HIGH-LEVEL PROGRAMMING 2
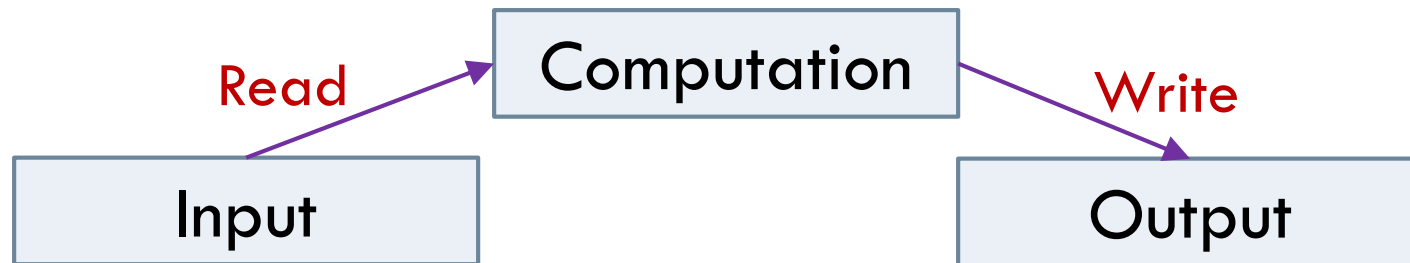
STL   by Prasanna Ghali

STL: Motivation

☐ There are two major aspects of computing: the computation and the data

STL: Motivation

- Things we [programmers] do with large amounts of data:
  - Collect data into containers
  - Organize data
  - Retrieve data items
  - Modify containers
  - Perform mathematical operations

STL: Motivation

☐ Problems we encounter:

  ☐ Infinite variations of data types

  ☐ Bewildering number of ways to store collections of data elements

 ☐ Huge variety of tasks we'd like to do with data collections

STL: Motivation

- We must generalize our code to cope with variations in data types, ways of storing data, and processing tasks

- We need code for common programming tasks so that we don't reinvent the wheel

What is STL?

