Back to Basics: Classic STL

Bob Steagall

CppCon 2021

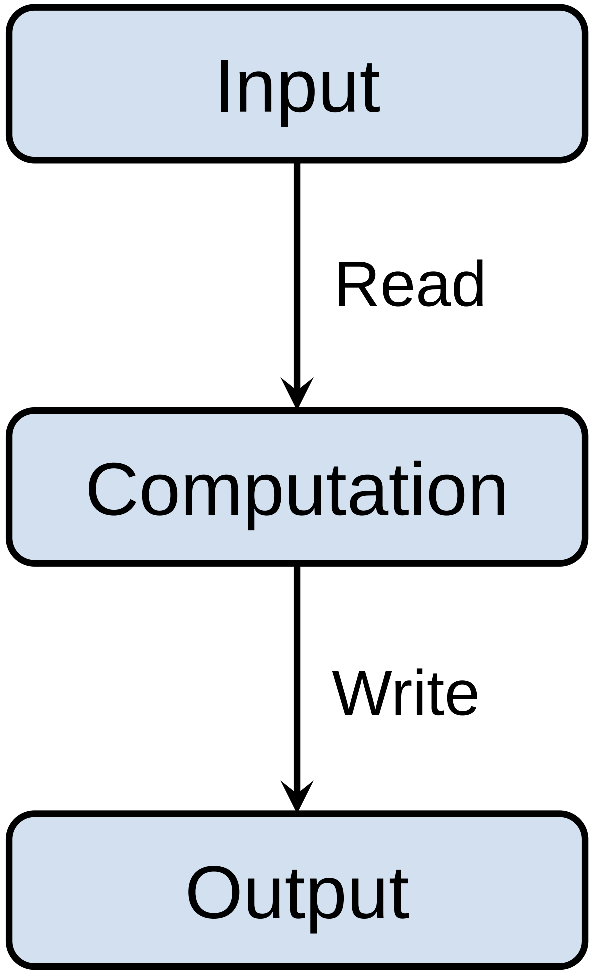
What is "Classic STL?"

The C++20 Standard Library

Language Support	General Utilities	Atomic Operations
Concepts	Containers	Thread Support
Diagnostics	Iterators	Numerics
Strings	Algorithms	Time
Ranges	Input/Output	Localization
	Regular Expressions	

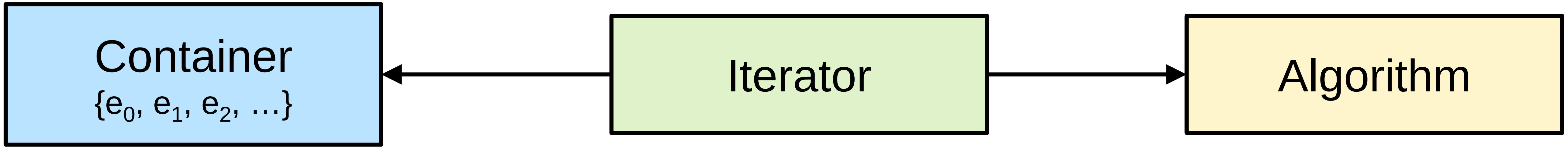
Rationale

- Data is almost always *collections* of *elements*
  - A virtually infinite number of data element types
- Each collection of elements has some *representation*
  - A large number of possible representations
- There are many kinds of processing (*algorithms*)
  - A very large number of algorithms
- In any given problem space, the choices are fewer
  - Call them N_T , N_R , and N_A
  - Traditionally, a combinatorial explosion of code – $N_T * N_R * N_A$
- We'd like a smaller number – $N_T + N_R + N_A$ – **this is the goal of the STL**



Key Principles

- *Containers* store *collections* of *elements*
- *Algorithms* perform operations upon collections of elements
- Containers and algorithms are entirely independent
- *Iterators* provide a common unit of information exchange between containers and algorithms



Containers Overview

- Containers hold a collection of elements
  - STL containers are implemented using a variety of basic data structures
  - Each STL container represents a **sequence** of elements
- Containers have an internal structure and ordering
  - We can observe this ordering
  - Sometimes we can control the ordering
- **Containers own the elements they hold**
  - Ownership means element lifetime management
  - Containers construct and destroy their member elements

Containers Overview

- Sequence containers
  - vector
  - deque
  - list
  - array (C++11)
  - forward_list (C++11)
- Associative containers
  - map
  - set
  - multimap
  - multiset
- Unordered associative containers
  - unordered_map (C++11)
  - unordered_set (C++11)
  - unordered_multimap (C++11)
  - unordered_multiset (C++11)
- Container adaptors
  - queue
  - stack
  - priority_queue

Iterators Overview

- Iterators typically provide a way of observing a container's elements and ordering
  - Some containers provide more than one way to observe elements

- Iterators *may* provide a way of modifying a container's elements

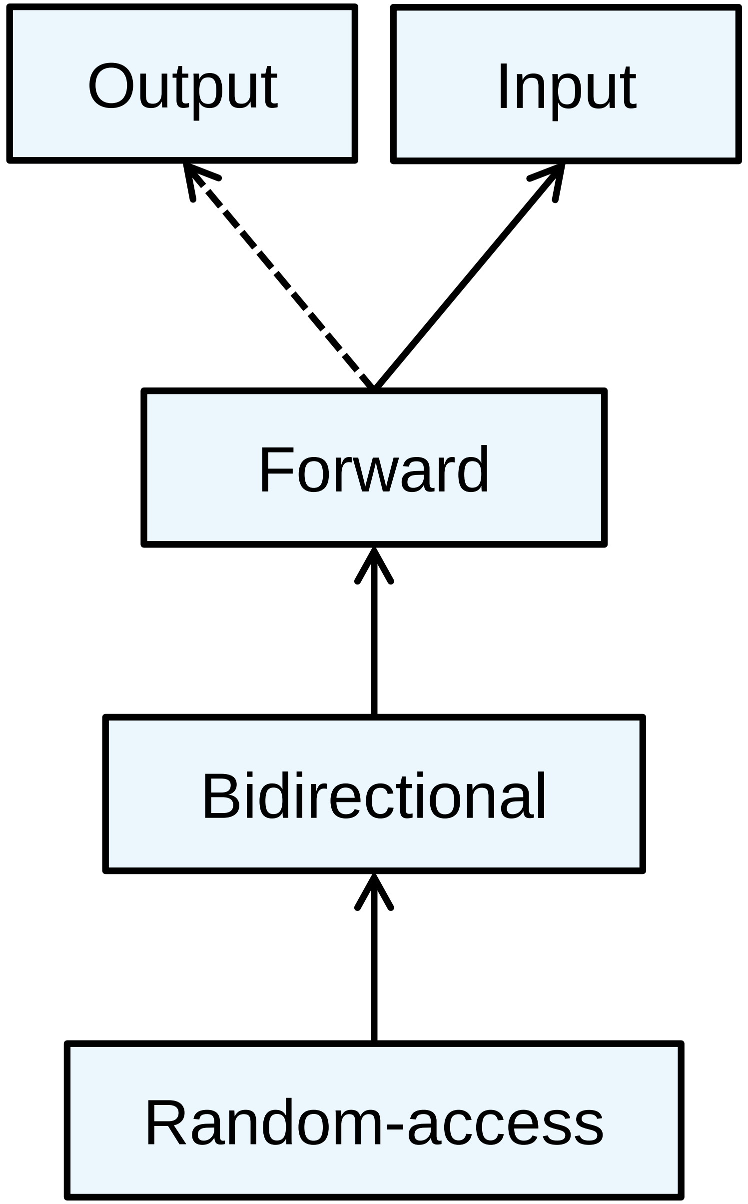
- An iterator's interface specifies
  - The complexity of observing and traversing a collection's elements
  - The manner in which elements are observed
  - Whether an element can be read from or written to

- **Iterators never own the elements to which they refer**

Iterators Overview

- Classic STL has five iterator categories
  - Output
  - Input
  - Forward
  - Bidirectional
  - Random-access

