<T, N>
std::map<T, V>
std::list<T>
…
```

Algorithms

```
std::find
std::all_of
std::any_of
std::transform
…
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Introduction

- Library of generic Containers & Algorithms
- Generic Iterator Interface allows for interoperability

Containers

```
std::vector<T>
std::array<T, N>
std::map<T, V>
std::list<T>
…
```

Implementation Effort $\mathcal{O}\left(N_C N_A\right)$
No Custom Containers

Algorithms

```
std::find
std::all_of
std::any_of
std::transform
…
```

Introduction

- Library of generic Containers & Algorithms
- Generic Iterator Interface allows for interoperability

Containers

```
std::vector<T>
std::array<T, N>
std::map<T, V>
std::list<T>
…
```

Iterators

```
InputIterator
OutputIterator
ForwardIterator
RandomAccessIterator
…
```

Algorithms

```
std::find
std::all_of
std::any_of
std::transform
…
```

Introduction

- Library of generic Containers & Algorithms
- Generic Iterator Interface allows for interoperability

Containers

```
std::vector<T>
std::array<T, N>
std::map<T, V>
std::list<T>
…
```

Iterators

```
InputIterator
OutputIterator
ForwardIterator
RandomAccessIterator
…
```

Algorithms

```
std::find
std::all_of
std::any_of
std::transform
…
```

→ Manageable Implementation effort $\mathcal{O}\left(N_{\mathrm{C}} + N_{\mathrm{A}}\right)$ vs $\mathcal{O}\left(N_{\mathrm{C}} N_{\mathrm{A}}\right)$

Introduction

- STL Containers are generic

```
vector<double> vec_d;
vector<int>    vec_i;
```

- Reduction for arbitrary container types

```
vector<double> vec;
list<double>   lst;

double vec_sum = reduce(cbegin(vec), cend(vec), 0.0);
double lst_sum = reduce(cbegin(lst), cend(lst), 0.0);
```

$$\sum_i v_i$$

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Introduction

- STL Containers are generic

```
vector<double> vec_d;
vector<int>    vec_i;
```

- Reduction for arbitrary container types

```
vector<double> vec;
list<double>   lst;

double vec_sum = reduce(cbegin(vec), cend(vec), 0.0);
double lst_sum = reduce(cbegin(lst), cend(lst), 0.0);
```

$$\sum_i v_i$$

```
double vec_sum = reduce(vec, 0.0);
double lst_sum = reduce(lst, 0.0);
```

C++20

FLATIRON INSTITUTE
Center for Computational Quantum Physics

STL Containers

An Overview

en.cppreference.com/w/cpp/container

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

`array` (C++11)	static contiguous array (class template)
`vector`	dynamic contiguous array (class template)
`deque`	double-ended queue (class template)
`forward_list` (C++11)	singly-linked list (class template)
`list`	doubly-linked list (class template)

FLATIRON INSTITUTE
Center for Computational Quantum Physics

An Overview

en.cppreference.com/w/cpp/container

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Cover most use-cases!

FLATIRON INSTITUTE
Center for Computational Quantum Physics

An Overview

en.cppreference.com/w/cpp/container

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)

```
auto a = array<int, 3>{};          // Construction
auto b = array<int, 3>{1, 2, 3};   // Initializer List
b.size();                          // How many elements?
int i = b[0];                      // Access
b[0]  = 4;                         // Assignment
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

An Overview

en.cppreference.com/w/cpp/container

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)

```
auto v = vector<int>{};         // Construction
auto w = vector<int>{1, 2, 3}; // Initializer List
w.size();                       // How many elements?
int i = w[0];                   // Access
w[0]  = 4;                      // Assignment
w.push_back(5);                 // Add new elements
w.resize(10);                   // Change to a particular size
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

An Overview

en.cppreference.com/w/cpp/container

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

`set`	collection of unique keys, sorted by keys (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

FLATIRON INSTITUTE
Center for Computational Quantum Physics

An Overview

en.cppreference.com/w/cpp/container

Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ amortized, $O(n)$ worst-case complexity).

unordered_set (C++11)	collection of unique keys, hashed by keys (class template)
unordered_map (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
unordered_multiset (C++11)	collection of keys, hashed by keys (class template)
unordered_multimap (C++11)	collection of key-value pairs, hashed by keys (class template)

FLATIRON INSTITUTE
Center for Computational Quantum Physics

The Iterator Interface

Defined in header <iterator>
Defined in namespace std

begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)

The Iterator Interface

Defined in header <iterator>
Defined in namespace std

begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)

The Iterator Interface

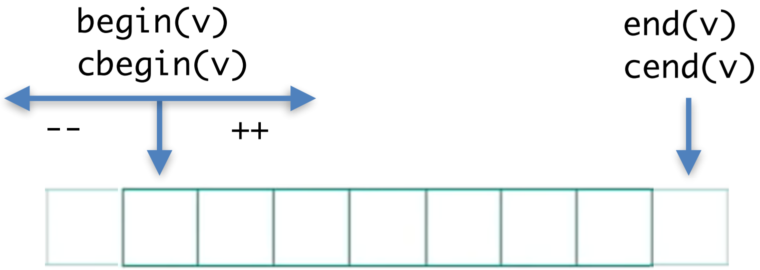
Defined in header <iterator>
Defined in namespace std

begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)

