STL

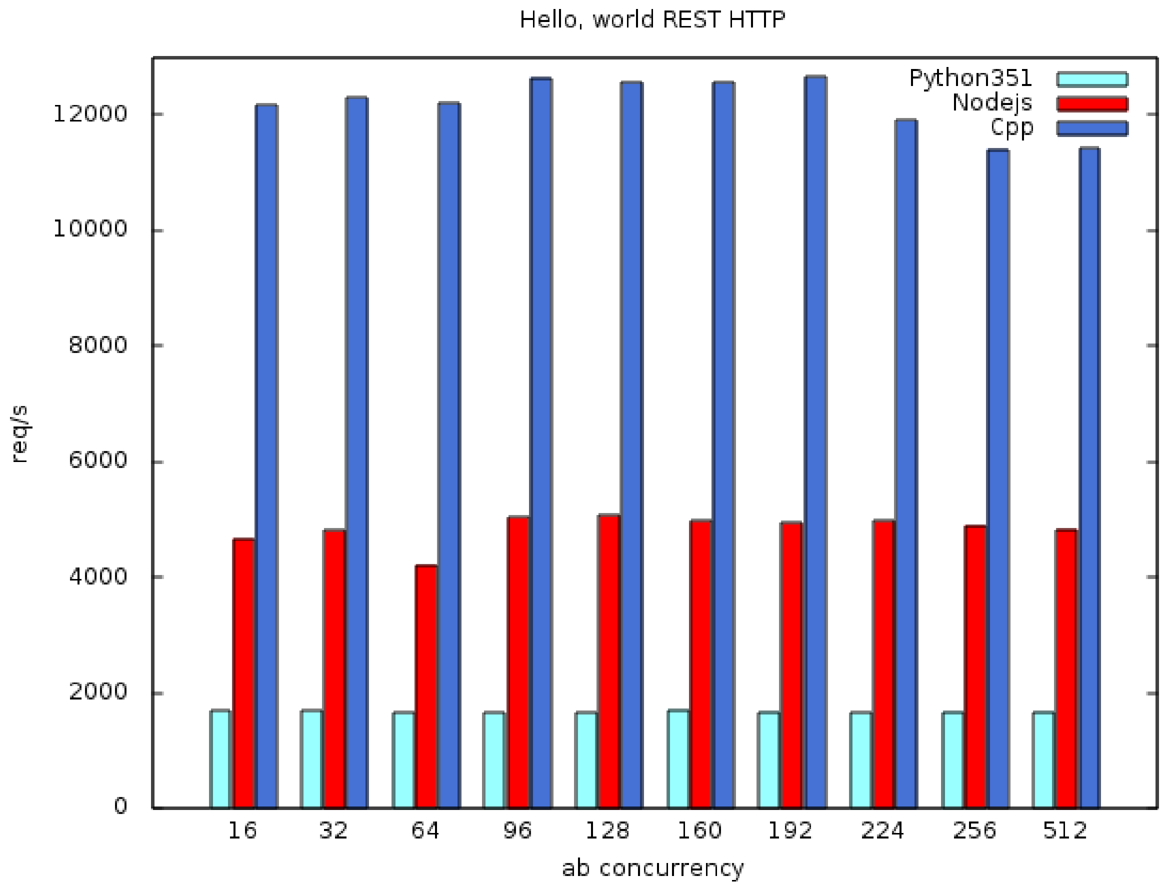
Praktyczne aspekty rozwoju oprogramowania

PWr, 21.03.2016

Sławomir Zborowski

tematy organizacyjne

speedy.sh/eb77Y


Hello, world REST HTTP
Python351
Nodejs
Cpp
req/s
0
2000
4000
6000
8000
10000
12000
16
32
64
96
128
160
192
224
256
512
ab concurrency

Agenda

- History and concepts
- Containers
- Iterators
- Algorithms
- Flaws
- Summary

History

- 1970+ - Stepanov, Musser
- 1993 - first paper
- 1994 - HP STL implementation
- 1998 - standardized

David R. Musser

Alexander Stepanov

Concepts

- Generic algorithms
- Container classes
- Iterators as a glue

Advantages and disadvantages

- Standardized
- Minimal overhead, efficiency-oriented
- Small
- Almost no inheritance
- Extensible
- Template syntax
- No constraints on template types (concepts, C++ flaw)
- Inconsistencies

Containers

Container traits

- Core of the STL
- Concepts of generic containers are older
- Strong genericity
- Backward compatibility

