A quick overview of the C++ Standard (Template) Library

Advanced Programming

Alberto Sartori

December 03, 2019

Outline

What is the standard library?

The standard library is the set of components specified by the ISO C++ standard ($\sim$ 1600 dense pages for C++17) and shipped with identical behavior (modulo performance) by every C++ implementation.

https://github.com/cplusplus/draft

The C++ Programming Language

The header files

Containers		
<vector>	One-dimensional resizable array	§31.4.2
<deque>	Double-ended queue	§31.4.2
<forward_list>	Singly-linked list	§31.4.2
<list>	Doubly-linked list	§31.4.2
<map>	Associative array	§31.4.3
<set>	Set	§31.4.3
<unordered_map>	Hashed associative array	§31.4.3.2
<unordered_set>	Hashed set	§31.4.3.2
<queue>	Queue	§31.5.2
<stack>	Stack	§31.5.1
<array>	One-dimensional fixed-size array	§34.2.1
<bitset>	Array of bool	§34.2.2

The header files

General Utilities		
<utility>	Operators and pairs	§35.5, §34.2.4.1
<tuple>	Tuples	§34.2.4.2
<type_traits>	Type traits	§35.4.1
<typeindex>	Use a type_info as a key or a hash code	§35.5.4
<functional>	Function objects	§33.4
<memory>	Resource management pointers	§34.3
<scoped_allocator>	Scoped allocators	§34.4.4
<ratio>	Compile-time rational arithmetic	§35.3
<chrono>	Time utilities	§35.2
<ctime>	C-style date and time	§43.6
<iterator>	Iterators and iterator support	§33.1

The header files

Algorithms		
<algorithm>	General algorithms	§32.2
<cstdlib>	bsearch(), qsort()	§43.7

The header files

Diagnostics		
<exception>	Exception class	§30.4.1.1
<stdexcept>	Standard exceptions	§30.4.1.1
<cassert>	Assert macro	§30.4.2
<cerrno>	C-style error handling	§13.1.2
<system_error>	System error support	§30.4.3

The header files

Strings and Characters		
<string>	String of T	Chapter 36
<cctype>	Character classification	§36.2.1
<cwctype>	Wide-character classification	§36.2.1
<cstring>	C-style string functions	§43.4
<cwchar>	C-style wide-character string functions	§36.2.1
<cstdlib>	C-style allocation functions	§43.5
<cuchar>	C-style multibyte characters	
<regex>	Regular expression matching	Chapter 37

The header files

Input/Output		
<iosfwd>	Forward declarations of I/O facilities	§38.1
<iostream>	Standard **iostream** objects and operations	§38.1
<ios>	**iostream** bases	§38.4.4
<streambuf>	Stream buffers	§38.6
<istream>	Input stream template	§38.4.1
<ostream>	Output stream template	§38.4.2
<iomanip>	Manipulators	§38.4.5.2
<sstream>	Streams to/from strings	§38.2.2
<cctype>	Character classification functions	§36.2.1
<fstream>	Streams to/from files	§38.2.1
<cstdio>	**printf()** family of I/O	§43.3
<cwchar>	**printf()**-style I/O of wide characters	§43.3

The header files

Localization		
<locale>	Represent cultural differences	Chapter 39
<clocale>	Represent cultural differences C-style	
<codecvt>	Code conversion facets	§39.4.6

The header files

Language Support		
<limits>	Numeric limits	§40.2
<climits>	C-style numeric scalar-limit macros	§40.2
<cfloat>	C-style numeric floating-point limit macros	§40.2
<cstdint>	Standard integer type names	§43.7
<new>	Dynamic memory management	§11.2.3
<typeinfo>	Run-time type identification support	§22.5
<exception>	Exception-handling support	§30.4.1.1
<initializer_list>	initializer_list	§30.3.1
<cstddef>	C library language support	§10.3.1
<cstdarg>	Variable-length function argument lists	§12.2.4
<csetjmp>	C-style stack unwinding	
<cstdlib>	Program termination	§15.4.3
<ctime>	System clock	§43.6
<csignal>	C-style signal handling	

