A quick overview of the Standard Template Library

Advanced Programming and Algorithmic Design

Alberto Sartori

November 30, 2017

Outline

Standard Template Library

What is an Iterator?

Design pattern

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

A generalization of a pointer

- indirect access (`operator*()`, `operator->()`)
- operations for moving to point to a new element (`operator++()`, `operator--()`)

Iterators in the STL

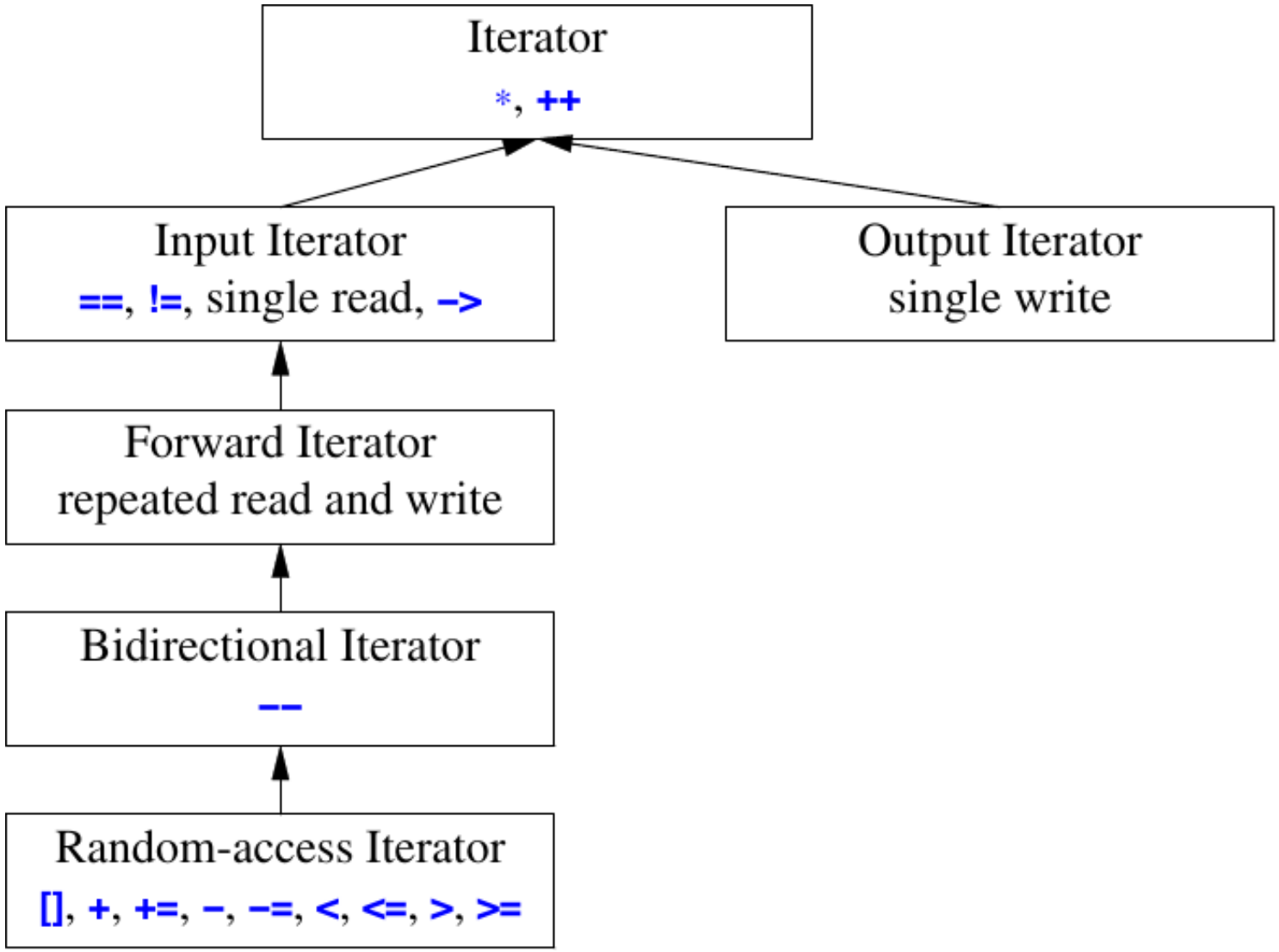
Their role

- Iterators are the glue that ties the standard-library alogorithms to their data
- Iterators are the mechanism used to minimize an algorithm's dependence on the data structures on which it operates.

Alex Stepanov

The reason that STL containers and algorithms work so well together is that they know nothing of each other.

Iterator categories



Does our iterator work?

```
template <typename T>
class List<T>::Iterator {
   ...
};
```

Does our iterator work?

```
#include <iterator>

...

template <typename T>
class List<T>::Iterator : public
   std::iterator<std::forward_iterator_tag, T> {
   ...
};
```

```
template <typename Cat,
          typename T,
          typename Dist = ptrdiff_t,
          typename Ptr = T*,
          typename Ref = T&>
struct iterator{
  using value_type = T;
  using difference_type = Dist;
  using pointer = Ptr;
  using reference = Ref;
  using iterator_category = Cat;
};
```

Containers

Definition

A container holds a sequence of objects

Two categories

