Back to Basics: Classic STL

Bob Steagall
CppCon 2021

KEWB COMPUTING

Overview

- Rationale
- History and design overview
- Iterators
- Containers
- Algorithms

Copyright © 2021 Bob Steagall

Goals and References

- Goals
  - Understand overall STL design
  - Understand iterators

- Recommended references
  - *The Standard C++ Library,* Second Edition

    Nicolai M. Josuttis – Addison-Wesley 2012

  - *Effective STL*

    Scott Meyers – O'Reilly 2001

  - *Programming: Principles and Practice Using C++*, Second Edition

    Bjarne Stroustrup – Addison-Wesley 2014

  - cppreference.com

Copyright © 2021 Bob Steagall

What is "Classic STL?"

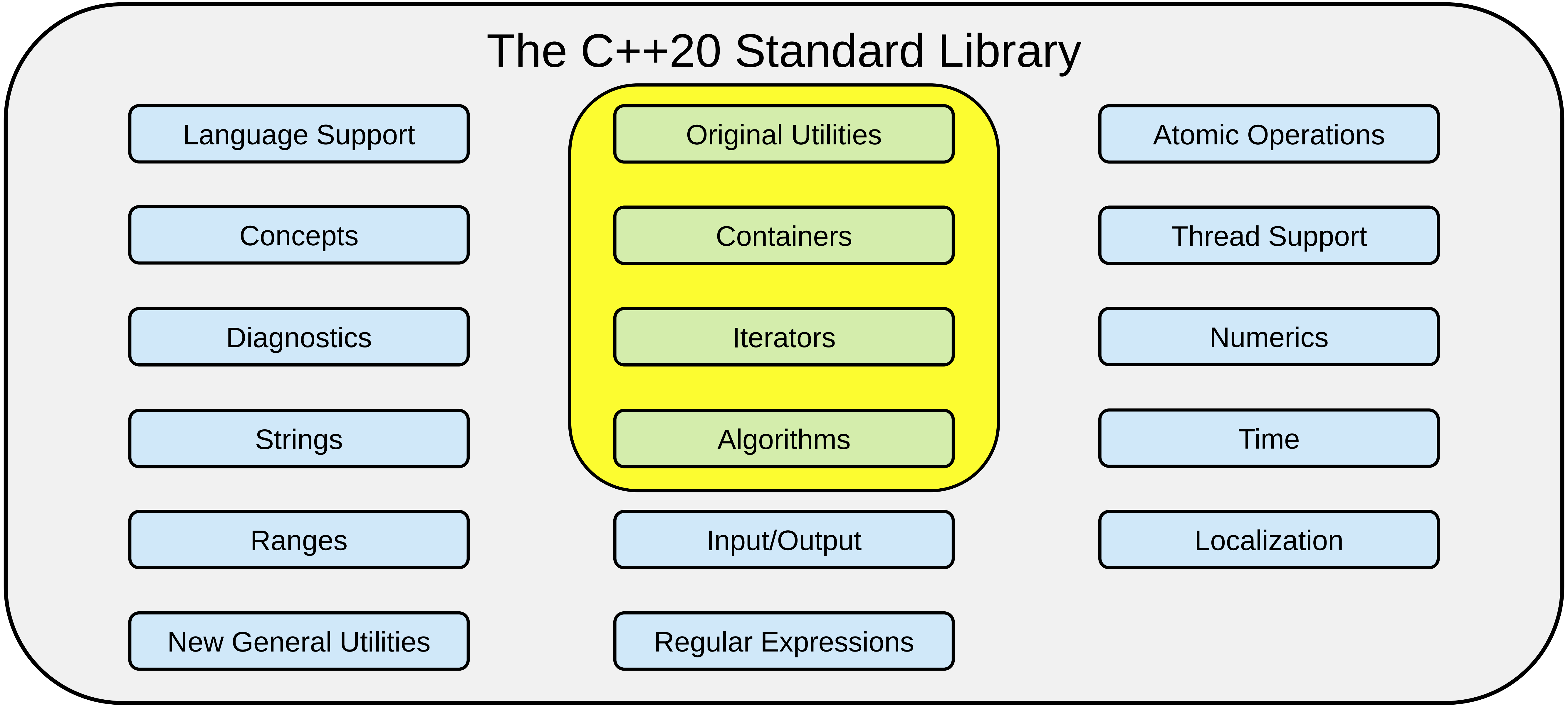
The C++20 Standard Library

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

Copyright © 2021 Bob Steagall

What is "Classic STL?"

- The short answer – containers + iterators + algorithms + some utilities



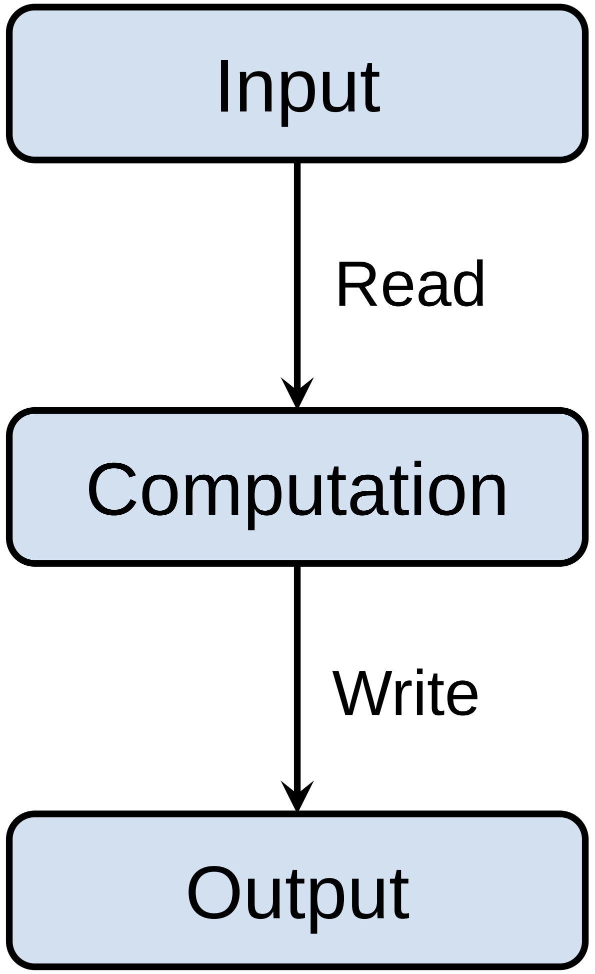
Copyright © 2021 Bob Steagall

Rationale

Copyright © 2021 Bob Steagall

Rationale

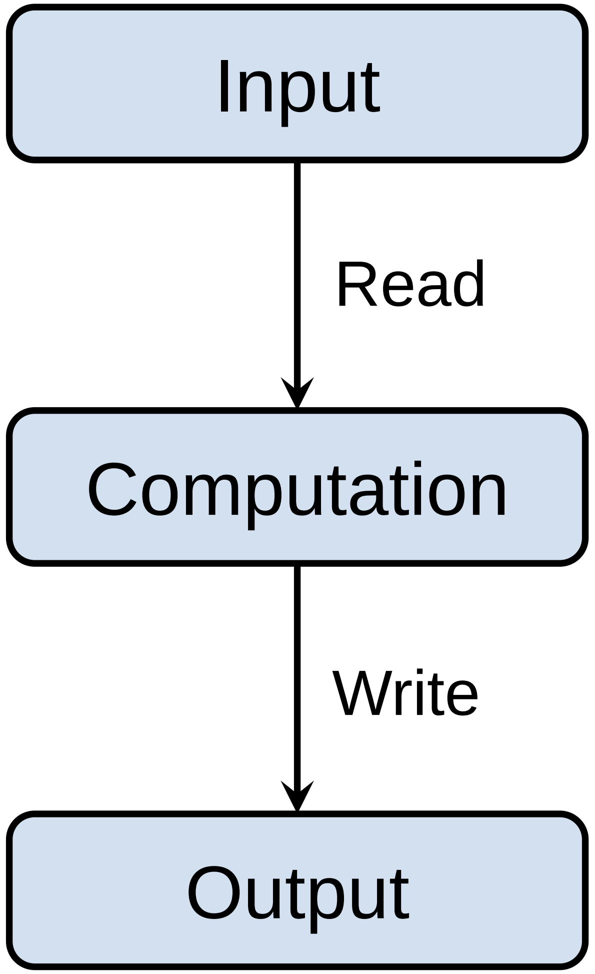
- We have some business problem to solve
- We begin with input data
- We read that data and perform computations
- We generate and write some desired output



Copyright © 2021 Bob Steagall

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**



Copyright © 2021 Bob Steagall

History and Overview of the STL

Copyright © 2021 Bob Steagall

A Brief STL History

- 1979, Alexander Stepanov begins exploring generic programming (GP)
- 1988, Stepanov and David Musser publish *Generic Programming*

Copyright © 2021 Bob Steagall

Generic Programming

Generic programming centers around the idea of **abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations** to produce a wide variety of useful software.

— David Musser, Alexander Stepanov
Generic Programming (1988)
[emphasis mine]

Copyright © 2021 Bob Steagall

Generic Programming

Following Stepanov, we can define generic programming without mentioning language features: **Lift algorithms and data structures from concrete examples to their most general and abstract form**.

— Bjarne Stroustrup
Evolving a language in and for the real world: C++ 1991-2006 (2007)
[emphasis mine]

Copyright © 2021 Bob Steagall

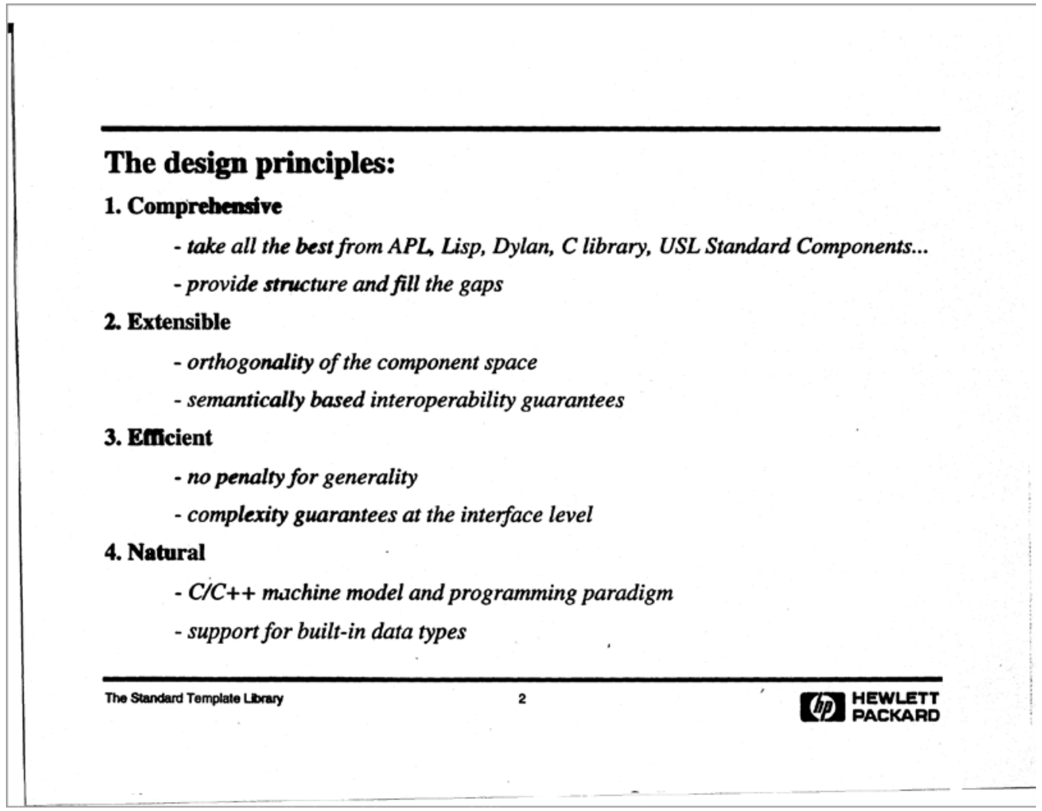
A Brief STL History

- 1979, Alexander Stepanov begins exploring generic programming (GP)
- 1988, Stepanov and David Musser publish *Generic Programming*
- 1992, Meng Lee joins Stepanov at HP Research Labs, where his team is experimenting with C and C++
- 1993, Stepanov presents the main ideas at the November WG21 meeting
- 1994, Stepanov and Lee create proposal for WG21 that was accepted later that year
- 1994-1998, much additional work; adding the original associative containers
- 1998, first ISO C++ Standard published
- 2011, C++11 is published, and with some new containers

Copyright © 2021 Bob Steagall

Original Design Principles

- Comprehensive
  - Take all the best from APL, Lisp, Dylan, C library, USL Standard Components…
  - Provide structure and fill the gaps
- Extensible
  - Orthogonality of the component space
  - Semantically based interoperability guarantees
- Efficient
  - No penalty for generality
  - Complexity guarantees at the interface level
- Natural
  - C/C++ machine model and programming paradigm
  - Support for built-in data types



The design principles:

1. Comprehensive

- take all the best from APL, Lisp, Dylan, C library, USL Standard Components...