The header files

Numerics		
<complex>	Complex numbers and operations	§40.4
<valarray>	Numeric vectors and operations	§40.5
<numeric>	Generalized numeric operations	§40.6
<cmath>	Standard mathematical functions	§40.3
<cstdlib>	C-style random numbers	§40.7
<random>	Random number generators	§40.7

The header files

Concurrency		
<atomic>	Atomic types and operations	§41.3
<condition_variable>	Waiting for an action	§42.3.4
<future>	Asynchronous task	§42.4.4
<mutex>	Mutual exclusion classes	§42.3.1
<thread>	Threads	§42.2

The header files

C Compatibility		
<cinttypes>	Aliases for common integer types	§43.7
<cstdbool>	C **bool**	
<ccomplex>	**<complex>**	
<cfenv>	Floating-point environment	
<cstdalign>	C alignment	
<ctgmath>	C "type generic math": **<complex>** and **<cmath>**	

The header files

Library Supported Language Features		
<new>	new and delete	§11.2
<typeinfo>	typeid() and type_info	§22.5
<iterator>	Range-for	§30.3.2
<initializer_list>	initializer_list	§30.3.1

We will focus on the STL ☺

We will not see the concurrency library ☹

```
int main(){
  // f and g are independent
  f();
  g();
}
```

We will not see the concurrency library ☹

```
#include <thread>

int main(){
  // f and g are independent
  std::thread t{ f };
  g();
  t.join();
}
```

We will not see the concurrency library ☹

```
#include <future>

int main(){
  // f and g are independent
  auto from_f = std::async( f );
  auto from_g = g();
  ...
  complicated( from_g, from_f.get() );
}
```

We will not see the concurrency library ☹

Link against `pthread`

```
$ c++ test.cpp -pthread
```

```
$ c++ test.cpp -c
$ c++ test.o -pthread
```