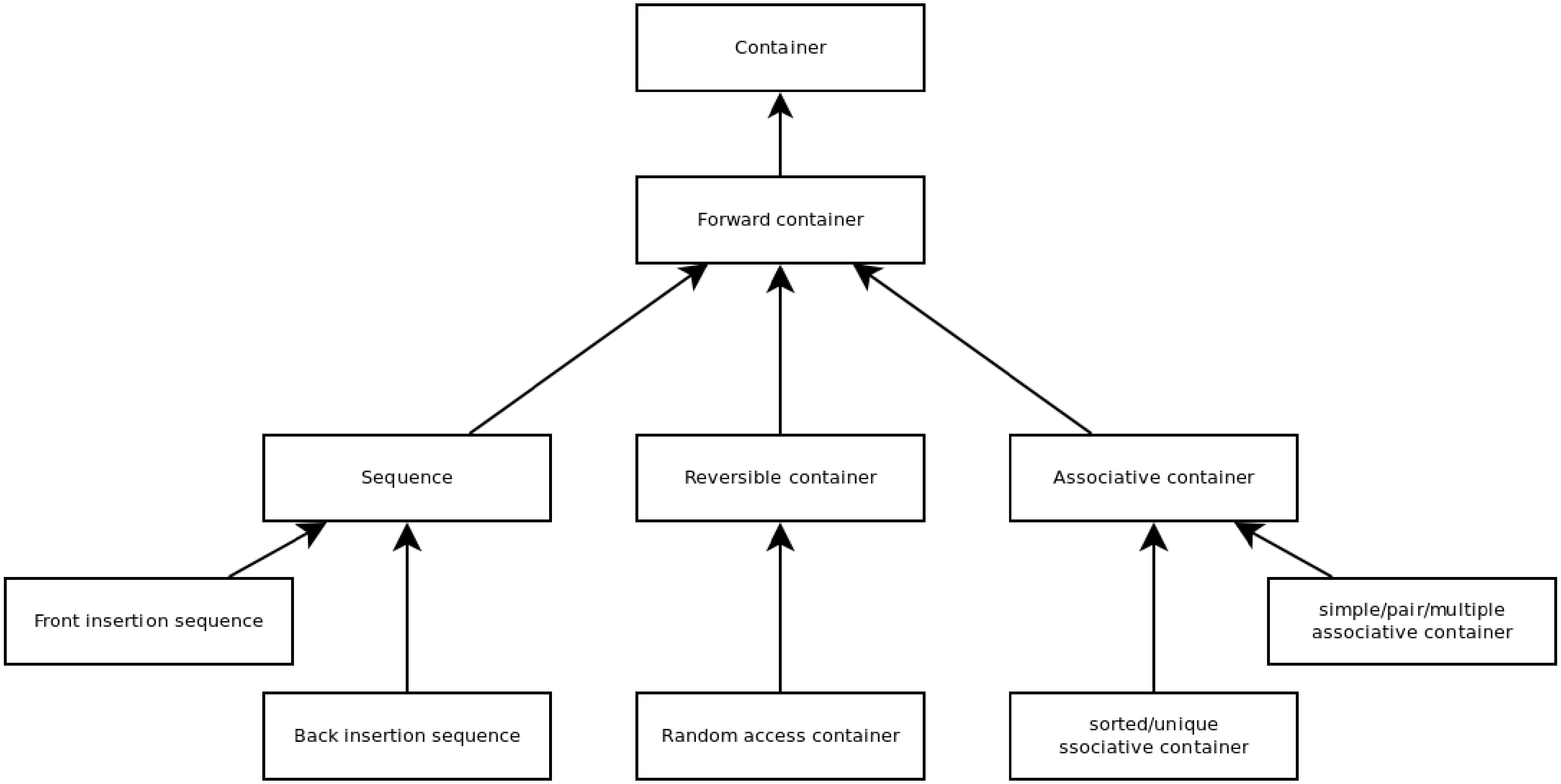
Container traits II

- Container concept
  - Owner of the element it contains
  - Defines iterator(s) that can be used to traverse
  - Provides methods to access elements
  - Order of elements is not strictly defined
- Important types
  - `value_type`
  - `iterator`
  - `const_iterator`

