Iterators

Nicolai M. Josuttis

josuttis.com

@NicoJosuttis

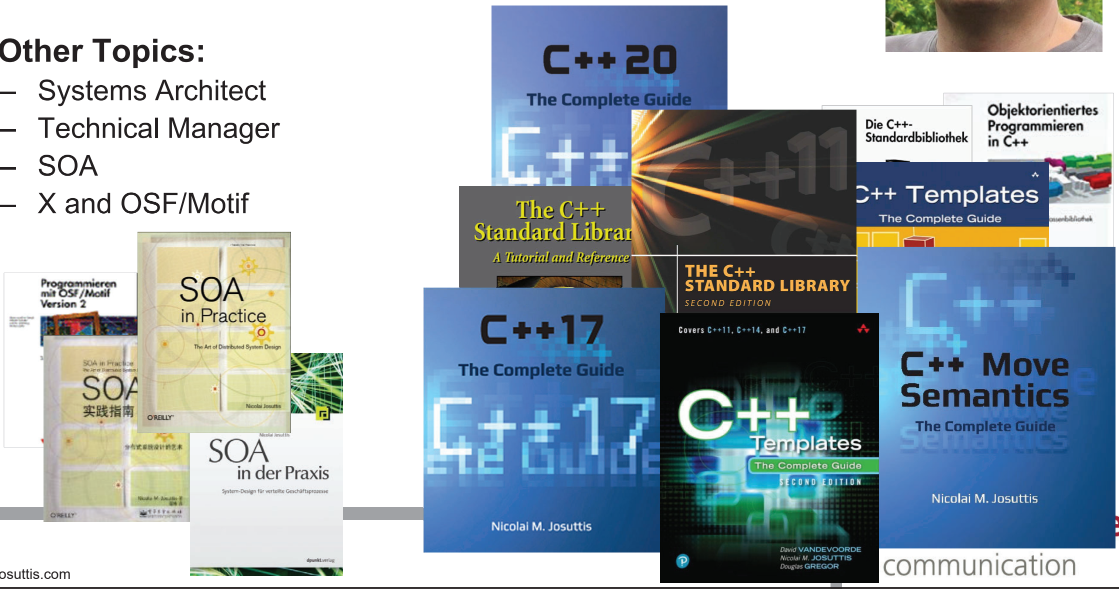
10/23

C++

josuttis | eckstein
IT communication

Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif


C++20 The Complete Guide
The C++ Standard Library A Tutorial and Reference
THE C++ STANDARD LIBRARY SECOND EDITION
Die C++-Standardbibliothek
Objektorientiertes Programmieren in C++
C++ Templates The Complete Guide
C++17 The Complete Guide
C++ Templates The Complete Guide SECOND EDITION
C++ Move Semantics The Complete Guide
Programmieren mit OSF/Motif Version 2
SOA in Practice
SOA 实践指南
SOA in der Praxis
Nicolai M. Josuttis

C++

Modern C++

Iterators

C++

josuttis | eckstein
IT communication

Loop Over Arrays

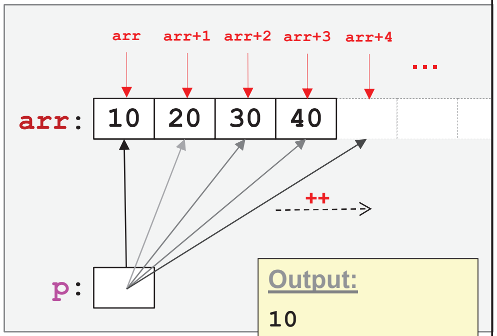
C/C++

- Two ways to iterate over the elements of an array
  - Using indexes
  - Using pointers

```
int arr[] = {10, 20, 30, 40};

// iterate over elements with index:
for (int i = 0; i < 4; ++i) {
  std::cout << arr[i] << '\n';
}
```

```
// iterate over elements with pointer:
for (int* p = arr; p < arr + 4; ++p) {
  std::cout << *p << '\n';
}
```

```
int* p = arr + 4;  // OK
std::cout << *p;  // Undefined Behavior
```



Output:
```
10
20
30
40
10
20
30
40
```

C++

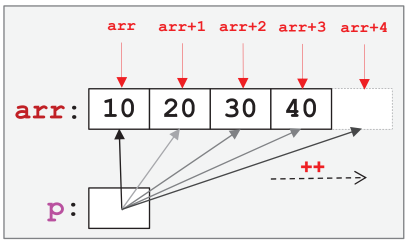
josuttis | eckstein
IT communication

Iterators: Generalization of Pointers that Iterate

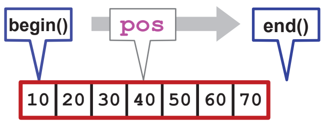
C++98/C++11

- **Iterate like a pointer over elements**
  - From `begin()` til `end()`

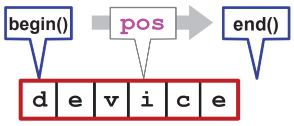
```
int arr[] = {10, 20, 30, 40};

for (int* p = arr; p < arr+4; ++p) {
  std::cout << *p << '\n';
}
```



```
std::vector<int> v{10, 20, 30, 40, 50, 60, 70};

for (std::vector<int>::iterator pos = v.begin(); pos < v.end(); ++pos) {
  std::cout << *pos << '\n';
}
```



```
std::string s{"device"};

for (std::string::iterator pos = s.begin(); pos < s.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

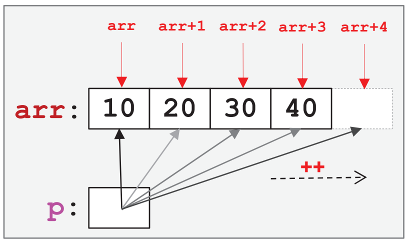


Iterators: Generalization of Pointers that Iterate

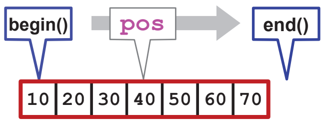
C++11

- **Iterate like a pointer over elements**
  - From `begin()` til `end()`

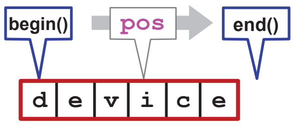
```
int arr[] = {10, 20, 30, 40};

