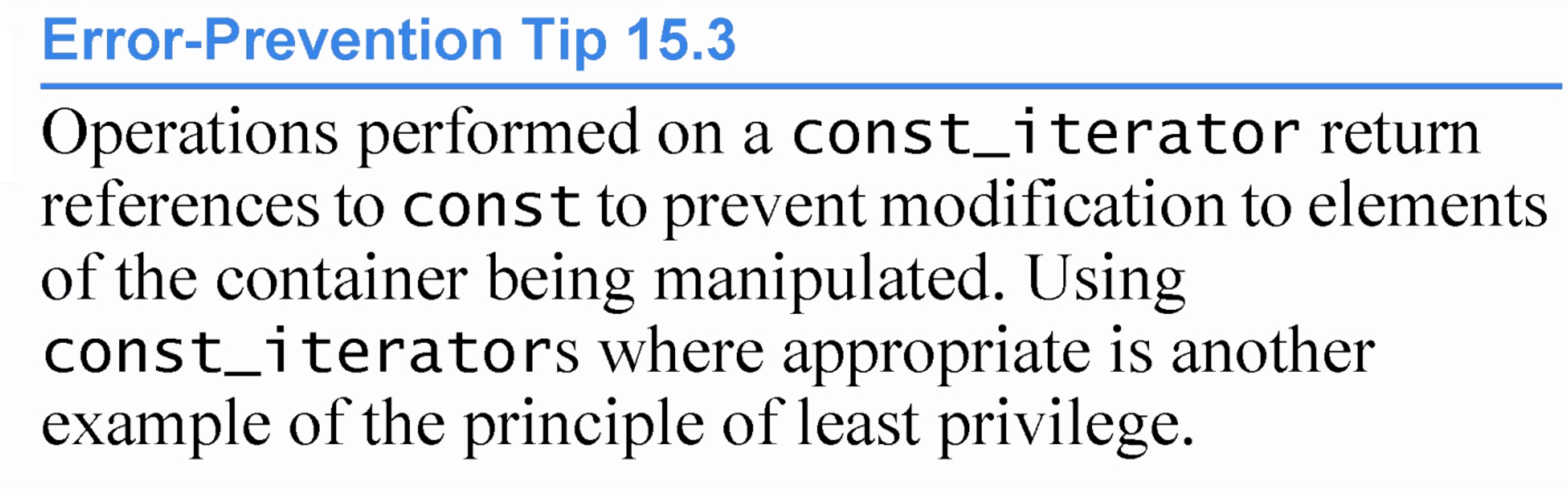
Introduction to Iterators

- This topic teaches how to
  - create iterators
  - use them to iterate over containers
  - hold state information

  dereference iterators
  - Increment
  - Decrement
  - point to the beginning of a container
  - point to the end of a container
  - use iterators with sequences
  - hierarchies of iterators
  - categories of iterators
  - random access iterators
  - predefined iterator typedefs
  - const iterators
  - Random
  - Forward
  - and bi-directional iterators



Error-Prevention Tip 15.3

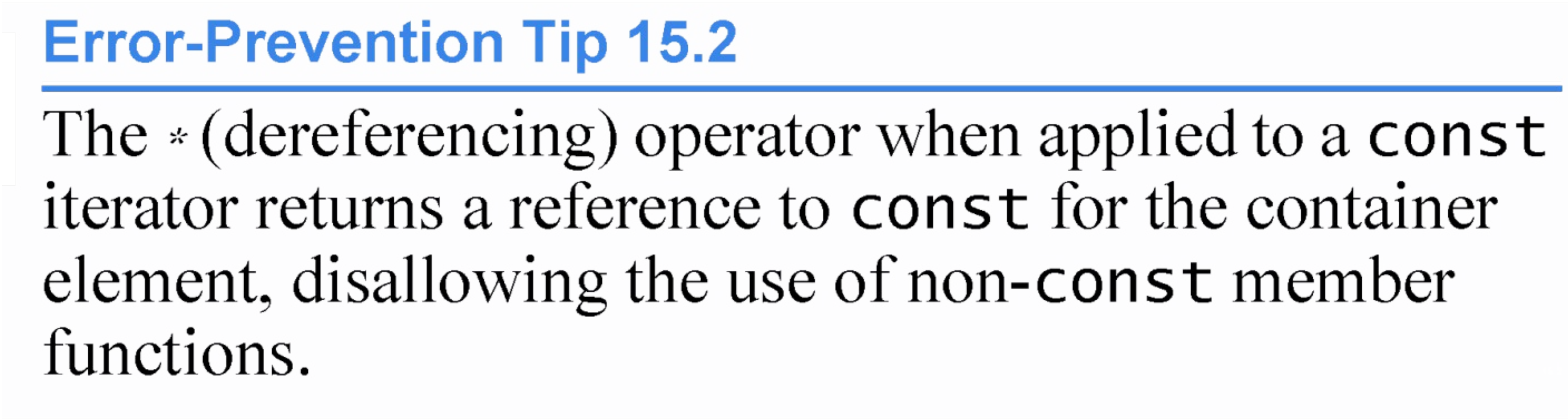
Operations performed on a `const_iterator` return references to `const` to prevent modification to elements of the container being manipulated. Using `const_iterator`s where appropriate is another example of the principle of least privilege.