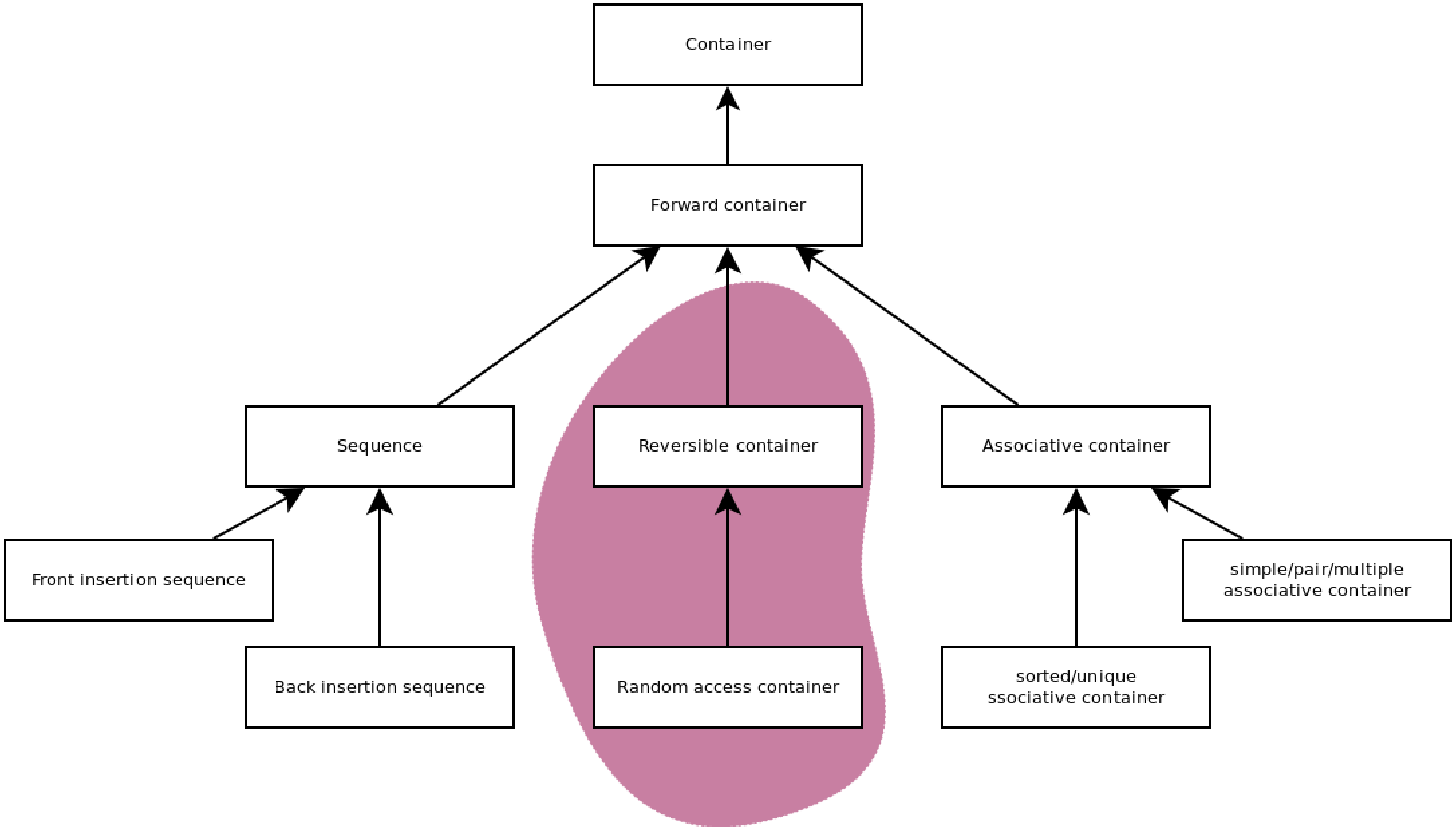
Container traits III

- Containers do not overlap
- Lifetime of contained element is always shorter (value semantic)
- An element can have only one container it belongs to
- **More**, **SGI**