for (int* p = arr; p < arr+4; ++p) {
  std::cout << *p << '\n';
}
```



```
std::vector<int> v{10, 20, 30, 40, 50, 60, 70};

for (auto pos = v.begin(); pos < v.end(); ++pos) {
  std::cout << *pos << '\n';
}
```



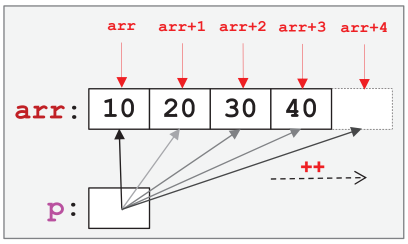
```
std::string s{"device"};

for (auto pos = s.begin(); pos < s.end(); ++pos) {
  std::cout << *pos << '\n';
}
```



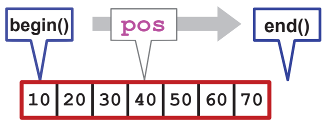
Half-Open Range

C++11

- **Iterate like a pointer over a collection**
  - `begin()` and `end()` as **half-open range**



```
int arr[] = {10, 20, 30, 40};

for (int* p = arr; p < arr+4; ++p) {
  std::cout << *p << '\n';
}
```



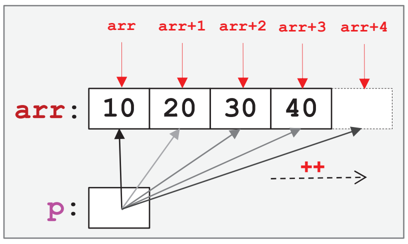
```
std::vector<int> v{10, 20, 30, 40, 50, 60, 70};

for (auto pos = v.begin(); pos < v.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

```
auto pos = v.end();    // OK
std::cout << *pos;     // Undefined Behavior
```

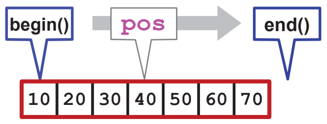
Half-Open Range

C++11

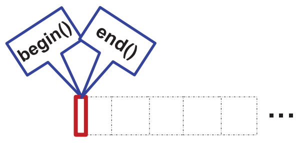
- **Iterate like a pointer over a collection**
  - `begin()` and `end()` as **half-open range**



```
int arr[] = {10, 20, 30, 40};

for (int* p = arr; p < arr+4; ++p) {
  std::cout << *p << '\n';
}
```



```
std::vector<int> v{10, 20, 30, 40, 50, 60, 70};

for (auto pos = v.begin(); pos < v.end(); ++pos) {
  std::cout << *pos << '\n';
}

std::vector<int> v2;   // empty

for (auto pos = v2.begin(); pos < v2.end(); ++pos) {
  std::cout << *pos << '\n';
}
```



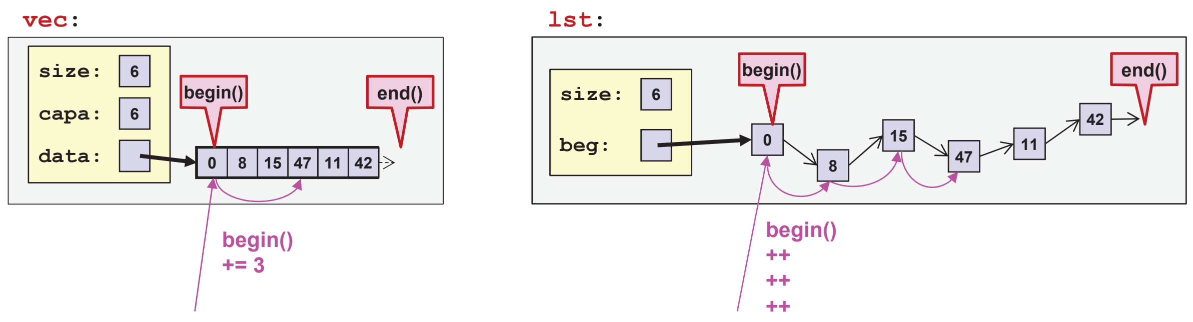
`begin() == end()` means empty

Modern C++

Why Iterators?

Index Operator vs. Iterator

C++20



```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::cout << vec[3];     // cheap
```

```
std::list<int> lst{0, 8, 15, 47, 11, 42};
std::cout << ~~lst[3];~~     // expensive
```

```
for (int i = 0; i < lst.size(); ++i) {
  … lst[i] …
}
```

calls:

`lst[0], lst[1], lst[2], lst[3], …`

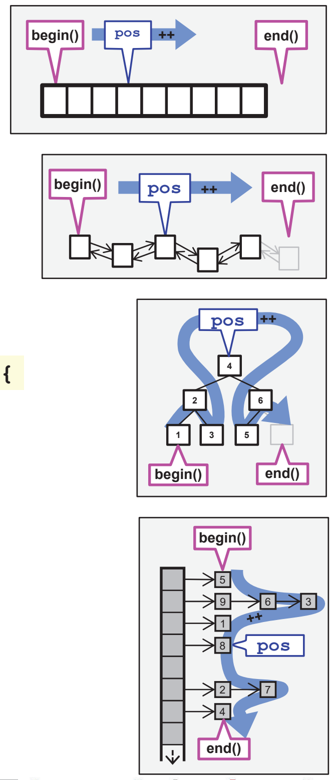
Iterating with Generic Code

C++11

Iterator API provided by all containers:

– `begin()` and `end()` yield *iterators*
  - to iterator over elements with `++`, `!=`, `*`, ...

```
// print all elements:
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
  std::cout << *pos << '\n';
}
```



C++

11

josuttis | eckstein
IT communication

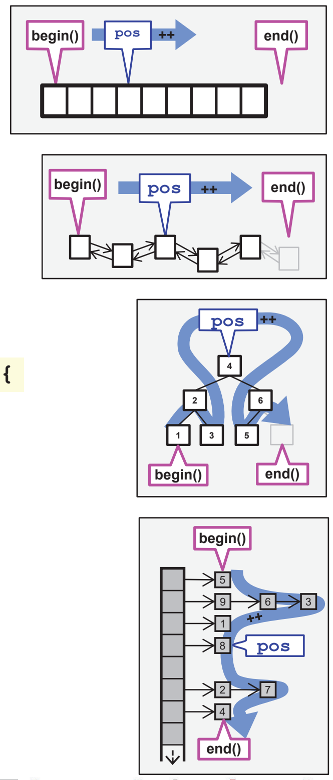
Iterating with Generic Code

C++11

Iterator API provided by all containers:

– `begin()` and `end()` yield *iterators*
  - to iterator over elements with `++`, `!=`, `*`, ...

```
template<typename T>
void printElems(const T& coll)
{
  for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << '\n';
  }
}
```



C++

12

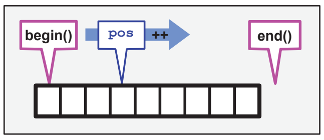
josuttis | eckstein
IT communication

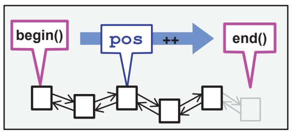
Iterating with Generic Code

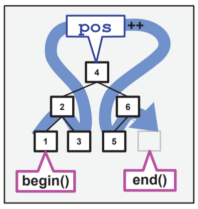
C++11

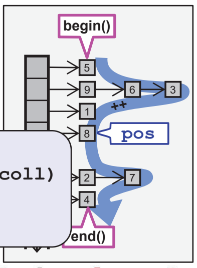
Iterator API provided by all containers:

– **`begin()`** and **`end()`** yield *iterators*
  • to iterator over elements with **`++`**, **`!=`**, **`*`**, ...









```
template<typename T>
void printElems(const T& coll)
{
  for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << '\n';
  }
}
```

```
std::vector<int> coll1;
...
printElems(coll1);
```

compiles:
```
void printElems(const std::vector<int>& coll)
{
  ...  // ++ just moves n bytes to the next element
}
```

```
std::set<std::string> coll2;
...
printElems(coll2);
```

compiles:
```
void printElems(const std::set<std::string>& coll)
{
  ...  // ++ navigates to the next element
}
```

C++

josuttis | eckstein
IT communication

Modern C++

How to Use Iterators?

C++

josuttis | eckstein
IT communication

Iterating with Generic Code

C++11

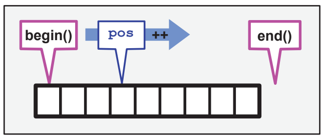
Iterator API provided by all containers:

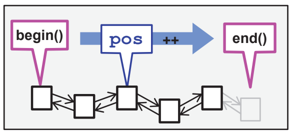
- **begin()** and **end()** yield *iterators*
  - to iterator over elements with **++**, **!=**, *****, ...
- Used by the range-based **for** loop

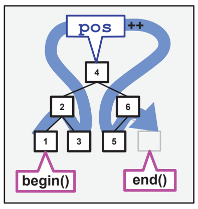
```
template<typename T>
void printElems(const T& coll)
{
  for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << '\n';
  }
}
```

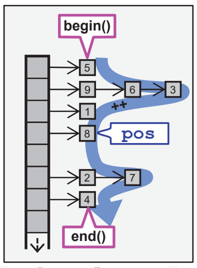
C++11

```
template<typename T>
void printElems(const T& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```









C++

josuttis | eckstein
IT communication

Iterating with Generic Code

C++20

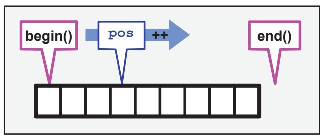
Iterator API provided by all containers:

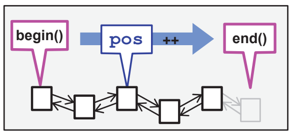
- **begin()** and **end()** yield *iterators*
  - to iterator over elements with **++**, **!=**, *****, ...
- Used by the range-based **for** loop

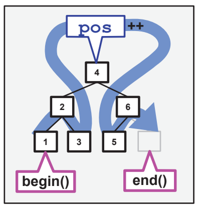
```
template<typename T>
void printElems(const T& coll)
{
  for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << '\n';
  }
}
```

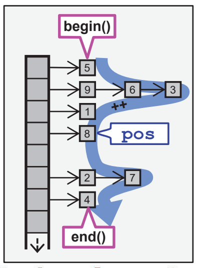
C++20

```
void printElems(const auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```









C++

josuttis | eckstein
IT communication

auto and cbegin() / cend()

C++11

- **To support auto for read-only iterations, we have:**
  - Type const_iterator
  - cbegin() and cend()
    crbegin() and crend()

The element the iterator refers to is const

```
template<typename T>
void processElements(T& coll)
{
  …
  for (typename T::const_iterator pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << '\n';
  }
}
```

*pos = 0;
does not compile

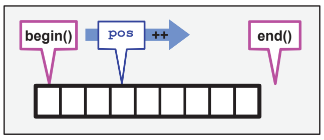
```
template<typename T>
void processElements(T& coll)
{
  …
  for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
    std::cout << *pos << '\n';
  }
}
```

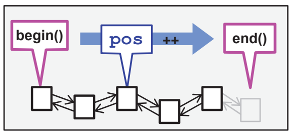
Partially broken by C++2x views

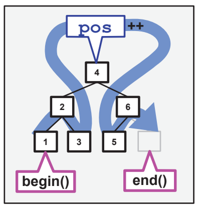
Iterators have different Categories

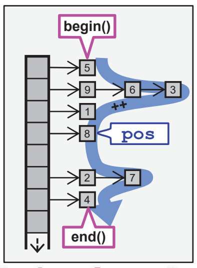
C++98

- **Random access iterators**
  - Can jump to and compare with any other position
    =, *, ++, ==, !=, --, +=, -=, <, <=, … [], -
  - **vector<>**, **array<>**, **deque<>**, **raw arrays**, **strings**
- **Bidirectional iterators**
  - Can iterate forward and backward
    =, *, ++, ==, !=, --
  - **list<>**, **associative** containers (**set<>**, **map<>**, ...)
- **Forward iterators**
  - Can iterate forward only
    =, *, ++, ==, !=
  - **forward_list<>**, **unordered** containers (hash tables)
- **Input iterators**
  - Can read elements only once
  - **istream_iterator<>**









Iterator Categories

C++11

// print all elements (for all containers):