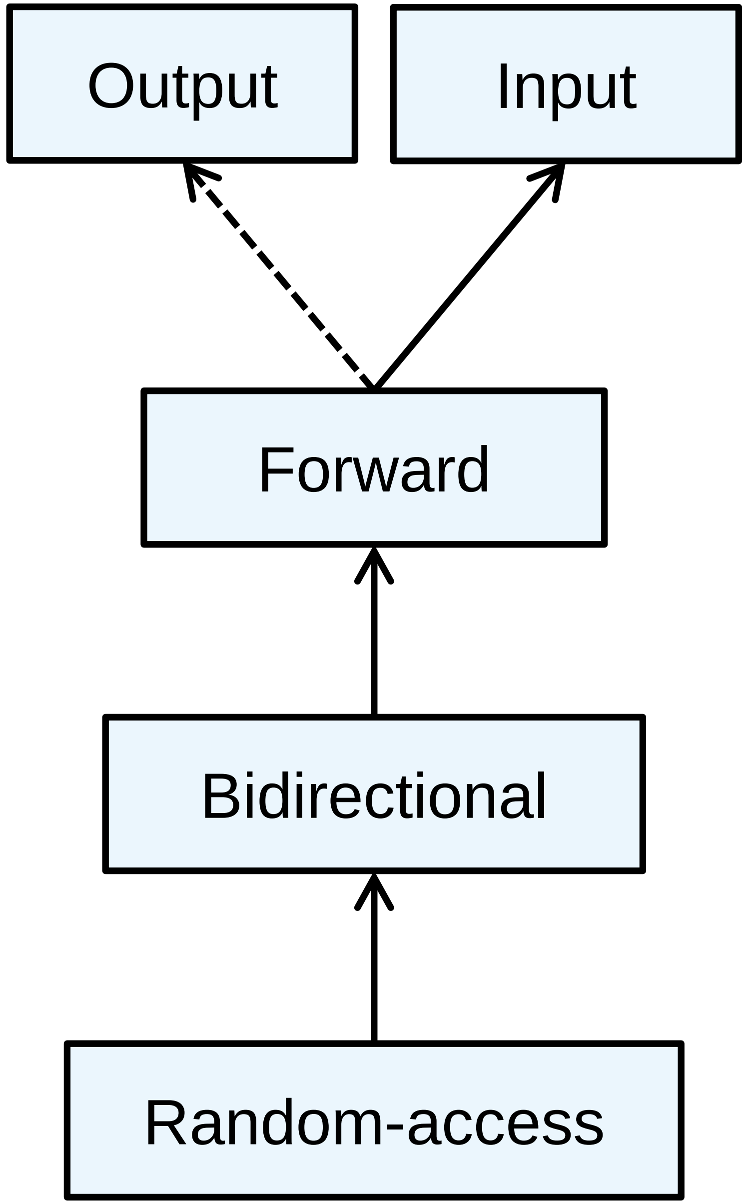
- Arranged in a hierarchy of *requirements*
  - <u>Not</u> public inheritance



Iterators

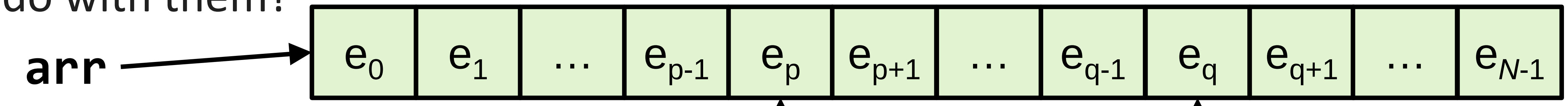
Regarding Iterators

- Where do the five iterator categories come from?
- What interface does each category provide?
- What is their time complexity?
- How are they related to containers?
- How are they used by the algorithms?
- Let's try a generic programming exercise and develop iterators from scratch



Referring to Elements in Arrays

- Consider pointers to 2 elements in an array of N objects
  - What can you do with them?

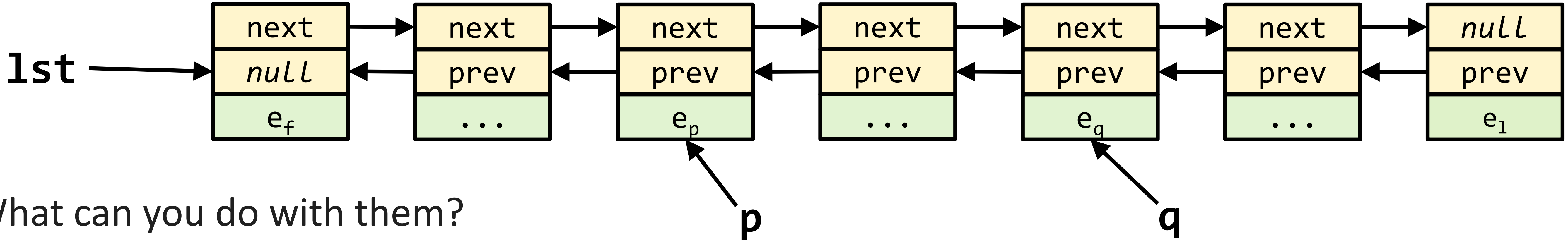


Action	Operation
Access element	`*p`
Access member of element	`p->mem`
Compare for equality of postion	`p == q,   p != q`
Move forward by 1	`++p,  p++`
Move backward by 1	`--p,  p--`
Make a copy (assign)	`q = p`
Access arbitrary element	`p[n]`
Move forward by arbitrary n	`p += n,   q = p + n`
Move backward by arbitrary n	`p -= n,   q = p - n`
Compare for relative position	`p < q,   p <= q,   p > q,   p >= q`
Find distance between two elements	`d = q - p`

O(1) - constant time!

Referring to Elements in Doubly-Linked Lists

- Consider pointers to 2 nodes in a simple doubly-linked list



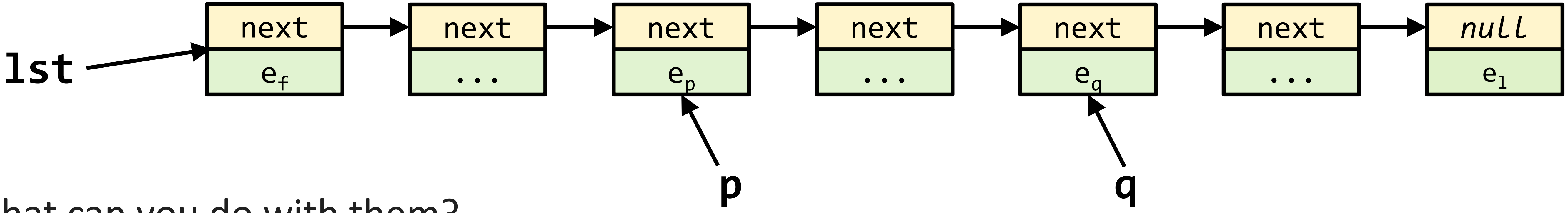
  - What can you do with them?

Action	Operation
Access element	`*p`
Access member of element	`p->`*`mem`*
Compare for equality of position	`p == q,   p != q`
Move forward by 1	`p = p->next`
Move backward by 1	`p = p->prev`
Make a copy (assign)	`q = p`

O(1) - constant time

Referring to Elements in Singly-Linked Lists

- Consider pointers to 2 nodes in a simple singly-linked list and



  - What can you do with them?

Action	Operation
Access element	`*p`
Access member of element	`p->mem`
Compare for equality of position	`p == q`, `p != q`
Move forward by 1	`p = p->next`
Make a copy (assign)	`q = p`