☐ Standard Template Library is part of C++ standard library that provides framework for common programming data structures and algorithms
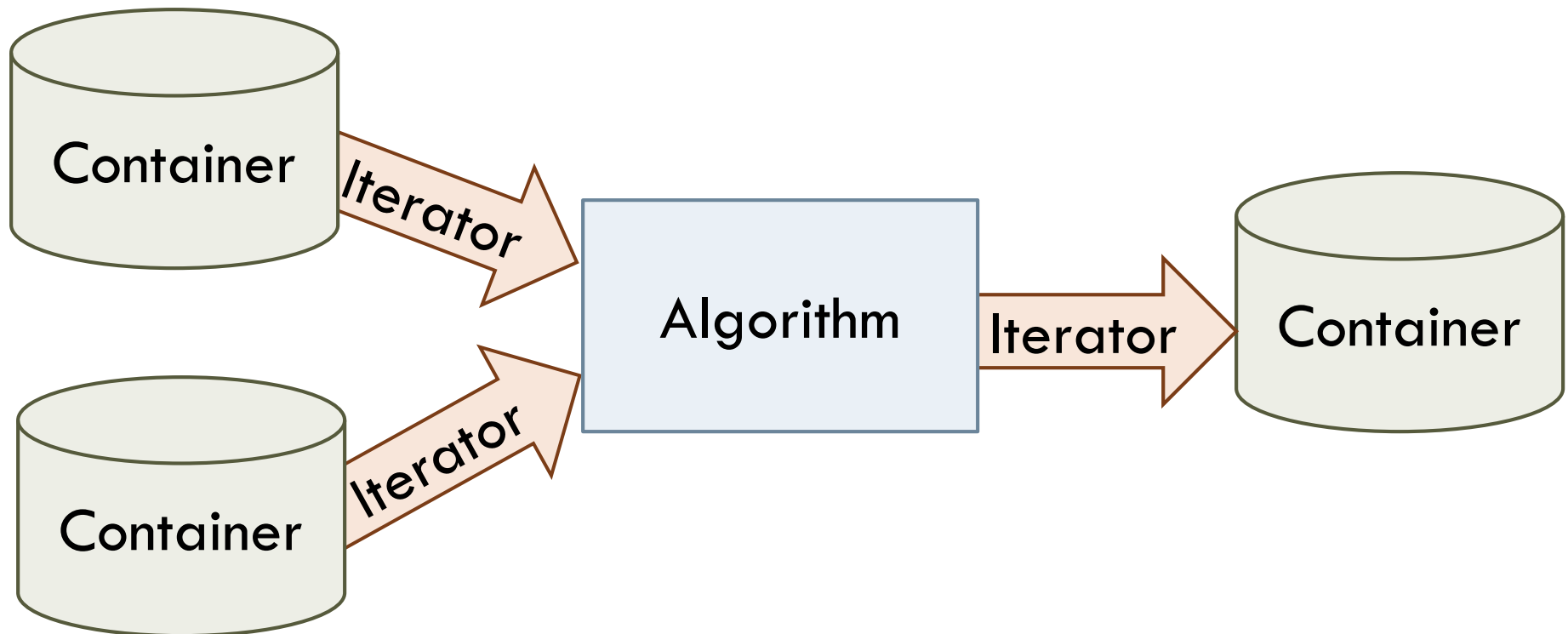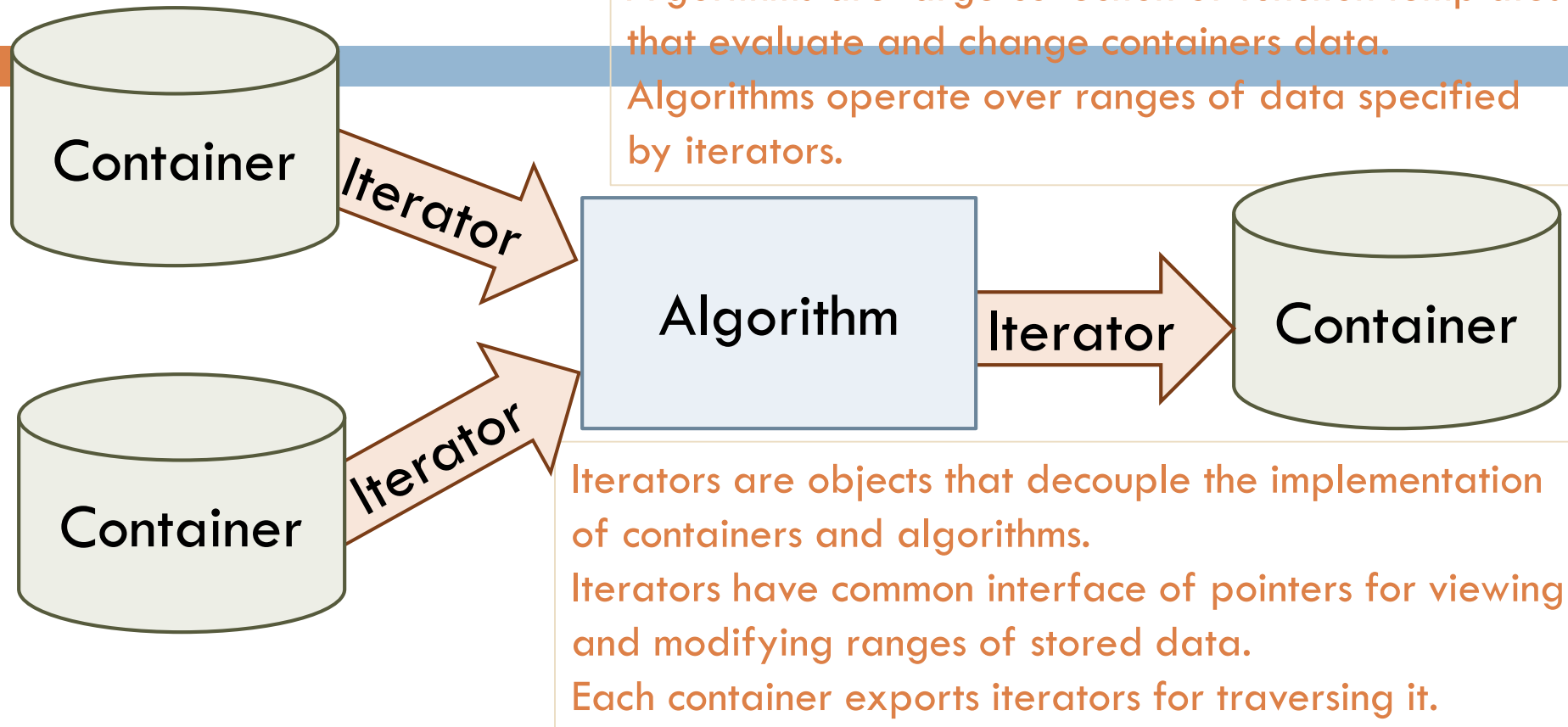
Why Use STL?