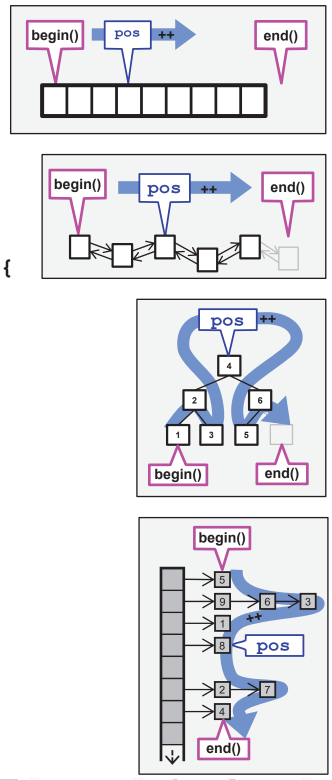
```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

// print every 2nd element (array, vector, deque only):

```
for (auto pos = coll.begin(); pos < coll.end(); pos += 2) {
  std::cout << *pos << '\n';
}
```

// print every 2nd element (for all containers):

```
for (auto pos = coll.begin(); pos != coll.end(); ) {
  std::cout << *pos << '\n';
  ++pos;
  if (pos != coll.end()) ++pos;
}
```



Iterating with Generic Code

C++11

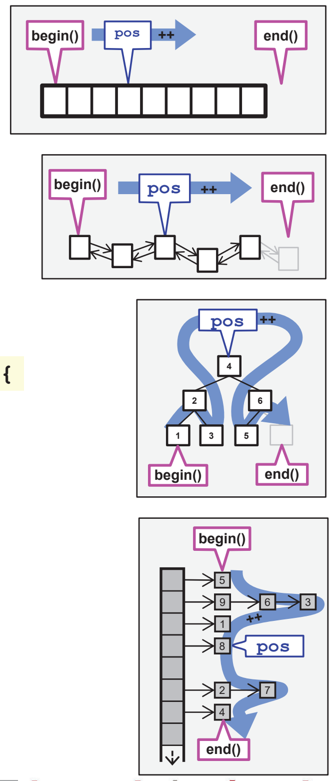
Iterator API provided by all containers:

– **begin()** and **end()** yield *iterators*
  - to iterator over elements with **++**, **!=**, *****, ...

// print all elements:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

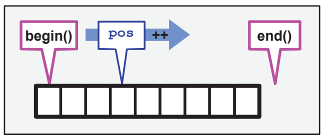
- **Prefer != over <**
  – < is not supported by all iterators (only random-access iterators)

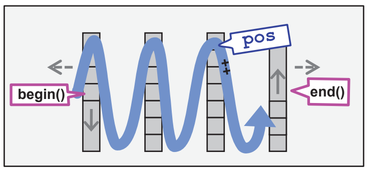


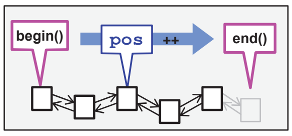
Range/Iterator Categories/Concepts since C++20

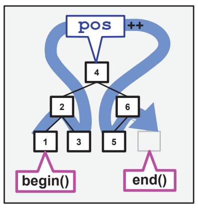
C++20

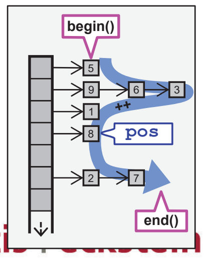
- **Contiguous range/iterator**
  - Can jump to and compare with any other position
    `=, *, ++, ==, !=, --, +=, -=, <, <=, … [], -`
  - Iterator may be raw pointer, range has `std::ranges::data()`
  - **vector<>**, **array<>**, **raw arrays**, **strings**
- **Random access range/iterator**
  - Can jump to and compare with any other position
    `=, *, ++, ==, !=, --, +=, -=, <, <=, … [], -`
  - **deque<>**
- **Bidirectional range/iterator**
  - Can iterate forward and backward
    `=, *, ++, ==, !=, --`
  - **list<>**, **associative** containers (**set<>**, **map<>**, ...)
- **Forward range/iterator**
  - Can iterate forward multiple times
    `=, *, ++, ==, !=`
  - **forward_list<>**, **unordered** containers (hash tables)











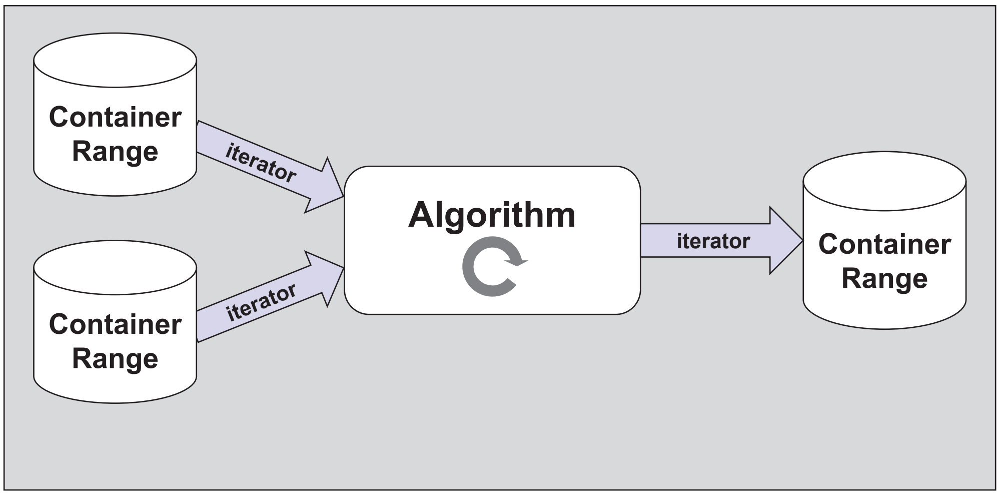
Modern C++

Iterators and Algorithms

Standard Template Library (STL)

C++98

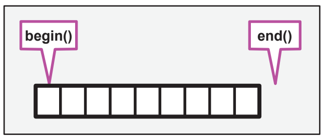
- **Data structures** as ranges
- **Algorithms**
- **Iterators** as glue interface



STL Algorithms

C++98 / C++11

Standard algorithms



- Process elements of half-open ranges
- Using the iterator interface for element access
- Generic to operate on different container types

```
#include <algorithm>    // for algorithms
#include <numeric>      // for numeric algorithms like accumulate()
...