Containers hierarchy



Containers hierarchy



Container

Forward container

Sequence

Reversible container

Associative container

Front insertion sequence

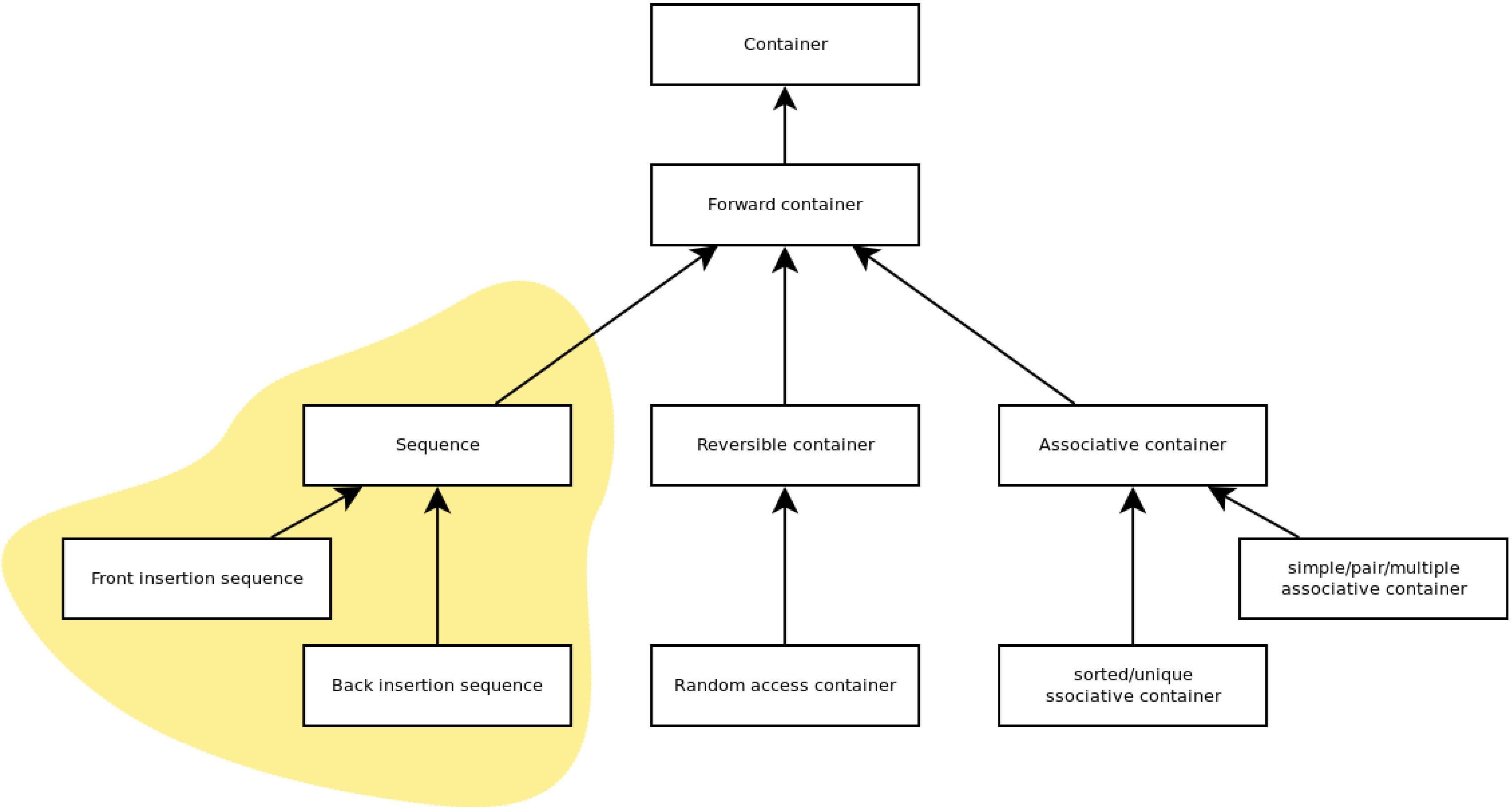
Back insertion sequence

Random access container

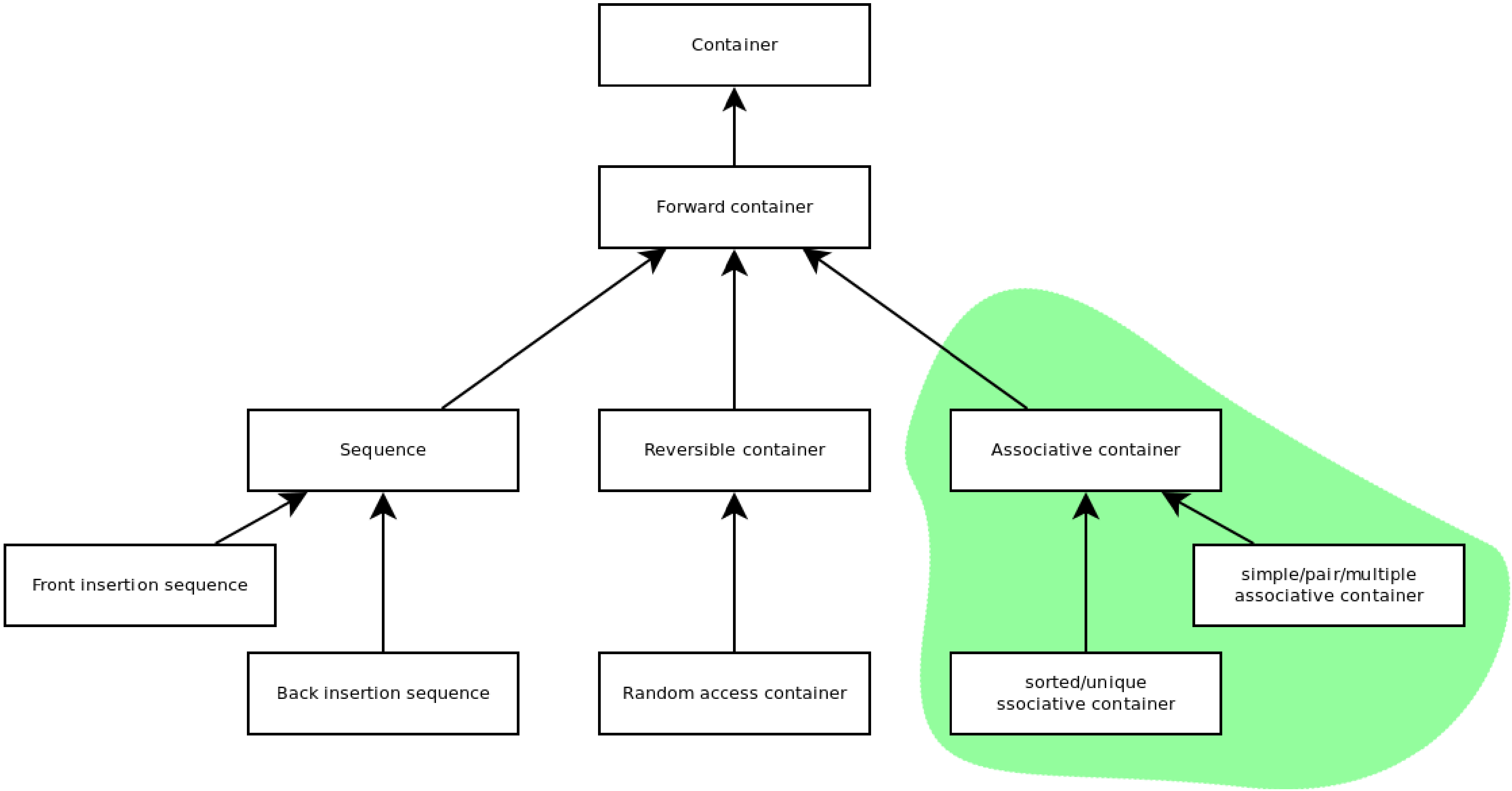
simple/pair/multiple associative container