- provide structure and fill the gaps

2. Extensible

- orthogonality of the component space

- semantically based interoperability guarantees

3. Efficient

- no penalty for generality

- complexity guarantees at the interface level

4. Natural

- C/C++ machine model and programming paradigm

- support for built-in data types

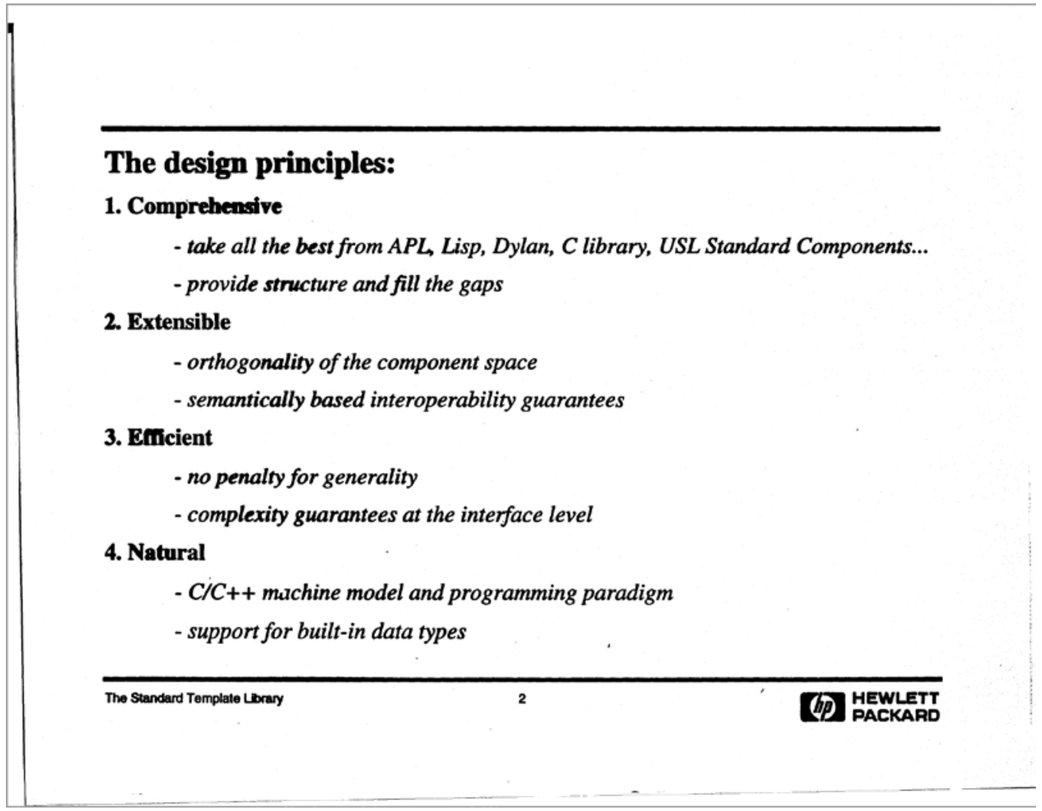
The Standard Template Library 2 HEWLETT PACKARD

Copyright © 2021 Bob Steagall

KEWB COMPUTING

Original Design Principles

- Comprehensive
  - Take all the best from APL, Lisp, Dylan, C library, USL Standard Components…
  - Provide structure and fill the gaps
- Extensible
  - Orthogonality of the component space
  - Semantically based interoperability guarantees
- Efficient
  - No penalty for generality
  - Complexity guarantees at the interface level
- Natural
  - C/C++ machine model and programming paradigm
  - Support for built-in data types



The design principles:

1. Comprehensive

- *take all the best from APL, Lisp, Dylan, C library, USL Standard Components...*
- *provide structure and fill the gaps*

2. Extensible

- *orthogonality of the component space*
- *semantically based interoperability guarantees*

3. Efficient

- *no penalty for generality*
- *complexity guarantees at the interface level*

4. Natural

- *C/C++ machine model and programming paradigm*
- *support for built-in data types*

The Standard Template Library 2 HEWLETT PACKARD

Copyright © 2021 Bob Steagall

Complexity and the Big-O Notation

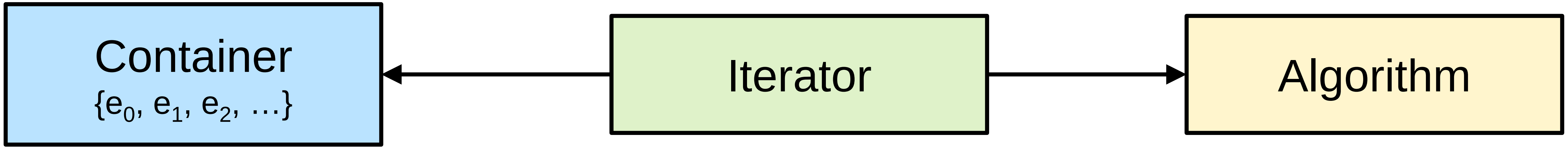
- **Complexity** refers to the *runtime cost* of an algorithm
- Big-O notation expresses the *relative complexity* of an algorithm

Type	Notation	Runtime Cost
Constant	$O(1)$	Independent of number of elements
Logarithmic	$O(log(n))$	Increases logarithmically with the number of elements
Linear	$O(n)$	Increases linearly with the number of elements
N-log-N	$O(n*log(n))$	Increases as a product of linear and logarithmic complexities
Quadratic	$O(n^2)$	Increases as the square of the number of elements

Copyright © 2021 Bob Steagall

Key Principles

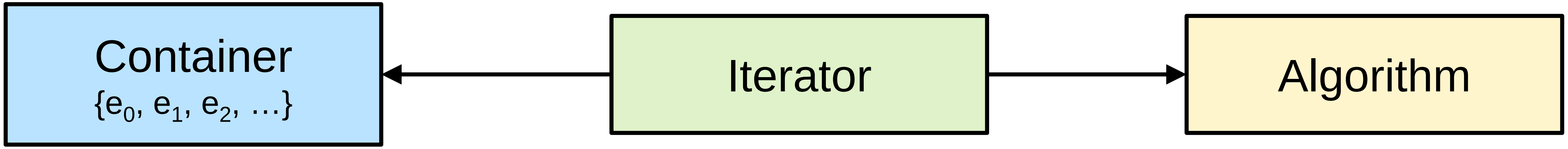
- *Containers* store *collections* of *elements*
- *Algorithms* perform operations upon collections of elements
- Containers and algorithms are entirely independent
- *Iterators* provide a common unit of information exchange between containers and algorithms



Copyright © 2021 Bob Steagall

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



Copyright © 2021 Bob Steagall

Complexity and Interfaces

- STL makes complexity guarantees by specifying *interfaces* and *requirements*

- Containers provide support for
  - Adding / removing elements
  - Accessing (reading / updating) elements via associated iterators
  - A container's iterators understand (and ***abstract***) that container's internal structure

- Iterators
  - Provide access to container elements through well-defined interfaces with strict guarantees

- Algorithms
  - Employ the well-defined interfaces provided by iterators
  - Have complexity based on the algorithm itself and the guarantees made by the iterators

Copyright © 2021 Bob Steagall

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

Copyright © 2021 Bob Steagall

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

Copyright © 2021 Bob Steagall

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

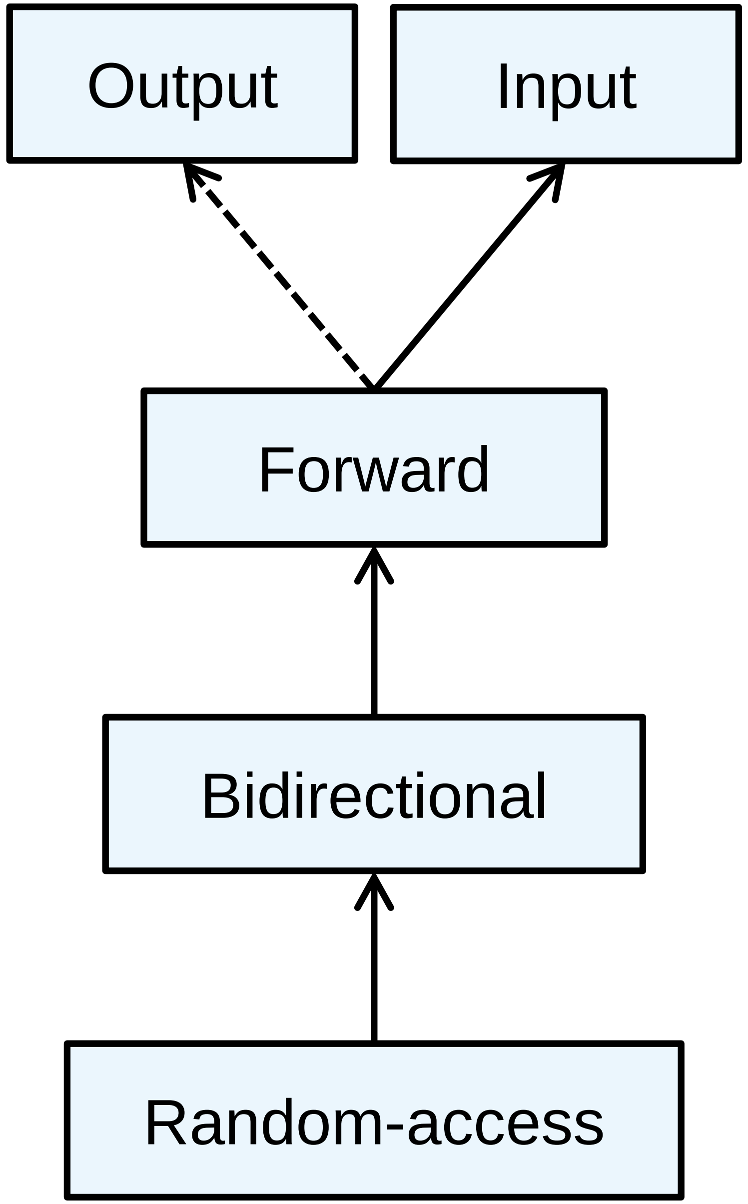
- **Iterators never own the elements to which they refer**

Copyright © 2021 Bob Steagall

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



Copyright © 2021 Bob Steagall

Algorithms Overview

- The algorithms process ranges of elements of a collection
  - Require at least one explicitly-specified iterator pair

- Algorithm categories
  - Non-modifying algorithms
  - Modifying algorithms
  - Removing algorithms
  - Mutating algorithms
  - Sorting algorithms
  - Sorted range algorithms
  - Numeric algorithms

