The New Algorithms

This lesson gives an overview of the new algorithms that are a part of C++17.

The new algorithms are in the `std` namespace. `std::for_each` and `std::for_each_n` require the header <algorithm>, but the remaining 6 algorithms require the header <numeric>.

Here is an overview of the new algorithms:

Algorithm	Description
`std::for_each`	Applies a unary ***callable*** to the range.
`std::for_each_n`	Applies a unary callable to the first n elements of the range.
`std::exclusive_scan`	Applies from the left a binary callable up to the ith (exclusive) element of the range. The left argument of the callable is the previous result. Stores intermediate results.
`std::inclusive_scan`	Applies from the left a binary callable up to the ith (inclusive) element of the range. The left argument of the callable is the previous result. Stores intermediate results.
`std::transform_exclusive_scan`	First applies a unary callable to the

`std::transform_exclusive_scan`	First applies a unary callable to the range and then applies `std::exclusive_scan`.
`std::transform_inclusive_scan`	First applies a unary callable to the range and then applies `std::inclusive_scan`.
`std::reduce`	Applies from the left a binary callable to the range.
`std::transform_reduce`	Applies first a unary callable to the range and then `std::reduce`.

Admittedly, this description is not easy to digest; therefore, I will first present an exhaustive example and then write about the functional heritage of these functions. I will ignore the new `std::for_each` algorithm. In contrast to the C++98 variant that returns a unary function, the additional C++17 variant returns nothing.

As far as I know, at the time this course is being written (June 2017) there is no standard-conforming implementation of the parallel STL available. Therefore, I used the HPX implementation to produce the output. The HPX (High-Performance ParalleX) is a framework that is a general purpose C++ runtime system for parallel and distributed applications of any scale.

```
// newAlgorithm.cpp


#include <algorithm>
#include <numeric>
#include <iostream>
#include <string>
#include <vector>

int main(){

  std::cout << std::endl;

  // for_each_n

  std::vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::for_each_n(std::execution::par,
                  intVec.begin(), 5, [](int& arg){ arg *= arg; });
```

```
std::cout << "for_each_n: ";
for (auto v: intVec) std::cout << v << " ";
std::cout << "\n\n";

// exclusive_scan and inclusive_scan
std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::exclusive_scan(std::execution::par,
                    resVec.begin(), resVec.end(), resVec.begin(), 1,
                    [](int fir, int sec){ return fir * sec; });

std::cout << "exclusive_scan: ";
for (auto v: resVec) std::cout << v << " ";
std::cout << std::endl;

std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8, 9};

std::inclusive_scan(std::execution::par,
                    resVec2.begin(), resVec2.end(), resVec2.begin(),
                    [](int fir, int sec){ return fir * sec; }, 1);

std::cout << "inclusive_scan: ";
for (auto v: resVec2) std::cout << v << " ";
std::cout << "\n\n";

// transform_exclusive_scan and transform_inclusive_scan
std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> resVec4(resVec3.size());
std::transform_exclusive_scan(std::execution::par,
                              resVec3.begin(), resVec3.end(),
                              resVec4.begin(), 0,
                              [](int fir, int sec){ return fir + sec; },
                              [](int arg){ return arg *= arg; });

std::cout << "transform_exclusive_scan: ";
for (auto v: resVec4) std::cout << v << " ";
std::cout << std::endl;

std::vector<std::string> strVec{"Only","for","testing","purpose"};
std::vector<int> resVec5(strVec.size());

std::transform_inclusive_scan(std::execution::par,
                              strVec.begin(), strVec.end(),
                              resVec5.begin(), 0,
                              [](auto fir, auto sec){ return fir + sec; },
                              [](auto s){ return s.length(); });

std::cout << "transform_inclusive_scan: ";
for (auto v: resVec5) std::cout << v << " ";
std::cout << "\n\n";

// reduce and transform_reduce
std::vector<std::string> strVec2{"Only","for","testing","purpose"};

std::string res = std::reduce(std::execution::par,
                  strVec2.begin() + 1, strVec2.end(), strVec2[0],
                  [](auto fir, auto sec){ return fir + ":" + sec; });

std::cout << "reduce: " << res << std::endl;

std::size_t res7 = std::transform_reduce(std::execution::par,
                   strVec2.begin(), strVec2.end(),
```

```
                      [](std::string s){ return s.length(); },
                      0, [](std::size_t a, std::size_t b){ return a + b; });


  std::cout << "transform_reduce: " << res7 << std::endl;

  std::cout << std::endl;

}
```