The Iterator Interface

Defined in header <iterator>
Defined in namespace std

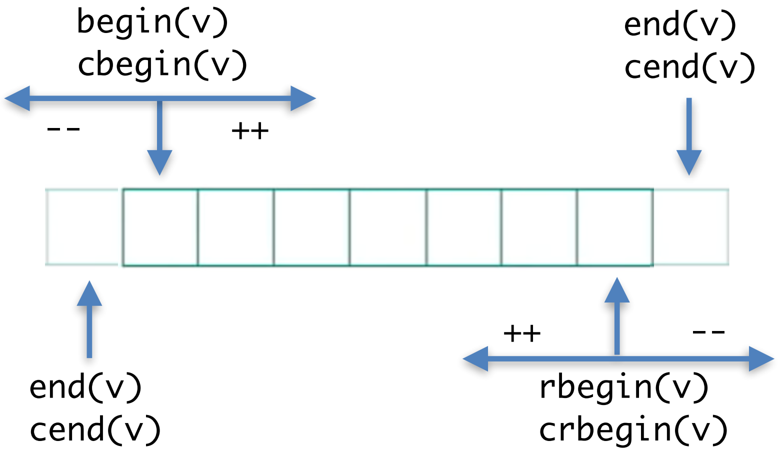
begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)



The Iterator Interface

Defined in header <iterator>
Defined in namespace std

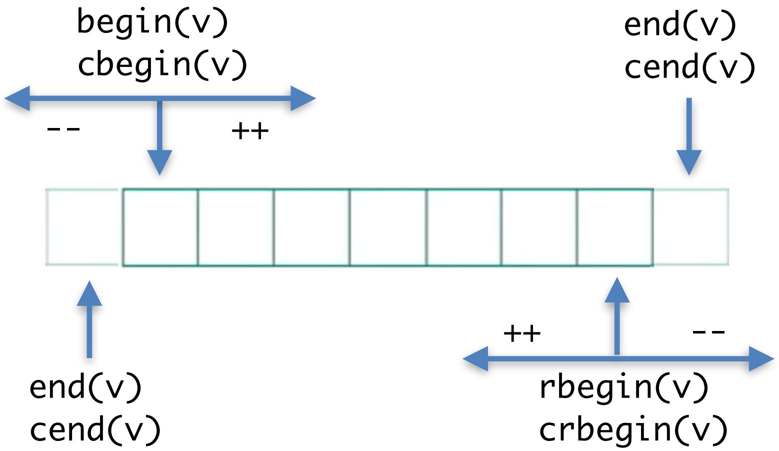
begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)



The Iterator Interface

Defined in header <iterator>
Defined in namespace std

begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)

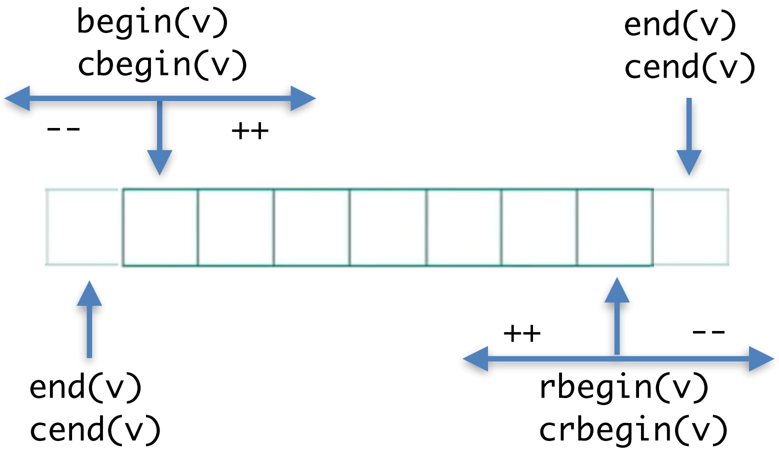



```
auto v = std::vector<int>{1, 2, 3};
int sum = 0;
for(auto it = cbegin(v); it != cend(v); ++it){
    sum += *it; // Dereference
}
```

The Iterator Interface

Defined in header <iterator>
Defined in namespace std

begin (C++11) **cbegin** (C++14)	returns an iterator to the beginning of a container or array (function template)
end (C++11) **cend** (C++14)	returns an iterator to the end of a container or array (function template)
rbegin **crbegin** (C++14)	returns a reverse iterator to a container or array (function template)
rend **crend** (C++14)	returns a reverse end iterator for a container or array (function template)
size (C++17) **ssize** (C++20)	returns the size of a container or array (function template)
empty (C++17)	checks whether the container is empty (function template)
data (C++17)	obtains the pointer to the underlying array (function template)




```cpp
auto v = std::vector<int>{1, 2, 3};
int sum = 0;
for(int i : v){
    sum += i;
}
```

```cpp
auto v = std::vector<int>{1, 2, 3};
int sum = 0;
for(auto it = cbegin(v); it != cend(v); ++it){
    sum += *it; // Dereference
}
```

STL Algorithms I

An Overview

en.cppreference.com/w/cpp/algorithm

cppreference.com   Create account   Search

Page   Discussion   View   Edit   History

C++ / Algorithm library

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past* the last element to inspect or modify.

Constrained algorithms

C++20 provides constrained versions of most algorithms in the namespace `std::ranges`. In these algorithms, a range can be specified as either an iterator-sentinel pair or as a single range argument, and projections and pointer-to-member callables are supported. Additionally, the return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm.

(since C++20)

```
std::vector<int> v = {7, 1, 4, 0, -1};
std::ranges::sort(v); // constrained algorithm
```

The header `<iterator>` provides a set of concepts and related utilities designed to ease constraining common algorithm operations.

Execution policies

Most algorithms have overloads that accept execution policies. The standard library algorithms support several execution policies, and the library provides corresponding execution policy types and objects. Users may select an execution policy statically by invoking a parallel algorithm with an execution policy object of the corresponding type.