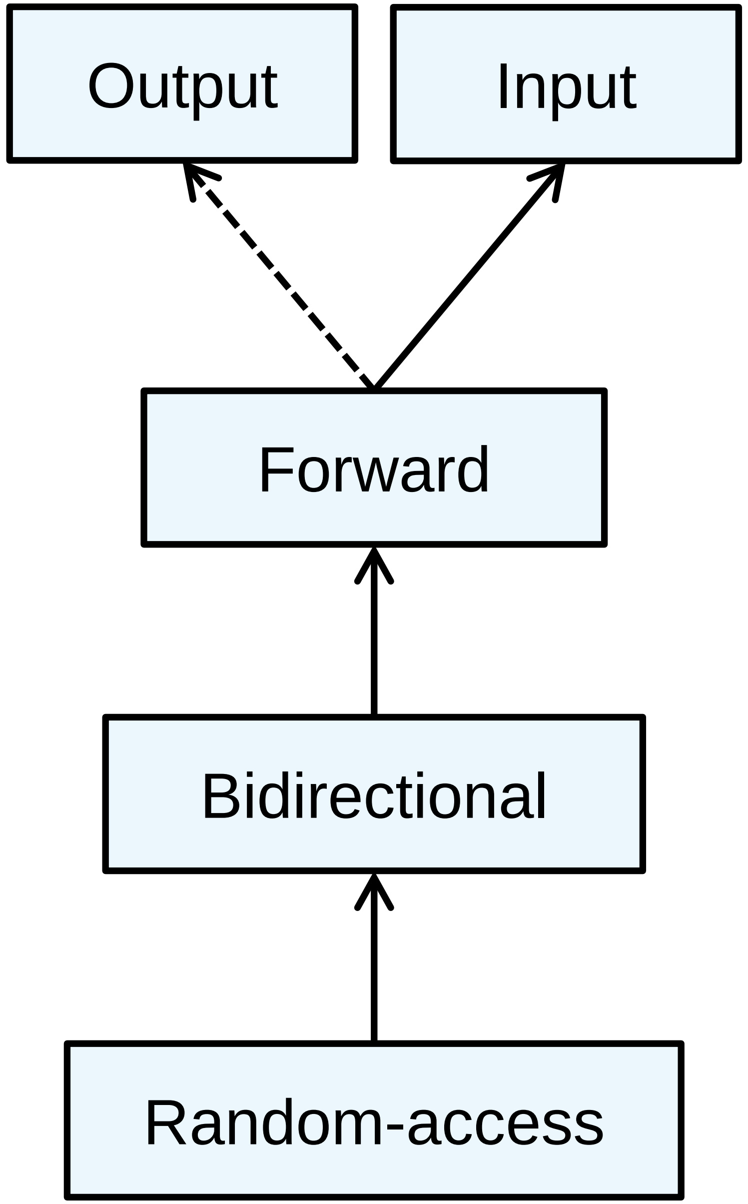
O(1) - constant time

Iterator Categories

Category	Operation
Output	Write forward, single-pass
Input	Read forward, single-pass
Forward	Access forward, multi-pass
Bidirectional	Access forward and backward, multi-pass
Random Access	Access arbitrary position, multi-pass

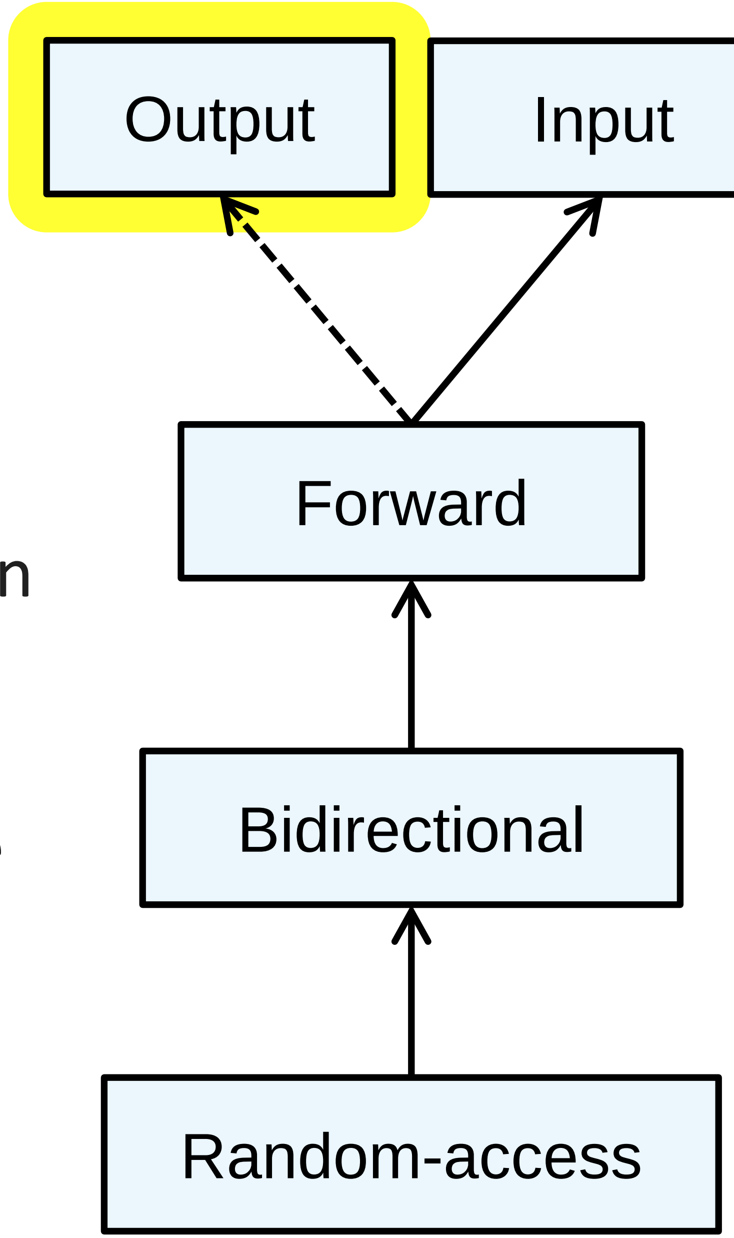
- Arranged in a hierarchy of *requirements*
  - Not public inheritance
  - Arrow to X means: "satisfies at least the requirements of X"
  - Dotted arrow means: "optional"

- Iterators that satisfy the requirements of output iterators are called *mutable* iterators



Output Iterators – Write Forward, Single-Pass

Expression	Action/Result
`Iter  q(p)`	Copy construction
`q = p`	Copy assignment
`*p`	Write to position one time
`++p`	Step forward, return new position
`p++`	Step forward, return old position

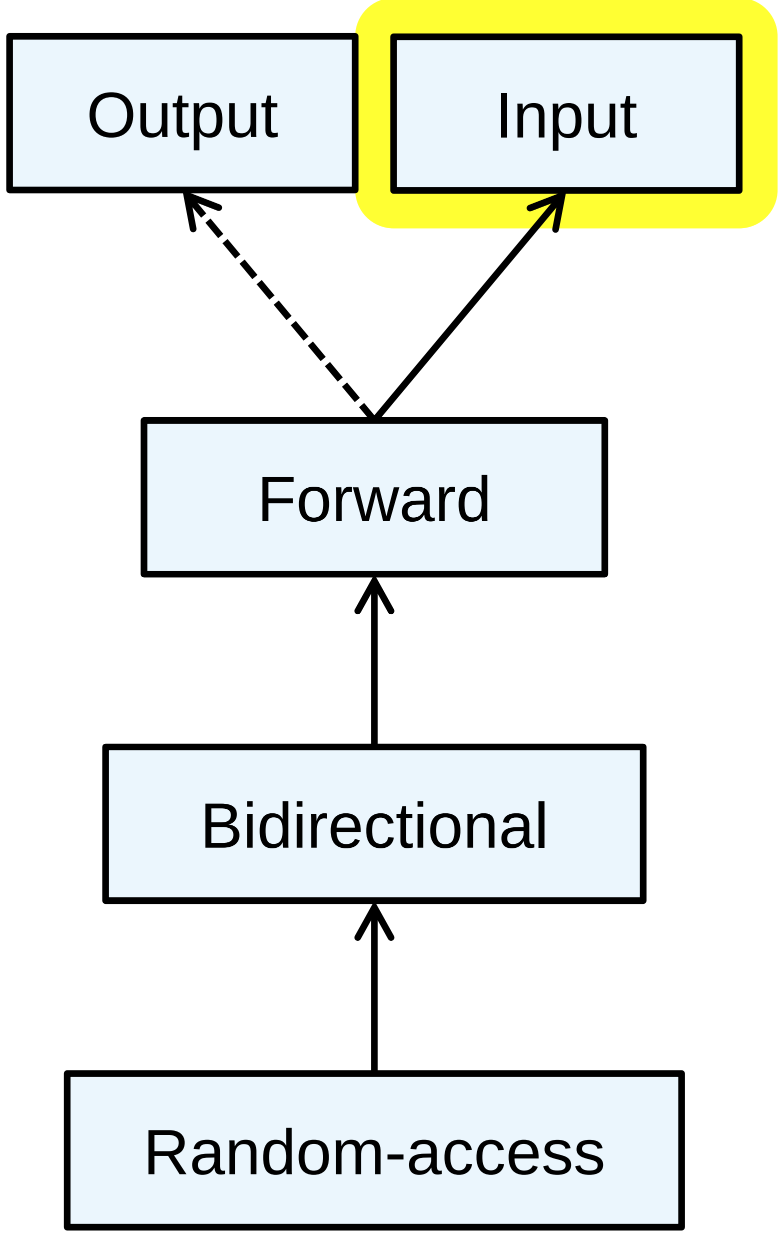


- The only valid use of the expression `*p` is on the left side of an assignment statement
- Comparison operators are not required – no end of sequence is assumed
  - Output iterators model an "infinite sink"
- `const_iterator` types provided by STL containers cannot be output iterators – `const_iterator`s permit only reading

Input Iterators – Read Forward, Single-Pass

Expression	Action/Result
`Iter q(p)`	Copy construction
`q = p`	Copy assignment
`*p`	Read access to element one time
`p->mem`	Read access member of element one time
`++p`	Move forward by 1, return new position
`p++`	Move forward by 1, possibly return old position
`p == q`	Return true if two iterators are equal
`p != q`	Return true if two iterators are different