sorted/unique ssociative container

Containers hierarchy

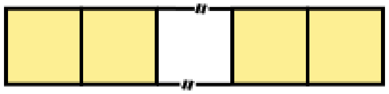


Containers hierarchy



array

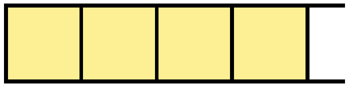
Operation	Method(s)	Complexity
Access	`[], at, front, back, data`	O(1)
Access	`begin, cbegin, rbegin, …`	O(1)
Modification	`fill, swap`	O(n)



- Continuous memory
- Random access
- Max size allowed depends on stack size

vector

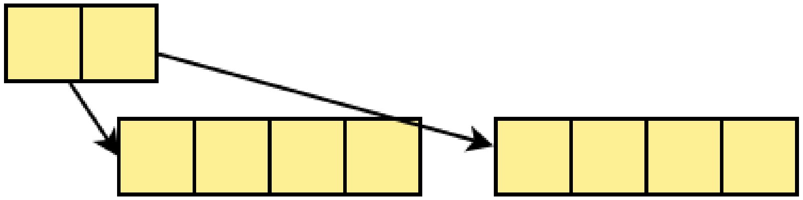
Operation	Method(s)	Complexity
Access	`[], at, front, back, data`	O(1)
Access	`begin, cbegin, rbegin,` …	O(1)
Modification	`insert, erase`	O(n)
Modification	`push_back, pop_back`	O(1) am.



- Continuous memory (on heap)
- Random access

deque

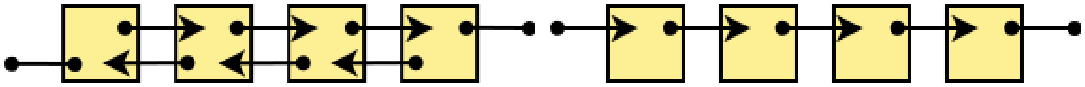
Operation	Method(s)	Complexity
Access	`front, back`	O(1)
Access	`[], at`	O(1)
Access	`begin, cbegin, rbegin, …`	O(1)
Modification	`insert, erase`	O(1) am.
Modification	`push_back, push_front`	O(1)



- multiple chunks
- implementation-defined

list, forward_list

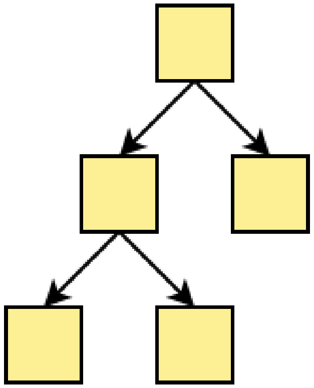
Operation	Method(s)	Complexity
Access	`front, [back]`	O(1)
Access	`begin, cbegin, rbegin,` …	O(1)
Modification	`insert, erase, push_*`	O(1)
Modification	`unique, reverse`	O(n)



- `forward_list` → smaller memory footprint, better efficiency

set, multiset

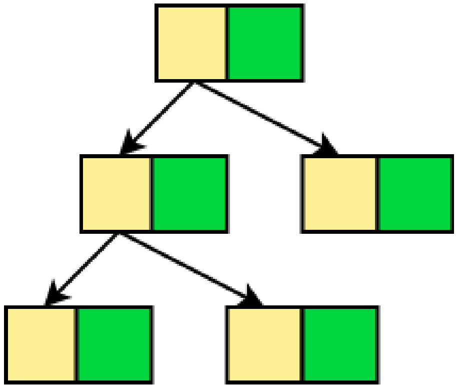
Operation	Method(s)	Complexity
Access	`begin, cbegin, rbegin,` …	O(1)
Modification	`insert, erase`	O(logN)
Lookup	`find, lower_bound,` …	O(logN)