- Sequence containers: provide access to sequences of elements
- Associative containers: provide associative lookup based on a key

Associative containers

- Ordered
- Unordered

Sequence containers

Sequence Containers	
vector<T,A>	A contiguously allocated sequence of Ts; the default choice of container
list<T,A>	A doubly-linked list of T; use when you need to insert and delete elements without moving existing elements
forward_list<T,A>	A singly-linked list of T; ideal for empty and very short sequences
deque<T,A>	A double-ended queue of T; a cross between a vector and a list; slower than one or the other for most uses

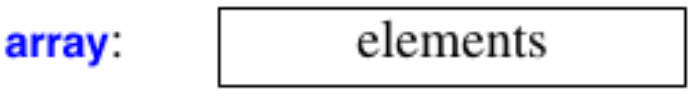
Ordered associative containers

Ordered Associative Containers (§iso.23.4.2) **C** is the type of the comparison; **A** is the allocator type	
map<K,V,C,A>	An ordered map from **K** to **V**; a sequence of (**K**,**V**) pairs
multimap<K,V,C,A>	An ordered map from **K** to **V**; duplicate keys allowed
set<K,C,A>	An ordered set of **K**
multiset<K,C,A>	An ordered set of **K**; duplicate keys allowed

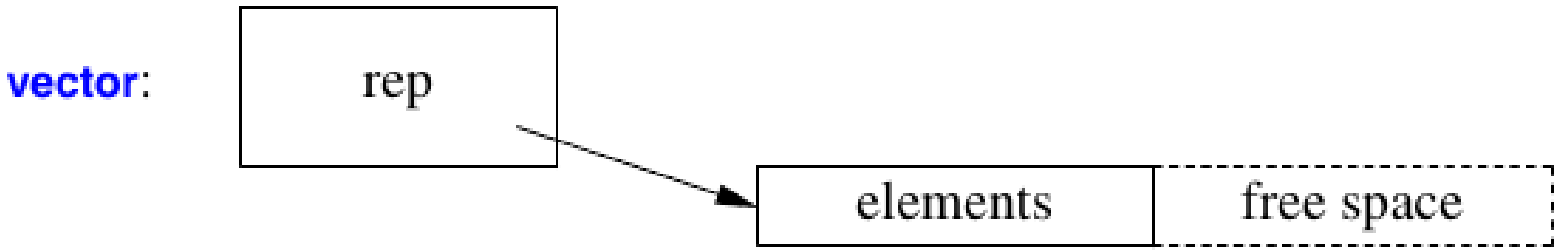
Unordered associative containers

Unordered Associative Containers (§iso.23.5.2) **H** is the hash function type; **E** is the equality test; **A** is the allocator type	
unordered_map<K,V,H,E,A>	An unordered map from **K** to **V**
unordered_multimap<K,V,H,E,A>	An unordered map from **K** to **V**; duplicate keys allowed
unordered_set<K,H,E,A>	An unordered set of **K**
unordered_multiset<K,H,E,A>	An unordered set of **K**; duplicate keys allowed