Standard library implementations (but not the users) may define additional execution policies as an extension. The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type is implementation-defined.

(since C++17)

Defined in header <execution>
Defined in namespace `std::execution`

`sequenced_policy` (C++17) `parallel_policy` (C++17) `parallel_unsequenced_policy` (C++17) `unsequenced_policy` (C++20)	execution policy types (class)
`seq` (C++17) `par` (C++17) `par_unseq` (C++17) `unseq` (C++20)	global execution policy objects (constant)

Defined in namespace `std`

`is_execution_policy` (C++17)	test whether a class represents an execution policy (class template)

Non-modifying sequence operations

Defined in header <algorithm>

`all_of` (C++11) `any_of` (C++11) `none_of` (C++11)	checks if a predicate is `true` for all, any or none of the elements in a range (function template)
`for_each`	applies a function to a range of elements (function template)
`for_each_n` (C++17)	applies a function object to the first n elements of a sequence (function template)
`count` `count_if`	returns the number of elements satisfying specific criteria (function template)
`mismatch`	finds the first position where two ranges differ (function template)

`find`
`find_if`
`find_if_not` (C++11)
	finds the first element satisfying specific criteria
	(function template)

`find_end`	finds the last sequence of elements in a certain range (function template)
`find_first_of`	searches for any one of a set of elements (function template)
`adjacent_find`	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
`search`	searches for a range of elements (function template)
`search_n`	searches a range for a number of consecutive copies of an element (function template)

Modifying sequence operations

Defined in header <algorithm>

`copy` `copy_if` (C++11)	copies a range of elements to a new location (function template)
`copy_n` (C++11)	copies a number of elements to a new location (function template)
`copy_backward`	copies a range of elements in backwards order (function template)
`move` (C++11)	moves a range of elements to a new location (function template)
`move_backward` (C++11)	moves a range of elements to a new location in backwards order (function template)
`fill`	copy-assigns the given value to every element in a range (function template)
`fill_n`	copy-assigns the given value to N elements in a range (function template)
`transform`	applies a function to a range of elements, storing results in a destination range (function template)
`generate`	assigns the results of successive function calls to every element in a range (function template)
`generate_n`	assigns the results of successive function calls to N elements in a range (function template)
`remove` `remove_if`	removes elements satisfying specific criteria (function template)
`remove_copy` `remove_copy_if`	copies a range of elements omitting those that satisfy specific criteria (function template)
`replace` `replace_if`	replaces all values satisfying specific criteria with another value (function template)
`replace_copy` `replace_copy_if`	copies a range, replacing elements satisfying specific criteria with another value (function template)

~~An Overview~~ A Selection en.cppreference.com/w/cpp/algorithm

cppreference.com Create account Search

Page Discussion View Edit History

C++ / **Algorithm library**

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past* the last element to inspect or modify.

Constrained algorithms

C++20 provides constrained versions of most algorithms in the namespace `std::ranges`. In these algorithms, a range can be specified as either an iterator-sentinel pair or as a single range argument, and projections and pointer-to-member callables are supported. Additionally, the return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm.

(since C++20)

```
std::vector<int> v = {7, 1, 4, 0, -1};
std::ranges::sort(v); // constrained algorithm
```

The header `<iterator>` provides a set of concepts and related utilities designed to ease constraining common algorithm operations.

Execution policies

Most algorithms have overloads that accept execution policies. The standard library algorithms support several execution policies, and the library provides corresponding execution policy types and objects. Users may select an execution policy statically by invoking a parallel algorithm with an execution policy object of the corresponding type.

Standard library implementations (but not the users) may define additional execution policies as an extension. The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type is implementation-defined.

(since C++17)

Defined in header <execution>
Defined in namespace std::execution

`sequenced_policy` (C++17) `parallel_policy` (C++17) `parallel_unsequenced_policy` (C++17) `unsequenced_policy` (C++20)	execution policy types (class)
`seq` (C++17) `par` (C++17) `par_unseq` (C++17) `unseq` (C++20)	global execution policy objects (constant)

Defined in namespace std

`is_execution_policy` (C++17)	test whether a class represents an execution policy (class template)

Non-modifying sequence operations

Defined in header <algorithm>

`all_of` (C++11) `any_of` (C++11) `none_of` (C++11)	checks if a predicate is `true` for all, any or none of the elements in a range (function template)
`for_each`	applies a function to a range of elements (function template)
`for_each_n` (C++17)	applies a function object to the first n elements of a sequence (function template)
`count` `count_if`	returns the number of elements satisfying specific criteria (function template)
`mismatch`	finds the first position where two ranges differ (function template)

`find`
`find_if`
`find_if_not` (C++11)
	finds the first element satisfying specific criteria (function template)

`find_end`	finds the last sequence of elements in a certain range (function template)
`find_first_of`	searches for any one of a set of elements (function template)
`adjacent_find`	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
`search`	searches for a range of elements (function template)
`search_n`	searches a range for a number of consecutive copies of an element (function template)

Modifying sequence operations

Defined in header <algorithm>

`copy` `copy_if` (C++11)	copies a range of elements to a new location (function template)
`copy_n` (C++11)	copies a number of elements to a new location (function template)
`copy_backward`	copies a range of elements in backwards order (function template)
`move` (C++11)	moves a range of elements to a new location (function template)
`move_backward` (C++11)	moves a range of elements to a new location in backwards order (function template)
`fill`	copy-assigns the given value to every element in a range (function template)
`fill_n`	copy-assigns the given value to N elements in a range (function template)
`transform`	applies a function to a range of elements, storing results in a destination range (function template)
`generate`	assigns the results of successive function calls to every element in a range (function template)
`generate_n`	assigns the results of successive function calls to N elements in a range (function template)
`remove` `remove_if`	removes elements satisfying specific criteria (function template)
`remove_copy` `remove_copy_if`	copies a range of elements omitting those that satisfy specific criteria (function template)
`replace` `replace_if`	replaces all values satisfying specific criteria with another value (function template)
`replace_copy` `replace_copy_if`	copies a range, replacing elements satisfying specific criteria with another value (function template)

