Ranges and algorithms

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: September 24, 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: iteratable things such as vectors

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

C++20 ranges

3. Range over vector

With

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

Code:

```
// rangestd/range.cpp
#include <algorithm>
#include <ranges>
namespace rng = std::ranges;
    /* ... */
  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

Find your solution of exercise **??** and rewrite the *norm* function to use a *for_each* lambda.

Be sure to include

```
// rangestd/norm2.cpp
#include <ranges>
#include <algorithm>
namespace rng = std::ranges;
// now you can write `rng::for_each'
```

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

Take a range, and make a new one of only the elements satisfying some condition:

Code:

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

Output:

```
1 3.3 7.7
```

Exercise 2: Element counting

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: Perfect numbers

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.

Iterators

11. Iterate without iterators

```
vector data{2,3,1};
sort( begin(data),end(data) ); // open to accidents
ranges::sort(data);
```

12. Begin and end iterator

Use independent of looping:

Code:

```
// stl/iter.cpp
    vector<int> v{1,3,5,7};
    auto pointer = v.begin();
    cout << "we start at "
         << *pointer << '\n';
    ++pointer;
    cout << "after increment:
    "
         << *pointer << '\n';

    pointer = v.end();
    cout << "end is not a
    valid element: "
         << *pointer << '\n';
    pointer--;
    cout << "last element: "
         << *pointer << '\n';
```

Output:

```
we start at 1
after increment: 3
end is not a valid
    ↪element: 0
last element: 7
```

13. Erase at/between iterators

Erase from start to before-end:

Code:

```
// iter/iter.cpp
vector<int>
    counts{1,2,3,4,5,6};
vector<int>::iterator second
    = counts.begin()+1;
auto fourth = second+2;
counts.erase(second,fourth);
cout << counts[0]
     << "," << counts[1] <<
    '\n';
```

Output:

```
1,4
```

(Also erasing a single element without end iterator.)

14. Insert at iterator

Insert at iterator: value, single iterator, or range:

Code:

```
// iter/iter.cpp
vector<int>
    counts{1,2,3,4,5,6},
  zeros{0,0};
auto after_one =
    zeros.begin()+1;
zeros.insert
  ( after_one,
    counts.begin()+1,
    counts.begin()+3 );
cout << zeros[0] << ","
     << zeros[1] << ","
     << zeros[2] << ","
     << zeros[3]
     << '\n';
```

Output:

```
0,2,3,0
```

15. Iterator arithmetic

```
auto first = myarray.begin();
first += 2;
auto last  = myarray.end();
last  -= 2;
myarray.erase(first,last);
```

Algorithms with iterators

16. Reduction operation

Default is sum reduction:

Code:

```
// stl/reduce.cpp
#include <numeric>
using std::accumulate;
    /* ... */
    vector<int> v{1,3,5,7};
    auto first = v.begin();
    auto last  = v.end();
    auto sum =
    accumulate(first,last,0);
    cout << "sum: " << sum <<
    '\n';
```

Output:

```
sum: 16
```

17. Reduction with supplied operator

Supply multiply operator:

Code:

```
// stl/reduce.cpp
using std::multiplies;
    /* ... */
    vector<int> v{1,3,5,7};
    auto first = v.begin();
    auto last  = v.end();
    ++first; last--;
    auto product =
      accumulate(first,last,2,

    multiplies<>());
    cout << "product: " <<
    product << '\n';
```

Output:

```
product: 30
```

18. Custom reduction function

```
// stl/reduce.cpp
class x {
public:
  int i,j;
  x() {};
  x(int i,int j) : i(i),j(j)
    {};
};
```

```
// stl/reduce.cpp
    std::vector< x > xs(5);
    auto xxx =
      std::accumulate
      (
    xs.begin(),xs.end(),0,
        [] ( int init,x x1 )
    -> int { return
    x1.i+init; }
        );
```

Write your own iterator

19. Vector iterator

Range-based iteration

```
for ( auto element : vec ) {
  cout << element;
}
```

is syntactic sugar around iterator use:

```
for (std::vector<int>::iterator elt_itr=vec.begin();
      elt_itr!=vec.end(); ++elt_itr) {
  element = *elt_itr;
  cout << element;
}
```

20. Custom iterators, 0

Recall that

Short hand:

```
vector<float> v;
for ( auto e : v )
  ... e ...
```

for:

```
for (
    vector<float>::iterator
    e=v.begin();
      e!=v.end(); e++ )
  ... *e ...
```

If we want

```
for ( auto e : my_object )
    ... e ...
```

we need an iterator class with methods such as *begin*, *end*, * and ++.

21. Custom iterators, 1

Ranging over a class with iterator subclass

Class:

```
// loop/iterclass.cpp
class NewVector {
protected:
  // vector data
  int *storage;
  int s;
    /* ... */
public:
  // iterator stuff
  class iter;
  iter begin();
  iter end();

};
```

Main:

```
// loop/iterclass.cpp
NewVector v(s);
  /* ... */
for ( auto e : v )
  cout << e << " ";
```

22. Custom iterators, 2

Random-access iterator:

```
// loop/iterclass.cpp
NewVector::iter& operator++();
int& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-( const NewVector::iter& other ) const;
NewVector::iter& operator+=( int add );
```

Exercise 4

Write the missing iterator methods. Here's something to get you started.

```
// loop/iterclass.cpp
class NewVector::iter {
private: int *searcher;
    /* ... */
NewVector::iter::iter( int *searcher )
  : searcher(searcher) {};
NewVector::iter NewVector::begin() {
  return NewVector::iter(storage); };
NewVector::iter NewVector::end()   {
  return NewVector::iter(storage+NewVector::s); };
```