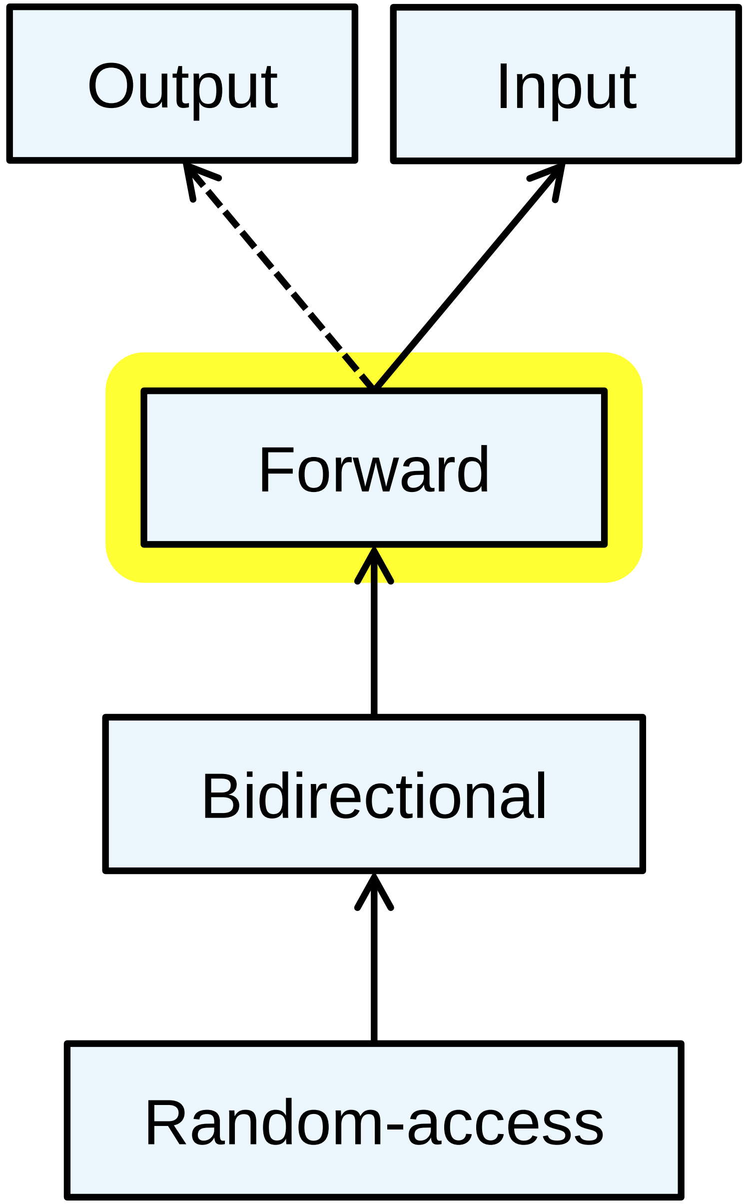
- `p == q` does not imply `++p == ++q`
- The comparison operators are provided to check whether an input iterator is equal to the past-the-end iterator
- All iterators that read values must provide at least the capabilities of input iterators; usually, they provide more



Forward Iterators – Access Forward, Multi-Pass

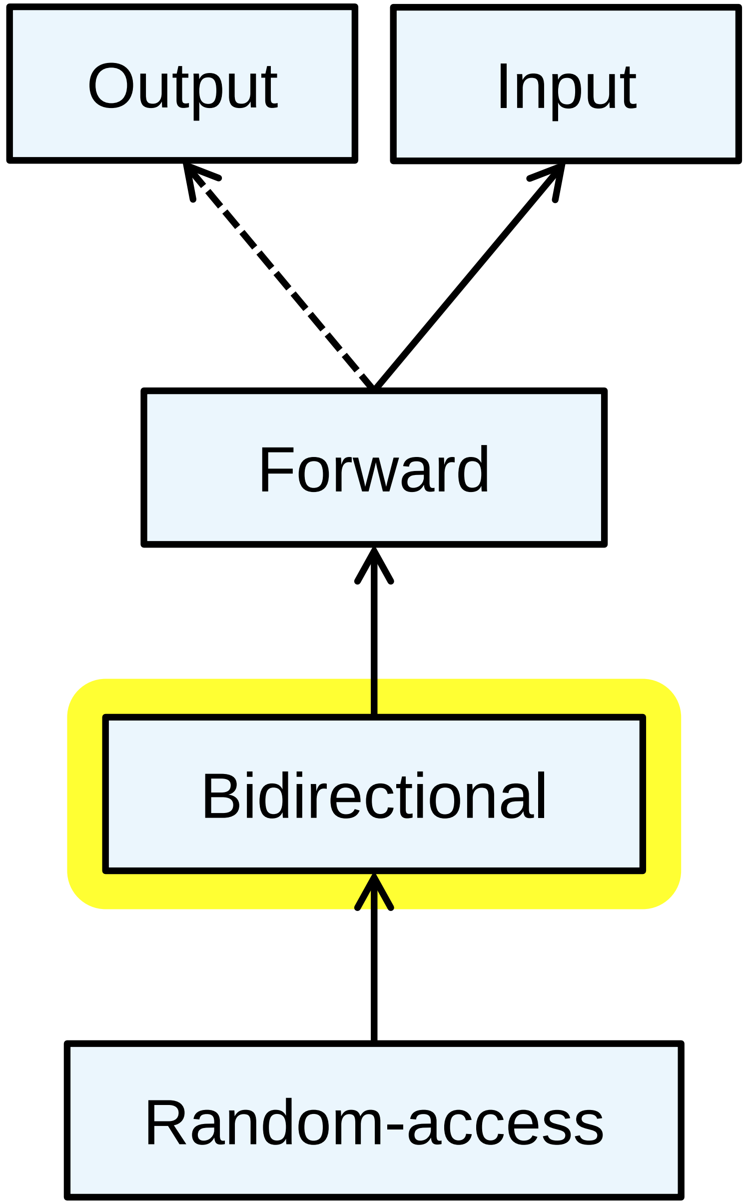
Expression	Action/Result
`Iter  q(p)`	Copy construction
`q = p`	Copy assignment
`*p`	Access element
`p->mem`	Access member of element
`++p`	Move forward by 1, return new position
`p++`	Move forward by 1, return old position
`p == q`	Return true if two iterators refer to the same position
`p != q`	Return true if two iterators refer to different positions
`Iter  p`	Default constructor, create singular value

- Additional capabilities and guarantees
  - `p` and `q` refer to the same position IFF `p == q`
  - `p == q` implies `++p == ++q`
  - Accessing an element (e.g., `*p`) does not change the iterator's position



Bidirectional Iterators – Access Forward/Backward, Multi-Pass

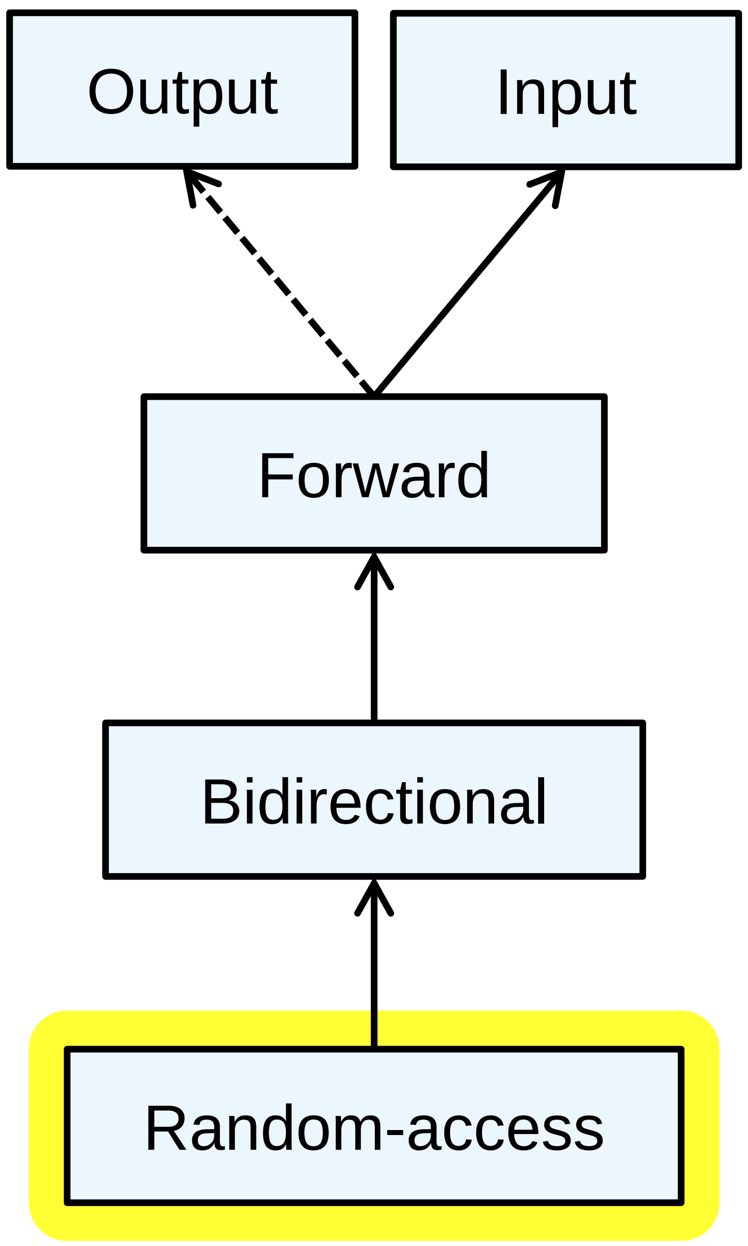
Expression	Action/Result
`Iter q(p)`	Copy construction
`q = p`	Copy assignment
`*p`	Access element
`p->mem`	Access member of element
`++p`	Move forward by 1, return new position
`p++`	Move forward by 1, return old position
`p == q`	Return true if two iterators refer to the same position
`p != q`	Return true if two iterators refer to different positions
`Iter p`	Default constructor, create singular value
`--p`	Move backward by 1, return new position
`p--`	Move backward by 1, return old position