Documentation

en.cppreference.com/w/cpp/algorithm/sort

std::sort

Defined in header `<algorithm>`

`template< class RandomIt >` `void sort( RandomIt first, RandomIt last );`	(1)	(until C++20)
`template< class RandomIt >` `constexpr void sort( RandomIt first, RandomIt last );`		(since C++20)
`template< class ExecutionPolicy, class RandomIt >` `void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt last );`	(2)	(since C++17)
`template< class RandomIt, class Compare >` `void sort( RandomIt first, RandomIt last, Compare comp );`	(3)	(until C++20)
`template< class RandomIt, class Compare >` `constexpr void sort( RandomIt first, RandomIt last, Compare comp );`		(since C++20)
`template< class ExecutionPolicy, class RandomIt, class Compare >` `void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt last, Compare comp );`	(4)	(since C++17)

Sorts the elements in the range `[first, last)` in ascending order. The order of equal elements is not guaranteed to be preserved.

1) Elements are compared using `operator<`.

3) Elements are compared using the given binary comparison function `comp`.

2,4) Same as (1,3), but executed according to `policy`. These overloads do not participate in overload resolution unless `std::is_execution_policy_v<std::decay_t<ExecutionPolicy>>` is true

Parameters

first, last - the range of elements to sort

policy - the execution policy to use. See execution policy for details.

comp - comparison function object (i.e. an object that satisfies the requirements of *Compare*) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

Documentation

en.cppreference.com/w/cpp/algorithm/sort

Example

Run Share Exit GCC 9.2 (C++2a)

Powered by Coliru online compiler

```cpp
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    // sort using the default operator<
    std::sort(s.begin(), s.end());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

Why use STL Algorithms?

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Why use STL Algorithms?

- Allow you to write expressive code
- Widely know → easy to read
- Tested and Debugged
- Optimal Performance

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Why use STL Algorithms?

- Allow you to write expressive code
- Widely know → easy to read
- Tested and Debugged
- Optimal Performance
- Parallel execution C++17 & C++20
- Compile-time execution

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Why use STL Algorithms?

- Allow you to write expressive code
- Widely know → easy to read
- Tested and Debugged
- Optimal Performance
- Parallel execution C++17 & C++20
- Compile-time execution

Know them & Use them!

FLATIRON INSTITUTE
Center for Computational Quantum Physics

What about performance?

godbolt.org/z/FVRSTG

C++ source #1

A Save/Load + Add new... Vim CppInsights C++

```cpp
#include <vector>


int f(std::vector<int> const & v){
    int res = 1;
    for(auto i : v){
        res *= i;
    }
    return res;
}
```

C++ source #2

A Save/Load + Add new... Vim CppInsights C++

```cpp
#include <vector>
#include <numeric>
#include <functional>

int f(std::vector<int> const & v){
    return std::accumulate(begin(v), end(v), 1, std::multiplies<int>{});
}
```

What about performance?

godbolt.org/z/FVRSTG

C++ source #1

```cpp
#include <vector>


int f(std::vector<int> const & v){
    int res = 1;
    for(auto i : v){
        res *= i;
    }
    return res;
}
```

C++ source #2

```cpp
#include <vector>
#include <numeric>
#include <functional>

int f(std::vector<int> const & v){
    return std::accumulate(begin(v), end(v), 1, std::multiplies<int>{});
}
```

x86-64 gcc (trunk) (Editor #1, Compiler #1) C++ — x86-64 gcc (trunk), -O3

```asm
f(std::vector<int, std::allocator<int> > const&):
        mov     rax, QWORD PTR [rdi]
        mov     rdx, QWORD PTR [rdi+8]
        mov     r8d, 1
        cmp     rax, rdx
        je      .L1
.L3:
        imul    r8d, DWORD PTR [rax]
        add     rax, 4
        cmp     rax, rdx
        jne     .L3
.L1:
        mov     eax, r8d
        ret
```

x86-64 gcc (trunk) (Editor #2, Compiler #2) C++ — x86-64 gcc (trunk), -O3

```asm
f(std::vector<int, std::allocator<int> > const&):
        mov     rdx, QWORD PTR [rdi+8]
        mov     rax, QWORD PTR [rdi]
        mov     r8d, 1
        cmp     rax, rdx
        je      .L1
.L3:
        imul    r8d, DWORD PTR [rax]
        add     rax, 4
        cmp     rax, rdx
        jne     .L3
.L1:
        mov     eax, r8d
        ret
```

What about performance?

godbolt.org/z/FVRSTG

C++ source #1

```cpp
#include <vector>


int f(std::vector<int> const & v){
    int res = 1;
    for(auto i : v){
        res *= i;
    }
    return res;
}
```

C++ source #2

```cpp
#include <vector>
#include <numeric>
#include <functional>

int f(std::vector<int> const & v){
    return std::accumulate(begin(v), end(v), 1, std::multiplies<int>{});
}
```

Identical Assembly!

x86-64 gcc (trunk) (Editor #1, Compiler #1) C++ — x86-64 gcc (trunk), -O3