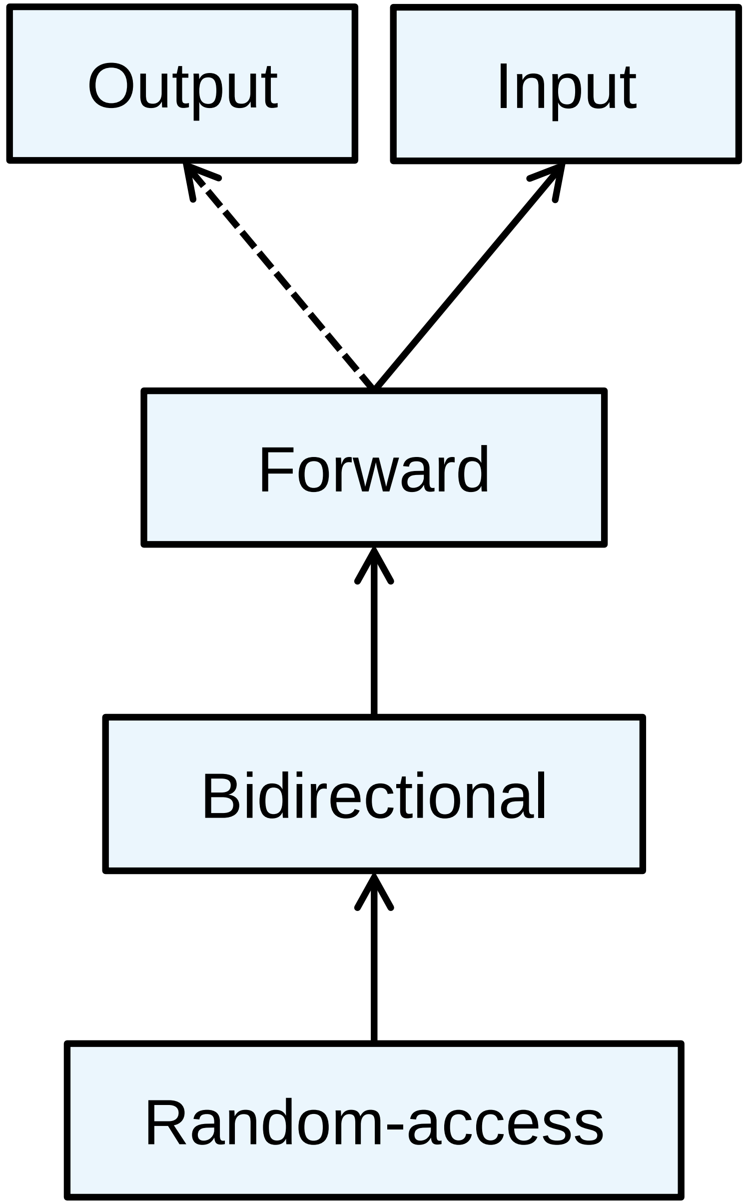
Copyright © 2021 Bob Steagall

Iterators

Copyright © 2021 Bob Steagall

Regarding Iterators

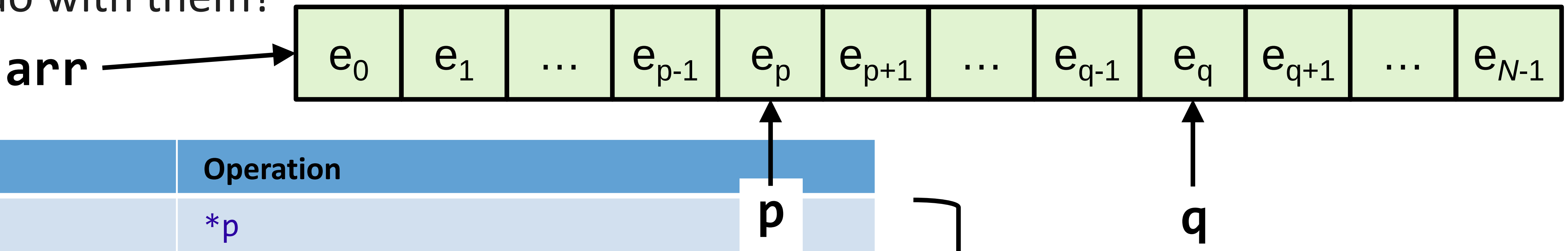
- Where do the five iterator categories come from?
- What interface does each category provide?
- What is their time complexity?
- How are they related to containers?
- How are they used by the algorithms?
- Let's try a generic programming exercise and develop iterators from scratch



Copyright © 2021 Bob Steagall

Referring to Elements in Arrays

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



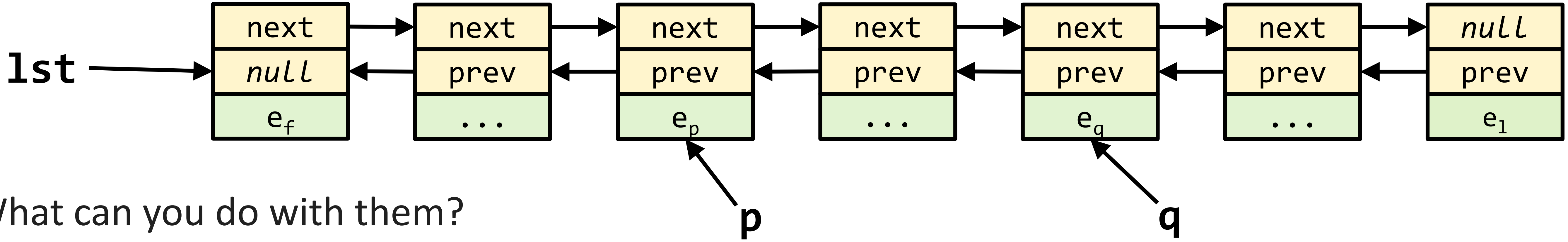
Action	Operation
Access element	`*p`
Access member of element	`p->mem`
Compare for equality of postion	`p == q,    p != q`
Move forward by 1	`++p,   p++`
Move backward by 1	`--p,   p--`
Make a copy (assign)	`q = p`
Access arbitrary element	`p[n]`
Move forward by arbitrary n	`p += n,    q = p + n`
Move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

Copyright © 2021 Bob Steagall

Referring to Elements in Doubly-Linked Lists

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



  - What can you do with them?

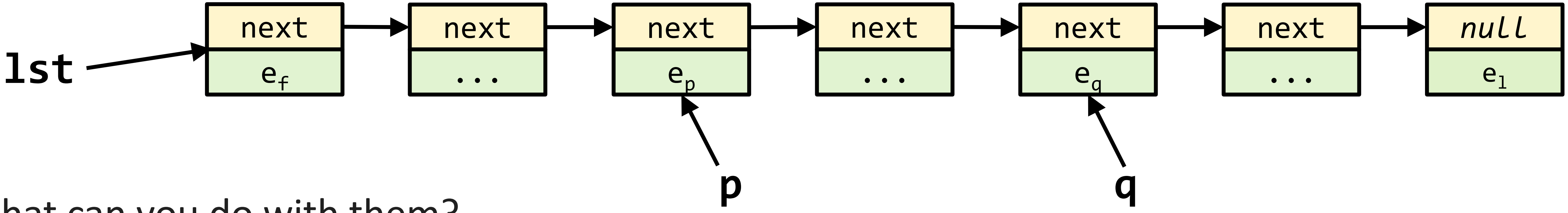
Action	Operation
Access element	`*p`
Access member of element	`p->mem`
Compare for equality of position	`p == q,    p != q`
Move forward by 1	`p = p->next`
Move backward by 1	`p = p->prev`
Make a copy (assign)	`q = p`

O(1) - constant time

Copyright © 2021 Bob Steagall

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

Copyright © 2021 Bob Steagall

Multi-Pass and Single-Pass Iteration

- Arrays, doubly-linked and singly-linked lists all support *multi-pass iteration*
  - Pointers to elements can be dereferenced more than once, with the same result each time
  - The sequence can be iterated over (traversed) more than once

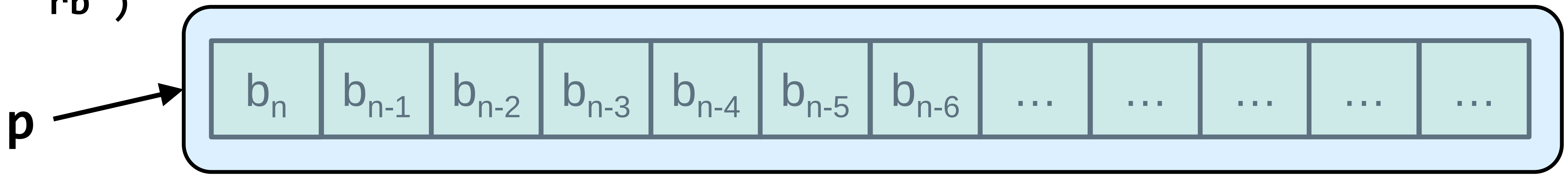
- What about sequences that can be traversed only once?
  - Some sequences support only *single-pass iteration*
  - An element can only be read from, or written to, a given position one time
  - The act of reading or writing irrevocably changes position
  - Reading from / writing to file streams, sockets, raw devices, etc.

Copyright © 2021 Bob Steagall

Reading Elements (Bytes) From a FILE Stream

- Consider a pointer to a FILE stream opened for input

```
FILE* p = fopen(name, "rb")
```



  - What can you do with it?

Action	Operation
Read element and advance	`b = fgetc(p)`
Compare for end-of-file equality	`b == EOF`, `feof(p)`
Make a copy (assign)	`q = p`

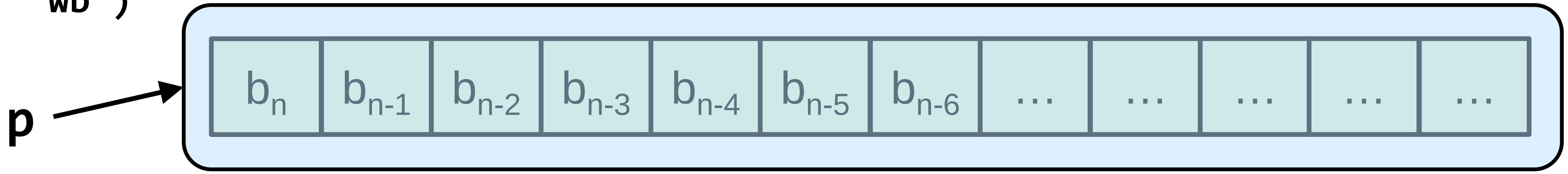
} O(1) - constant time

Copyright © 2021 Bob Steagall

Writing Elements (Bytes) To a FILE Stream

- Consider a pointer to a FILE stream opened for output

```
FILE* p = fopen(name, "wb")
```



  - What can you do with it?