- Additional capabilities and guarantees
  - `p == q` implies `--p == --q`
  - `--(++p) == p`

Random-Access Iterators – Arbitrary Access, Multi-Pass

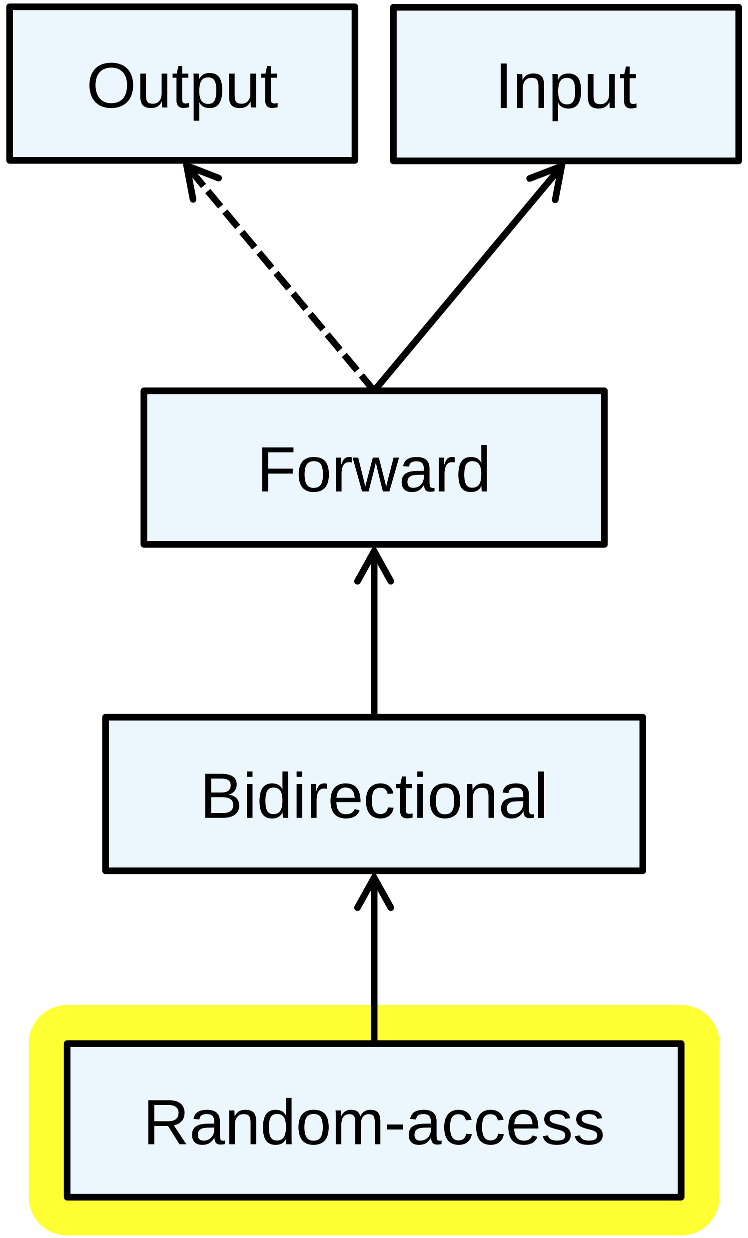
Expression	Action/Result
`Iter q(p)`	Copy construction
`q = p`	Copy assignment
`*p`	Access element
`p->mem`	Access member of element
`++p`	Move forward by 1, return new position
`p++`	Move forward by 1, return old position
`p == q`	Return true if two iterators refer to the same position
`p != q`	Return true if two iterators refer to different positions
`Iter p`	Default constructor, create singular value
`--p`	Move backward by 1, return new position
`p--`	Move backward by 1, return old position



- Additional capabilities and guarantees
  - Emulate pointers
  - Provide operators for iterator arithmetic, analogous to pointer arithmetic
  - Provide relational operators to compare position

Random-Access Iterators – Arbitrary Access, Multi-Pass

Expression	Action/Result
`p[n]`	Access element at nth position
`p += n`	Move forward by n elements (backward if n < 0)
`p -= n`	Move backward by n elements (forward if n < 0)
`p + n`, `n + p`	Return iterator pointing n elements forward (backward if n < 0)
`p – n`	Return iterator pointing n elements backward (forward if n < 0)
`p – q`	Return the distance between positions
`p < q`	True if p is before q in the sequence
`p <= q`	True if p is not after q in the sequence
`p > q`	True if p is after q in the sequence
`p >= q`	True if p is not before q in the sequence



- Additional capabilities and guarantees
  - Emulate pointers
  - Provide operators for iterator arithmetic, analogous to pointer arithmetic
  - Provide relational operators to compare position

Containers Overview

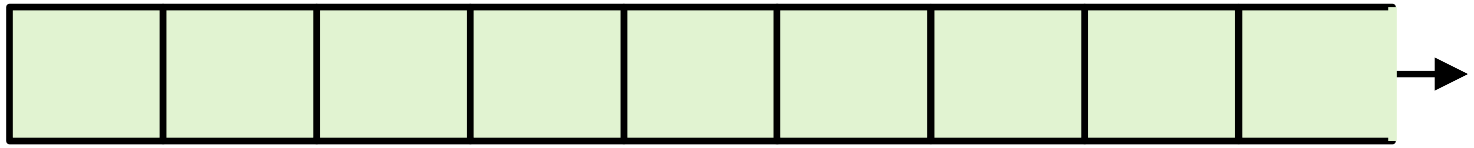
- Sequence containers
  - Represent ordered collections where an element's position is independent of its value
  - Usually implemented using arrays or linked lists
  - `vector`, `deque`, `list`, `array`*, `forward_list`*

- Associative containers
  - Represent sorted collections where an element's position depends only on its value
  - Usually implemented using binary search trees
  - `map`, `set`, `multimap`, `multiset`

- Unordered associative containers*
  - Represent unsorted collections where an element's position is irrelevant
  - Implemented using hash tables
  - `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`

Sequence Container: Vector

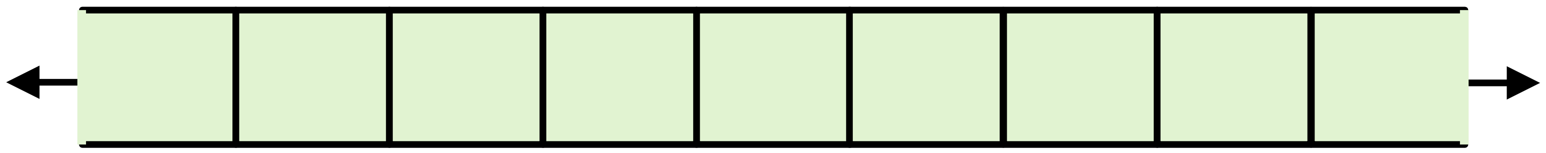
```
template<class T, class Allocator = allocator<T>>
class vector;
```



- Features
  - Supports amortized constant time insert and erase operations at its end
  - Supports linear time insert and erase operations in its middle
  - Provides const and mutable **random-access** iterators
  - Provides const and mutable element indexing
  - Supports changing element values
  - Uses contiguous storage for all element types except `bool`