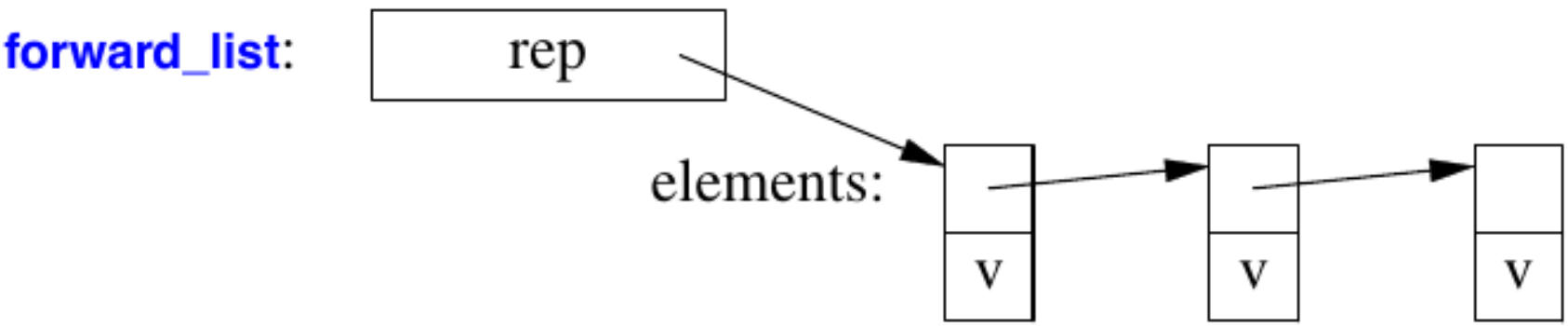
Array



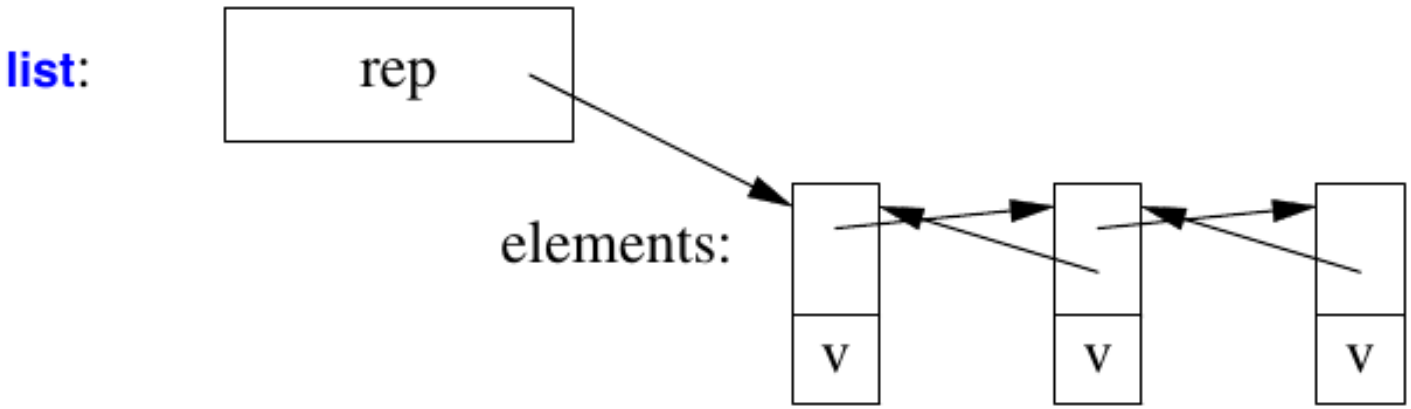
Vector



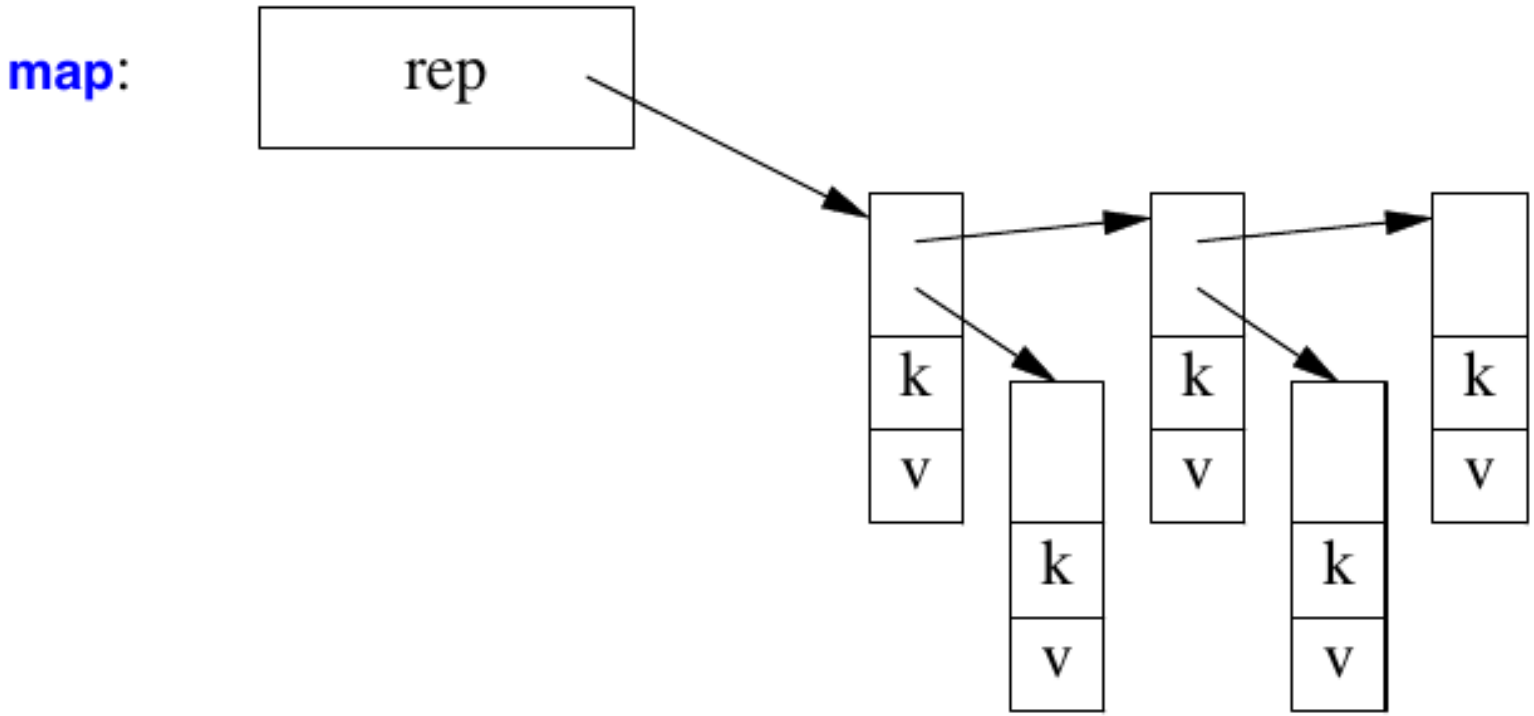
Forward list



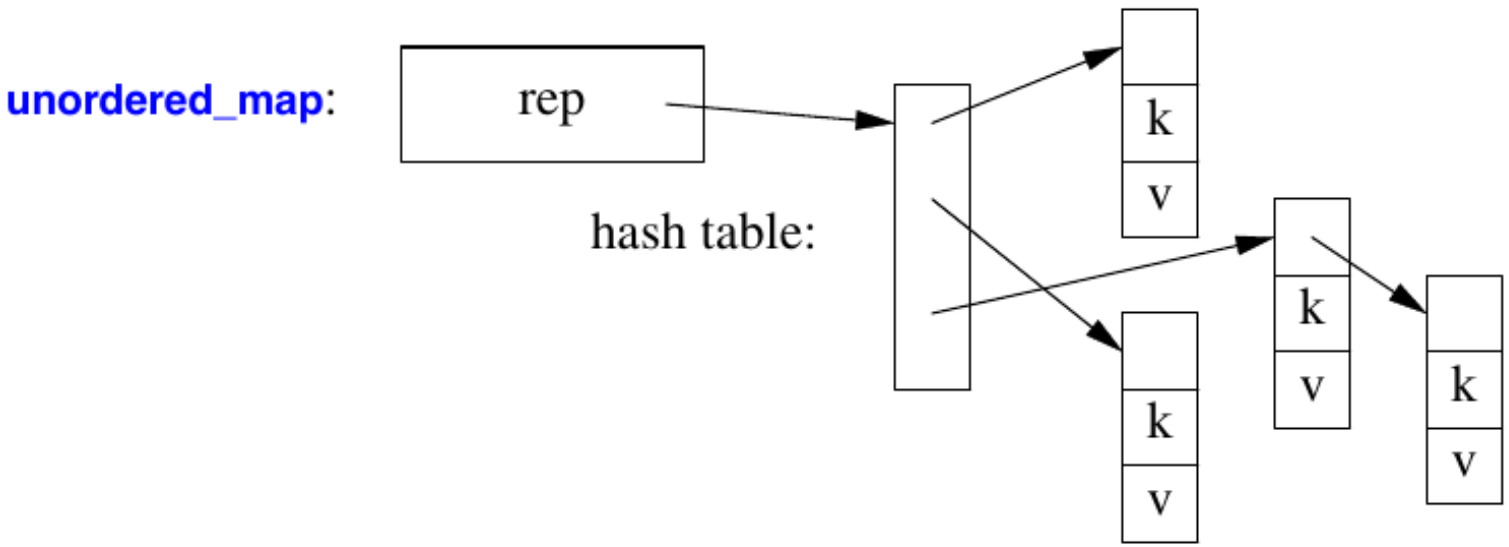
List



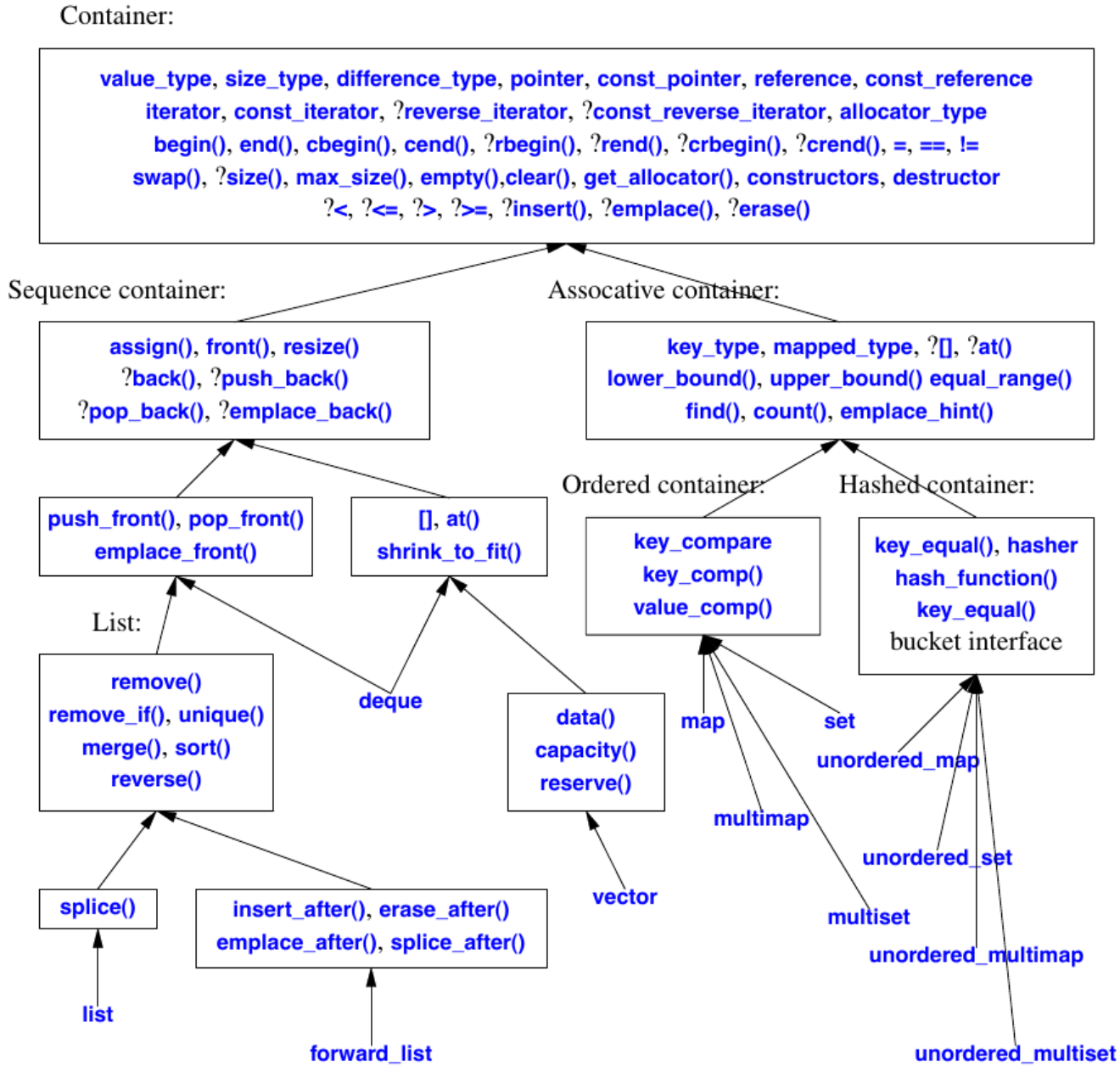
Map



Unordered map



Operations and types



Operation complexity

Standard Container Operation Complexity					
	[] §31.2.2	List §31.3.7	Front §31.4.2	Back §31.3.6	Iterators §33.1.2
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
built-in array	const				Ran
valarray	const				Ran
bitset	const				

Prime numbers

```
#include <vector>

int main(){
  std::vector<int> primes;

  primes.emplace_back(2);

  for (int i=3; i<=max; ++i)
    if (is_prime(i))
      primes.emplace_back(i);

  for (const auto& x: primes)
    std::cout << x << std::endl;
}
```

Word count

```
#include <map>

int main(){
  std::map<std::string, int> words;

  for (std::string s; std::cin>>s;)
    ++words[s];

  for (const auto& x: words)
  std::cout << x.first << ": "
            << x.second << std::endl;
}
```

Word count

```
#include <map>

int main(){
  std::unordered_map<std::string, int> words;

  for (std::string s; std::cin>>s;)
    ++words[s];

  for (const auto& x: words)
  std::cout << x.first << ": "
            << x.second << std::endl;
}
```