Action	Operation
Write element and advance	`fputc(b, p)`
Make a copy (assign)	`q = p`

O(1) - constant time

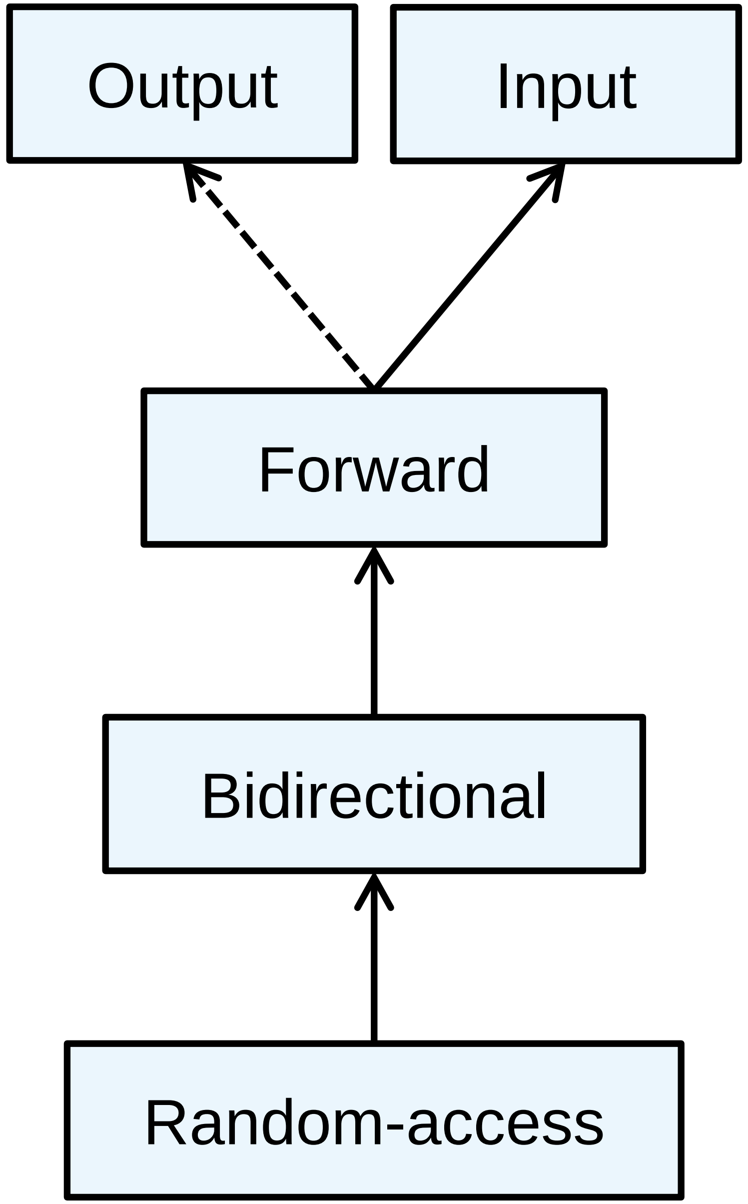
Copyright © 2021 Bob Steagall

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"

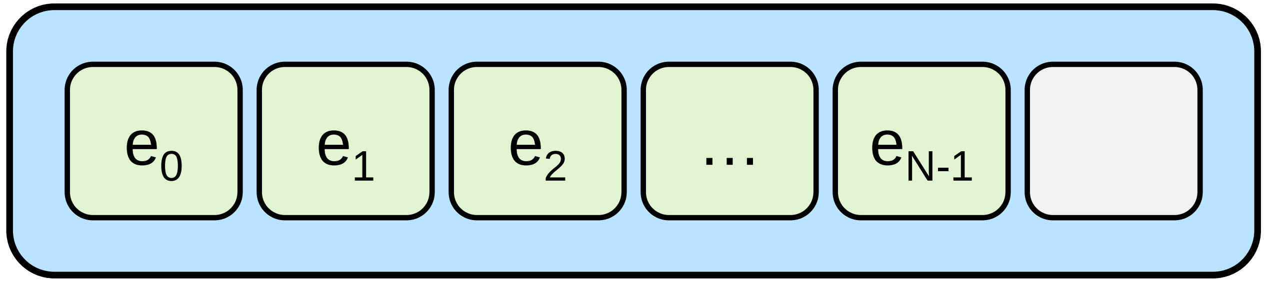
- Iterators that satisfy the requirements of output iterators are called *mutable* iterators



Copyright © 2021 Bob Steagall

Iterator Ranges

- Let's think about sequences in terms of *positions*
  - By fiat, a sequence of N elements has N+1 positions
  - The first N positions contain elements and are *dereferenceable*
  - Assume the last position contains nothing and is therefore *non-dereferenceable*
  - You can point/refer to the last position, but you cannot read from it or write to it

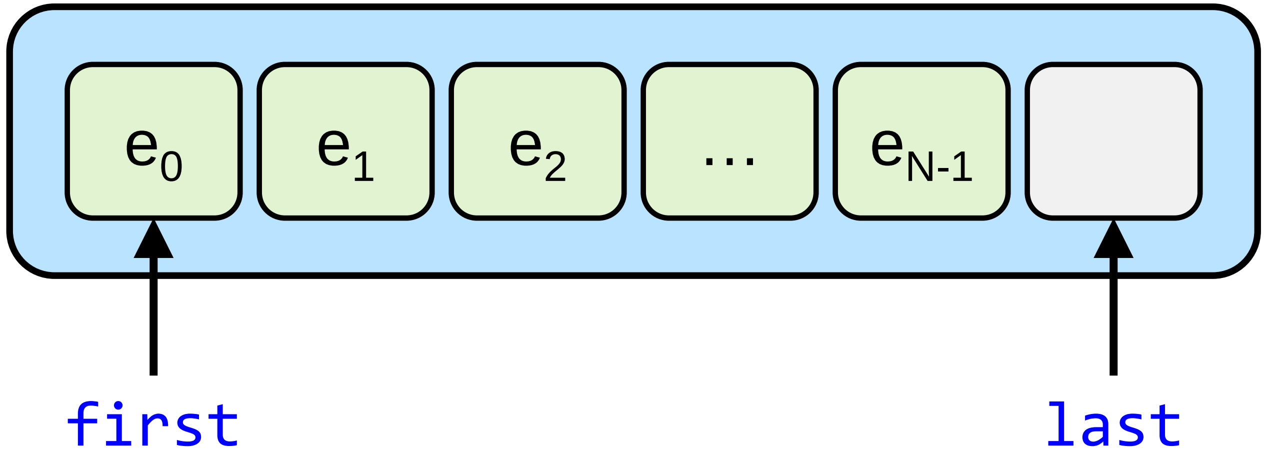


Copyright © 2021 Bob Steagall

Iterator Ranges

- In the STL, iteration over sequences is based on the idea of *iterator ranges*

- An iterator range is represented by a pair of iterators -- `[first, last)`
  - This pair represents a *half-open interval* over the sequence of elements
  - `first` refers to the first element **included** in the sequence
  - `last` refers to the non-dereferenceable, "one-past-the-end" (PTE) position **excluded** from the sequence



Copyright © 2021 Bob Steagall

Iterator Ranges

- Q: Why use ranges described by half-open intervals?

- A: It makes testing for loop termination very simple
  - Loops only need to test for iterator equality
  - Indexing not required
  - Location in memory is irrelevant

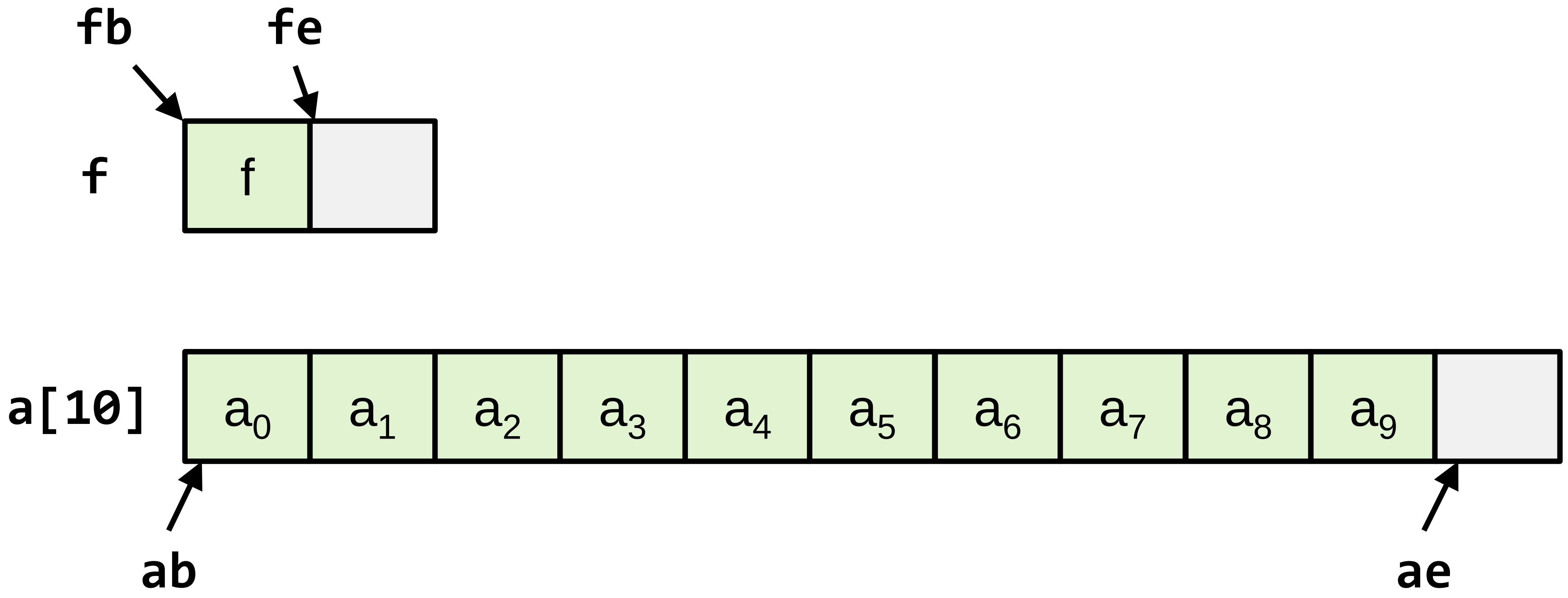
```
iterator  f = get_position_of_first_element_in_sequence();
iterator  l = get_one_past_end_position_in_sequence();

//- Works for all iterator types except OutputIterator
//
for (;  f != l;  ++f)
{
    some_function(*f);
}
```


Copyright © 2021 Bob Steagall

Iterator Ranges

- Q: How can they work?
- A: It depends on the container / sequence
  - Containers that store elements contiguously in memory rely on ability to get a pointer to the "next-position-after"

```
Foo  f;
Foo* fb = &f;
Foo* fe = pfb + 1;

Foo  a[10];
Foo* ab = &a[0];
Foo* ae = ab + 10;
```

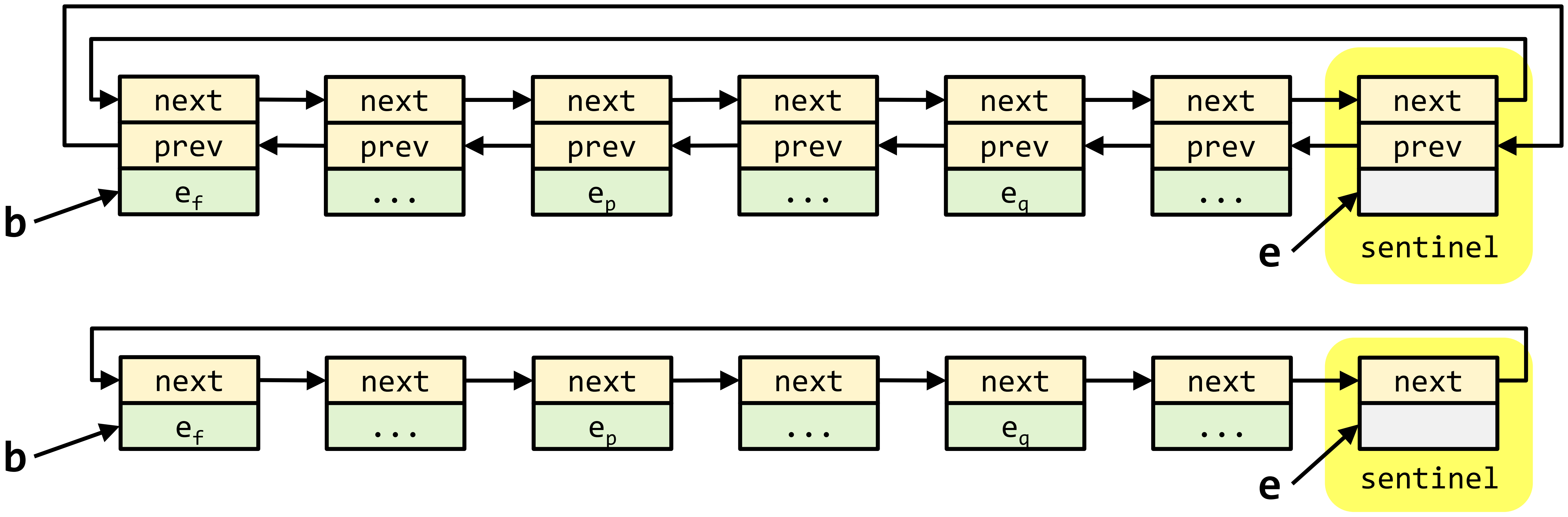


Copyright © 2021 Bob Steagall

Iterator Ranges

- Q: How can they work?