Sequence Container: Deque

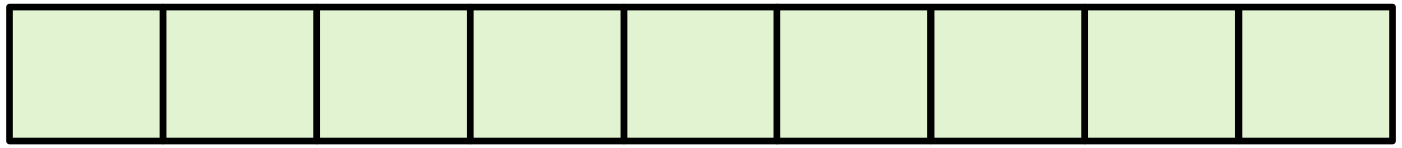
```
template<class T, class Allocator = allocator<T>>
class deque;
```



- Features
  - Supports amortized constant time insert and erase operations at both ends
  - Supports linear time insert and erase operations in its middle
  - Provides const and mutable **random-access** iterators
  - Provides const and mutable element indexing
  - Supports changing element values

Sequence Container: Array

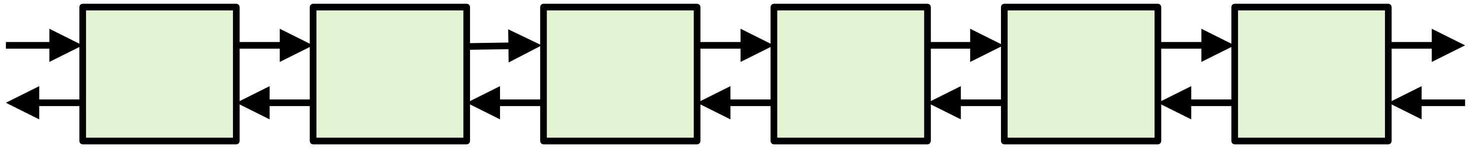
```
template<class T, size_t N>
class array;
```



- Features
  - Manages a fixed-sized sequence of objects in an internal C-style array
  - Provides const and mutable **random-access** iterators
  - Provides const and mutable element indexing
  - Supports changing element values
  - Uses contiguous storage for all element types

Sequence Container: List

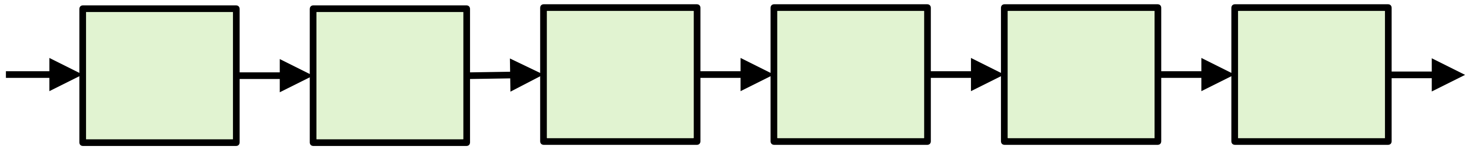
```
template<class T, class Allocator = allocator<T>>
class list;
```



- Features
  - Supports constant time insert and erase operations anywhere in the sequence
  - Provides const and mutable **bidirectional** iterators
  - Supports changing element values
  - Provides member functions for splicing, sorting, and merging
  - Usually implemented as a doubly-linked list

Sequence Container: Forward List

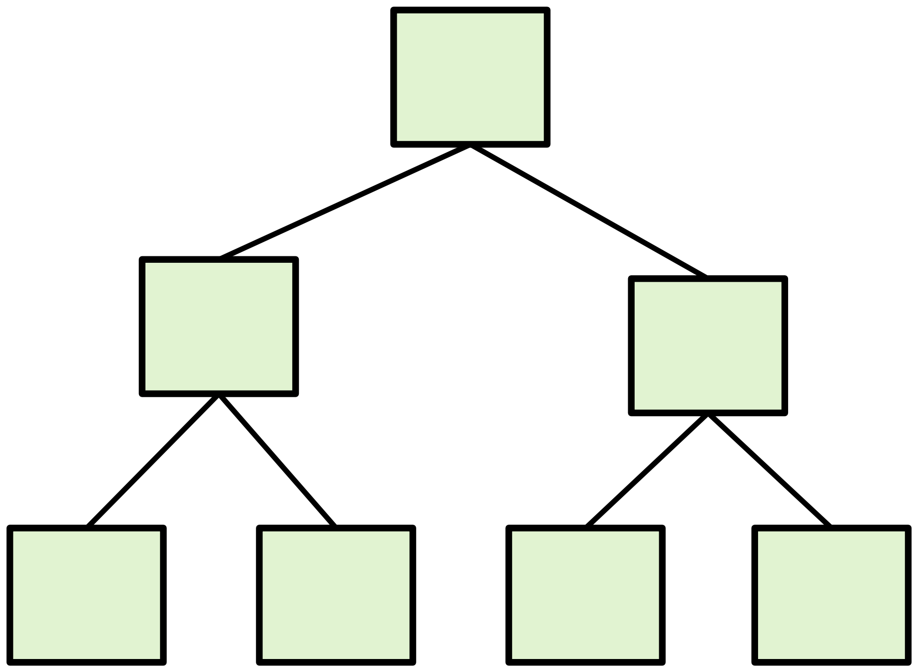
```
template<class T, class Allocator=allocator<T>>
class forward_list;
```



- Features
  - Supports constant time insert and erase operations anywhere in the sequence
  - Provides const and mutable **forward** iterators
  - Supports changing element values
  - Provides member functions for splicing
  - Usually implemented as a singly-linked list

Associative Containers: Set

```
template<class Key,
         class Compare = less<Key>,
         class Allocator = allocator<Key>>
class set;
```

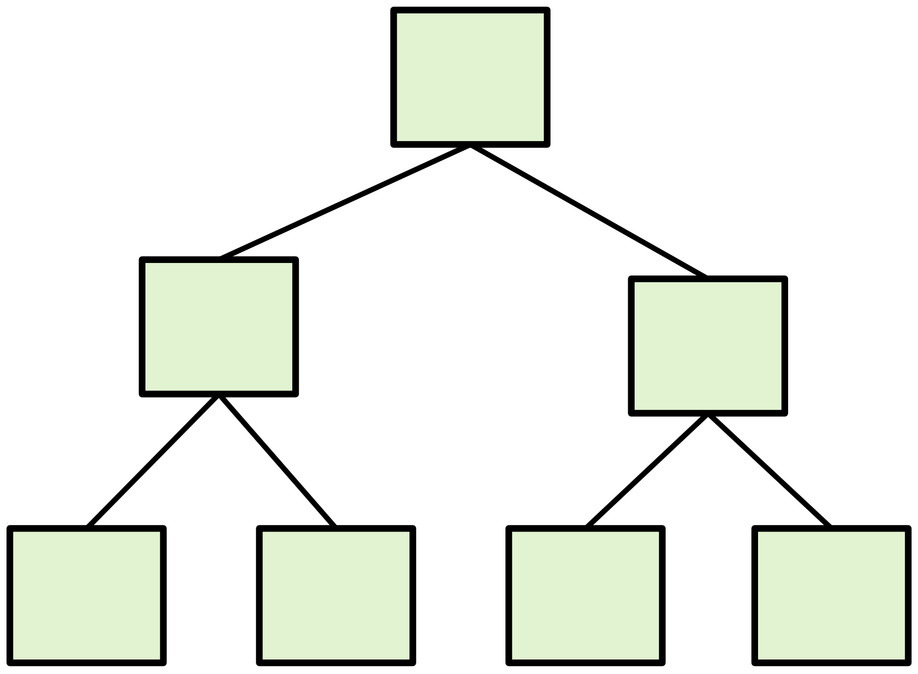


- Features
  - Supports logarithmic time element lookup
  - Elements of type `Key` are sorted according to `Compare`
  - Element values are **unique**
  - Provides const **bidirectional** iterators
  - Usually implemented as a binary search tree

Associative Container: Multiset

```
template<class Key,
         class Compare = less<Key>,
         class Allocator = allocator<Key>>
class multiset;
```

