

Lambda Functions & Standard Template Library

Nils Wentzell
Oct 20, 2022

Lambda Functions

A simple Example

C++

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

Python

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

Lambda Functions

Integrate a generic function on the interval [0, 1]

```
double integrate(auto f) {  
    double r = 0.0;  
    for (auto x: {0.0, 0.1, ..., 0.9}) r += 0.1*f(x);  
    return r;  
}
```

```
double r1 = integrate( cos );
```

```
double r2 = integrate( [](double x){ return cos(2*x); } );
```

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

```
double r3 = integrate( [r2](double y){ return pow(y, r2); } ); // y^r2
```



Capture it first

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

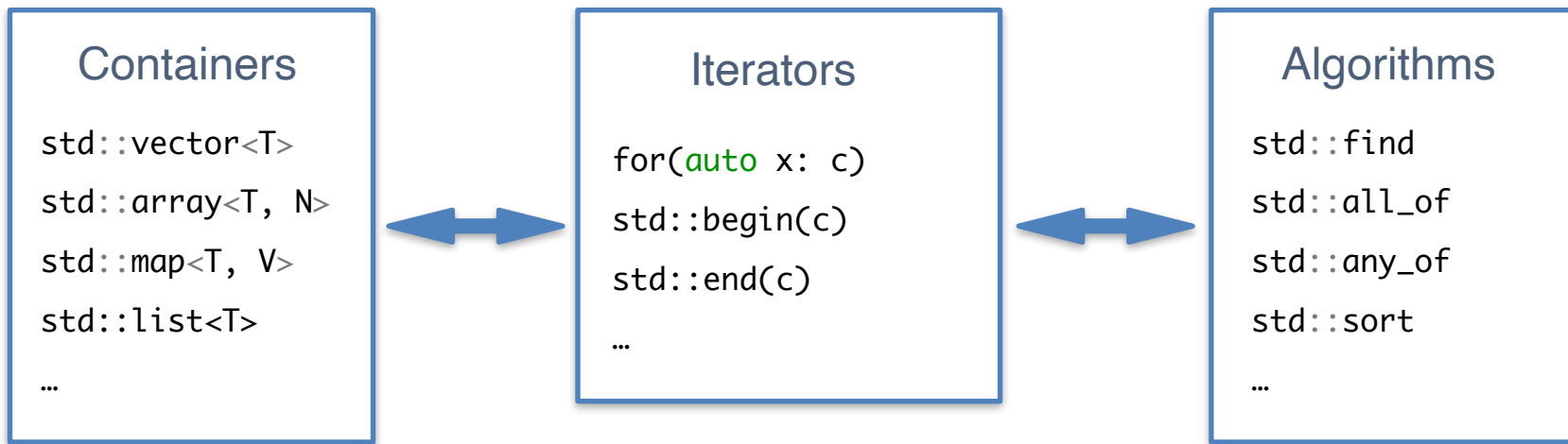
[captures] (params) -> ret { statements; }

[captures]

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

Standard Template Library (STL)

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



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)

STL Containers

en.cppreference.com/w/cpp/container

array (C++11)	static contiguous array (class template)	→ On Stack
vector	dynamic contiguous array (class template)	→ On Heap

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

```
auto a = array{1, 2, 3}; // C++17: auto-deduce element-type & size
```


STL Algorithms

en.cppreference.com/w/cpp/algorithm

- Set of generic algorithms
- Widely known → easy to read
- Tested & Debugged
- Optimal Performance
- Parallel execution C++17/20
- Composable

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.