- A: It depends on the container / sequence
  - Node-based containers can use *sentinel nodes*

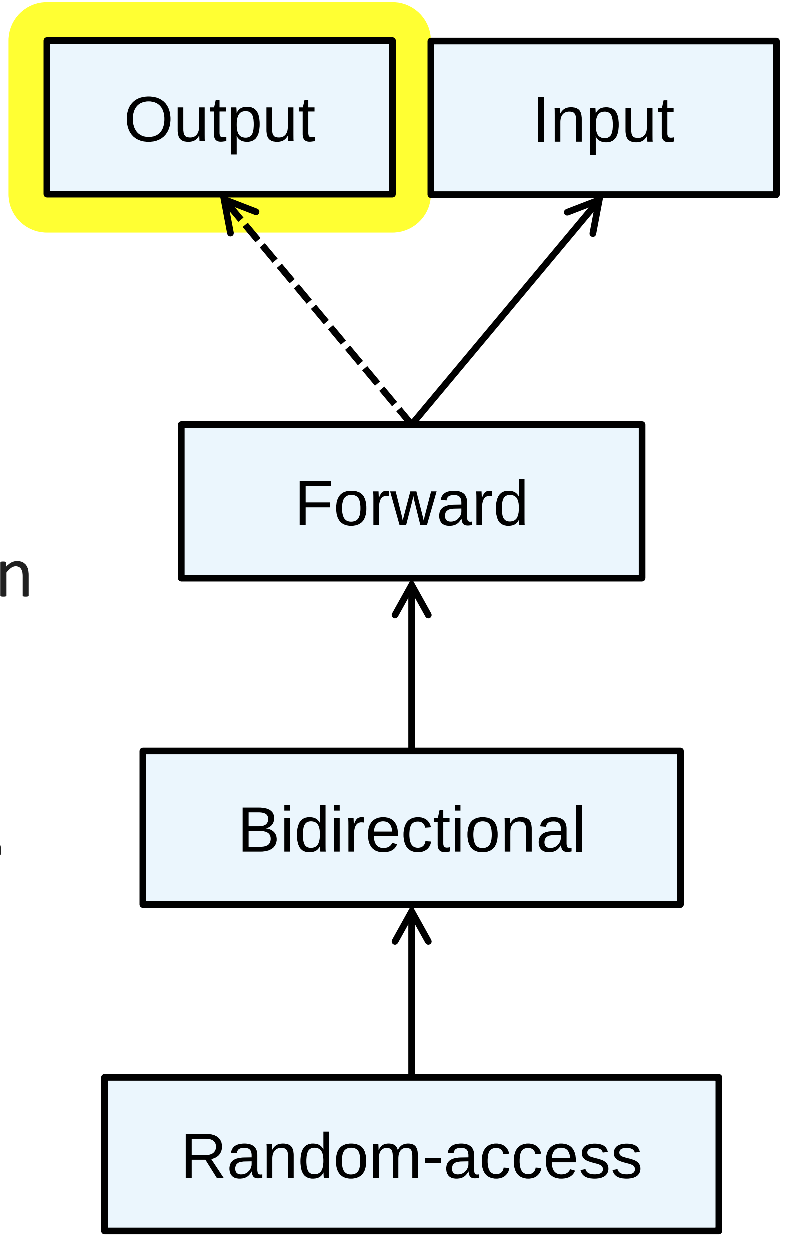


Copyright © 2021 Bob Steagall

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

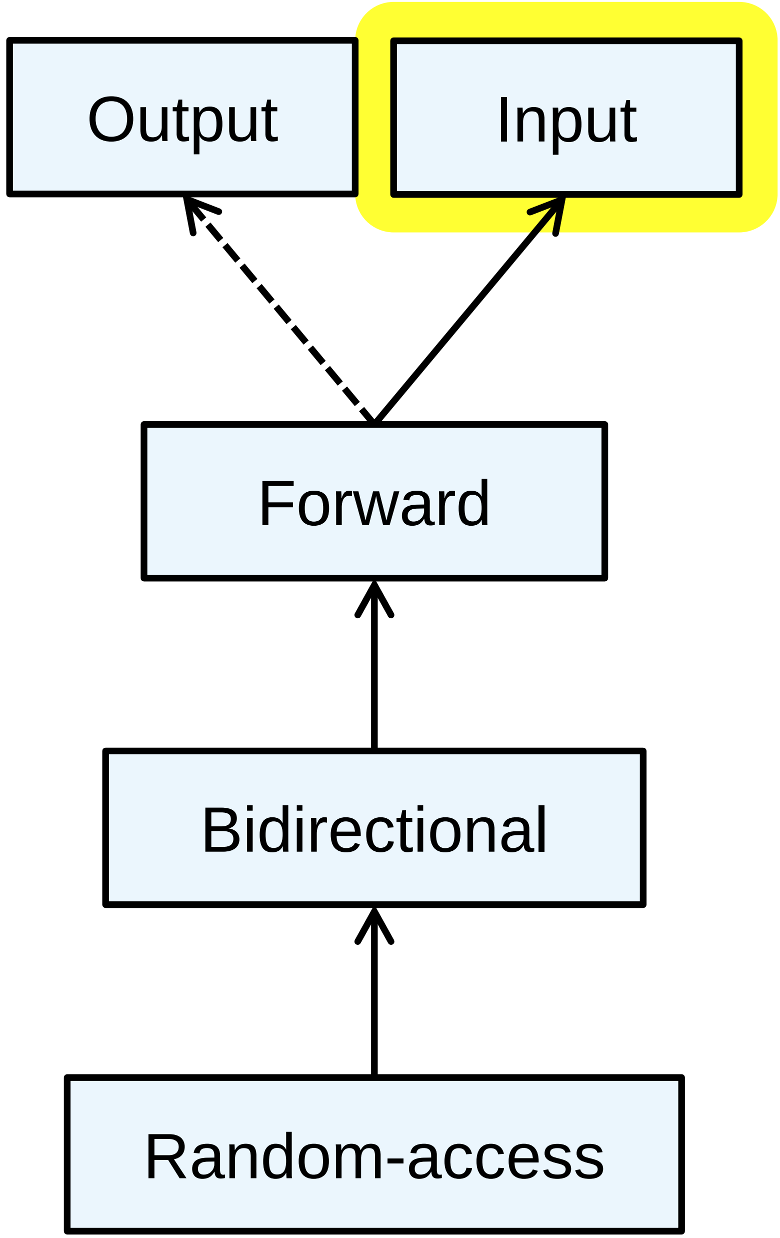


Copyright © 2021 Bob Steagall

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

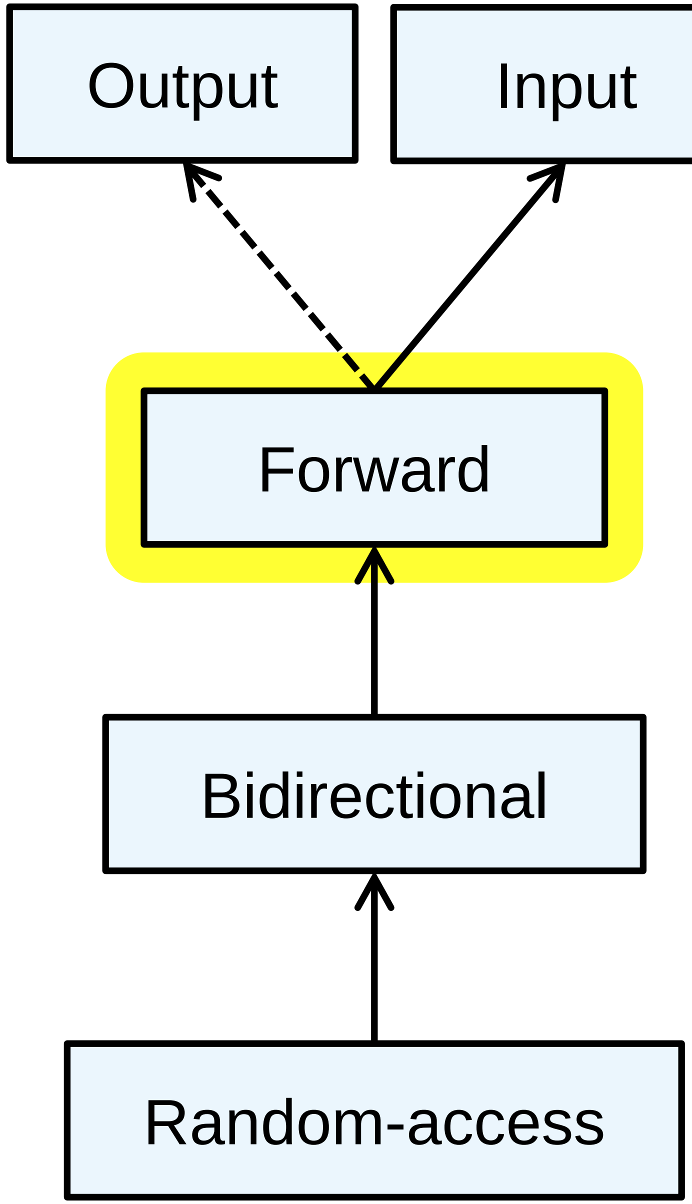


Copyright © 2021 Bob Steagall

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

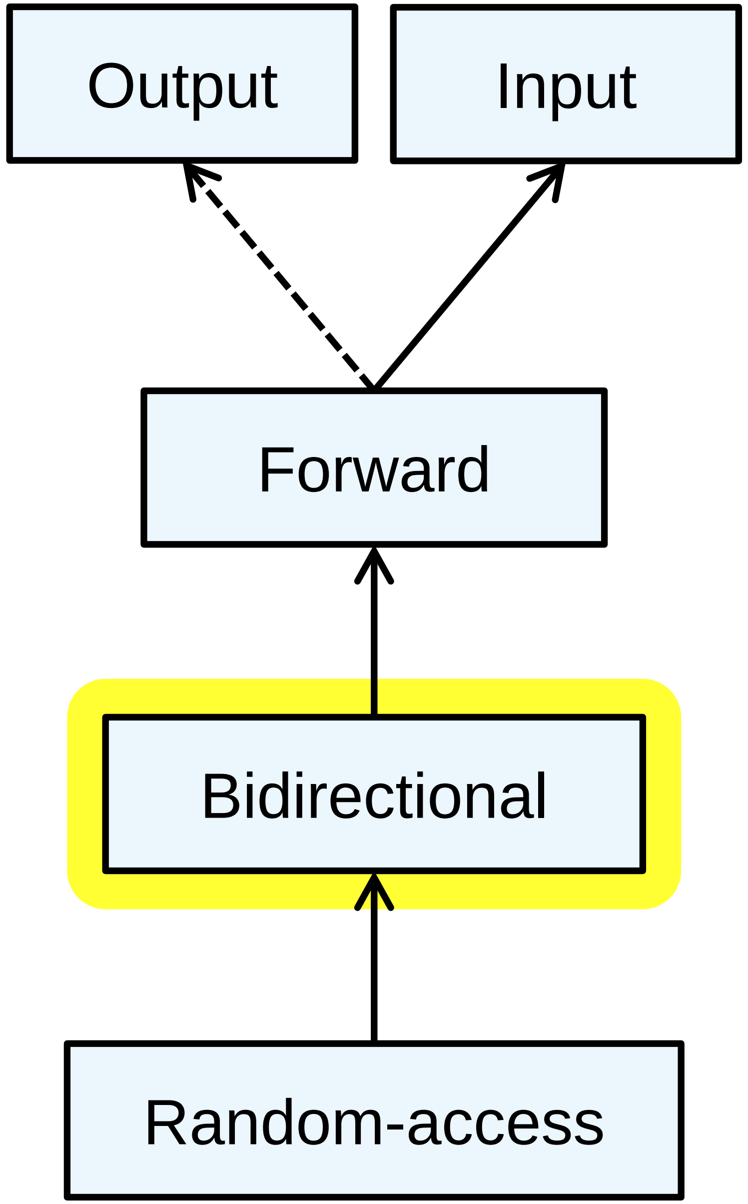


Copyright © 2021 Bob Steagall

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

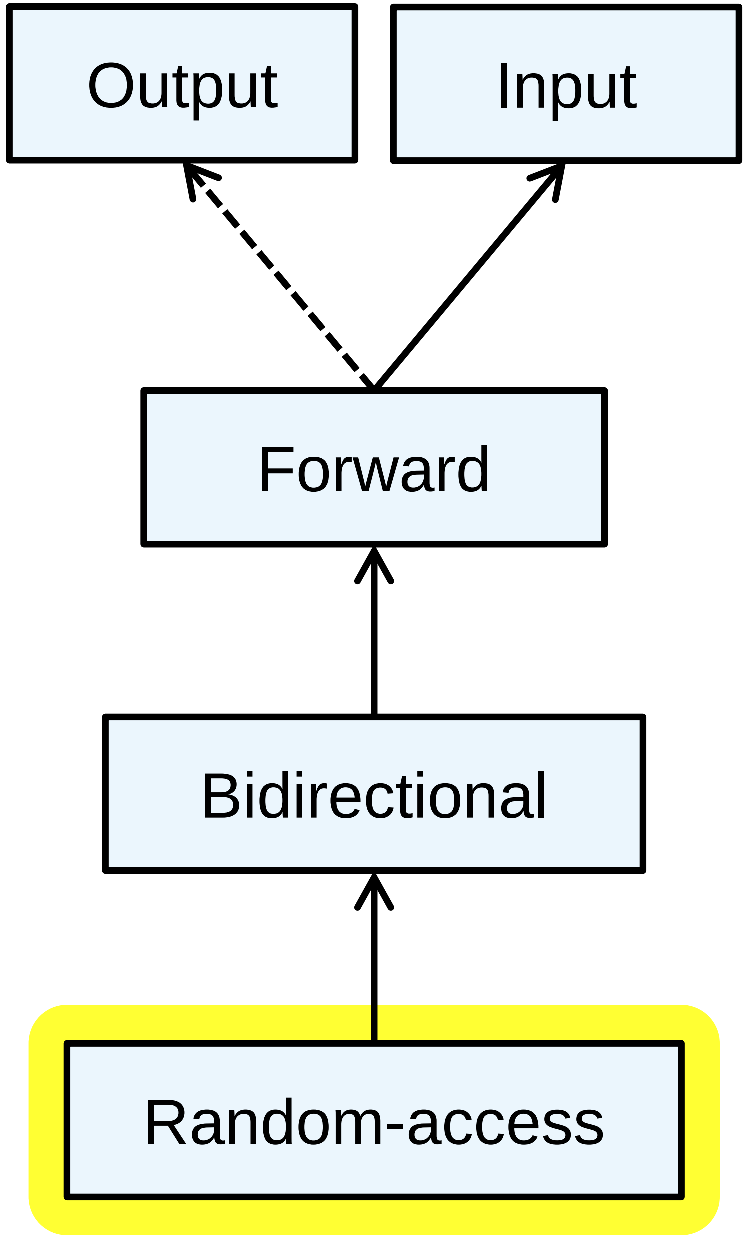
- Additional capabilities and guarantees
  - `p == q` implies `--p == --q`
  - `--(++p) == p`



Copyright © 2021 Bob Steagall

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

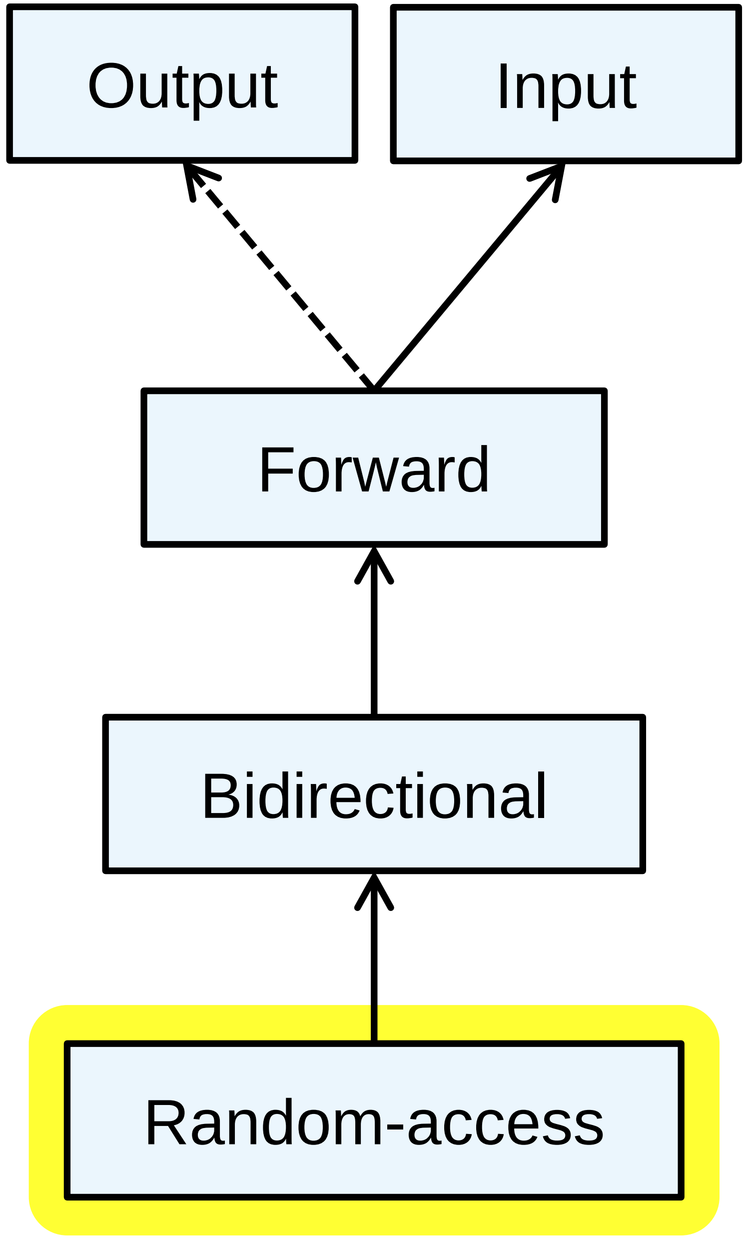


- Additional capabilities and guarantees
  - Emulate pointers
  - Provide operators for iterator arithmetic, analogous to pointer arithmetic
  - Provide relational operators to compare position

Copyright © 2021 Bob Steagall

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

Copyright © 2021 Bob Steagall

Iterator Adaptors

- Reverse iterators
  - `template<class Iter> reverse_iterator;`
  - Iterates backward from the end of a sequence to the beginning
  - Models a bidirectional iterator when `Iter` is bidirectional
  - Models a random-access iterator when `Iter` is random-access

- Insert iterators (inserters)
  - `template<class Container> back_insert_iterator;`
  - `template<class Container> front_insert_iterator;`
  - `template<class Container> insert_iterator;`
  - Models an output iterator that inserts elements at the back / front / interior of a container

Copyright © 2021 Bob Steagall

Containers

Copyright © 2021 Bob Steagall

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`

Copyright © 2021 Bob Steagall

Common Container Interface

- Every STL container provides a common set of nested type aliases

```
template< ... >
class container
{
    ...

    using value_type      = ...
    using reference       = ...
    using const_reference = ...

    using iterator        = ...
    using const_iterator  = ...
    using size_type       = ...
    using difference_type = ...
    ...
```


Copyright © 2021 Bob Steagall