STL algorithms

- about 80 algorithms in `<algorithm> and <numeric>`
- operate on *sequences*
  - pair of iterators for inputs $[b : e)$
  - single iterator for output $[b2 : b2 + (e - b))$
- can take functions of function objects
- report failure by returning the end of the sequence

Examples

Sequences

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

int main(){
   std::vector<double> v1;
   ...
   std::vector<double> v2(v1.size());
   std::sort(v1.begin(), v1.end());
   std::copy(v1.begin(), v1.end(), v2.begin());
}
```

Examples

Sequences

```
#include <numeric>
#include <vector>

int main(){
   std::vector<double> v1;
   ...
   double sum{0};
   sum = std::accumulate(v1.begin(),v1.end(),sum);
}
```

Examples

Predicates

```
#include <numeric>
#include <vector>

double my_f(const double& a, const double& b){
 if(b == 2.2)
  return a;
 return a+b;
}
int main(){
 std::vector<double> v1;
 ...
 double sum{0};
 sum = std::accumulate(first,last,sum,my_f);
}
```

Examples

Predicates

```
#include <numeric>
#include <vector>
int main(){
 std::vector<double> v1;
 ...
 auto my_f = [](const double& a, const double &b)
     -> double {
   double res = 0;
   (b==2.2 ? res = a : res= a+b);
   return res;
 };
 double sum{0};
 sum = std::accumulate(first,last,sum,my_f);
}
```

Examples

Failure check

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

int main(){
  std::vector<double> v1;
  ...
  auto it = std::find(v1.begin(), v1.end(), 2.2);

  if(it != v1.end())
    std::cout << "found " << *it << std::endl;
  else
    std::cout << "not found\n";
}
```

Function objects

- defined in `<functional>`
- comparison criteria
- predicates (functions returning `bool`)
- arithmetic operations

Predicates

Predicates (§iso.20.8.5, §iso.20.8.6, §iso.20.8.7)	
p=equal_to<T>(x,y)	p(x,y) means x==y when x and y are of type T
p=not_equal_to<T>(x,y)	p(x,y) means x!=y when x and y are of type T
p=greater<T>(x,y)	p(x,y) means x>y when x and y are of type T
p=less<T>(x,y)	p(x,y) means x<y when x and y are of type T
p=greater_equal<T>(x,y)	p(x,y) means x>=y when x and y are of type T
p=less_equal<T>(x,y)	p(x,y) means x<=y when x and y are of type T
p=logical_and<T>(x,y)	p(x,y) means x&&y when x and y are of type T
p=logical_or<T>(x,y)	p(x,y) means x||y when x and y are of type T
p=logical_not<T>(x)	p(x) means !x when x is of type T
p=bit_and<T>(x,y)	p(x,y) means x&y when x and y are of type T
p=bit_or<T>(x,y)	p(x,y) means x|y when x and y are of type T
p=bit_xor<T>(x,y)	p(x,y) means x^y when x and y are of type T