- handling duplicate values:
  - `insert()` → no effect
  - `emplace()` → no effect
- `multiset`: duplicate values are allowed

map, multimap

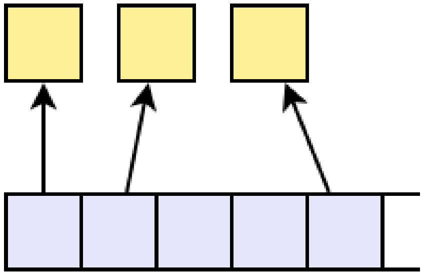
Operation	Method(s)	Complexity
Access	`begin, cbegin, rbegin,` ...	O(1)
Modification	`insert, erase`	O(logN)
Lookup	`find, lower_bound,` ...	O(logN)



- handling duplicate values:
  - `insert()` → no effect
  - `emplace()` → no effect
  - `operator[]` → update
- `multimap`: duplicate keys are OK

unordered_set, unordered_multiset

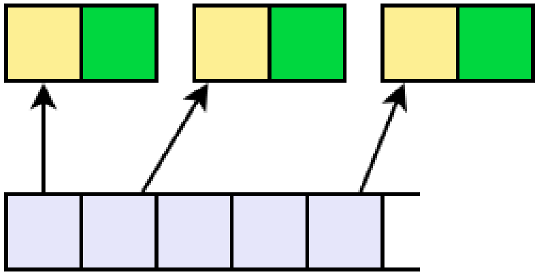
Operation	Method(s)	Complexity
Access	`begin, cbegin, rbegin,` …	O(1)
Modification	`insert, erase`	O(1) am.
Lookup	`find, count,` …	O(1) am.



- handling duplicate values:
  - `insert()` → no effect
  - `emplace()` → no effect

unordered_map, unordered_multimap

Operation	Method(s)	Complexity
Access	`begin, cbegin, rbegin,` ...	O(1)
Modification	`insert, erase`	O(1) am.
Lookup	`find, count,` ...	O(1) am.



- handling duplicate values:
  - `insert()` → no effect
  - `emplace()` → no effect
  - `operator[]` → update

Adapters

- **stack**
  - FILO build over sequence container (vector, **deque**, list)
  - provides top(), push(), pop()
- **queue**
  - FIFO build over sequence container (**deque**, list)
  - provides front(), back(), push(), pop()
- **priority_queue**
  - build over sequence container (**vector**, deque)
  - compare type needed (weak ordering)
  - provides top(), push(), pop()

Other containers and adapters

- string, wstring
- valarray
- bitset
- **rope**

flat containers

- **sorted vector > set ?**
- set
  - fast insertion O(logN)
  - fast search O(logN)
  - complicated structure (red-black tree)
- sorted vector
  - O(logN)
  - can outperform set in search. why?
  - less complicated structure
  - worse insertion in middle O(N), very good at end O(1)
- cache matter

backward compatibility

- c style arrays?
- std::begin returns:
  - possibly cons-qualified iterator - c.begin()
  - pointer to beginning of array
- std::end returns:
  - possibly cons-qualified iterator - c.end()
  - pointer to past-last element of array
- since c++11

Containers - summary

- vector - fast random access
- list - fast insertion and erasure
- deque - double-ended queue
- set/map - ordered
- unordered_set/map - unordered
- Adapters
- std::begin(), std::end()
- flat containers

vector<bool>?