Common Container Interface

- Every STL container provides a common set of functions

```
template< ... >
class container
{
    ...

    iterator        begin();
    iterator        end();

    const_iterator  begin() const;
    const_iterator  end() const;

    const_iterator  cbegin() const;
    const_iterator  cend() const;
    ...
```


Copyright © 2021 Bob Steagall

Common Bidirectional Container Interface

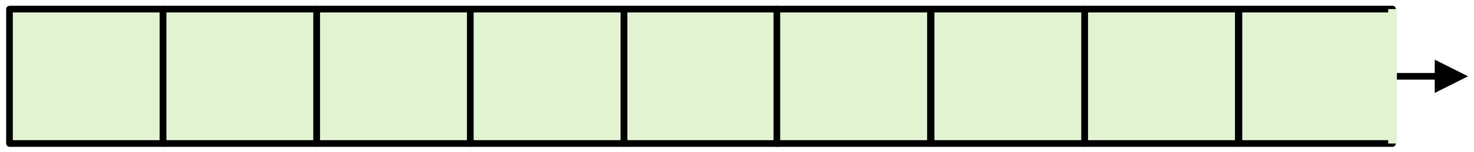
- Bidirectional containers provide additional aliases and functions

```
template< ... >
class bidirectional_container
{
    ...

    using reverse_iterator       = ...
    using const_reverse_iterator = ...

    reverse_iterator        rbegin();
    reverse_iterator        rend();

    const_reverse_iterator  rbegin() const;
    const_reverse_iterator  rend() const;

    const_reverse_iterator  crbegin() const;
    const_reverse_iterator  crend() const;
    ...
```


Copyright © 2021 Bob Steagall

Sequence Container: Vector

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

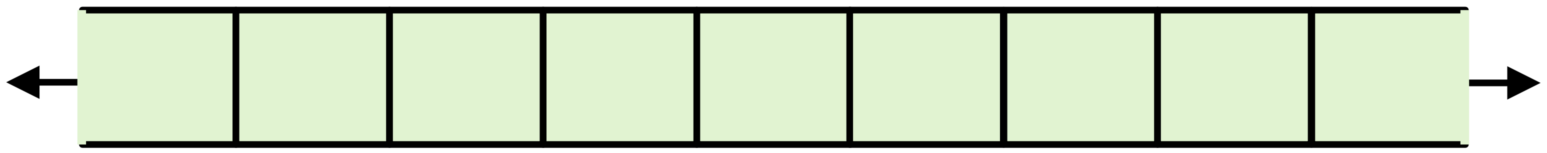


- Features
  - Supports amortized constant time insert and erase operations at its end
  - Supports linear time insert and erase operations in its middle
  - Provides const and mutable **random-access** iterators
  - Provides const and mutable element indexing
  - Supports changing element values
  -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

Copyright © 2021 Bob Steagall

Sequence Container: Deque

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

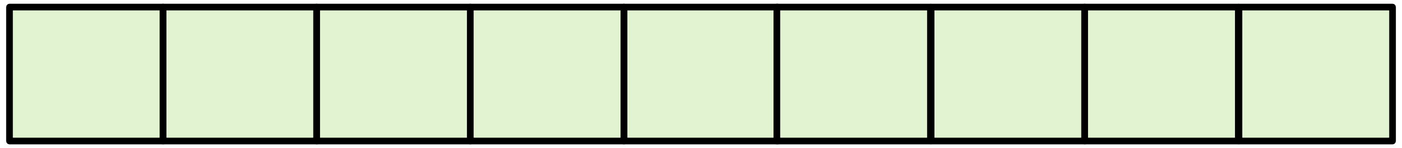


- Features
  - Supports amortized constant time insert and erase operations at both ends
  - Supports linear time insert and erase operations in its middle
  - Provides const and mutable **random-access** iterators
  - Provides const and mutable element indexing
  - Supports changing element values

Copyright © 2021 Bob Steagall

Sequence Container: Array

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

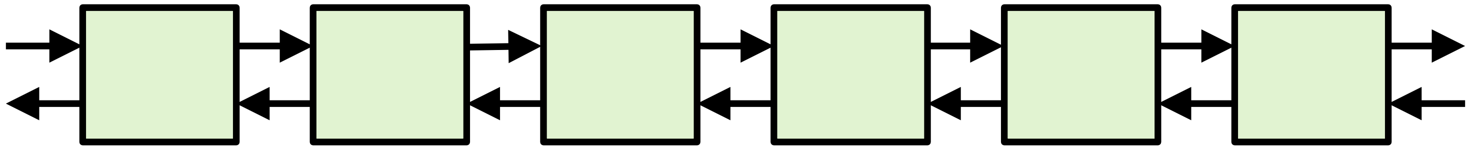


- Features
  - Manages a fixed-sized sequence of objects in an internal C-style array
  - Provides const and mutable **random-access** iterators
  - Provides const and mutable element indexing
  - Supports changing element values
  - Uses contiguous storage for all element types