□ Minimizes programming work by providing
- Uniform access to data
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast and efficient retrieval, addition, and deletion of data
- Industry standard versions of most common algorithms such as copy, find, search, sort, permute, partition, …
- Benefit from innovations in data structures and algorithms without having to master these techniques

Benefits of Knowing/Using STL

☐ Read unknown number of integers from a file and write only unique integers in sorted [descending] order to another file

What is STL?

Container

Iterator

Container

Iterator

Algorithm

Iterator

Container

Algorithms are large collection of function templates that evaluate and change containers data.
Algorithms operate over ranges of data specified by iterators.

Iterators are objects that decouple the implementation of containers and algorithms.
Iterators have common interface of pointers for viewing and modifying ranges of stored data.
Each container exports iterators for traversing it.

Containers are class templates whose purpose is to contain objects [including containers and containers of containers and so on].
They provide broad range of important data structures that are easy to use, and are quite efficient.

Deeper Overview of STL

☐ Logically divided into six generic components that interoperate with rest of standard library

Algorithms	←	Functions
Iterators	←	Adapters
Containers		
Allocators		
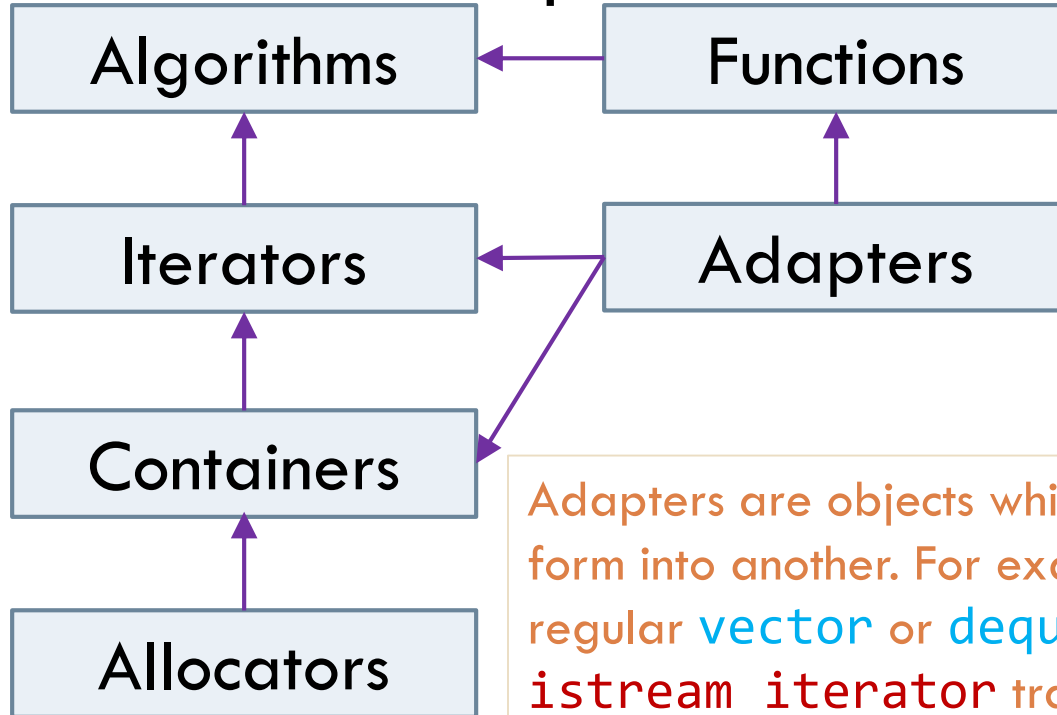
Function objects are used as callbacks by algorithms to give them more flexibility.
Thus, you can use an algorithm to suit your needs even if that need is very special or complex.