- *Iterators* have many similarities to pointers and are used to point to *first-class container* elements and for other purposes
- Iterators hold *state* information sensitive to the particular containers on which they operate
  - Thus, iterators are implemented for each type of container
- Certain iterator operations are uniform across containers
  - For example, the *dereferencing operator* (*) dereferences an iterator so that you can use the element to which it points
- The *++ operation on an iterator* moves it to the container's *next element*
  - Much as incrementing a pointer into a built-in array aims the pointer at the next array element

- *First-class containers* provide member functions `begin` and `end`
  - Function `begin`
    - Returns an iterator pointing to the *first* element of the container
  - Function `end`
    - Returns an iterator pointing to the *first element past the end of the container* (one past the end)
    - A non-existent element that's used to determine when the end of a container is reached

- If iterator `i` points to a particular element
  - `++i` points to the "next" element
  - `*i` refers to the element pointed to by `i`
- The iterator resulting from `end`
  - Is typically used in an equality or inequality comparison
  - To determine whether the "moving iterator" has reached the end of the container
- An object of a container's `iterator` type
  - Refers to a container element that *can* be modified
- An object of a container's `const_iterator` type
  - Refers to a container element that *cannot* be modified

- We use iterators with sequences (also called ranges)
  - Sequences can be in containers
  - Or they can be input sequences
  - Or output sequences



Error-Prevention Tip 15.2

The * (dereferencing) operator when applied to a `const` iterator returns a reference to `const` for the container element, disallowing the use of non-`const` member functions.