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

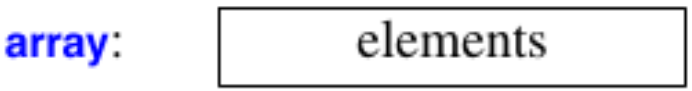
Ordered associative containers

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

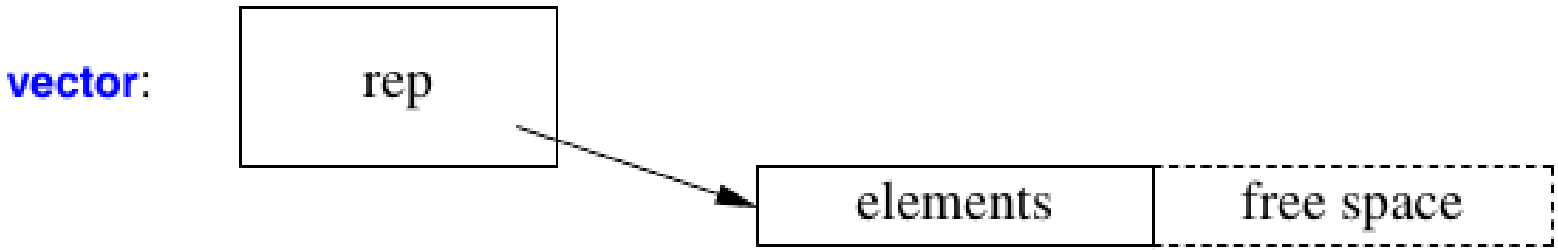
Unordered associative containers

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