Copyright © 2021 Bob Steagall

Sequence Container: List

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

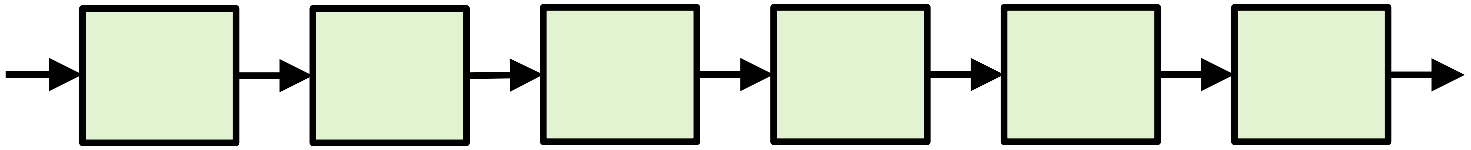


- Features
  - Supports constant time insert and erase operations anywhere in the sequence
  - Provides const and mutable **bidirectional** iterators
  - Supports changing element values
  - Provides member functions for splicing, sorting, and merging
  - Usually implemented as a doubly-linked list

Copyright © 2021 Bob Steagall

Sequence Container: Forward List

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



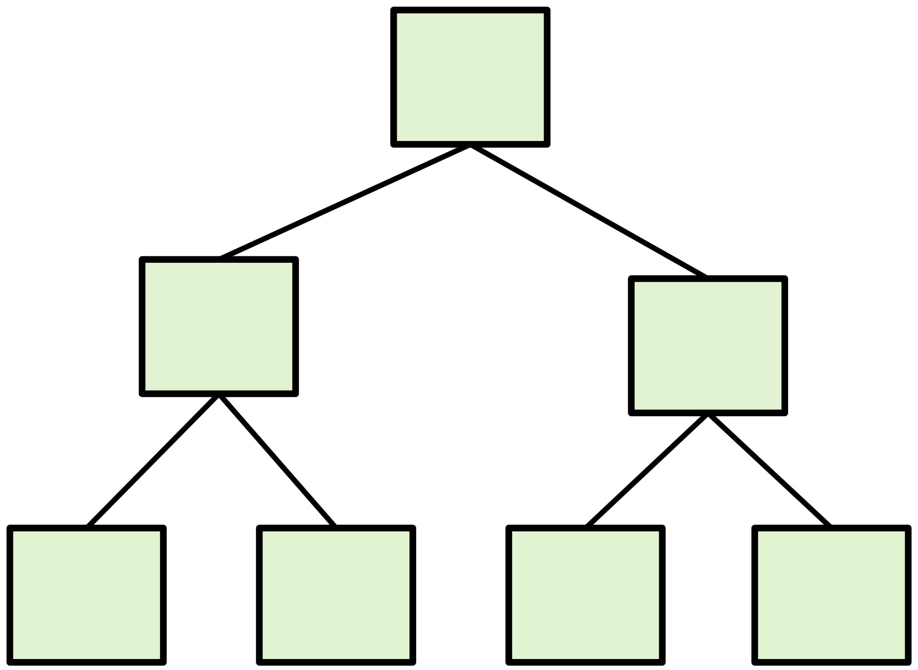
- Features
  - Supports constant time insert and erase operations anywhere in the sequence
  - Provides const and mutable **forward** iterators
  - Supports changing element values
  - Provides member functions for splicing
  - Usually implemented as a singly-linked list

Copyright © 2021 Bob Steagall

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

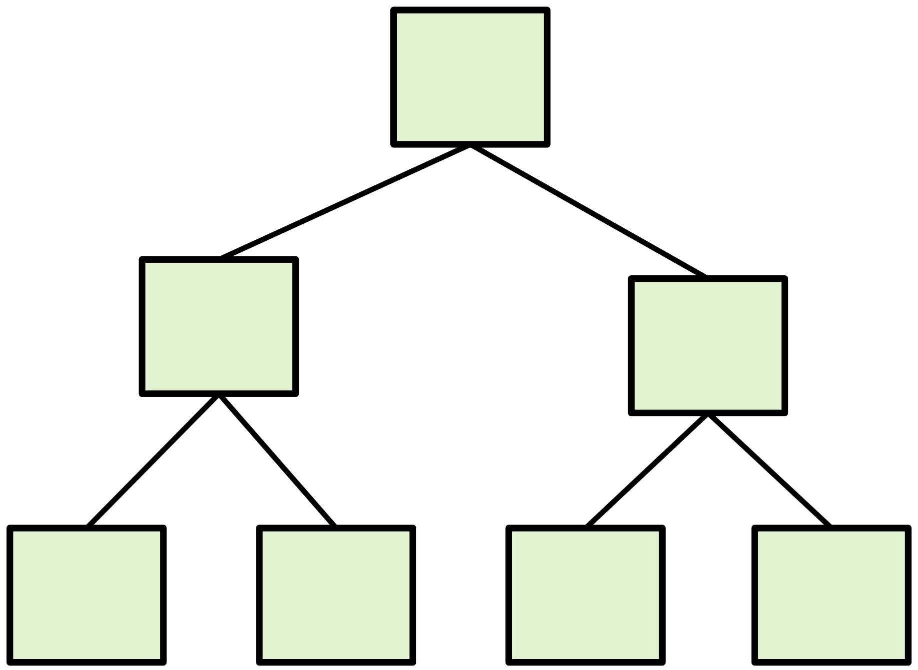


Copyright © 2021 Bob Steagall

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

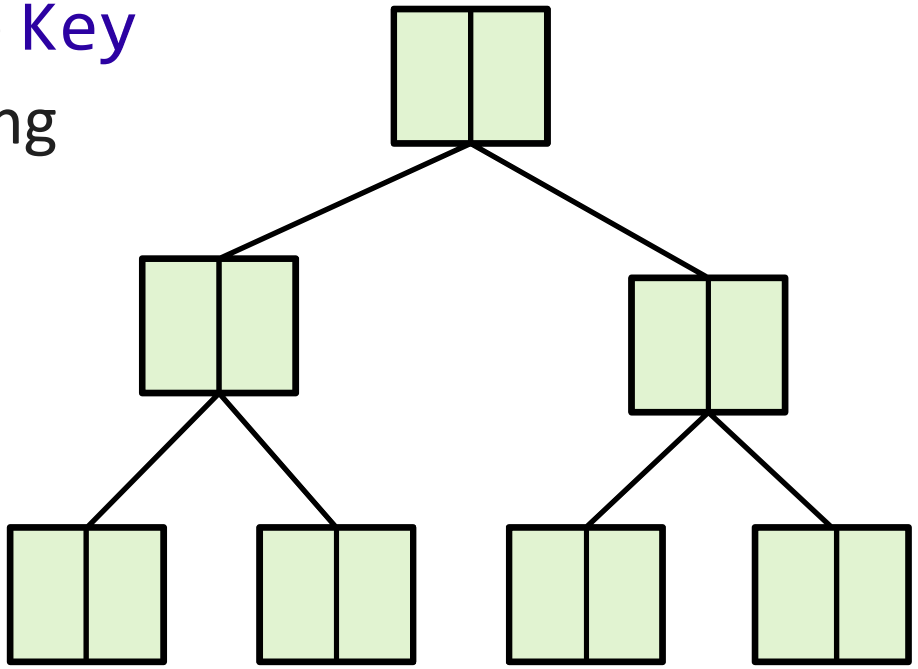


Copyright © 2021 Bob Steagall

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

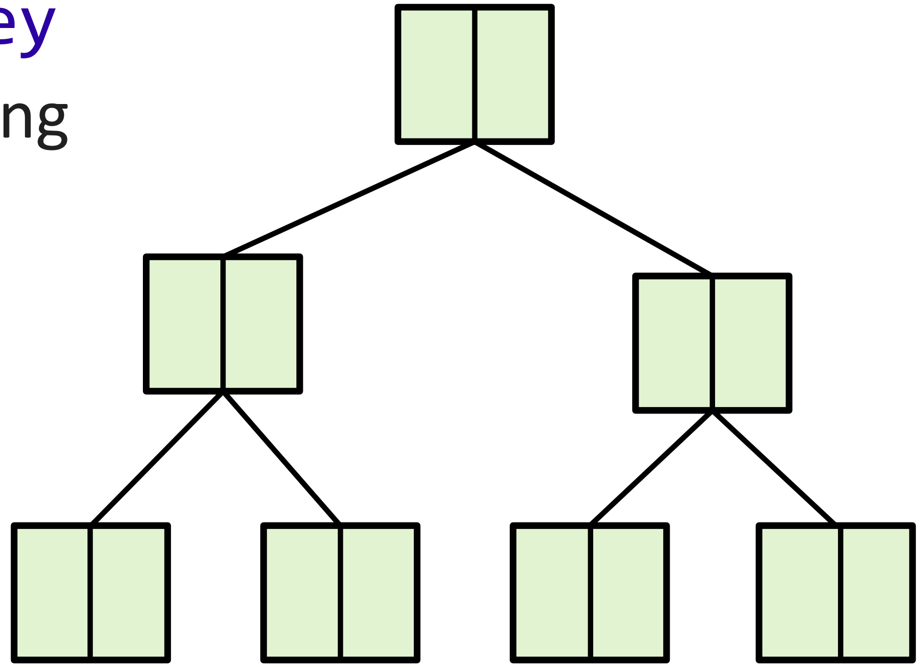


Copyright © 2021 Bob Steagall

Associative Container: Multimap

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

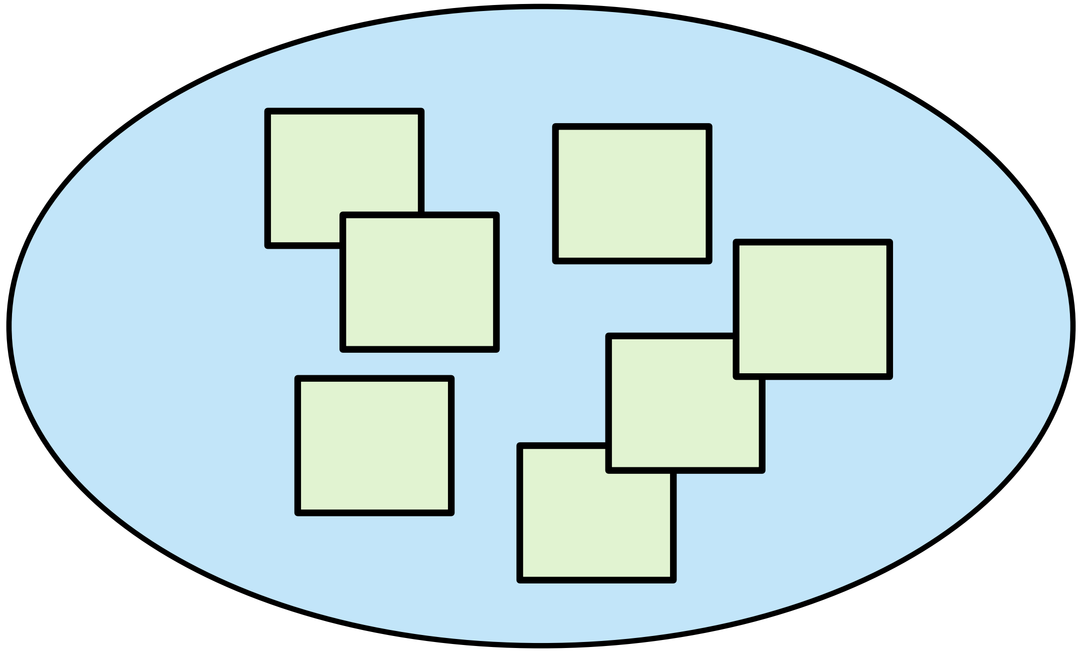
- Features
  - Supports logarithmic time lookup of a type `Val` based a type `Key`
  - Elements of type `pair<const Key, Val>` are sorted according to `Compare`
  - Key values are **not unique**
  - Provides const and mutable **bidirectional** iterators
    - Mutable iterators permit the `Val` member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