Arithmetic operations

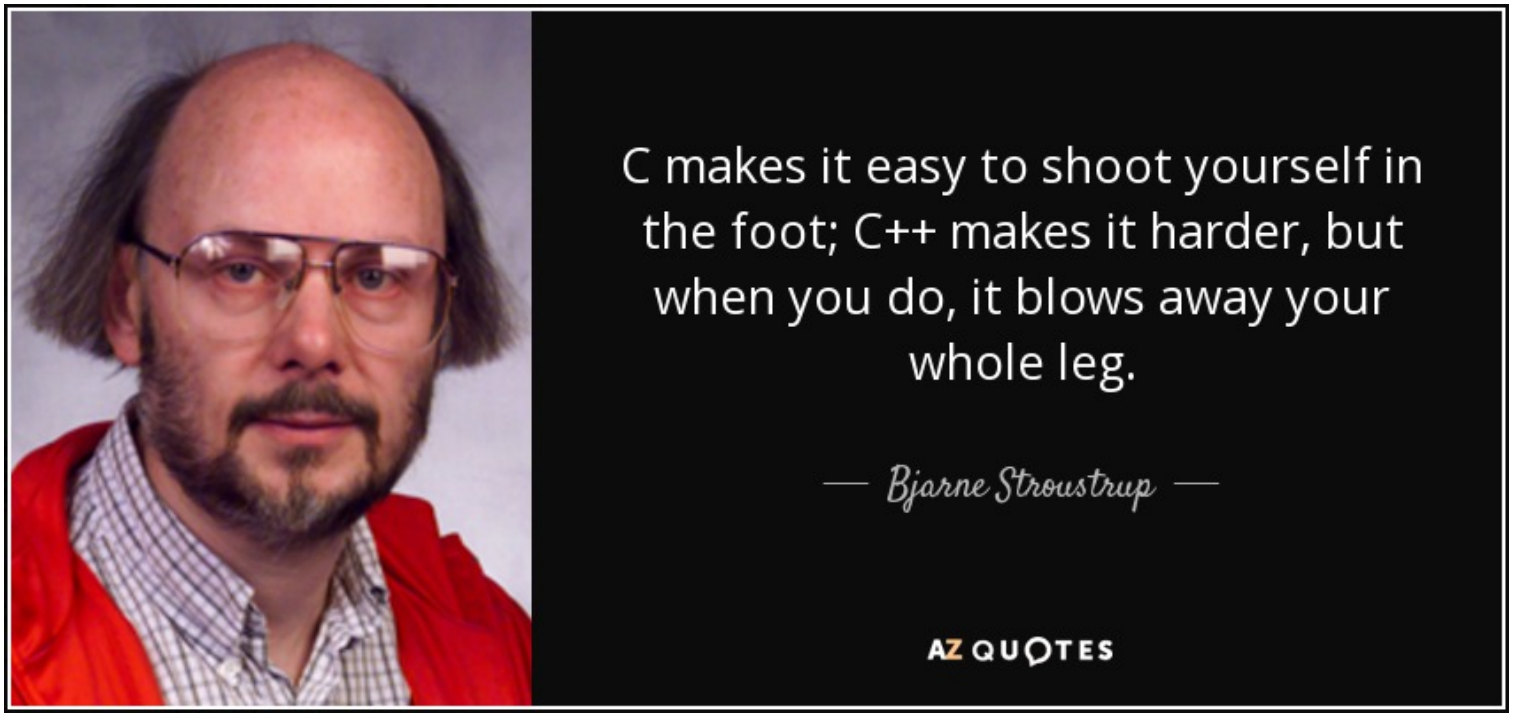
Arithmetic Operations (§iso.20.8.4)	
f=plus<T>(x,y)	f(x,y) means x+y when x and y are of type T
f=minus<T>(x,y)	f(x,y) means x–y when x and y are of type T
f=multiplies<T>(x,y)	f(x,y) means x∗y when x and y are of type T
f=divides<T>(x,y)	f(x,y) means x/y when x and y are of type T
f=modulus<T>(x,y)	f(x,y) means x%y when x and y are of type T
f=negate<T>(x)	f(x) means –x when x is of type T

Decreasing sort

```
#include <algorithm>
#include <vector>
#include <functional>

int main(){
   std::vector<double> v1;
   ...
   std::sort(v1.begin(), v1.end(),
             std::greater<double>{});
}
```

My comparison

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

template <typename num>
struct my_comparison{
  bool operator()(const num& a, const num& b) {
     return a > b;}
};

int main(){
  std::vector<double> v1;
  ...
  std::sort(v1.begin(), v1.end(),
            my_comparison<double>{});
}
```


C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.
— Bjarne Stroustrup —
AZ QUOTES