Adapters are objects which transform an object from one form into another. For example, `stack` adapter transforms a regular `vector` or `deque` into a LIFO container, while `istream_iterator` transforms an input stream into iterator

Allocators allow clients of container classes to customize allocation/deallocation and construction/destruction of elements
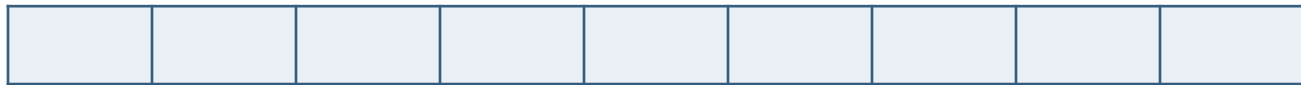
Containers

- Containers manage a collection of elements
- Four kinds of containers:
  - Sequence containers
  - Associative containers
  - Unordered containers
  - Adapter containers

Sequence Containers
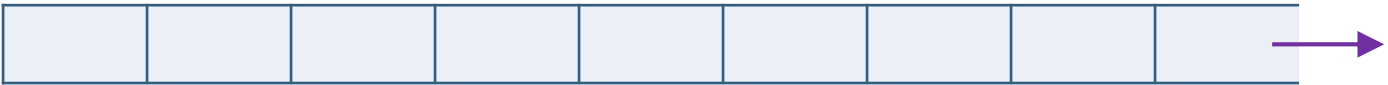
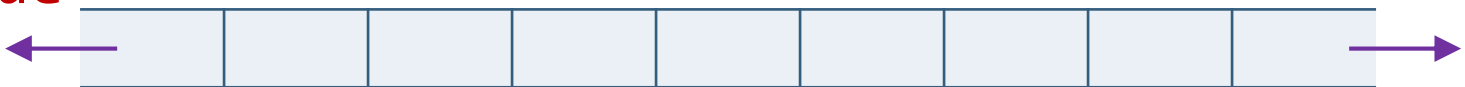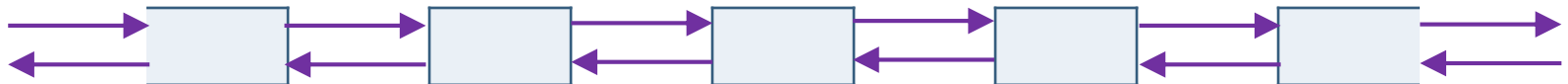□ Ordered collections in which every element has a certain position independent of its value
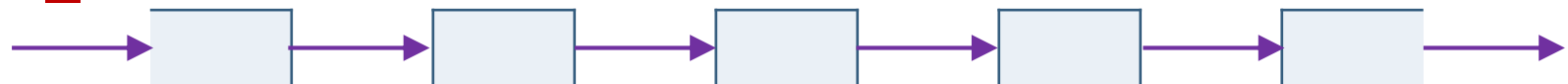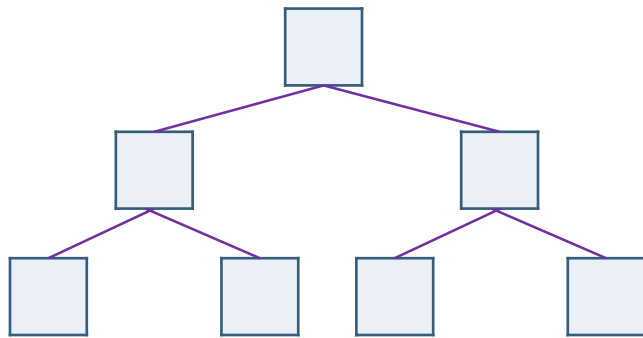
array

vector

deque

list

forward_list

Associative Containers

□ Sorted collections in which element's position depends on its *value* [set/multiset] or *key* [map/multimap] due to certain sorting criterion

set/multiset
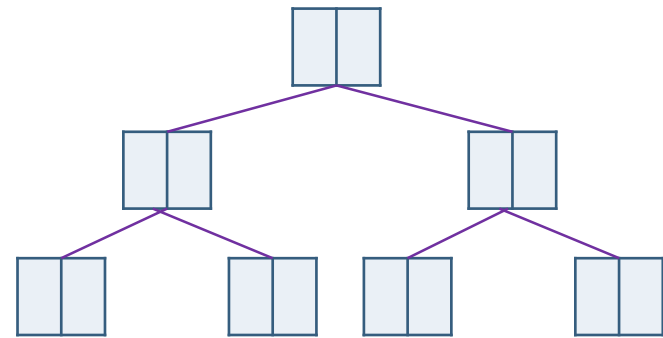
map/multimap

Unordered Containers

- Not covered in HLP2

Adapter Containers

- Wrappers around existing containers that provide a new interface: stack, queue, priority_queue

```
namespace std {

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

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

template <typename T, typename Container = vector<T>,
        typename Compare = less<typename Container::value_type>>
class priority_queue;

}
```

Accessing Elements

Container std::	operator[]	at	front	back	data	top
array	√	√	√	√	√	
vector	√	√	√	√	√	
deque	√	√	√	√		
forward_list			√			
list			√	√		
stack						√
queue			√	√		
set/multiset						
map/multimap	√	√				

What is set/multiset?