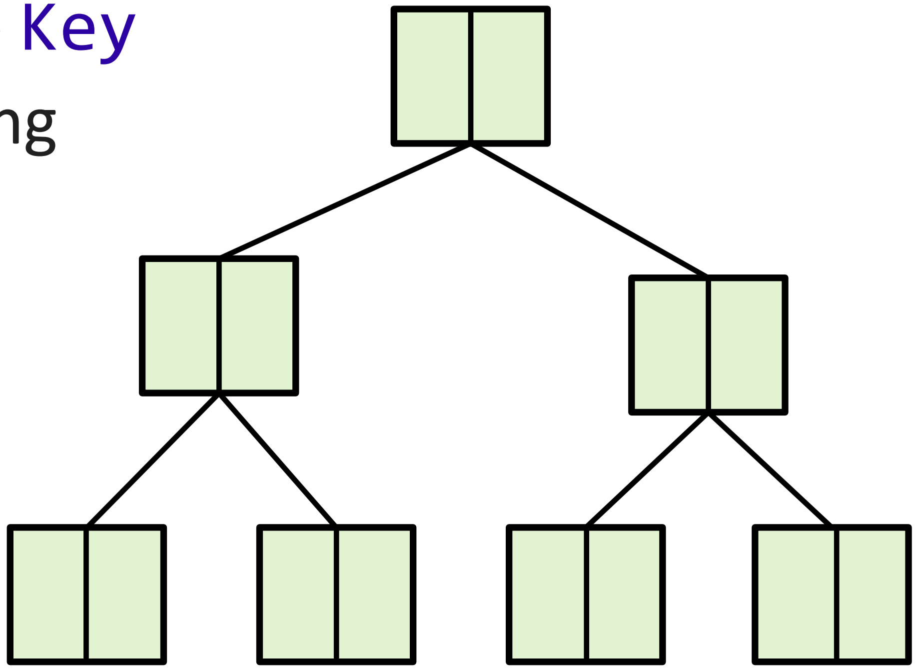
- Features
  - Supports logarithmic time element lookup
  - Elements of type `Key` are sorted according to `Compare`
  - Element values are **not unique**
  - Provides const **bidirectional** iterators
  - Usually implemented as a binary search tree



Associative Container: Map

```
template<class Key, class Val,
         class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, Val>>>
class map;
```

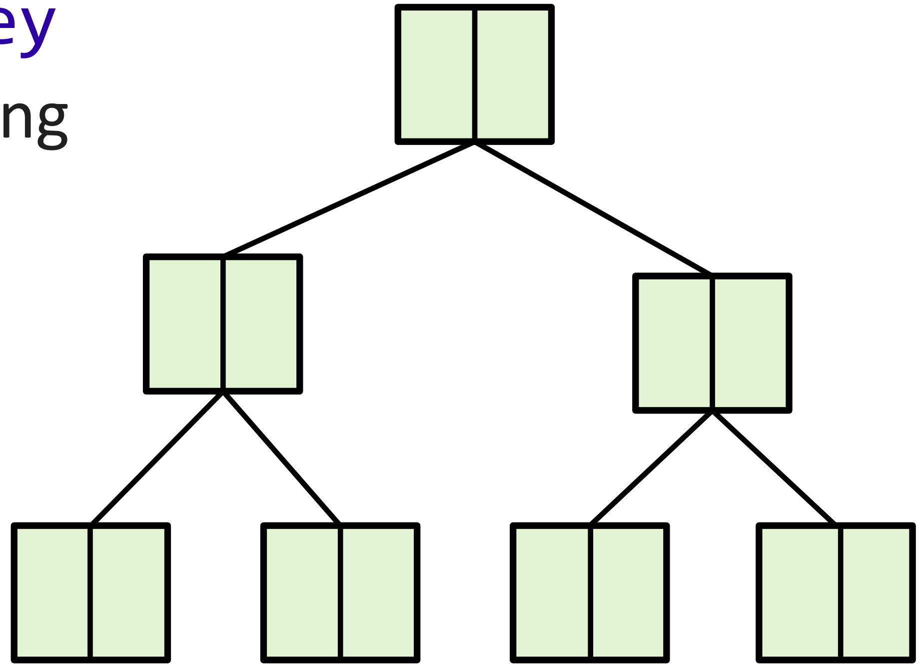
- Features
  - Supports logarithmic time lookup of a type `Val` based on a type `Key`
  - Elements of type `pair<const Key, Val>` are sorted according to `Compare`
  - Key values are **unique**
  - Provides const and mutable **bidirectional** iterators
    - Mutable iterators permit the `Val` member of `pair<const Key, Val>` to be modified
  - Usually implemented as a binary search tree
  - Can be used as an associative array



Associative Container: Multimap

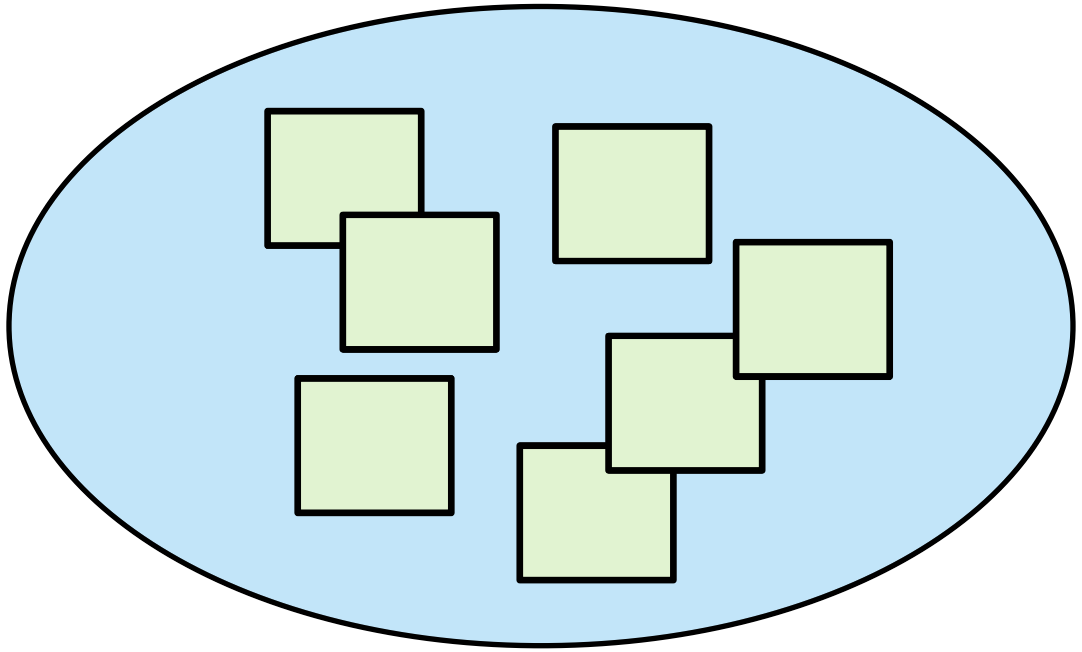
```
template<class Key, class Val,
         class Compare=less<Key>,
         class Allocator = allocator<pair<const Key, Val>>>
class multimap;
```

- Features
  - Supports logarithmic time lookup of a type `Val` based a type `Key`
  - Elements of type `pair<const Key, Val>` are sorted according to `Compare`
  - Key values are **not unique**
  - Provides const and mutable **bidirectional** iterators
    - Mutable iterators permit the `Val` member of `pair<const Key, Val>` to be modified
  - Usually implemented as a binary search tree
  - Can be used as a dictionary



Unordered Associative Container: Unordered Set

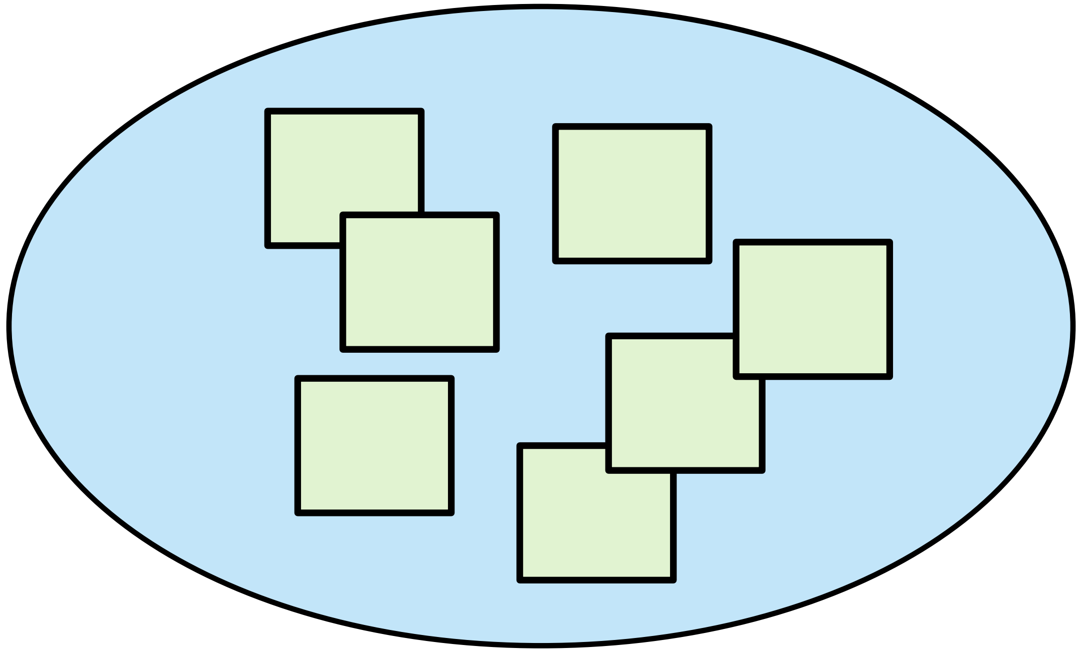
```
template<class Key,
         class Hash = hash<Key>,
         class Pred = equal_to<Key>,
         class Allocator = allocator<Key>>
class unordered_set;
```



