

Ranges and algorithms

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: September 26, 2024

1. Range-based iteration

You have seen

```
for ( auto n : set_of_integers )  
    if ( even(n) )  
        do_something(n);
```

Can we do

```
for ( auto n : set_of_integers  
      and even ) // <= not actual syntax  
    do_something(n);
```

or even

```
// again, not actual syntax  
apply( set_of_integers and even,  
        do_something );
```

2. Range algorithms

Algorithms: for-each, find, filter, transform ...

Ranges: iterable things such as vectors

Views: transformations of ranges, such as picking only even numbers

3. Range over vector

With

```
// rangestd/range.cpp  
vector<int> v{2,3,4,5,6,7};
```

Code:

```
// rangestd/range.cpp  
#include <ranges>  
namespace rng = std::ranges;  
#include <algorithm>  
/* ... */  
rng::for_each  
    ( v,  
      [] (int i) {  
          cout << i << " ";  
      }  
    );
```

Output:

2 3 4 5 6 7

4. Range with accumulation

Capture a global accumulator by reference:

Code:

```
// rangestd/range.cpp  
count = 0;  
rng::for_each  
    ( v,  
      [&count] (int i) {  
        count += (i<5); }  
    );  
cout << "Under five: "  
      << count << '\n';
```

Output:

Under five: 3

Exercise 1

Revisit the vector normalization and rewrite the *norm* function to use a *for_each* algorithm.

Also rewrite the *scale* function using a *for_each* algorithm.

5. Range composition

Pipeline of ranges and views:

```
// rangestd/range.cpp  
vector<int> v{2,3,4,5,6,7};
```

Code:

```
// rangestd/range.cpp  
count = 0;  
rng::for_each  
  ( v | rng::views::drop(1),  
    [&count] (int i) {  
      count += (i<5); }  
  );  
cout << "minus first: "  
      << count << '\n';
```

Output:

```
minus first: 2
```

'pipe operator'

6. Filter

Filter a range by some condition:

Code:

```
// rangestd/filter.cpp
vector<float> numbers
{1,-2.2,3.3,-5,7.7,-10};
for ( auto n :
      numbers
      |
      std::ranges::views::filter
        ( [] (int i) -> bool {
            return i>0; } )
      )
  cout << n << " ";
cout << '\n';
```

Output:

1 3.3 7.7

Exercise 2

Change the filter example to let the lambda count how many elements were > 0 .

7. Range composition

Code:

```
// range/filtertransform.cpp
vector<int> v{ 1,2,3,4,5,6 };
/* ... */
auto times_two_over_five = v
    | rng::views::transform
      ( [] (int i) {
          return 2*i; } )
    | rng::views::filter
      ( [] (int i) {
          return i>5; } );
```

Output:

Original vector:
1, 2, 3, 4, 5, 6,
Times two over five:
6 8 10 12

8. Quantor-like algorithms

Code:

```
// rangestd/of.cpp
vector<int>
    integers{1,2,3,5,7,10};
auto any_even =
    std::ranges::any_of
        ( integers,
          [=] (int i) -> bool {
              return i%2==0; }
        );
if (any_even)
    cout << "there was an
        even\n";
else
    cout << "none were even\n";
```

Output:

there was an even

Also *all_of*, *none_of*

9. Reductions

accumulate and *reduce*:
tricky, and not in all compilers.
See above for an alternative.

10. Iota and take

Code:

```
// rangestd/iota.cpp
#include <ranges>
namespace rng = std::ranges;
    /* ... */
    for ( auto n :
        rng::views::iota(2,6) )
        cout << n << '\n';
    cout << "===\n";
    for ( auto n :
        rng::views::iota(2)
        | rng::views::take(4) )
        cout << n << '\n';
```

Output:

```
2
3
4
5
===
2
3
4
5
```

Exercise 3

A perfect number is the sum of its own divisors:

$$6 = 1 + 2 + 3$$

Output the perfect numbers.

(at least 4 of them)

Use only ranges and algorithms, no explicit loops.

1. Write a lambda expression to compute the sum of the factors of a number
2. Use *iota* to iterate over numbers: if one is equal to the sum of its factors, print it out.
3. Use *filter* to pick out these numbers.