Copyright © 2021 Bob Steagall

Unordered Associative Container: Unordered Set

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

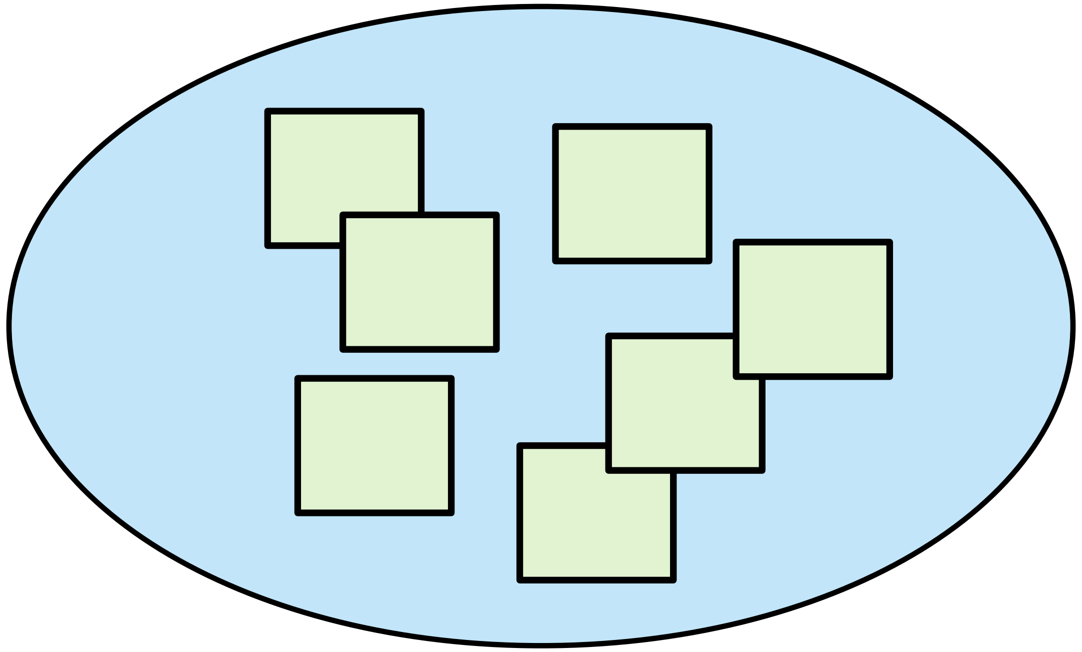


- Features
  - Supports amortized constant time element lookup
  - Elements of type `Key` are stored internally in an order determined by Hash
  - Element values are **unique**
  - Provides const **forward** iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests `Key`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

Copyright © 2021 Bob Steagall

Unordered Associative Container: Unordered Multiset

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



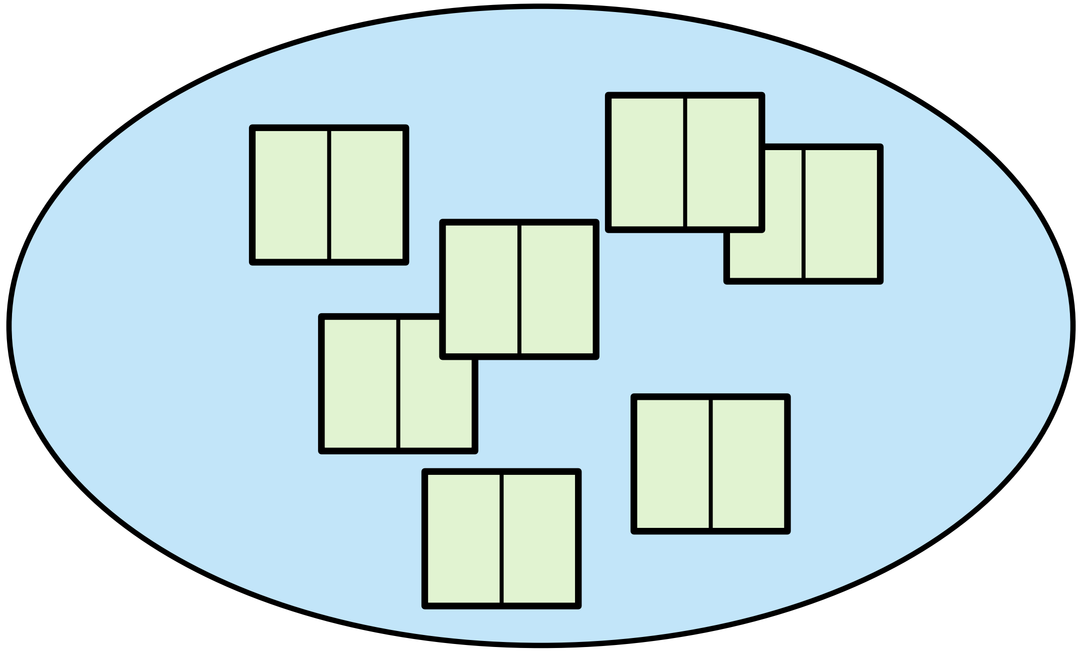
- Features
  - Supports amortized constant time element lookup
  - Elements of type `Key` are stored internally in an order determined by Hash
  - Element values are **not unique**
  - Provides const **forward** iterators
  - Implemented as a hash table – `Hash` helps determine ordering, `Pred` tests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

Copyright © 2021 Bob Steagall

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

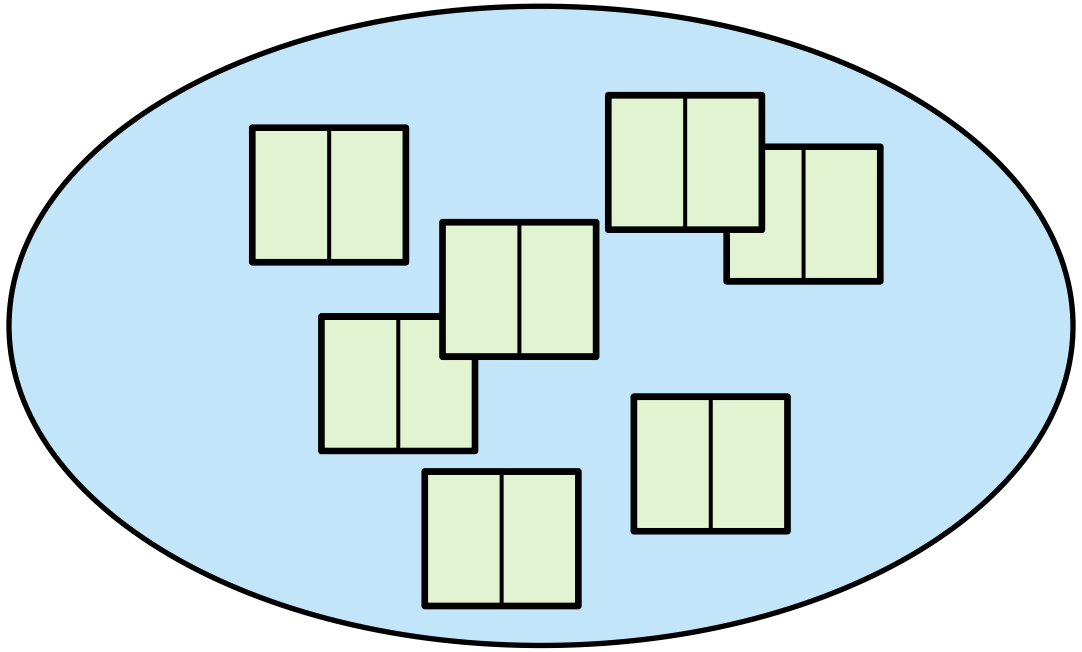


Copyright © 2021 Bob Steagall

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



Copyright © 2021 Bob Steagall

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

Copyright © 2021 Bob Steagall

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

Copyright © 2021 Bob Steagall

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`)

Copyright © 2021 Bob Steagall

Algorithms

Copyright © 2021 Bob Steagall

Algorithms

- There's large number of algorithms provided by STL (well over 100)
  - Multiple versions of almost all
  - Parallel implementations of some

- Algorithm categories
  - Non-modifying algorithms
  - Modifying algorithms
  - Removing algorithms
  - Mutating algorithms
  - Sorting algorithms
  - Sorted range algorithms
  - Numeric algorithms

Copyright © 2021 Bob Steagall

Algorithms - Declaration of `sort`

- `sort`
  - **Action**: Sorts the elements in the range `[first, last)` in non-descending order; the order of equivalent elements is not guaranteed to be preserved; Elements are compared using the given binary comparison function comp
  - **Complexity**: O(*N·log(N)*), where N = `std::distance(first, last)` comparisons

```
template<class RandomIter, class Compare>
void
sort(RandomIter first, RandomIter last, Compare comp);
```


Copyright © 2021 Bob Steagall

Algorithms - Declaration of `lower_bound`

- `lower_bound`
  - **Action**: Returns an iterator pointing to the first element in the range `[first, last)` that is not less than (i.e., greater than or equal to) `value`, or `last` if no such element is found
  - **Complexity**: the number of comparisons performed is logarithmic in the distance between `first` and `last` (at most *log2*(*last* - *first*) + O(*1*) comparisons)

    For non-random-access iterators, the number of iterator increments is linear

```
template<class ForwardIter, class T>
ForwardIt
lower_bound(ForwardIter first, ForwardIter last, const T& value);
```


Copyright © 2021 Bob Steagall

Algorithms – A Sample remove_copy_if

- remove_copy_if
  - **Action**: copies elements from the range [first, last), to another range beginning at dest, omitting the elements which satisfy specific criteria; source and destination ranges cannot overlap; returns an iterator to the element past the last element copied
  - **Complexity**: exactly std::distance(first, last) applications of the predicate.

```
template<class InputIter, class OutputIter, class UnaryPredicate>
OutputIt
remove_copy_if(InputIter first, InputIter last, OutputIter dest, UnaryPredicate pred)
{
    for (;  first != last;  ++first)
    {
        if (!pred(*first))
        {
            *dest++ = *first;
        }
    }
    return dest;
}
```

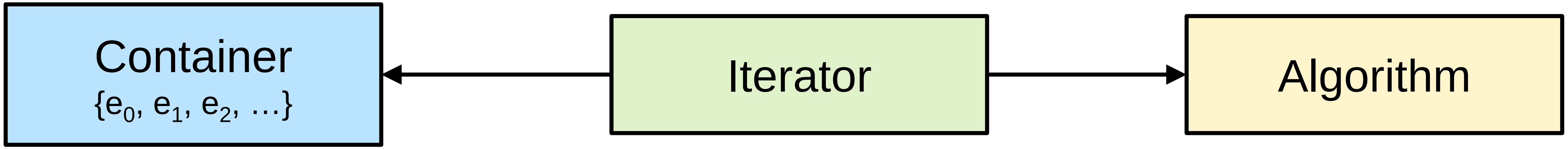

Copyright © 2021 Bob Steagall

Summary

Copyright © 2021 Bob Steagall

Summary: 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**



- STL makes complexity guarantees by specifying *interfaces* and *requirements*

Copyright © 2021 Bob Steagall

Summary: On the Brilliance of the STL

- Four important positive qualities
  - Speed
  - Efficiency
  - Extensibility
  - Elegance

- The STL separates data structures from algorithms, and ties them together with iterators
  - It is remarkable can be done with only 5 iterator categories

- The underlying ideas have become embedded into our way of thinking

Copyright © 2021 Bob Steagall

Thank You for Attending!

Talk: `github.com/BobSteagall/CppCon2021`

Blog: `bobsteagall.com`

Copyright © 2021 Bob Steagall