

Lambda Functions & STL Algorithms

Nils Wentzell
Jan 30, 2020

Syntax

[captures]

(params) -> ret { statements; }

A simple Example

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

Syntax

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

A simple Example

C++

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

Python

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

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 * step;
}
```

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

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

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.

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

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);
```

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);
```


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;
}();
```

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

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;  
}();
```

STL Algorithms I

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!

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)

unordered_set (C++11)

unordered_map (C++11)

unordered_multiset (C++11)

unordered_multimap (C++11)

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
```

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
```

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.

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

(since C++17)

sequenced_policy (C++17)
parallel_policy (C++17) execution policy types
parallel_unsequenced_policy (C++17) (class)
unsequenced_policy (C++20)

seq (C++17)
par (C++17) global execution policy objects
par_unseq (C++17) (constant)
unseq (C++20)

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) checks if a predicate is **true** for all, any or none of the elements in
any_of (C++11) a range
none_of (C++11) (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) move_backward (C++11)	moves a range of elements to a new location (function template)
fill	moves a range of elements to a new location in backwards order (function template)
fill_n	copy-assigns the given value to every element in a range (function template)
transform	copy-assigns the given value to N elements in a range (function template)
generate	applies a function to a range of elements, storing results in a destination range (function template)
generate_n	assigns the results of successive function calls to every element in a range (function template)
remove remove_if	assigns the results of successive function calls to N elements in a range (function template)
remove_copy remove_copy_if	removes elements satisfying specific criteria (function template)
replace replace_if	copies a range of elements omitting those that satisfy specific criteria (function template)
replace_copy replace_copy_if	replaces all values satisfying specific criteria with another value (function template)
	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.

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

(since C++17)

sequenced_policy (C++17)
parallel_policy (C++17) execution policy types
parallel_unsequenced_policy (C++17) (class)
unsequenced_policy (C++20)

seq (C++17)
par (C++17) global execution policy objects
par_unseq (C++17) (constant)
unseq (C++20)

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) checks if a predicate is **true** for all, any or none of the elements in a range
any_of (C++11)
none_of (C++11) (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)

std::sort

Defined in header `<algorithm>`

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

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`<`.
- 2) Elements are compared using the given binary comparison function `comp`.
- 3) 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
- 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);
```

Example

Run

Share

Exit

GCC 9.2 (C++2a)

Powered by [Coliru](#) online compiler

```
1 #include <algorithm>
2 #include <functional>
3 #include <array>
4 #include <iostream>
5
6 int main()
7 {
8     std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
9
10    // sort using the default operator<
11    std::sort(s.begin(), s.end());
12    for (auto a : s) {
13        std::cout << a << " ";
14    }
15    std::cout << '\n';
16 }
```

Output:

0 1 2 3 4 5 6 7 8 9

Why use STL Algorithms?

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

Know them & Use them!

What about performance?

godbolt.org/z/FVRSTG

The image shows two side-by-side code editors. The left editor, titled 'C++ source #1', contains the following code:

```
1 #include <vector>
2
3
4 int f(std::vector<int> const & v){
5     int res = 1;
6     for(auto i : v){
7         res *= i;
8     }
9     return res;
10 }
```

The right editor, titled 'C++ source #2', contains the following code:

```
1 #include <vector>
2 #include <numeric>
3 #include <functional>
4
5 int f(std::vector<int> const & v){
6     return std::accumulate(begin(v), end(v), 1, std::multiplies<int>{});
7 }
```

A red rectangular box is overlaid on the bottom right of the image, containing the text 'Identical Assembly!' in a bold, red, sans-serif font.

A simple Example

- Find the maximum absolute difference between any two elements

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

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;  
}
```

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

A simple Example

```
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

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;  
}
```

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

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;  
}
```

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

Lambda Functions & STL Algorithms

Custom 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),  
    [](op_t x, op_t y) { return (x.tau < y.tau); });
```


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.

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

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

$$(v_0, v_1, v_2, \dots, v_n) \rightarrow g(\dots g(g(v_0, v_1), v_2), \dots, v_n)$$

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.

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

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

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

reduce (accumulate)

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



- 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; });
```

reduce (accumulate)

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



- 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

transform

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

- Squaring elements $v_i \rightarrow v_i^2$

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

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

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

transform

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

- Squaring elements $v_i \rightarrow v_i^2$

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

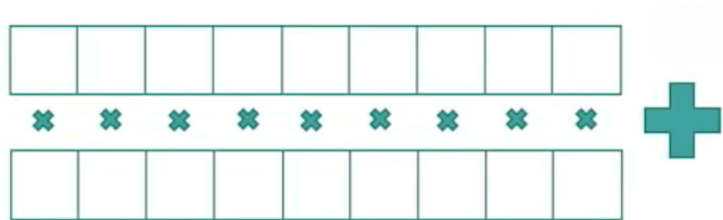
- Logical or $(v_i, w_i) \rightarrow v_i || w_i$

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

transform_reduce (inner_product)

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

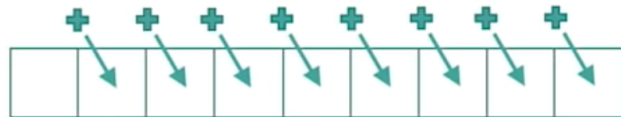
$$(v_0, \dots, v_n), (w_0, \dots, w_n) \rightarrow g(v_0, w_0) \square \dots \square g(v_n, w_n)$$



- 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<>{});

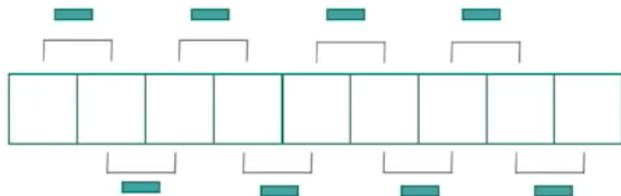
Other useful algorithms

inclusive_scan (partial_sum)



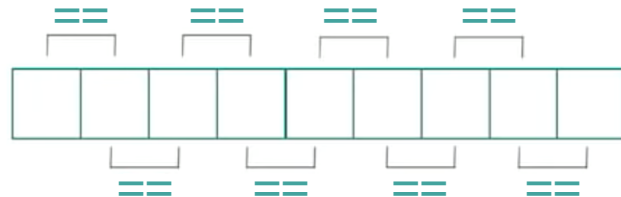
$v_i \rightarrow v_0 \square \dots \square v_i$ Why not partial_reduce ?

adjacent_difference



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

adjacent_find



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.; });
```


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);`

- 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
```

Summary

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

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);
```

Ranges — An Outlook

C++20

en.cppreference.com/w/cpp/ranges

Example

Run this code

```
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> ints{0,1,2,3,4,5};
    auto even = [](int i){ return 0 == i % 2; };
    auto square = [](int i) { return i * i; };

    for (int i : ints | std::views::filter(even) | std::views::transform(square)) {
        std::cout << i << ' ';
    }
}
```

Output:

0 4 16

Thank you for your attention!