- Features
  - Supports amortized constant time element lookup
  - Elements of type `Key` are stored internally in an order determined by Hash
  - Element values are **unique**
  - Provides const **forward** iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests `Key` equivalence

Unordered Associative Container: Unordered Multiset

```
template<class Key,
         class Hash = hash<Key>,
         class Pred = equal_to<Key>,
         class Allocator = allocator<Key>>
class unordered_multiset;
```

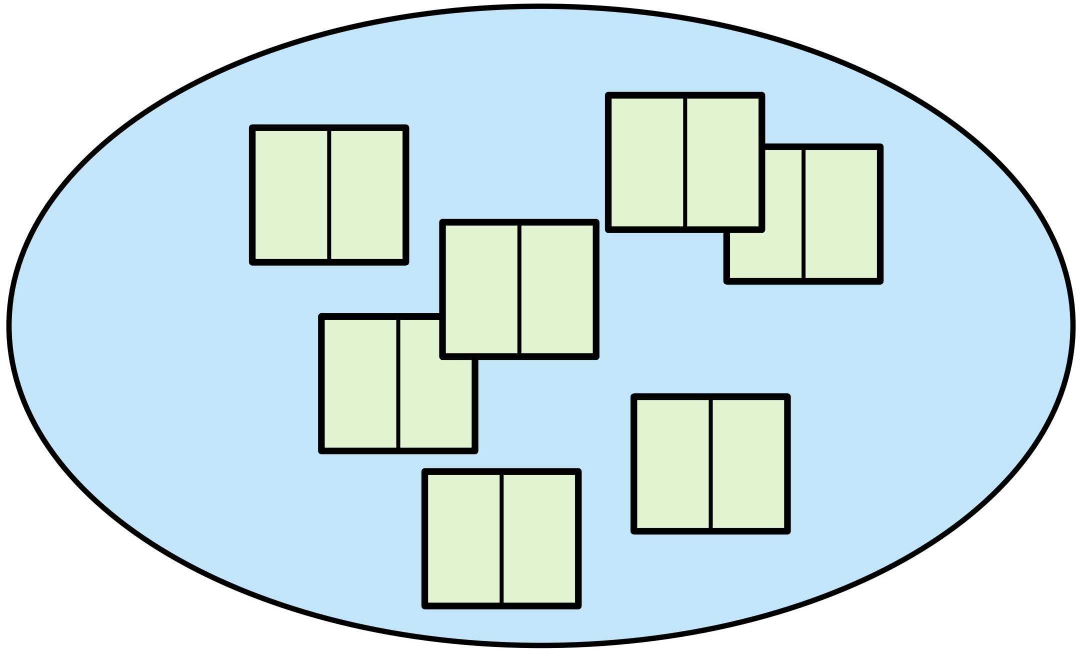


- Features
  - Supports amortized constant time element lookup
  - Elements of type `Key` are stored internally in an order determined by Hash
  - Element values are **not unique**
  - Provides const **forward** iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests `Key` equivalence

Unordered Associative Container: Unordered Map

```
template<class Key, class Val,
         class Hash = hash<Key>,
         class Pred = equal_to<Key>,
         class Allocator = allocator<pair<const Key, Val>>>
class unordered_map;
```

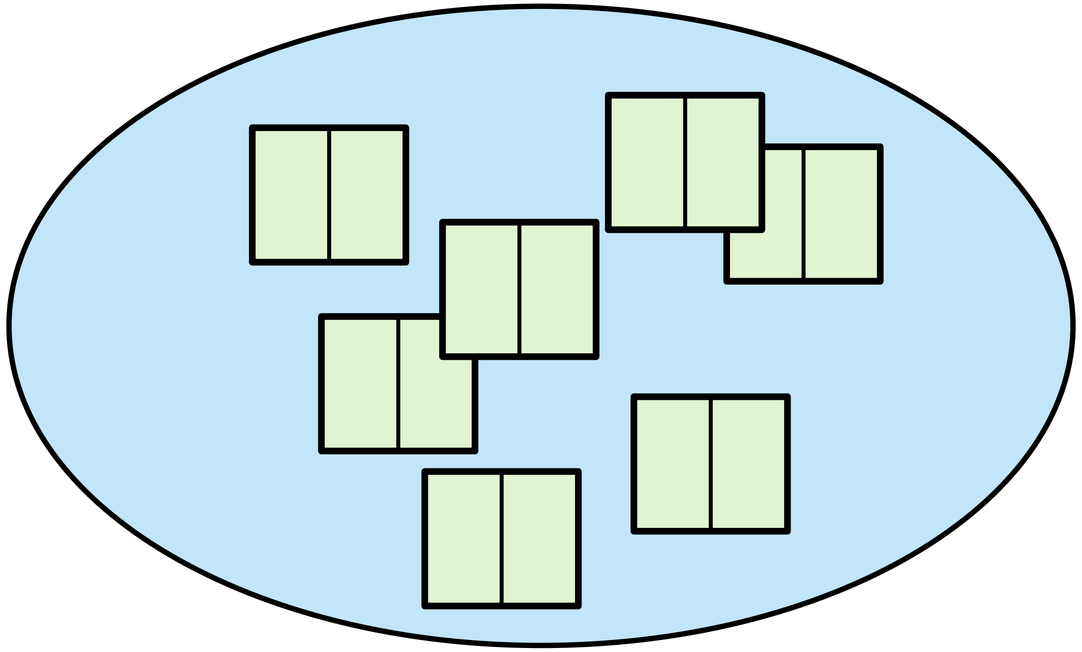
- Features
  - Supports amortized constant time lookup of a type `Val` based on a type `Key`
  - Elements are of type `pair<const Key, Val>`
  - `Key` values are **unique**
  - Provides const and mutable **forward** iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests `Key` equivalence
  - Can be used as an associative array



Unordered Associative Container: Unordered Multimap

```
template<class Key, class Val,
         class Hash = hash<Key>,
         class Pred = equal_to<Key>,
         class Allocator = allocator<pair<const Key, Val>>>
class unordered_multimap;
```

- Features
  - Supports amortized constant time lookup of a type `Val` based on a type `Key`
  - Elements are of type `pair<const Key, Val>`
  - `Key` values are **not unique**
  - Provides const and mutable **forward** iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests `Key` equivalence
  - Can be used as a dictionary



Container Adaptor: Stack

```
template<class T, class Container = deque<T>>
class stack;
```

- Features
  - Wrapper type that implements a classic LIFO stack
  - Amortized constant time `push()` and `pop()` operations
  - Constant time access to next element with `top()`
  - Works with `vector`, `deque`, `list`, and `forward_list`
- Requirements from `Container`
  - Amortized constant time `push_back()` and `pop_back()` member functions
  - Constant time `back()` member function

Container Adaptor: Queue

```
template<class T, class Container = deque<T>>
class queue;
```

- Features
  - Wrapper type that implements a classic FIFO queue
  - Amortized constant time `push()` and `pop()` operations
  - Constant time access to next element with `front()` and last element with `back()`
  - Works with `vector`, `deque`, `list`, and `forward_list`

- Requirements from `Container`
  - Amortized constant time `push_back()` and `pop_front()` member functions
  - Constant time `front()` and `back()` member functions

Container Adaptor: Priority Queue

```
template<class T, class Container = deque<T>>
class priority_queue;
```

- Features
  - Wrapper type that implements a classic priority queue (AKA heap)
  - Logarithmic time `push()` and `pop()` operations
  - Constant time access to next element with `top()`
- Requirements from `Container`
  - Amortized constant time `push_back()` and `pop_back()` member functions
  - Constant time `front()` member function
  - Random-access iterators (works with `vector` and `deque`)