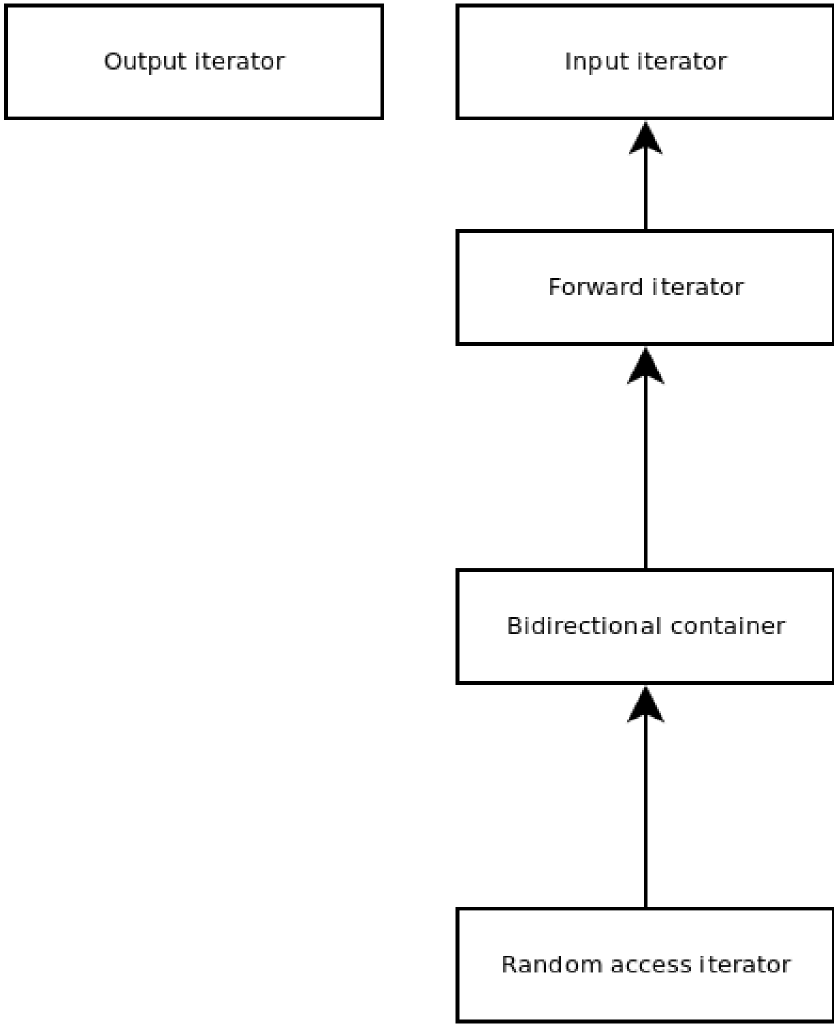
exercise: dup

Iterators

How do you pair socks?

Make it O(1)!

Iterators hierarchy



Input/Output iterators

- Input iterator
  - For sequential input operations
  - Incremented after each read
  - Must support at least `operator==`, `operator!=`
- Output iterator
  - For sequential output operations
  - Incremented after each write

Forward and reversible iterators

- Forward iterator
  - For one-direction iteration
  - Can decrement value multiple times
  - Multiple passes allowed
- Bidirectional iterator
  - Forward iterator with `operator--` support

Random access iterators

- Most complete in terms of functionality
- Can be used like pointers

Algorithms

Algorithms

- Approximately **90 algorithms** in STL
- Of which about 20 new with C++11
- Multiple categories
  - searching (e.g. find(), find_if()) - difference?
  - sorting (e.g. sort())
  - mutating (e.g. transform())
  - numerical (e.g. accumulate(), count_if())
- Most of algorithms take following form:
  function (iterator, iterator, ...);

sum up whole container

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

return acc;
```

```
return accumulate(v.begin(), v.end(), 0);
```

✓ Less verbose, with a name

find maximum/minimum

```
int max = v[0];
for (int i = 1; i < v.size(); ++i) {
    if (v[i] > max) {
        max = v[i];
    }
}
```

```
*max_element(v.begin(), v.end());
```

There are also:

`min_element`, `minmax_element`, `min`, `max`, `minmax`

any in line?

```
bool result = false;
for (int i = 0; i < v.size(); ++i) {
    if (v[i] < 5) {
        result = true;
        break;
    }
}
```

```
any_of(v.begin(), v.end(),
        [](int const e) { return e < 5; });
```

Also: `none_of`, `all_of`, …

rotate

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

```
rotate(v.begin(), v.begin() + v.size() / 2, v.end()) ?
```

0	1	2	3	4	5	6	7	8	9
6	7	8	9	10	1	2	3	4	5

```
rotate(v.begin(), v.begin() + (v.size() / 2) + 2, v.end()) ?
```

0	1	2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10	1	2

partial_sum

0	1	2	3	4
1	2	3	4	5

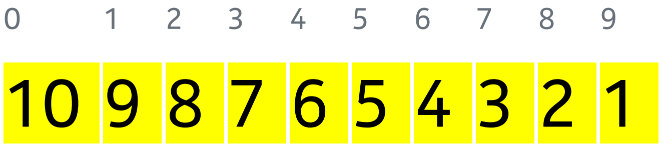
```
partial_sum(v.begin(), v.end(), result.begin());
```

0	1	2	3	4
1	3	6	10	15

```
partial_sum(v.begin(), v.end(), result.begin(), multiplies<int>()); ?
```

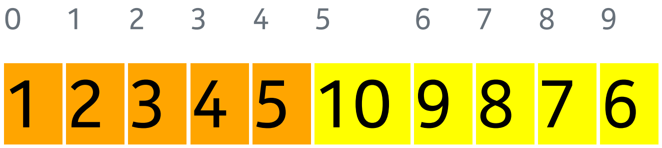
0	1	2	3	4
1	2	6	24	120

partial_sort



0	1	2	3	4	5	6	7	8	9
10	9	8	7	6	5	4	3	2	1

```
partial_sort(v.begin(), v.begin() + v.size() / 2, v.end()) ?
```



0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	10	9	8	7	6

exercise: cheap

partition

0	1	2	3	4	5	6	7	8	9
0.9	1.1	1.9	0.2	0.8	0.6	1.4	0.0	0.5	0.3