// sort elements:
std::sort(coll.begin(), coll.end());

// find the element with the highest value:
auto maxPos = std::max_element(coll.begin(),coll.end());
if (maxPos != coll.end()) {
  std::cout << "max: " << *maxPos;
}

// process the sum of all elements:
auto sum = std::accumulate(coll.begin(), coll.end(),
                           0);    // initial value (and return type) for the sum
```

`end()` signals: none found (here: range was empty)

Implementation of STL Algorithms

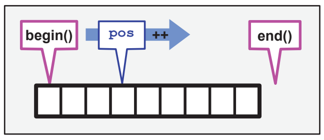
C++98 / C++11

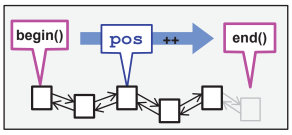
Algorithms are generic:

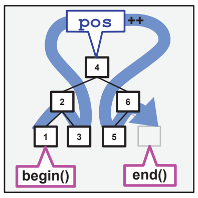
- Using the iterator interface of all containers
- Provided all operations are supported

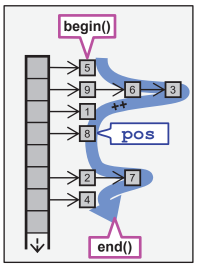
```
template <typename IterT, typename ValueT>
ValueT accumulate (IterT beg, IterT end,  // range
                   ValueT val)            // initial value
{
  for (IterT pos = beg; pos != end; ++pos) {
    val = val + *pos;     // add value of each element
  }
  return val;
}
```

```
std::vector<long> coll{1, 2, 3, 4, 5};

auto res1 = accumulate(coll.begin(), coll.end(),      // range
                       0L);                           // initial value
std::cout << res1 << '\n';     // prints: 15

std::set<std::string> words{"one", "two", "three", "four"};
auto res2 = accumulate(words.begin(), words.end(),    // range
                       std::string{});                // initial value
std::cout << res2 << '\n';     // prints: fouronethreetwo
```









C++

josuttis | eckstein IT communication

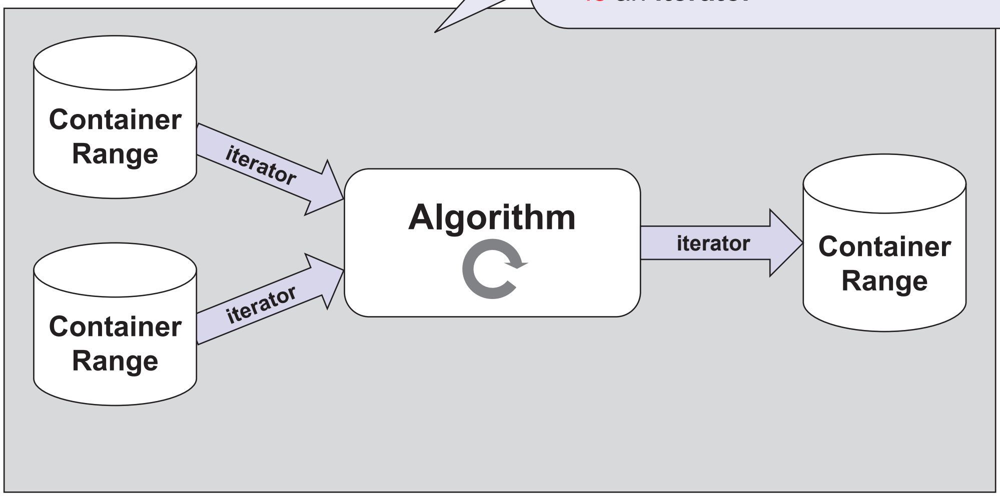
Standard Template Library (STL)

C++98

- **Data structures as ranges**
- **Algorithms**
- **Iterators as glue interface**

Pure abstractions:

- Everything that *behaves* like a **container**, *is* a **container**
- Everything that *behaves* like an **iterator**, *is* an **iterator**



C++

josuttis | eckstein IT communication

STL Algorithms for Pointers

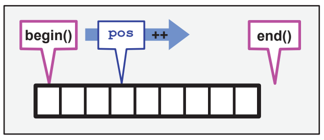
C++98

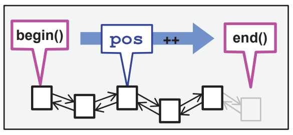
Algorithms are generic:

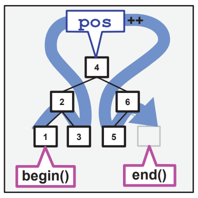
– Using the iterator interface of all containers
– Provided all operations are supported

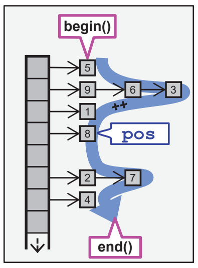
```
template <typename IterT, typename ValueT>
ValueT accumulate (IterT beg, IterT end, // range
                   ValueT val)           // initial value
{
  for (IterT pos = beg; pos != end; ++pos) {
    val = val + *pos;    // add value of each element
  }
  return val;
}
```

```
double arr[] = {1.1, 2.2, 3.3, 4.4, 5.5};

double res = accumulate(arr, arr + 5,    // range with pointers
                        0.0);            // initial value
std::cout << res << '\n';      // prints: 16.5
```







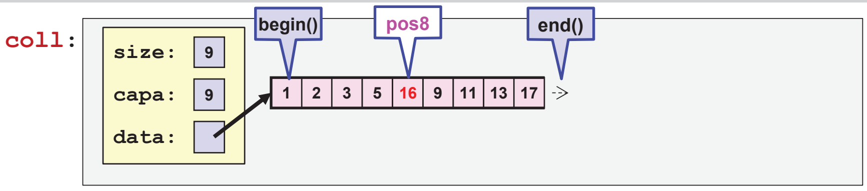


Modern C++

Pitfalls of Iterators

Using Vector Iterators

C++11



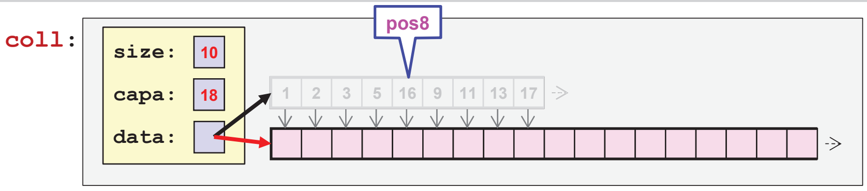
```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> coll{1, 2, 3, 5, 8, 9, 11, 13, 17};

  for (int elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';

  auto pos8 = std::find(coll.begin(), coll.end(), 8);
  if (pos8 != coll.end()) {
    std::cout << "8 found\n";
    *pos8 *= 2;
  }