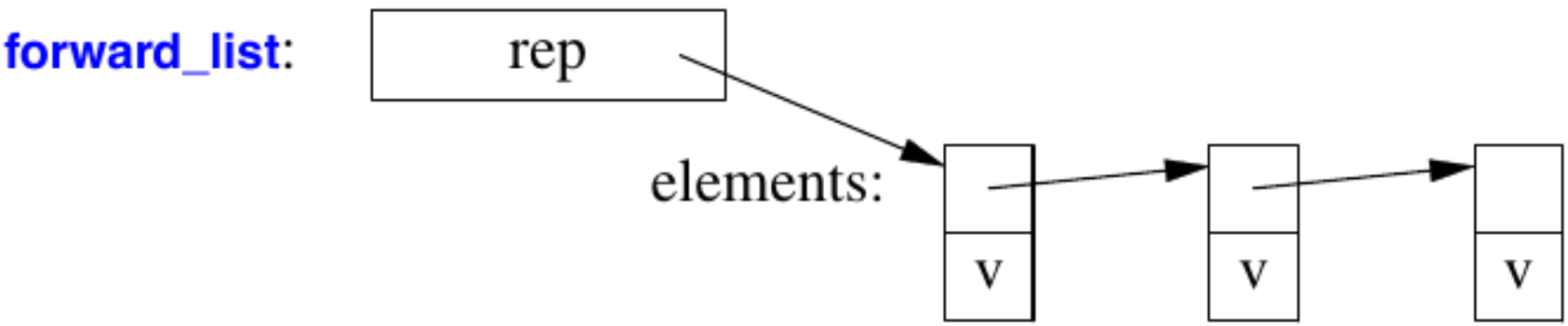
Array



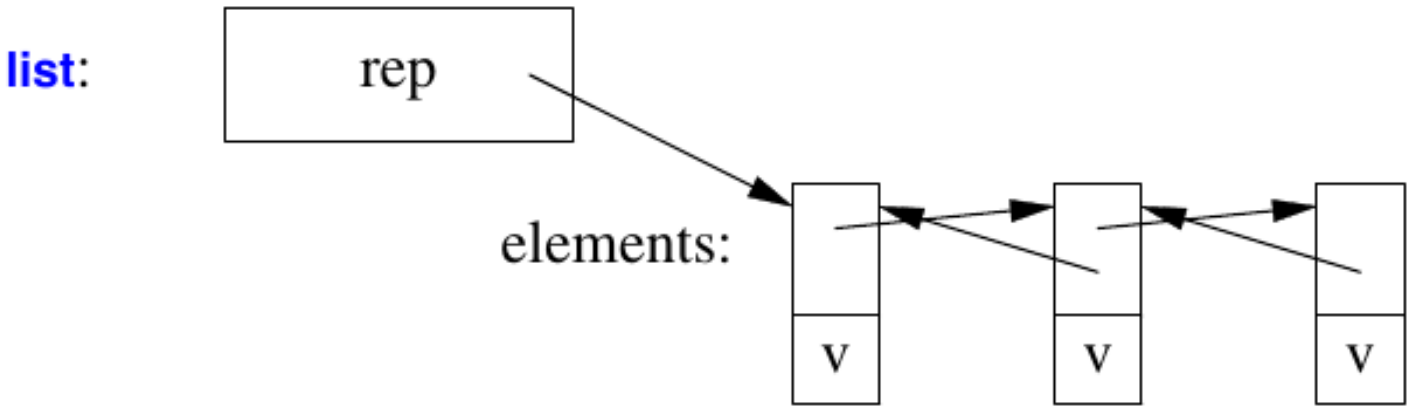
Vector



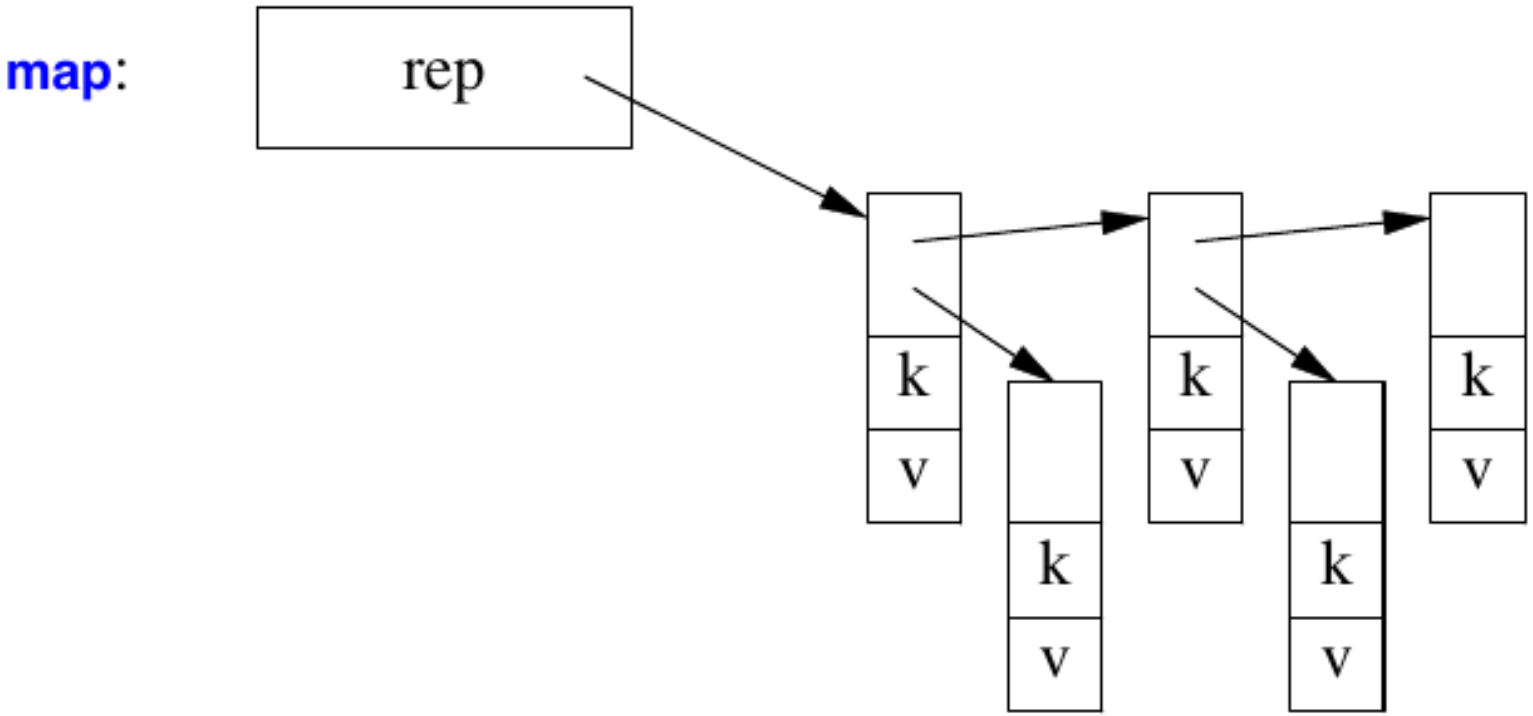
Forward list



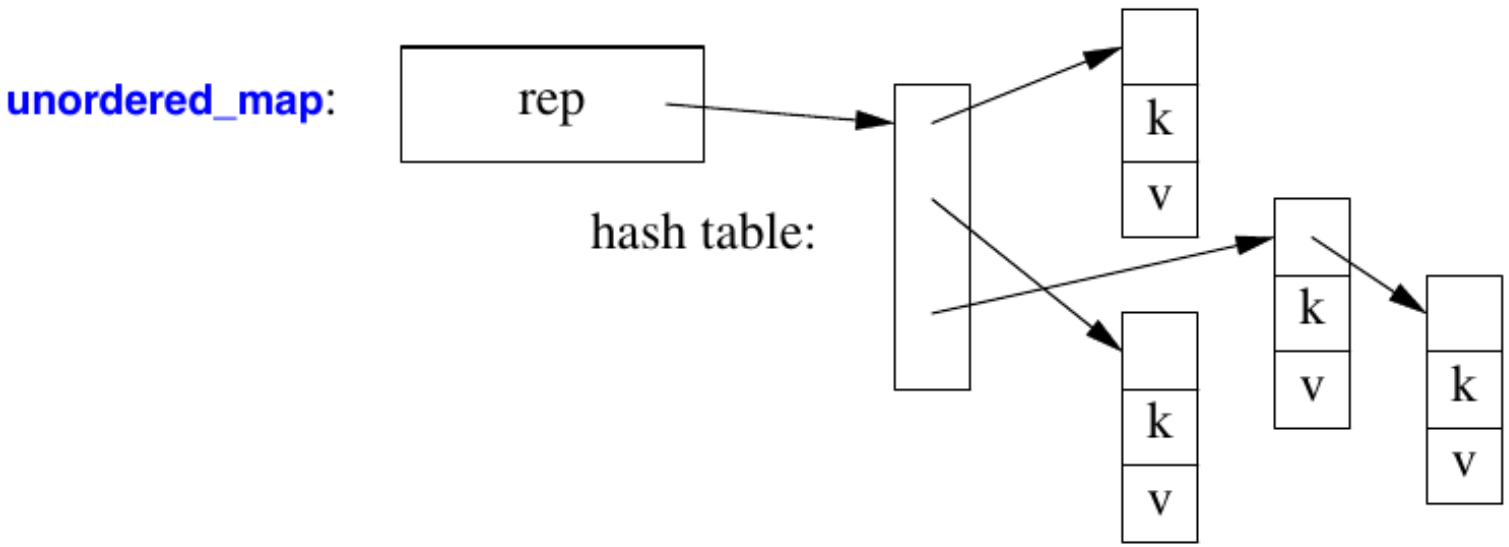
List



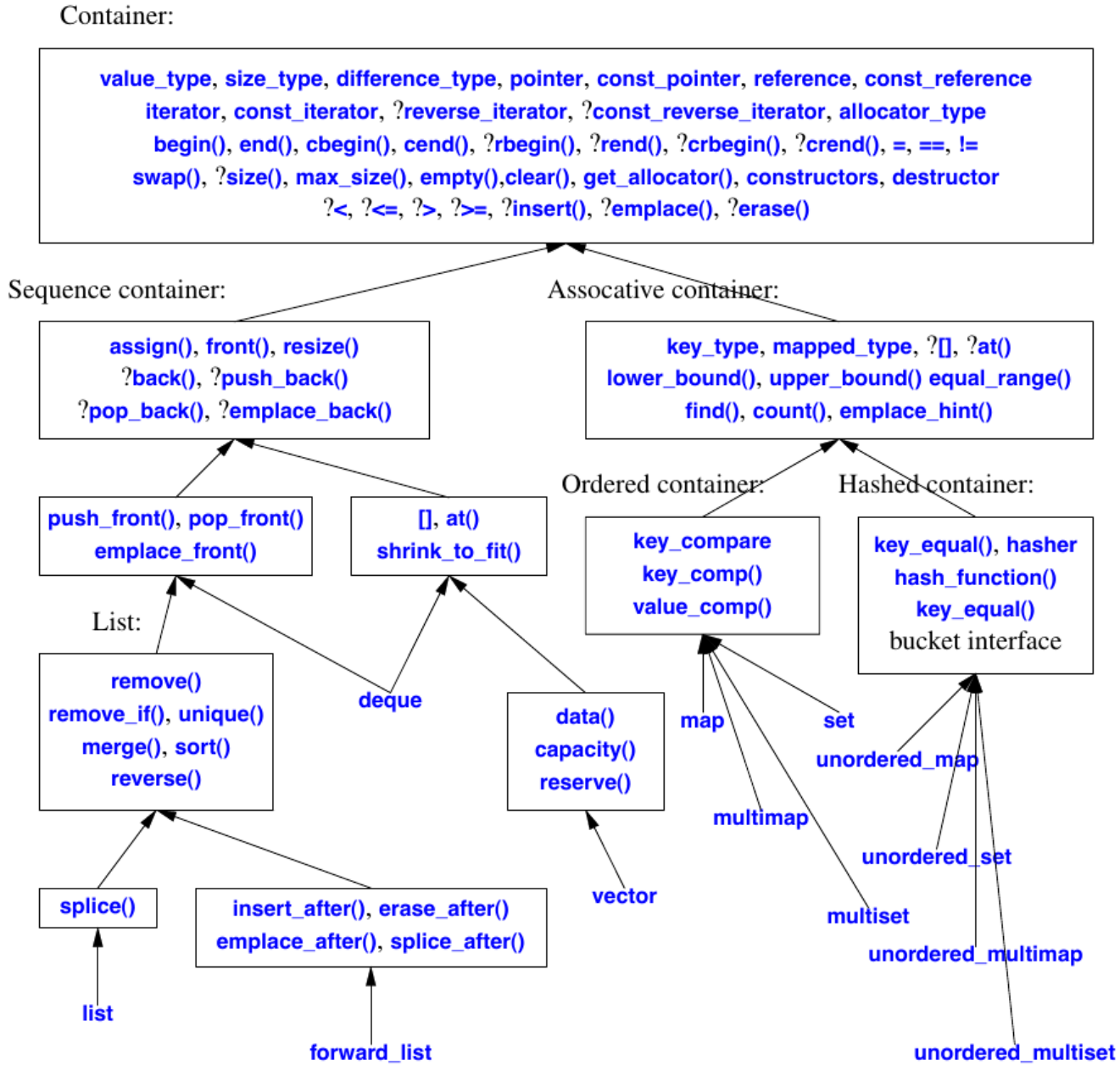
Map