Defined in header <algorithm>		searches a range for a number of consecutive copies of an element (function template)	
all_of (C++11) any_of (C++11) none_of (C++11) ranges::all_of (C++20) ranges::any_of (C++20) ranges::none_of (C++20) for_each ranges::for_each (C++20) for_each_n (C++17) ranges::for_each_n (C++20) count count_if ranges::count (C++20) ranges::count_if (C++20) mismatch ranges::mismatch (C++20) find find_if find_if_not (C++11) ranges::find (C++20) ranges::find_if (C++20) ranges::find_if_not (C++20) ranges::find_last ranges::find_last_if ranges::find_last_if_n find_end ranges::find_end (C++20) find_first_of ranges::find_first_of (C++20) adjacent_find ranges::adjacent_find (C++20) search ranges::search (C++20)	search_n ranges::search_n (C++20) ranges::starts_with (C++20) ranges::ends_with (C++20) Modifying sequence operations Defined in header <algorithm> copy copy_if (C++11) ranges::copy (C++20) ranges::copy_if (C++20) copy_n (C++11) ranges::copy_n (C++20) copy_backward ranges::copy_backward (C++20) move (C++11) ranges::move (C++20) move_backward (C++11) ranges::move_backward (C++20) fill ranges::fill (C++20) fill_n ranges::fill_n (C++20) transform ranges::transform (C++20) generate ranges::generate (C++20) generate_n ranges::generate_n (C++20)	remove remove_if ranges::remove (C++20) ranges::remove_if (C++20) remove_copy remove_copy_if ranges::remove_copy (C++20) ranges::remove_copy_if (C++20) replace replace_if ranges::replace (C++20) ranges::replace_if (C++20) replace_copy replace_copy_if ranges::replace_copy (C++20) ranges::replace_copy_if (C++20) swap swap_ranges ranges::swap_ranges (C++20) iter_swap reverse ranges::reverse (C++20) reverse_copy ranges::reverse_copy (C++20) rotate ranges::rotate (C++20) rotate_copy ranges::rotate_copy (C++20) shift_left shift_right (C++20) ranges::shift_left (C++23) ranges::shift_right (C++23)	removes elements satisfying specific criteria (function template) removes elements satisfying specific criteria (inplace) copies a range of elements omitting those that satisfy specific criteria (function template) copies a range of elements omitting those that satisfy specific criteria (inplace) replaces all values satisfying specific criteria with another value (function template) replaces all values satisfying specific criteria with another value (inplace) copies a range, replacing elements satisfying specific criteria with another value (function template) copies a range, replacing elements satisfying specific criteria with another value (function template) swaps the values of two objects (function template) swaps two ranges of elements (function template) swaps two ranges of elements (inplace) swaps the elements pointed to by two iterators (function template) reverses the order of elements in a range (function template) reverses the order of elements in a range (inplace) creates a copy of a range that is reversed (function template) creates a copy of a range that is reversed (inplace) rotates the order of elements in a range (function template) rotates the order of elements in a range (inplace) copies and rotate a range of elements (function template) copies and rotate a range of elements (inplace) shifts elements in a range (function template) shifts elements in a range (inplace) shifts elements in a range (inplace)

std::sort

- Simple Example

```
auto v = vector{ 2, 1, 3 };  
sort(begin(v), end(v)); // -> 1, 2, 3  
  
std::ranges::sort(v); // C++20
```

- A custom sort

```
// Find traversal order from smallest to largest  
auto v = vector{ 2, 1, 3, 6, 5, 4 };  
  
// Create index vector 0, 1, 2, .. N-1  
auto l = vector<int>(v.size());  
iota(begin(l), end(l), 0);  
  
sort(begin(l), end(l), [&v](int i, int j){ return v[i] < v[j]; });  
// -> 1, 0, 2, 5, 4, 3
```

reduce (accumulate)

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



- The default case $\sum_i v_i$

```
auto v = vector{ 2, 1, 3 };  
reduce(begin(v), end(v), 0); // -> 6
```

- Custom reduction $\prod_i v_i$

```
auto v = vector{ 2, 1, 3 };  
reduce(begin(v), end(v), 1, [](int i, int j){ return i*j; }); // -> 6
```

reduce (accumulate)

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



- The default case $\sum_i v_i$

```
auto v = vector{ 2, 1, 3 };  
reduce(begin(v), end(v), 0); // -> 6
```