}
```

Output:

```
1 2 3 5 8 9 11 13 17
8 found
```

Using Vector Iterators

C++11



```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> coll{1, 2, 3, 5, 8, 9, 11, 13, 17};

  for (int elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';

  auto pos8 = std::find(coll.begin(), coll.end(), 8);
  if (pos8 != coll.end()) {
    std::cout << "8 found\n";
    *pos8 *= 2;
  }

  coll.push_back(15);      // append one element

  if (pos8 != coll.end()) {
    *pos8 *= 2;                  // fatal runtime ERROR: undefined behavior
  }
}
```

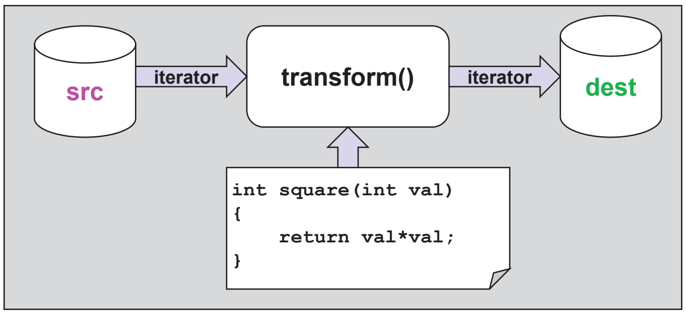
Output:

```
1 2 3 5 8 9 11 13 17
8 found
```

`std::transform()` with 4 Arguments

C++98

```
#include <algorithm>
#include <list>
#include <vector>

int square(int val)
{
  return val * val;
}
```



```
int square(int val)
{
    return val*val;
}
```

```
void foo()
{
  std::list<int> src;
  std::vector<int> dest;
  ...
  // transform all elements of the source range to the square in the destination range
  // - overwrites; does not insert
  // - precondition: dest.size() >= src.size()
  std::transform(src.begin(), src.end(),     // source range
                 dest.begin(),               // begin of destination range
                 square);                    // transformation
  ...
}
```

C++

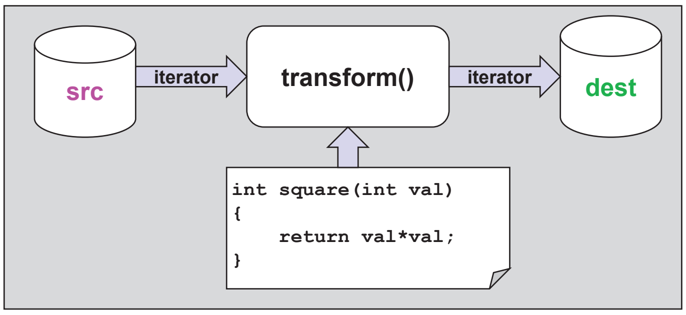
31

josuttis | eckstein IT communication

`std::transform()` with 4 Arguments

C++98/C++11

```
#include <algorithm>
#include <list>
#include <vector>

int square(int val)
{
  return val * val;
}
```



```
int square(int val)
{
    return val*val;
}
```

```
void foo()
{
  std::list<int> src{1, 2, 3, 4, 5, 6};     // some elements
  std::vector<int> dest;                    // empty !

  // transform all elements of the source range to the square in the destination range
  std::transform(src.begin(), src.end(),     // source range
                 dest.begin(),               // begin of destination range
                 square);                    // transformation
  ...
}
```

Fatal runtime error (undefined behavior)

C++

32

josuttis | eckstein IT communication

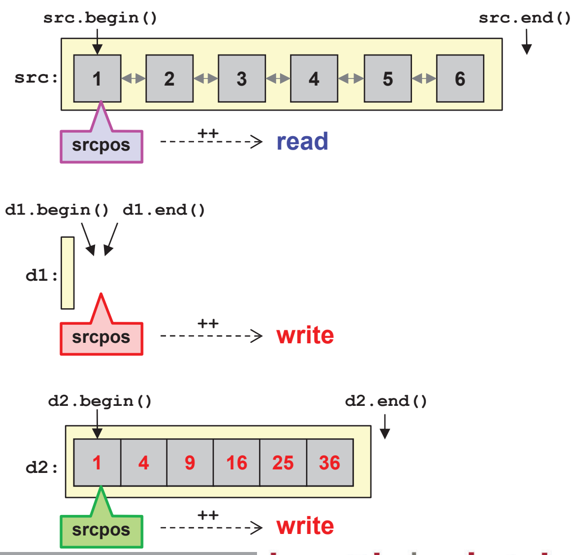
Writing Algorithms

C++98/C++11

- **Output iterators overwrite**
  - Refer to a location for an element and don't know where ranges end
  - Similar to raw pointers

  => There must be elements to overwrite

```
std::list<int> src{1, 2, 3, 4, 5, 6};

std::vector<int> d1;
std::transform(src.begin(), src.end(),
               d1.begin(),     // ERROR
               square);


std::vector<int> d2;
d2.resize(src.size()); // set size big enough
std::transform(src.begin(), src.end(),
               d2.begin(),     // OK
               square);
```



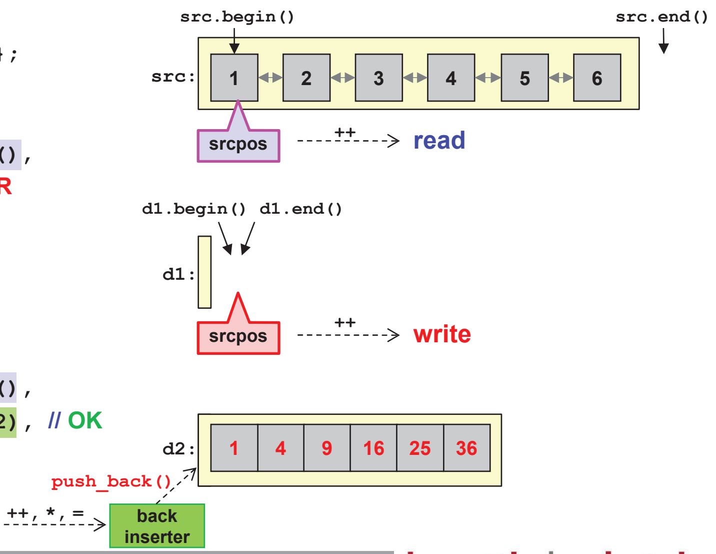
Using Inserters

C++98/C++11

- **Inserters**
  - Iterators that know the objects they iterate over
  - Insert instead of overwrite

