Ranges

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: November 19, 2025

1. Begin and end iterator

Use independent of looping:

Code:

```
1 // stl/iter.cpp
2 vector<int> v{1,3,5,7};
3 auto pointer = v.begin();
4 cout << "we start at "
5      << *pointer << '\n';
6 ++pointer;
7 cout << "after increment: "
8      << *pointer << '\n';
9
10 pointer = v.end();
11 cout << "end is not a valid element:
       "
12      << *pointer << '\n';
13 pointer--;
14 cout << "last element: "
15      << *pointer << '\n';
```

Output:

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

2. Iterator arithmetic

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

Algorithms with iterators

3. Vector iterator

Range-based iteration

```
1 for ( auto element : vec ) {
2   cout << element;
3 }
```

is syntactic sugar around iterator use:

```
1 for (std::vector<int>::iterator elt_itr=vec.begin();
2       elt_itr!=vec.end(); ++elt_itr) {
3   element = *elt_itr;
4   cout << element;
5 }
```

4. Copy range

Copy a begin/end range of one container
to an iterator in another container::

Code:

```
1 // iter/iter.cpp
2 vector<int> counts{1,2,3,4};
3 vector<int> copied(5);
4 copy( counts.begin(),counts.end(),
5       copied.begin()+1 );
6 cout << copied[0]
7      << ", " << copied[1]
8      << ".." << copied[4] << '\n';
```

Output:

```
1 0, 1..4
```

(No bound checking, so be careful!)

5. Erase at/between iterators

Erase from start to before-end:

Code:

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

Output:

```
1 1,4
```

(Also erasing a single element without end iterator.)

6. Insert at iterator

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

Code:

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

Output:

```
1 0,2,3,0
```

C++20 ranges

7. Iterate without iterators

The ranges library has range variants for iterator-based functions:

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

8. For each

`for_each`: for each element of a range, apply a lambda:

Code:

```
1 // rangestd/range.cpp
2 #include <ranges>
3 namespace rng = std::ranges;
4 #include <algorithm>
5     /* ... */
6   vector<int> v{2,3,4,5,6,7};
7   rng::for_each
8     ( v, [] (int i) { cout << i << "
        "; } );
```

Output:

```
1 2 3 4 5 6 7
```

9. For each'

Use capture to do a reduction:

Code:

```
1 // rangestd/range.cpp
2 count = 0;
3 rng::for_each
4   ( v,
5     [&count] (int i) {
6       count += (i<5); }
7     );
8 cout << "Under five: "
9      << count << '\n';
```

Output:

```
1 Under five: 3
```

10. Any

Quantified predicates over a range:

Code:

```
1 // rangestd/of.cpp
2 vector<int> integers{1,2,3,5,7,10};
3 auto any_even =
4   std::ranges::any_of
5     ( integers,
6       [] (int i) -> bool {
7         return i%2==0; }
8      );
9 if (any_even)
10   cout << "there was an even\n";
11 else
12   cout << "none were even\n";
```

Output:

```
1 there was an even
```

11. Reduction operation

Default is sum reduction:

```
1 // range/sumsquare.cpp
2 vector<float> elements{.5f,1.f,1.5f};
3 auto sumsq =
4   rng::accumulate
5   ( elements
6     | rng::views::transform( [] (auto e) { return e*e; } ),
7     0.f );
8 cout << "Sum of squares: " << sumsq << '\n';
```

12. Reduction with supplied operator

Supply multiply operator:

Code:

```
1 // range/reduce.cpp
2 vector<int> v{1,3,5,7};
3 auto product =
4   rng::accumulate
5     (v,2,multiplies<>());
6 cout << "multiplied: " << product <<
        '\n';
```

Output:

```
1 multiplied: 210
```

13. For loop as view

Range of integers from `std::range::iota`
finite or infinite::

Code:

```
1 // rangestd/iota.cpp
2 #include <ranges>
3 namespace rng = std::ranges;
4     /* ... */
5   for ( auto n :
6           rng::views::iota(2,6) )
7     cout << n << '\n';
8   cout << "===\n";
9   for ( auto n :
10          rng::views::iota(2)
11          | rng::views::take(4) )
12    cout << n << '\n';
```

Output:

```
1 2
2 3
3 4
4 5
5 ===
6 2
7 3
8 4
9 5
```

14. Stream composition

Code:

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

Output:

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

Exercise 1

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 the numbers $1 \ldots 10,1000$: if one is equal to the sum of its factors, print it out.
3. Use `filter` to pick out these numbers.

More ranges

15. Zipped views

Zip ranges together with `ranges::views::zip`, giving a tuple:

Code:

```
1 // rangestd/zip.cpp
2 vector a { 10, 20, 30, 40, 50 };
3 vector<string> b
4   { "one", "two", "three", "four" };
5
6 for (const auto& [num, name] :
7         rng::views::zip(a, b))
8   cout << name << " -> " << num <<
       '\n';
```

Output:

```
1 one -> 10
2 two -> 20
3 three -> 30
4 four -> 40
```

16. Sliding window

Return adjacent elements with `ranges::adjacent` as a tuple.

```
1 // range/adjacent.cpp
2   for ( auto [ lt,mi,rt ] : values | views::adjacent<3> )
3     cout << fixed
4          << setw(3) << lt << ","
5          << setw(3) << mi << ","
6          << setw(3) << rt << '\n';
```

Write your own iterator

17. Vector iterator

Range-based iteration

```
1 for ( auto element : vec ) {
2   cout << element;
3 }
```

is syntactic sugar around iterator use:

```
1 for (std::vector<int>::iterator elt_itr=vec.begin();
2       elt_itr!=vec.end(); ++elt_itr) {
3   element = *elt_itr;
4   cout << element;
5 }
```

18. Custom iterators, 0

Recall that

Short hand:

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

for:

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

If we want

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

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

19. 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 << " ";
```

20. 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 2

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); };
```