Iterators

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: October 4, 2022

Begin/end iterator

1. Containers have iterators

Containers such as `vector`, `map` have `begin/end` iterators.

'Pointer' (not in the technical sense) to first and one-beyond-last element.

2. Begin and end iterator

Use independent of looping:

Code:

```
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

```
[stl] iter:

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

(Note: the `auto` actually stands for `vector::iterator`)

3. (In case you know C)

This is not a C-style pointer dereference,
but rather an overloaded oeprator.

4. Copy range

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

Code:

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

Output

```
[iter] copy:

0, 1..4
```

(No bound checking, so be careful!)

5. Beyond begin/end

- An iterator is a little like a pointer
  (into anything iteratable)
- *begin* / *end*
- pointer-arithmetic and 'dereferencing':

```
auto element_ptr = my_vector.begin();
element_ptr++;
cout << *element_ptr;
```

- allows operations (erase, insert) on containers:
  erase/insert elements at some location given by an iterator

6. Erase at/between iterators

Erase from start to before-end:

Code:

```
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

```
[iter] erase2:

1,4
```

(Also single element without end iterator.)

7. Insert at iterator

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

Code:

```
    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

```
[iter] insert2:

0,2,3,0
```

8. Reconstruct index

Find 'index' by getting the distance between two iterators:

Code:

```
vector<int> numbers{1,3,5,7,9};
auto it=numbers.begin();
while ( it!=numbers.end() ) {
  auto d =
     distance(numbers.begin(),it);
  cout << "At distance " << d
       << ": " << *it << '\n';
  it++;
}
```

Output

```
[loop] distance:

At distance 0: 1
At distance 1: 3
At distance 2: 5
At distance 3: 7
At distance 4: 9
```

Algorithms

9. Reduction operation

Default is sum reduction:

Code:

```
#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

```
[stl] accumulate:

sum: 16
```

10. Reduction with supplied operator

Supply multiply operator:

Code:

```
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

```
[stl] product:

product: 30
```

11. Use lambda to find any of

Here is an example using any_of to find whether there is any even element in a vector:

Code:

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

Output

```
[range] anyof:

there was an even
```

12. For each, very simple example

Apply something to each array element:

Code:

```
#include <algorithm>
  /* ... */
    vector<int>
     ints{2,3,4,5,7,8,13,14,15};
    for_each( ints.begin(),ints.end(),
              [] ( int i ) -> void {
                cout << i << '\n';
              }
              );
```

Output

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

13. For any

Reduction with boolean result:

See if any element satisfies a test

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
bool there_was_an_8 =
  any_of( ints.begin(),ints.end(),
          [] ( int i ) -> bool {
            return i==8;
          }
          );
cout << "There was an 8: " <<
 boolalpha << there_was_an_8 <<
 '\n';
```

Output

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

(Why wouldn't you use a `accumulate` reduction?)

Exercise 1

Use `for_each` to sum the elements of a vector.

Hint: the problem is how to treat the sum variable. Do not use a global variable!

14. Capture by reference

Capture variables are normally by value, use ampersand for reference. This is often used in *algorithm* header.

Code:

```
vector<int> moreints{8,9,10,11,12};
int count{0};
for_each
  ( moreints.begin(),moreints.end(),
    [&count] (int x) {
      if (x%2==0)
        count++;
    } );
cout << "number of even: " << count
   << '\n';
```

Output

```
[stl] counteach:

number of even: 3
```

15. For each, with capture

Capture by reference, to update with the array elements.

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
int sum=0;
for_each( ints.begin(),ints.end(),
          [&sum] ( int i ) ->
 void {
            sum += i;
          }
          );
cout << "Sum = " << sum << '\n';
```

Output

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

16. Sorting

Iterator syntax:
(see later for ranges)

```
sort( myvec.begin(),myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```
sort( myvec.begin(),myvec.end(),
      [] (int i,int j) { return i>j; }
    );
```