```
std::list<int> src{1, 2, 3, 4, 5, 6};

std::vector<int> d1;
std::transform(src.begin(), src.end(),
               d1.begin(),     // ERROR
               square);


std::vector<int> d2;

std::transform(src.begin(), src.end(),
               std::back_inserter(d2),  // OK
               square);
```

helper function (needs parentheses)



Output of the Following Program?

C++98/C++11

```
int main()
{
   std::list<int> coll{
     6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6
   };

   for (int elem : coll) {                   // print all elements
     std::cout << elem << ' ';
   }
   std::cout << '\n';

   // remove all elements with value 3
   std::remove(coll.begin(), coll.end(),     // range
               3);                           // value

   for (int elem : coll) {                   // print all elements again
     std::cout << elem << ' ';
   }
   std::cout << '\n';
}
```

Output:

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

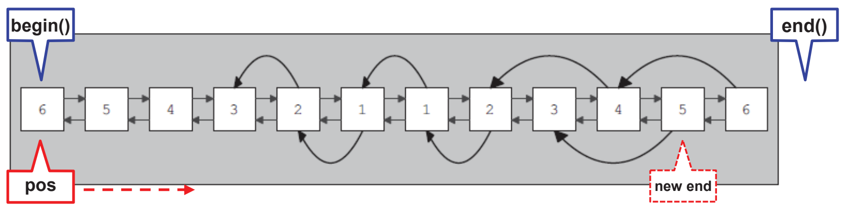
Removing Algorithms

C++98

Output:

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

- **Content after `std::remove(...,3)`:**
- **Removing algorithms do not remove**
  - Instead, they **replace removed elements** and return the new end
  - Reason:
    - Iterators operate on elements, not on containers
      - Can only read, write, and go to another value



"Removing" Algorithms

C++98/C++11

```
int main()
{
   std::list<int> coll{
     6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6
   };

   for (int elem : coll) {                  // print all elements
     std::cout << elem << ' ';
   }
   std::cout << '\n';

  // remove all elements with value 3
  auto newEnd = std::remove(coll.begin(), coll.end(),     // range
                            3);                           // value

   for (auto pos = coll.begin(); pos != newEnd; ++pos) {  // print elems up to new end
     std::cout << *pos << ' ';
   }
   std::cout << '\n';
}
```

Output:

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

"Removing" Algorithms and Views

C++20

```
int main()
{
   std::list<int> coll{
     6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6
   };

   for (int elem : coll) {                  // print all elements
     std::cout << elem << ' ';
   }
   std::cout << '\n';

  // remove all elements with value 3
  auto newEnd = std::remove(coll.begin(), coll.end(),     // range
                            3);                           // value

   for (int elem : std::ranges::subrange{coll.begin(), newEnd}) {  // elems til new end
     std::cout << elem << ' ';
   }
   std::cout << '\n';
}
```

Output:

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

"Removing" Algorithms and Views

C++20

```
int main()
{
   std::list<int> coll{
     6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6
   };

   for (int elem : coll) {                    // print all elements
     std::cout << elem << ' ';
   }
   std::cout << '\n';

   // print all elements not having value 3:
   auto not3 = [] (const auto& elem) {
                  return elem != 3;
               };
   for (int elem : coll | std::views::filter(not3)) {
     std::cout << elem << ' ';
   }
   std::cout << '\n';
}
```

Output:

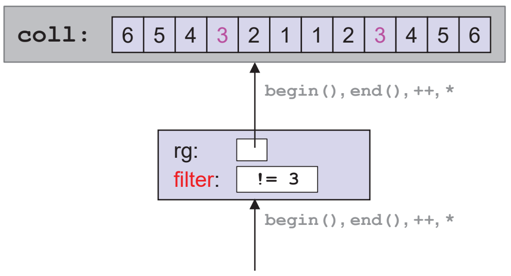
```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
```

C++

josuttis | eckstein
IT communication

"Removing" Algorithms and Views

C++20



```
   std::list<int> coll{6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6};

   // print all elements not having value 3:
   auto not3 = [] (const auto& elem) {
                  return elem != 3;
               };
   for (int elem : coll | std::views::filter(not3)) {
     std::cout << elem << ' ';
   }
   std::cout << '\n';
```

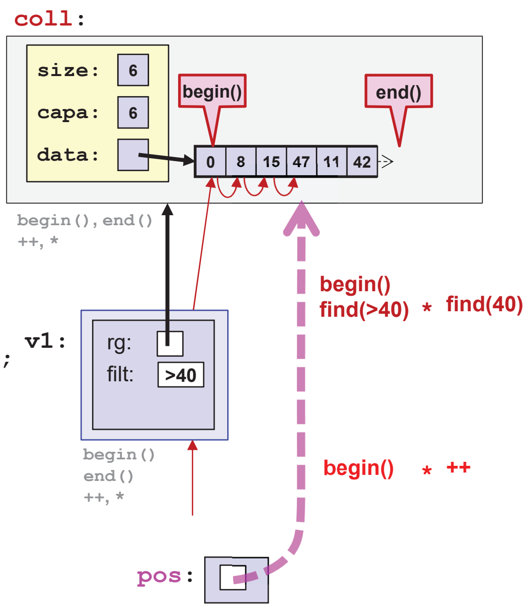
C++

josuttis | eckstein
IT communication

Filter Views Intern

C++20