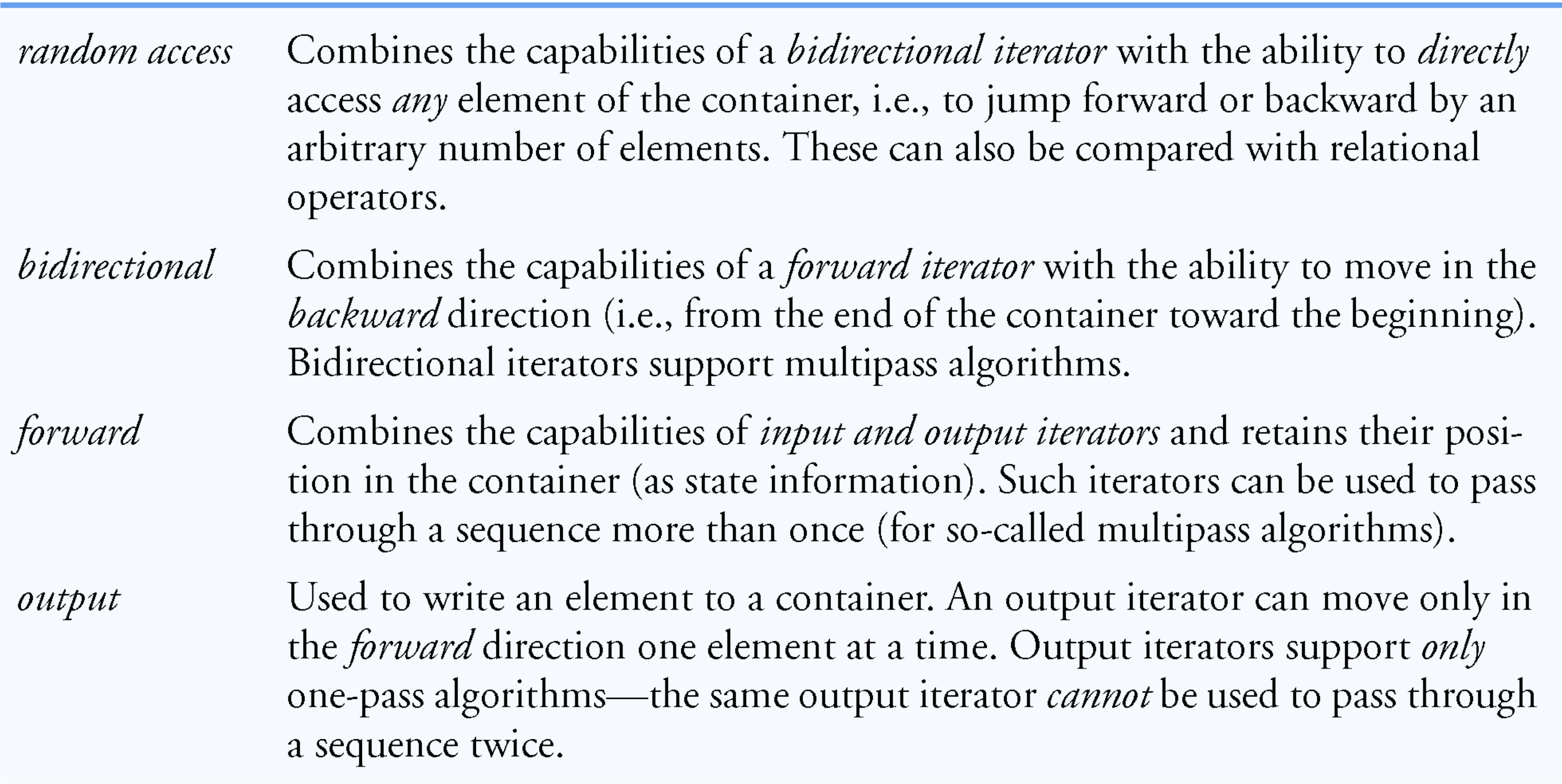
Each iterator category provides a specific set of functionality

As you follow an iterator hierarchy from bottom to top

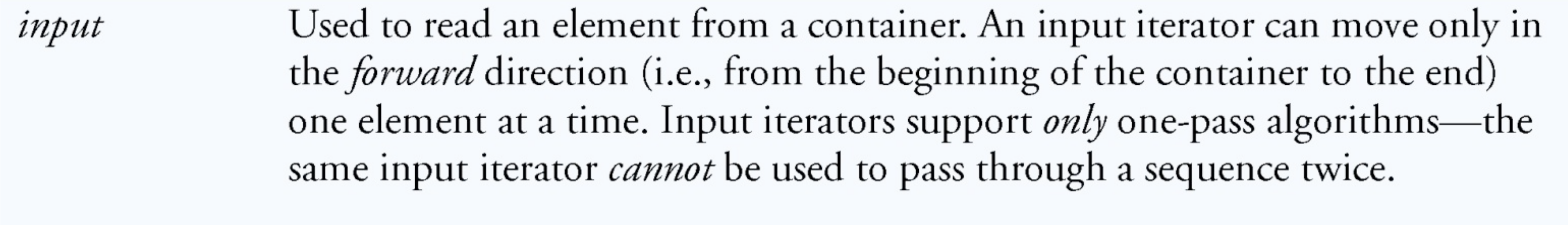
- Each iterator category supports all the functionality of the categories *below* it in the figure

Thus the "weakest" iterator types are at the bottom and the most powerful one is at the top