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 <unordered_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;
}
```

What is an Iterator?

Design pattern [GoF]

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

Stepanov

Iterator is a coordinate.

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

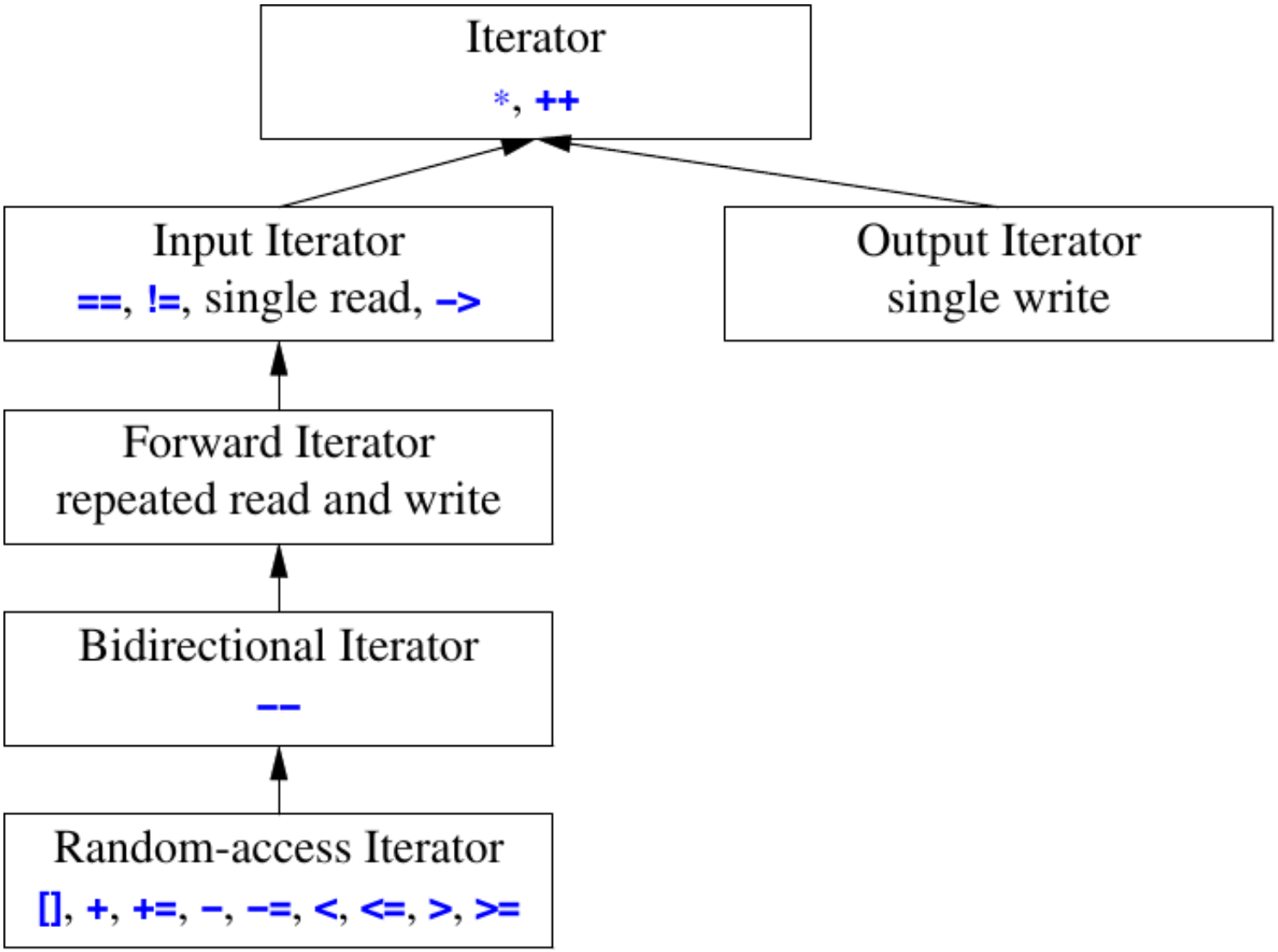
- Iterators are the glue that ties the standard-library alogorithms to their data
- Iterators are the mechanism used to minimize an algorithm's dependence on the data structures on which it operates.

Alex Stepanov

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