```asm
f(std::vector<int, std::allocator<int> > const&):
        mov     rax, QWORD PTR [rdi]
        mov     rdx, QWORD PTR [rdi+8]
        mov     r8d, 1
        cmp     rax, rdx
        je      .L1
.L3:
        imul    r8d, DWORD PTR [rax]
        add     rax, 4
        cmp     rax, rdx
        jne     .L3
.L1:
        mov     eax, r8d
        ret
```

x86-64 gcc (trunk), -O3

```asm
f(std::vector<int, std::allocator<int> > const&):
        mov     rdx, QWORD PTR [rdi+8]
        mov     rax, QWORD PTR [rdi]
        mov     r8d, 1
        cmp     rax, rdx
        je      .L1
.L3:
        imul    r8d, DWORD PTR [rax]
        add     rax, 4
        cmp     rax, rdx
        jne     .L3
.L1:
        mov     eax, r8d
        ret
```

Algorithm Categories

Algorithm Categories

- Non-modifying Sequence Operations `find(cbegin(v), cend(v), 0)`

Flatiron Institute
Center for Computational Quantum Physics

Algorithm Categories

- Non-modifying Sequence Operations `find(cbegin(v), cend(v), 0)`
- Modifying Sequence Operations `copy(cbegin(v), cend(v), begin(w))`

FLATIRON INSTITUTE
Center for Computational Quantum Physics

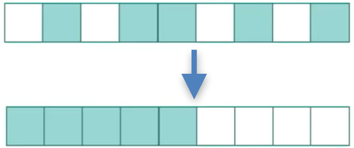
Algorithm Categories

- Non-modifying Sequence Operations `find(cbegin(v), cend(v), 0)`
- Modifying Sequence Operations `copy(cbegin(v), cend(v), begin(w))`
- Permutation Operations `sort(begin(v), end(v))`

FLATIRON INSTITUTE
Center for Computational Quantum Physics

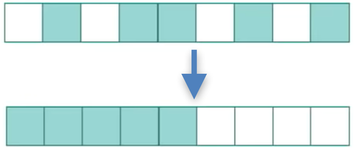
Algorithm Categories

- Non-modifying Sequence Operations `find(cbegin(v), cend(v), 0)`
- Modifying Sequence Operations `copy(cbegin(v), cend(v), begin(w))`
- Permutation Operations `sort(begin(v), end(v))`
- Partitioning Operations `partition(cbegin(v), cend(v), cnd)`



Algorithm Categories

- Non-modifying Sequence Operations `find(cbegin(v), cend(v), 0)`
- Modifying Sequence Operations `copy(cbegin(v), cend(v), begin(w))`
- Permutation Operations `sort(begin(v), end(v))`
- Partitioning Operations `partition(cbegin(v), cend(v), cnd)`

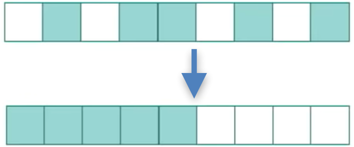


- Numeric Operations `reduce(cbegin(v), cend(v), 0)`

FLATIRON INSTITUTE
Center for Computational Quantum Physics

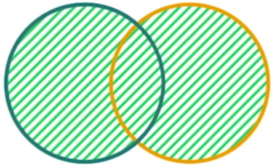
Algorithm Categories

- Non-modifying Sequence Operations `find(cbegin(v), cend(v), 0)`
- Modifying Sequence Operations `copy(cbegin(v), cend(v), begin(w))`
- Permutation Operations `sort(begin(v), end(v))`
- Partitioning Operations `partition(cbegin(v), cend(v), cnd)`



- Numeric Operations `reduce(cbegin(v), cend(v), 0)`
- Operations on Sets

```
set_union(cbegin(s), cend(s),
          cbegin(t), cend(t),
          begin(r))
```



A simple Example

- Find the maximum absolute difference between any two elements

```
int solve() {
  vector v = {2, 1, 3, 5, 4};
  // ...





}
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

A simple Example

```
int solve() {
  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;
}
```

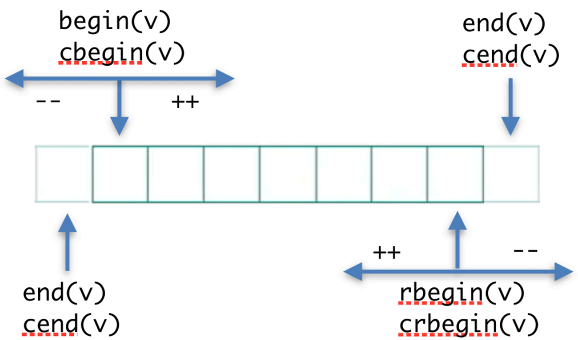
A simple Example

```
int solve() {
  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(v[i] - v[j]));
    }
  }
  return ans;
}
```

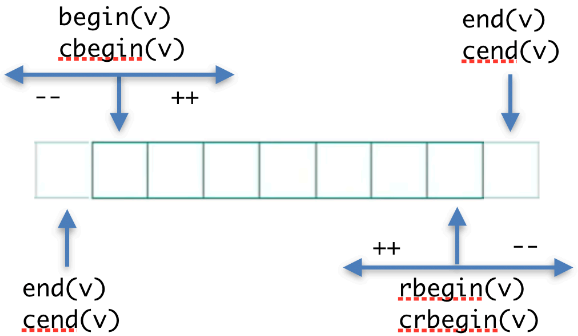
$\boxed{\mathcal{O}(N^2)\ \text{Complexity}}$