```
std::vector<int> coll{0, 8, 15, 47, 11, 42};

auto greater40 = [] (const auto& elem) {
                    return elem > 40;
                 };
auto v1 = coll | std::views::filter(greater40);

auto pos = v1.begin();
std::cout << *pos;
++pos;
std::cout << *pos;
…
while (pos != v1.end())
```



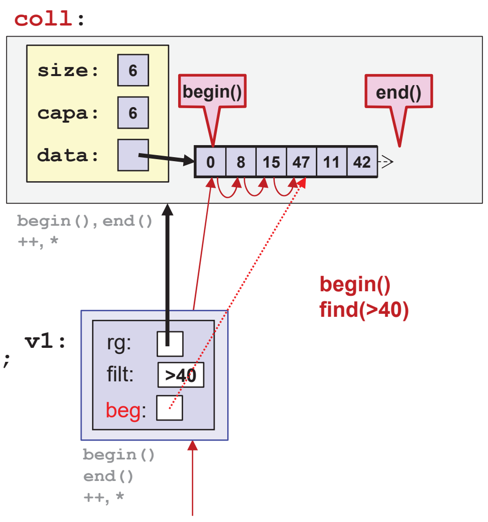
C++

josuttis | eckstein
IT communication

Filter Views Cache begin()

C++20

```
std::vector<int> coll{0, 8, 15, 47, 11, 42};

auto greater40 = [] (const auto& elem) {
                    return elem > 40;
                 };
auto v1 = coll | std::views::filter(greater40);

auto pos = v1.begin();
```



C++

josuttis | eckstein
IT communication

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42, 1};
std::list<int> lst{0, 8, 15, 47, 11, 42, 1};

print(vec);
print(lst);

auto gt40 = [] (const auto& elem) {return elem > 40};
for (const auto& elem : vec | std::views::filter(gt40)) {    // OK
  std::cout << elem << ' ';
}
print(vec | std::views::filter(gt40));        // ERROR

print(vec | std::views::drop(3));            // OK
print(lst | std::views::drop(3));            // ERROR
```

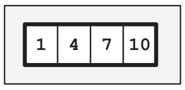
Output:

```
0 8 15 47 11 42 1
0 8 15 47 11 42 1

47 42
ERROR
47 11 42 1
ERROR
```

Using the Filter View

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);
```



```
auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

// add 2 to even elements:
for (int& i : collEven) {
  i += 2;
}
print(coll);

// add 2 to even elements:
for (int& i : collEven) {
  i += 2;
}
print(coll);
```

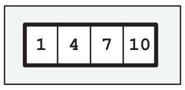
Output:

```
1 4 7 10

1 6 7 12

1 8 7 14
```

Using the Filter View

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);
```



```
auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);
```

// increment even elements:

```
for (int& i : collEven) {
  i += 1;          // Runtime Error: UB: predicate broken
}
print(coll);
```

// increment even elements:

```
for (int& i : collEven) {
  i += 1;          // Runtime Error: UB: predicate broken
}
print(coll);
```

Output:

```
1 4 7 10
1 5 7 11
1 6 7 11
```

Using the Filter View

C++20

- **Main use case of a filter:**
  - Fix an attribute that some elements might have

has undefined behavior:

[range.filter.iterator]:
Modification of the element a filter_view::*iterator* denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

// as a shaman:

```
for (auto& m : monsters | std::views::filter(isDead)) {
  m.resurrect();   // undefined behavior: because no longer dead
  m.burn();        // OK (because it is still dead)
}
```

Thanks to Patrice Roy for this example

Summary

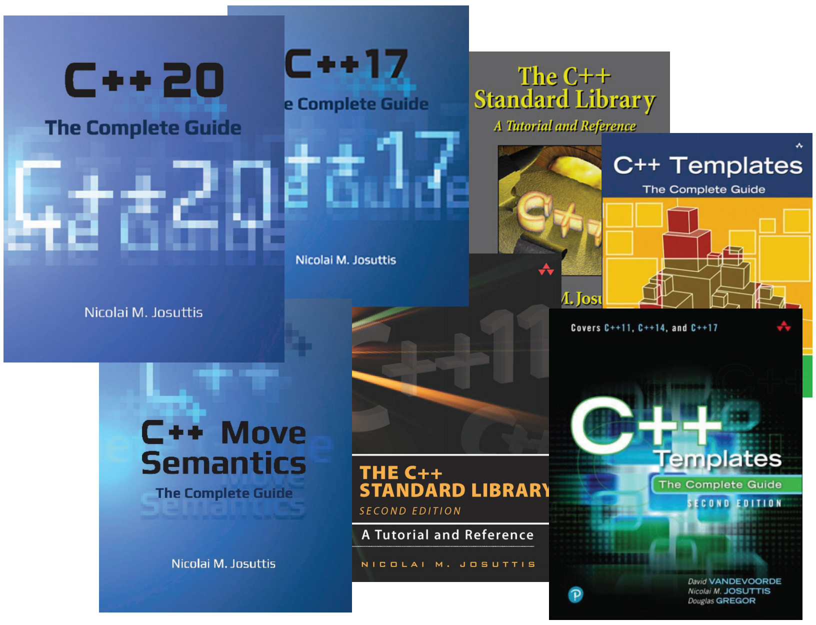
Iterators

- **Key role for C++**
  - Glue between ranges and algorithms
- **Pure abstraction**
  - Everything that behaves like an iterator is an iterator
- **Different categories with different abilities**
- **Do not know their ranges (in general)**
  - Don't know where the end is
  - Cannot insert/remove
- **Use iterators with care**
  - Referenced range has to be valid
  - Don't compare iterators not referring to the same range
- **Iterators of C++2x filter views cache begin**
  - Results in break idioms and unexpected behavior

Thank You!

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis


C++20 The Complete Guide Nicolai M. Josuttis
C++17 The Complete Guide Nicolai M. Josuttis
The C++ Standard Library A Tutorial and Reference
C++ Templates The Complete Guide
C++ Move Semantics The Complete Guide Nicolai M. Josuttis
THE C++ STANDARD LIBRARY SECOND EDITION A Tutorial and Reference NICOLAI M. JOSUTTIS
Covers C++11, C++14, and C++17 C++ Templates The Complete Guide SECOND EDITION David VANDEVOORDE Nicolai M. JOSUTTIS Douglas GREGOR