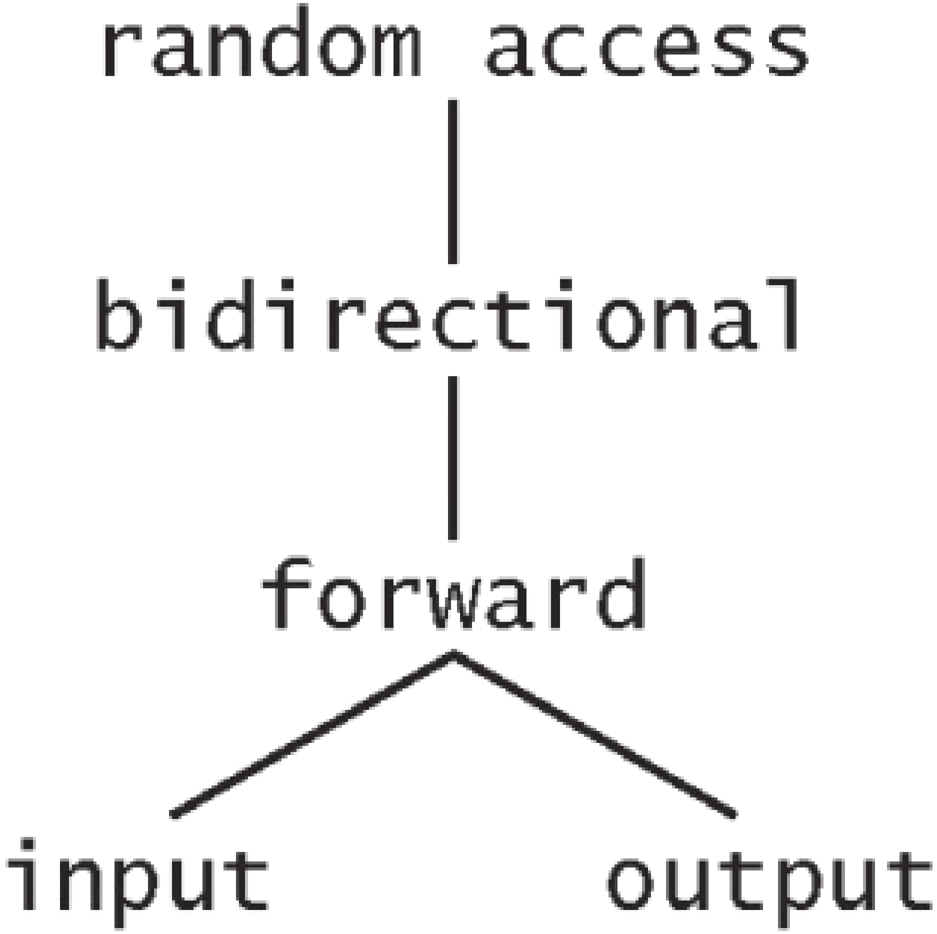
- Note that this is *not* an inheritance hierarchy



random access	Combines the capabilities of a *bidirectional iterator* with the ability to *directly* access *any* element of the container, i.e., to jump forward or backward by an arbitrary number of elements. These can also be compared with relational operators.
bidirectional	Combines the capabilities of a *forward iterator* with the ability to move in the *backward* direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
forward	Combines the capabilities of *input and output iterators* and retains their position in the container (as state information). Such iterators can be used to pass through a sequence more than once (for so-called multipass algorithms).
output	Used to write an element to a container. An output iterator can move only in the *forward* direction one element at a time. Output iterators support *only* one-pass algorithms—the same output iterator *cannot* be used to pass through a sequence twice.



input	Used to read an element from a container. An input iterator can move only in the *forward* direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support *only* one-pass algorithms—the same input iterator *cannot* be used to pass through a sequence twice.