A simple Example

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



A simple Example




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

A simple Example

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

$\mathcal{O}(N \log N)$ Complexity

FLATIRON INSTITUTE
Center for Computational Quantum Physics

A simple Example

```
int solve() {
  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;
}
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

A simple Example

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

$\mathcal{O}(N)$ Complexity

FLATIRON INSTITUTE
Center for Computational Quantum Physics

A simple Example

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

FLATIRON INSTITUTE
Center for Computational Quantum Physics

A simple Example

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

A simple Example

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

C++20

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Lambda Functions in C++

Syntax

[captures] (params) -> ret { statements; }

Center for Computational Quantum Physics

Syntax

[captures] (params) -> ret { statements; }

A simple Example

```
auto l = [](int i) -> int { return i+1;};
int u = l(2);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Syntax

[captures] (params) -> ret { statements; }

A simple Example

```
auto l = [](int i) { return i+1;};
int u = l(2);
```

FLATIRON INSTITUTE
Center for Computational
Quantum Physics

Syntax

[captures] (params) -> ret   { statements; }

A simple Example

C++

```cpp
auto l = [](int i) { return i+1;};
int u = l(2);
```

Python

```python
l = lambda i : i + 1
u = l(2)
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Syntax

[captures] (params) -> ret { statements; }

Integrate a generic function on the interval [a, b]

```
template <typename F>
double integrate(F f, double a, double b) {
 const int N = 1000;
 double r = 0, step = (b - a) / N;
 for (int i = 0; i < N; ++i) r += f(a + step * i);
 return r;
}
```

$$\int_0^1 dx \cos(2x)$$

```
double r1 = integrate( [](double x){ return cos(2*x); }, 0, 1);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Syntax

`[ captures ]` `( params ) -> ret   { statements; }`

[captures]

What outside variables are available, by value or by reference.

(params)

How to invoke it.

-> ret

Return type. Will be `auto` deduced if omitted.

{ statements; }

The body of the lambda function.

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Different ways to capture

[captures] (params) -> ret { statements; }

[captures]

- [=] Capture all by copy
- [&] Capture all by reference
- [a] Capture a by copy
- [&a] Capture a by reference
- [&, a] Capture all by reference and a by copy
- [=, &a] Capture all by copy and a by reference

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Different ways to capture

[captures] (params) -> ret { statements; }

- Earlier in scope

```
MyClass w{};
```

- Capture by reference

```
auto lam = [&w] (int i) { return f(w, i); };
lam(42);
```

- Capture by copy, Parameter by `const &`

```
int i = 10;
auto g = [i] (MyClass const & w) { return f(w, i); };
g(w);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Polymorphic lambdas

[captures] (params) -> ret { statements; }

- Use `auto` to define generic lambdas

```
auto four_times = [] (auto s) { return 4.0 * s; };
auto n = four_times(4);

auto I = complex<double>{0.0, 1.0};
auto cplx = four_times(I);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Polymorphic lambdas

[captures] (params) -> ret { statements; }

- Use `auto` to define generic lambdas

```
auto four_times = [] (auto s) { return 4.0 * s; };
auto n = four_times(4);

using namespace std::complex_literals;
auto cplx = four_times(1i);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Immediately invoked lambdas

[captures] (params) -> ret { statements; }

- Initialization of variables

```
int N = 10;
…
const int x = [N]() {
  int res = 1;
  for (int i = 2; i <= N; i += 2) { // this could be a
    res += i;                      // long and complicated
  }                                // calculation
  return res;
}();
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Immediately invoked lambdas

[captures] (params) -> ret { statements; }

- Initialization of variables

```
int N = 10;
…
const int x = [N]() {
  int res = 1;
  for (int i = 2; i <= N; i += 2) {
    res += i;
  }
  return res;
}();
```

- No need to define free function
- Retain Locality

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Generalized Captures

[captures] (params) -> ret { statements; }

- Initialization of variables

```
const int x = [N = 10]() {
  int res = 1;
  for (int i = 2; i <= N; i += 2) {
    res += i;
  }
  return res;
}();
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

STL Algorithms & Lambda Functions

Introduction — Revisiting `sort`

- The simple use-case

```
auto v = vector<int>{ 2, 1, 3 };
sort(begin(v), end(v));
```

Introduction — Revisiting `sort`

- The simple use-case

```
auto v = vector<int>{ 2, 1, 3 };
sort(begin(v), end(v));
```

- A custom sort

```
struct op_t {
  double tau;
  // whatever ...
};
auto v = vector<op_t>{};
//...
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Introduction — Revisiting sort

- The simple use-case

```
auto v = vector<int>{ 2, 1, 3 };
sort(begin(v), end(v));
```

- A custom sort

```
struct op_t {
  double tau;
  // whatever ...
};
auto v = vector<op_t>{};
//...

// Sort v according to tau
sort(begin(v), end(v),
  [](double x, double y) { return (x.tau < y.tau); });
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

all_of and any_of

- All values greater 10?

```
vector<int> v;
all_of(cbegin(v), cend(v), [](int i){ return i > 10; });
```

Flatiron Institute
Center for Computational Quantum Physics

all_of and any_of

- All values greater 10?

```
vector<int> v;
all_of(cbegin(v), cend(v), [](int i){ return i > 10; });
```

- Any values negative?