```
while( first != last )
```

Iterator categories



How to implement our own iterator?

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

How to implement our own iterator?

```
#include <iterator>
...
template <typename T>
class List<T>::Iterator{
  typename List<T>::node* current;
  public:
  using value_type = T;
  using difference_type = std::ptrdiff_t;
  using iterator_category =
    std::forward_iterator_tag;
  using reference = value_type&;
  using pointer = value_type*;
  ...
```

How to implement our own iterator?

```
    ...
    reference operator*() {
      return current->value; }
    pointer operator->() { return &**this; }
    Iterator& operator++() {
      current = current->next;
      return *this;
    }
    friend
    bool operator==(const Iterator&, const
       Iterator&);
    friend
    bool operator!=(const Iterator&, const
       Iterator&);
};
```

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 or function objects
- report failure (e.g. not found) 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

User-defined functions

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

double my_f(const double& a, const double& b){
 if(std::abs(b - 2.2) < 1e-12)
  return a;
 return a+b;
}
int main(){
 std::vector<double> v1;
 ...
 double sum{0};
 sum = std::accumulate(first,last,sum,my_f);
}
```

Examples

Lambda functions

```
#include <numeric>
#include <vector>
int main(){
 std::vector<double> v1;
 ...
 auto my_f = [](const double& a, const double &b)
      -> double {
   return ( (std::abs(b-2.2) < 1e-12) ? a : a+b);
 };
 double sum{0};
 sum = std::accumulate(first,last,sum,my_f);
}
```

Examples

Generic lambdas (since C++14)

```
#include <numeric>
#include <vector>
int main(){
 std::vector<double> v1;
 ...
 auto my_f = [](const auto& a, const auto& b) {
   return ( (std::abs(b-2.2) < 1e-12) ? a : a+b);
 };
 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>{});
}
```

Lambda

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

int main(){
   std::vector<double> v1;
   ...
   std::sort(v1.begin(), v1.end(),
             [](const auto& a, const auto& b)
                { return a>b; } );
}
```