random access

bidirectional

forward

input

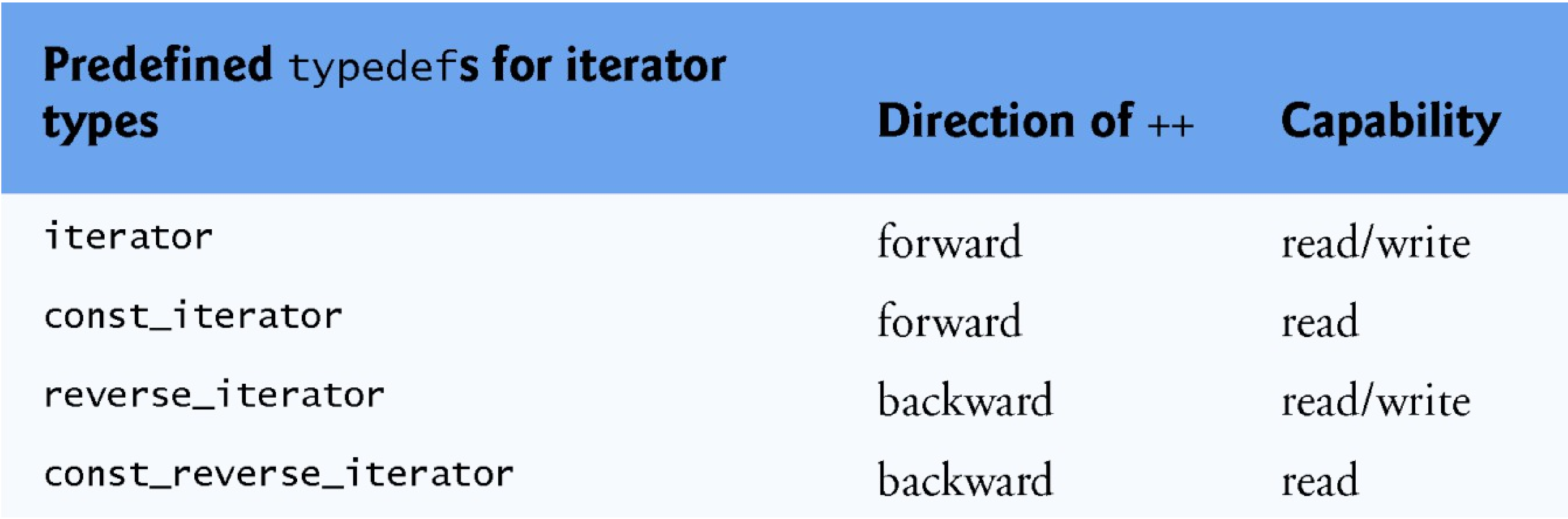
output

- The iterator category that each container supports determines whether that container can be used with specific algorithms
  - *Containers that support random-access iterators can be used with all Standard Library algorithms*
    - With the exception that if an algorithm requires changes to a container's size, the algorithm can't be used on built-in arrays or `array` objects
  - Pointers into *built-in* arrays can be used in place of iterators with most algorithms

- The following Figure shows the iterator category of each container
  - The first-class containers, `string`s and built-in arrays are all traversable with iterators