```
vector<double> x;
any_of(cbegin(x), cend(x), [](double d){ return d < 0.; });
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

iota and generate

- A range of integers

```
auto v = vector<int>(10);
iota(begin(v), end(v), 0);
// 0 1 2 3 4 5 6 7 8 9
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

iota and generate

github.com/TRIQS/triqs/blob/2.2.x/test/itertools/itertools.cpp

github.com/TRIQS/triqs/blob/2.2.x/itertools/itertools.hpp

- A range of integers

```
auto v = vector<int>(10);
iota(begin(v), end(v), 0);
// 0 1 2 3 4 5 6 7 8 9
```

```
itertools::range(0,10);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

iota and generate

- A range of integers

```
auto v = vector<int>(10);
iota(begin(v), end(v), 0);
// 0 1 2 3 4 5 6 7 8 9
```

- A list of squares

```
auto v = vector<int>(10);
generate(begin(v), end(v),
  [i = 0] () mutable { ++i; return i*i; });
// 1 4 9 16 25 36 49 64 81 100
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

reduce (accumulate)

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.

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

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.

$$(v_0, v_1) \to g(v_0, v_1)$$

FLATIRON INSTITUTE
Center for Computational Quantum Physics

`reduce` (accumulate)

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.

$$(v_0, v_1) \to g(v_0, v_1)$$

$$(v_0, v_1, v_2) \to g(g(v_0, v_1), v_2)$$

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

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.

$$(v_0, v_1) \to g(v_0, v_1)$$

$$(v_0, v_1, v_2) \to g(g(v_0, v_1), v_2)$$

$$(v_0, v_1, v_2, \ldots, v_n) \to g(\ldots g(g(v_0, v_1), v_2), \ldots, v_n)$$

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

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.

$$(v_0, v_1) \rightarrow v_0 \square v_1$$

$$(v_0, v_1, v_2) \rightarrow v_0 \square v_1 \square v_2$$

$$(v_0, v_1, v_2, \ldots, v_n) \rightarrow v_0 \square v_1 \square v_2 \square \ldots \square v_n$$

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

$$(v_0, \ldots, v_n) \rightarrow v_0 \square \ldots \square v_n$$

- The default use-case $\sum_i v_i$

```
auto v = vector<int>{ 2, 1, 3 };
reduce(cbegin(v), cend(v), 0);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

$$(v_0, \ldots, v_n) \to v_0 \square \ldots \square v_n$$

- The default use-case $\sum_i v_i$

```
auto v = vector<int>{ 2, 1, 3 };
reduce(cbegin(v), cend(v), 0);
```

- Custom reduction $\prod_i v_i$

```
auto v = vector<int>{ 2, 1, 3 };
reduce(cbegin(v), cend(v), 1, [](int i, int j){ return i * j; });
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

reduce (accumulate)

$$(v_0, \ldots, v_n) \to v_0 \square \ldots \square v_n$$

- The default use-case $\sum_i v_i$

```
auto v = vector<int>{ 2, 1, 3 };
reduce(cbegin(v), cend(v), 0);
```

- Custom reduction $\prod_i v_i$

```
auto v = vector<int>{ 2, 1, 3 };
reduce(cbegin(v), cend(v), 1, multiplies<>{});
```

en.cppreference.com/w/cpp/header/functional

FLATIRON INSTITUTE
Center for Computational Quantum Physics

transform

$v_i \to f(v_i)$ $(v_i, w_i) \to g(v_i, w_i)$

transform

$v_i \to f(v_i)$ $\quad\quad (v_i, w_i) \to g(v_i, w_i)$

- Squaring elements $\quad v_i \to v_i^2$

```
auto v = vector<int>{ 2, 1, 3 };
transform(cbegin(v), cend(v), begin(v),
  [](int i){ return i * i; });
```

transform

$v_i \to f(v_i)$ $(v_i, w_i) \to g(v_i, w_i)$

- Squaring elements $v_i \to v_i^2$

```
auto v = vector<int>{ 2, 1, 3 };
transform(cbegin(v), cend(v), begin(v),
  [](int i){ return i * i; });
```

- Logical or $(v_i, w_i) \to v_i || w_i$

```
vector<bool> a, b;
transform(cbegin(a), cend(a), cbegin(b), begin(a),
  [](bool l, bool r){ return l || r; });
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

transform

$v_i \to f(v_i)$ $\qquad (v_i, w_i) \to g(v_i, w_i)$

- Squaring elements $\quad v_i \to v_i^2$

```
auto v = vector<int>{ 2, 1, 3 };
transform(cbegin(v), cend(v), begin(v),
  [](int i){ return i * i; });
```

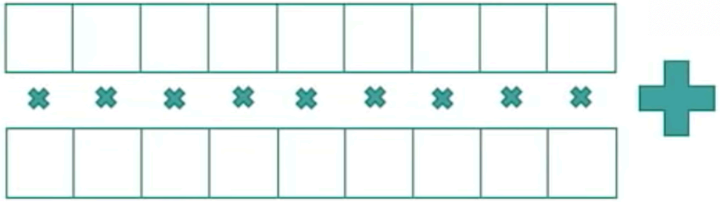
- Logical or $\quad (v_i, w_i) \to v_i || w_i$

```
vector<bool> a, b;
transform(cbegin(a), cend(a), cbegin(b), begin(a),
  logical_or<>{});
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

transform_reduce (inner_product)

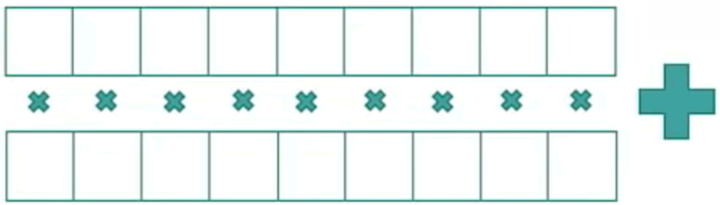
$$(v_0, \ldots, v_n) \to f(v_0) \square \ldots \square f(v_n)$$



transform_reduce (inner_product)

$(v_0, \ldots, v_n) \rightarrow f(v_0) \square \ldots \square f(v_n)$

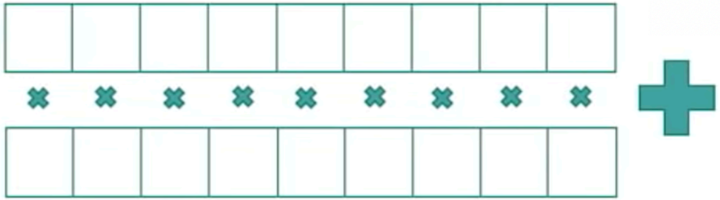
$(v_0, \ldots, v_n), (w_0, \ldots, w_n) \rightarrow g(v_0, w_0) \square \ldots \square g(v_n, w_n)$



transform_reduce (inner_product)

$$(v_0, \ldots, v_n) \to f(v_0) \square \ldots \square f(v_n)$$

$$(v_0, \ldots, v_n), (w_0, \ldots, w_n) \to g(v_0, w_0) \square \ldots \square g(v_n, w_n)$$



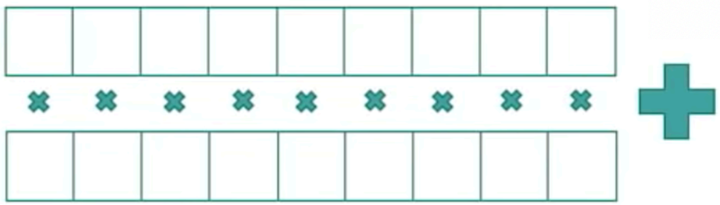
- A vector product $\sum_i v_i w_i$