- Custom reduction $\prod_i v_i$

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

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

STL Cheat Sheets

hackingcpp.com/cpp/cheat_sheets.html

Algorithms

`minmax_element(@begin, @end) → { @minimum, @maximum }`

C++11

`any_of (@begin, @end, f(○)→bool) →` true, if **f** yields true for *any*, *all* or *none* elements in the input range

`all_of (@begin, @end, f(○)→bool) →`

`none_of (@begin, @end, f(○)→bool) →` false otherwise

C++11

`sort(@begin, @end, f(○,○)→bool)`
`sort(@begin, @end, std::less)`

`stable_sort(@begin, @end, compare(○,○)→bool)`
`= ○ < ○`

compare_case_insensitive

"stable" = preserves the relative order of equivalent elements

Containers

vector<T>

dynamic array

C++'s "default" container

#include <vector>

```
std::vector<int> v {1,2,3,4,5,6};
v.reserve(9);
cout << v.capacity(); // 9
cout << v.size(); // 6
v.push_back(7); // appends '7'
v.insert(v.begin(), 0); // prepends '0'
v.pop_back(); // removes last
v.erase(v.begin()+2); // removes 3rd
v.resize(20, 0); // size ⇒ 20
```

contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

and much more...

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(begin(v), end(v), 0);
```



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

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

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 (Ranges)

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{ 2, 1, 3 };  
transform(begin(v), end(v), begin(v),  
    [](int i){ return i * i; });
```

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

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

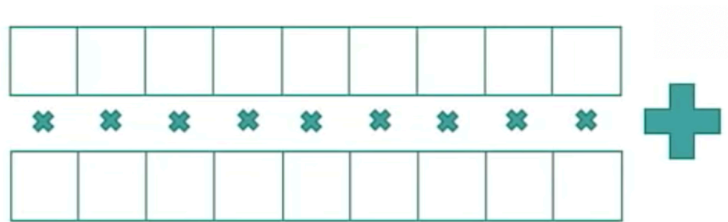
- Logical or $(v_i, w_i) \rightarrow v_i || w_i$

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

transform_reduce (inner_product)

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

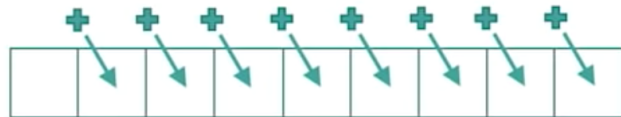
$$(v_0, \dots, v_n), (w_0, \dots, w_n) \rightarrow g(v_0, w_0) \square \dots \square g(v_n, w_n)$$



- A vector product $\sum_i v_i w_i$
vector<double> x, y;
// ...
transform_reduce(begin(x), end(x), begin(y), 0.0);
- Vector Distance $\sum_i (v_i - w_i)^2$
vector<int> v, w;
// ...
transform_reduce(begin(v), end(v), begin(w), 0,
 [](int i, int j) -> int { return (i - j) * (i - j); },
 std::plus<>{});

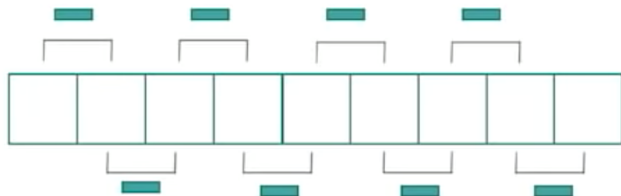
Other useful algorithms

inclusive_scan (partial_sum)



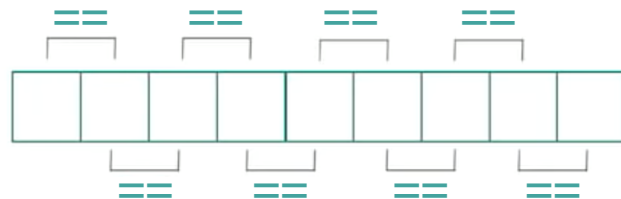
$v_i \rightarrow v_0 \square \dots \square v_i$ Why not partial_reduce ?

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(begin(v), end(v), [](int i){ return i > 10; });
```

- Any values negative?

```
vector<double> x;  
any_of(begin(x), end(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
```