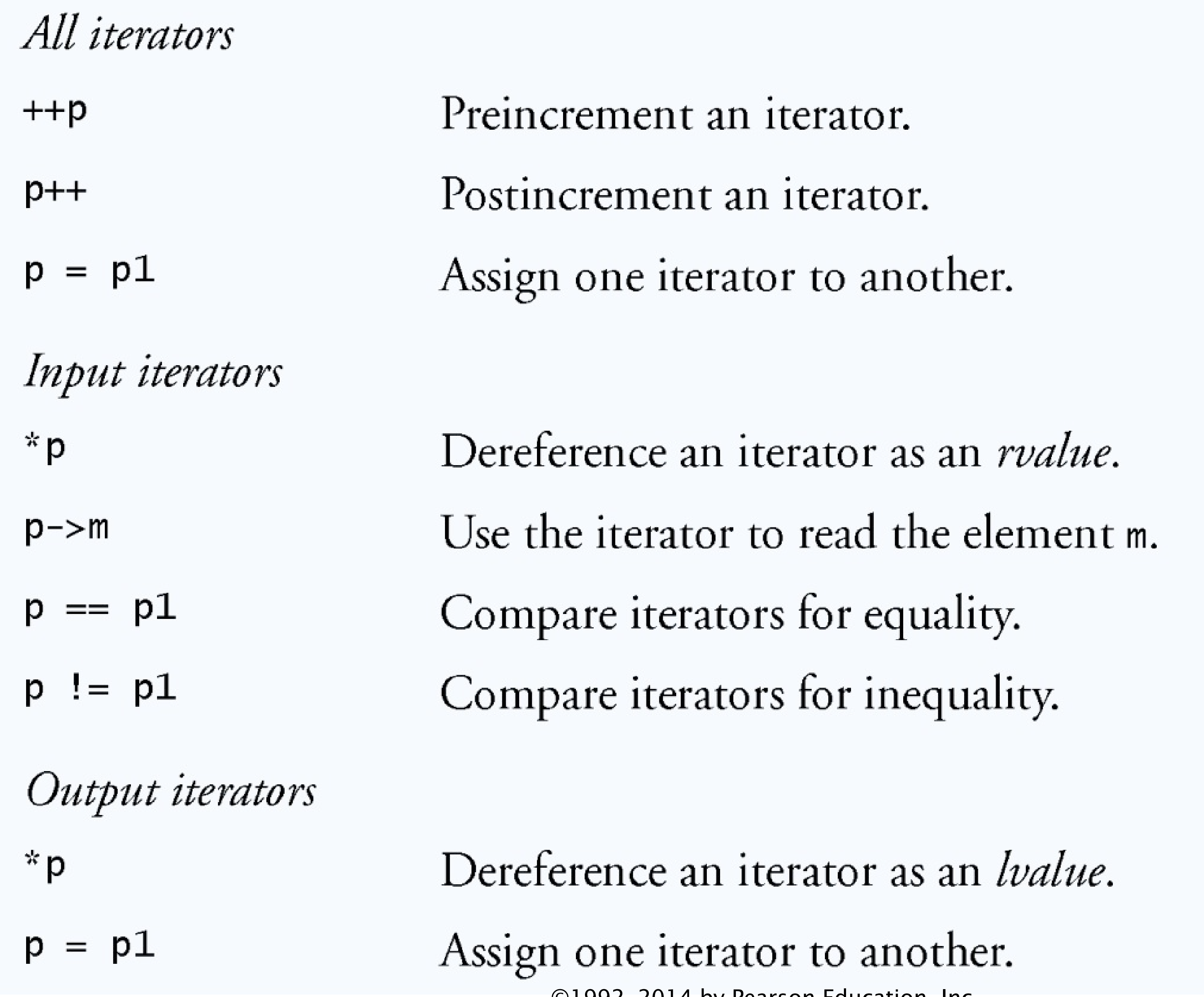
Container	Iterator type	Container	Iterator type
Sequence containers (first class)		*Unordered associative containers (first class)*	
`vector`	random access	`unordered_set`	bidirectional
`array`	random access	`unordered_multiset`	bidirectional
`deque`	random access	`unordered_map`	bidirectional
`list`	bidirectional	`unordered_multimap`	bidirectional
`forward_list`	forward		
Ordered associative containers (first class)		*Container adapters*	
`set`	bidirectional	`stack`	none
`multiset`	bidirectional	`queue`	none
`map`	bidirectional	`priority_queue`	none
`multimap`	bidirectional		

- Not every `typedef` is defined for every container
  - We use `const` versions of the iterators for traversing `const` *containers*
  - We use non-`const` containers that should not be modified
  - We use *reverse iterators* to traverse containers in the reverse direction



Predefined `typedef`s for iterator types	Direction of `++`	Capability
`iterator`	forward	read/write
`const_iterator`	forward	read
`reverse_iterator`	backward	read/write
`const_reverse_iterator`	backward	read

- Iterators must provide default constructors, copy constructors and copy assignment operators.
- A *forward* iterator supports ++ and all of the *input* and *output* iterator capabilities.
- A *bidirectional* iterator supports -- and all the capabilities of *forward* iterators.
- A *random access iterator* supports all operations
- For input iterators and output iterators, it’s not possible to save the iterator then use the saved value later



All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
`p = p1`	Assign one iterator to another.
Input iterators	
`*p`	Dereference an iterator as an *rvalue*.
`p->m`	Use the iterator to read the element `m`.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator as an *lvalue*.
`p = p1`	Assign one iterator to another.

Iterator operation	Description
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions

Iterator operation	Description
`p < p1`	Return `true` if iterator `p` is *less than* iterator `p1` (i.e., iterator `p` is *before* iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is *less than or equal to* iterator `p1` (i.e., iterator `p` is *before* iterator `p1` or *at the same location* as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is *greater than* iterator `p1` (i.e., iterator `p` is *after* iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is *greater than or equal to* iterator `p1` (i.e., iterator `p` is *after* iterator `p1` or *at the same location* as iterator `p1` in the container); otherwise, return `false`.

- This topic taught how to
  - create iterators
  - use them to iterate over containers
  - hold state information
  - dereference iterators
  - Increment
  - Decrement
  - point to the beginning of a container
  - point to the end of a container
  - use iterators with sequences
  - hierarchies of iterators
  - categories of iterators
  - random access iterators
  - predefined iterator typedefs
  - const iterators
  - Random
  - Forward
  - and bi-directional iterators