```
vector<double> x, y;
// ...
transform_reduce(cbegin(x), cend(x), cbegin(y), 0.0);
```

transform_reduce (inner_product)

$$(v_0, \ldots, v_n) \rightarrow f(v_0) \square \ldots \square f(v_n)$$

$$(v_0, \ldots, v_n), (w_0, \ldots, w_n) \rightarrow g(v_0, w_0) \square \ldots \square g(v_n, w_n)$$



- A vector product $\sum_i v_i w_i$

```
vector<double> x, y;
// ...
transform_reduce(cbegin(x), cend(x), cbegin(y), 0.0);
```

- Vector Distance $\sum_i (v_i - w_i)^2$

```
vector<int> v, w;
// ...
transform_reduce(cbegin(v), cend(v), cbegin(w), 0,
  [](int i, int j) -> int { return (i - j) * (i - j); },
  std::plus<>{});
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Other useful algorithms

Other useful algorithms

`inclusive_scan` (partial_sum)

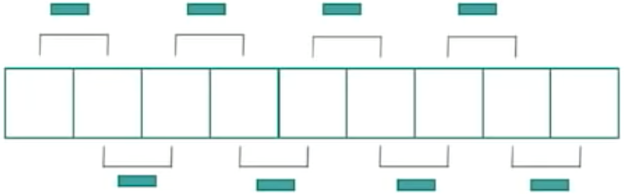
$v_i \rightarrow v_0 \Box \ldots \Box v_i$ Why not `partial_reduce` ?

Other useful algorithms

`inclusive_scan` (`partial_sum`)

$v_i \to v_0 \square \ldots \square v_i$ Why not `partial_reduce` ?

`adjacent_difference`



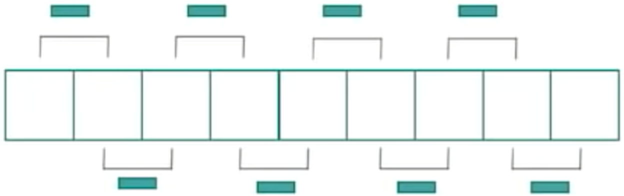
$v_0 \to v_0$ $v_i \to v_{i-1} \square v_i$

Other useful algorithms

`inclusive_scan` (`partial_sum`)

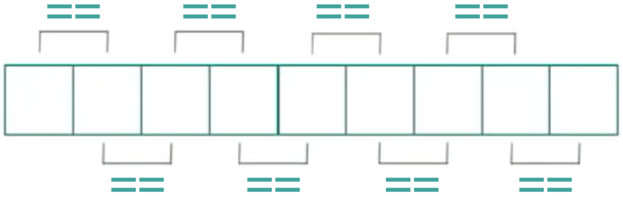
$v_i \to v_0 \square \ldots \square v_i$ Why not `partial_reduce` ?

`adjacent_difference`



$v_0 \to v_0$ $v_i \to v_{i-1} \square v_i$

`adjacent_find`



A note on Parallelism

en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

- Most STL Algorithms can be easily run in parallel

```
auto v = vector<int>(1e5, 1);
reduce(cbegin(v), cend(v), 0);
```

```
#include <execution>
auto v = vector<int>(1e5, 1);
return reduce(std::execution::par, cbegin(v), cend(v), 0);
```

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Ranges — An Outlook

C++20

en.cppreference.com/w/cpp/ranges

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Summary

- Algorithms + Lambdas are incredibly useful!
- In particular `transform_reduce`
  - `generate`, `reduce`, `transform`, `inclusive_scan`, `adjacent_difference`, `adjacent_find`
- Even more powerful and expressive in C++20/23
  - Parallel execution, Compact Syntax, Composability (RangeTS)

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Thank you for your attention!

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Exercise

Given a vector of integers $(v_0, \ldots, v_n)$, generate a vector of indices $(l_0, \ldots, l_n)$ that label the elements from smallest to largest, i.e. $v_{l_i} \leq v_{l_{i+1}} \quad \forall i \in \{0,\ldots,n-1\}$

Solve it in the Browser: bit.ly/2PdGBNi

FLATIRON INSTITUTE
Center for Computational Quantum Physics

FLATIRON INSTITUTE
Center for Computational Quantum Physics