```
    partition(v.begin(), v.end(), [](double a) { return a < 1.0; }) ?
```

0	1	2	3	4	5	6	7	8	9
0.9	0.3	0.5	0.2	0.8	0.6	0.0	1.4	1.9	1.1

```
    stable_partition(v.begin(), v.end(), [](double a) { return a < 1.0; }) ?
```

0	1	2	3	4	5	6	7	8	9
0.9	0.2	0.8	0.6	0.0	0.5	0.3	1.1	1.9	1.4

stable_sort

- sort vs stable_sort

0	1	2	3	4	5	6	7	8	9
4.2	4.1	3.2	3.1	2.2	2.1	1.2	1.1	0.2	0.1

```
stable_sort(v.begin(), v.end(), comp_as_ints) ?
```

0	1	2	3	4	5	6	7	8	9
0.2	0.1	1.2	1.1	2.2	2.1	3.2	3.1	4.2	4.1

remove duplicates

```
set<int> result;
for (int const e: v) {
    result.insert(e);
}
```

```
vector<int> result;
unique(v.begin(), v.end());
```

std::unique removes only consecutive duplicates

unique vs unique_copy

0	1	2	3	4	5	6	7	8	9
1	1	4	2	4	5	5	5	6	1

```
unique(v.begin(), v.end()) ?
```

0	1	2	3	4	5	6	7	8	9
1	4	2	4	5	6	1	?	?	?

```
unique_copy(v.begin(), v.end(), back_inserter(result)) ?
```

0	1	2	3	4	5	6
1	4	2	4	5	6	1

```
std::back_inserter ?
```

find difference

```
vector<int>::iterator it = v1.begin();
for (int i = 0; i < v1.size(); ++i) {
    if (v1[i] != v2[i]) {
        advance(it, i);
    }
}
```

```
mismatch(v1.begin(), v1.end(), v2.begin());
```

Returns pair of iterators at which difference occurs

nth_element

0	1	2	3	4	5	6	7	8	9
4	9	9	4	3	3	7	8	8	8

```
nth_element(v.begin(), v.begin() + 4, v.end()); ?
```

0	1	2	3	4	5	6	7	8	9
4	3	3	4	7	8	8	8	9	9

transform

0	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9

```
    transform(v1.begin(), v1.end(), v2.begin(), inc) ?
```

0	1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9	10

- unary vs binary

copy

0	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, ","))
```

- 1,2,3,4,5,6,7,8,9,

exercise: pow2

iota

```
int start = -2;
for (int i = 0; i < v.size(); ++i) {
    v[i] = start + i;
}
```

```
iota(v.begin(), v.end(), -2);
```

iota? wtf!

Common mistakes

- Container classes used as bases
- Removing elements from container may invalidate iterators
- Containers and std::auto_ptr
- Premature optimization when choosing a container class
- Same algorithm, two containers
- Valid iterator, wrong container

erase-remove idiom

- What does remove?
- What does erase then?

Update value in set?

Loops aren't smart

```
for (auto const& e: elements) {
    if (e.isOutdated()) {
        elements.erase(e);
    }
}
```

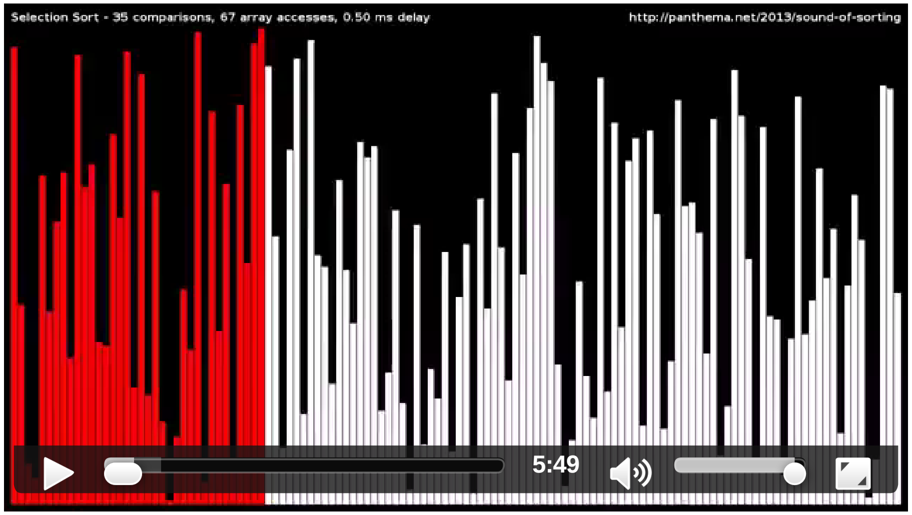
- Undefined behavior...

Common mistakes

STL - summary

- Small
- Efficient
- Extensible
- Powerful

Q & A



Selection Sort - 35 comparisons, 67 array accesses, 0.50 ms delay

http://panthema.net/2013/sound-of-sorting