I apply the new algorithms to a `std::vector<int>` (line 16) and a `std::vector<std::string>` (line 57).

The `std::for_each_n` algorithm in line 17 maps the first `n` ints of the vector to their powers of 2.

`std::exclusive_scan` (line 26) and `std::inclusive_scan` (line 36) are quite similar; both apply a binary operation to their elements. The difference is that `std::exclusive_scan` excludes the last element in each iteration.

Let me explain the `std::transform_exclusive_scan` in line 47. In the first step, I apply the lambda function `[](int arg){ return arg *= arg; }` to each element of the range `resVec3.begin()` to `resVec3.end()`. In the second step, I apply the binary operation `[](int fir, int sec){ return fir + sec; }` to the intermediate vector. This means to sum up all elements using 0 as the initial value. The result is placed in `resVec4.begin()`.

The `std::transform_inclusive_scan` function in line 60 is similar. This function maps each element to its length.

The `std::reduce` function puts `":"` characters between every two elements of the input vector. The resulting string should not start with a `":"` character; therefore, the range starts at the second element (`strVec2.begin() + 1`) and uses the first element of the vector `strVec2[0]` as the initial element.

I will now discuss the `std::transform_reduce` function in line 79. First of all, the C++ algorithm `transform` is often called map in other languages; therefore, we can also call `std::transform_reduce` `std::map_reduce`. `std::transform_reduce` is the well-known parallel MapReduce algorithm implemented in C++. Accordingly, `std::transform_reduce` maps a unary callable (`[](std::string s){ return s.length(); }`) onto a range and reduces the pair to a output value: `[](std::size_t a, std::size_t b){ return a + b; }`.

Studying the output of the program will help you.

> ### More overloads
>
> All C++ variants of reduce and scan have more overloads. In the simplest form, you can invoke them without an initial element and without a binary callable. If you do not use an initial element, the first element will be used. If you do not use a binary callable, the addition will be used as the binary operation.

In the next lesson, I will discuss new algorithms from a functional perspective.

```
main = do let ints = [1..9]
          let strings =["Only","for","testing","purpose"]
          print (map (\a -> a * a) ints)
          print (scanl (*) 1 ints)
          print (scanl (+) 0 ints)
          print (scanl (+) 0 . map(\a -> a * a) $ints)
          print (scanl1 (+) . map(\a -> length a) $strings)
          print (foldl1 (\l r -> l++ ":" ++r) strings)
          print (foldl (+) 0 . map (\a -> length a) $strings)
```

(1) and (2) define a list of integers and a list of strings. In (3), I apply the lambda function `(\a -> a * a)` to the list of integers. That being said, (4) and (5) are more sophisticated. The expression (4) multiplies (*) all pairs of integers starting with the 1 as neutral element of multiplication. Expression (5) does the corresponding for addition. Expressions (6), (7), and (9) are, for the imperative eye, quite challenging. You have to read them from right to left. `scanl1 (+) . map(\a -> length)` (7) is a function composition. The dot (.) symbol composes the two functions. The first function maps each element to its length, the second function adds the list of lengths together. (9) is similar to (7), the difference being that `foldl` produces one value and requires an initial element that is in case 0. Now expression (8) should be readable; it successively joins two strings with the ":" character.