- set and multiset containers sort their elements automatically according to a certain sorting criterion
  - multisets allow duplicates, whereas sets do not
- Search functions have logarithmic complexity [because they're implemented as binary tree]
- Automatic sorting imposes constraint that element value cannot be changed

Internal Structure of set/multiset

```
std::set<int> si{12,10,11,8,9,7,6,5,4,3,2,1};
```
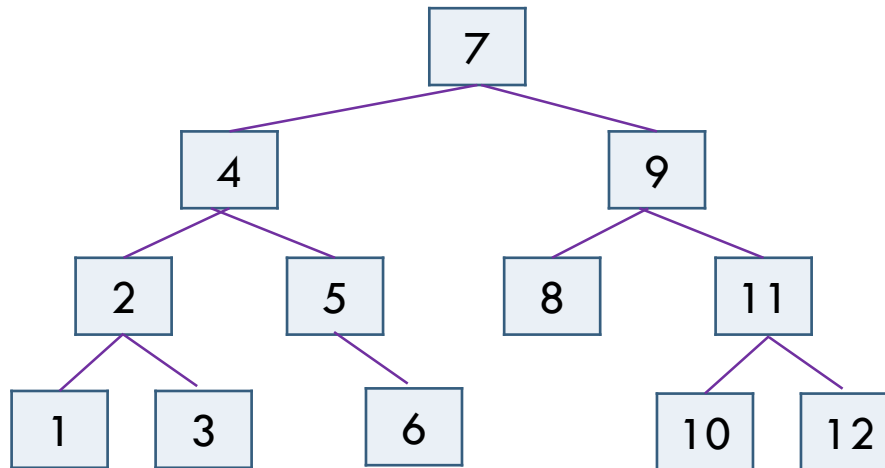
set/multiset Operations

- Search functions have logarithmic complexity [because they're implemented as binary tree]
- Automatic sorting imposes constraint that element value cannot be changed
- See *set.cpp* and *die-set.cpp*

What is map/multimap?

□ Ordered sequence of (key,value) pairs in which you can look up a value based on a key

  ▪ Also known as called *associative arrays, hash tables, red-black trees*

Map type	Associative	Ordered	Mapped	Unique keys	Dynamic
map	√	√	√	√	√
multimap	√	√	√		√

map/multimap: Key Operations

- Map supports many operations of which four are key:
  - Inserting a key/value pair
  - Checking whether a particular key exists
  - Querying which value is associated with given key
  - Removing an existing key/value pair
- See *map.cpp* for details …

When to Use Which Container (1/2)

	array	vector	deque	list	forward_list	Associative Containers
Internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree
Element type	Value	Value	Value	Value	Value	set: value map: value/key
Duplicates	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere
Random access	Yes	Yes	Yes	No	No	No
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast

When to Use Which Container (2/2)

- By default, you should use a vector

- If you insert and/or remove elements often at beginning and end of sequence, you should use deque

- If you insert, remove, and move elements often in middle of container, consider using list

- If you often need to search for elements according to certain criterion, use multiset

- To process key/value pairs, use multimap or map

- If you need associative array, use map

- If you need dictionary, use multimap

Sequences and Iterators (1/3)

- Sequence is central concept of STL
  - Sequence is collection of data [not necessarily elements of container]
  - Sequence has beginning and end
  - Sequence can be traversed from its beginning to end
- Iterator is object that provides abstract interface of pointer by identifying an element of sequence

first: last:

Sequences and Iterators (2/3)

☐ Iterator is object that provides abstract interface of pointer by identifying an element of sequence

☐ Pair of iterators define sequence of elements using half-open range [first : last)

first:

last:

...

Arrows from one element to next indicate that if we've an iterator from one element we can get an iterator to the next

Sequences and Iterators (3/3)

- Half-open range [first : last) has two advantages:
  - Simple end criterion for loops that iterate over elements in sequence – they simply continue as long as last is not reached
  - Avoids special handling for empty ranges – for empty ranges, first is equal to last

Iterators: Key Idea of STL (1/3)

- ☐ Iterators are fundamental methodology that decouples implementation of data structures and algorithms
  - ☐ Any algorithm can operate on any container or any combination of containers

- ☐ To program m algorithms and n data structures, we need $m \times n$ implementations

- ☐ With iterators, total programming effort is $m + n$ implementations!!!
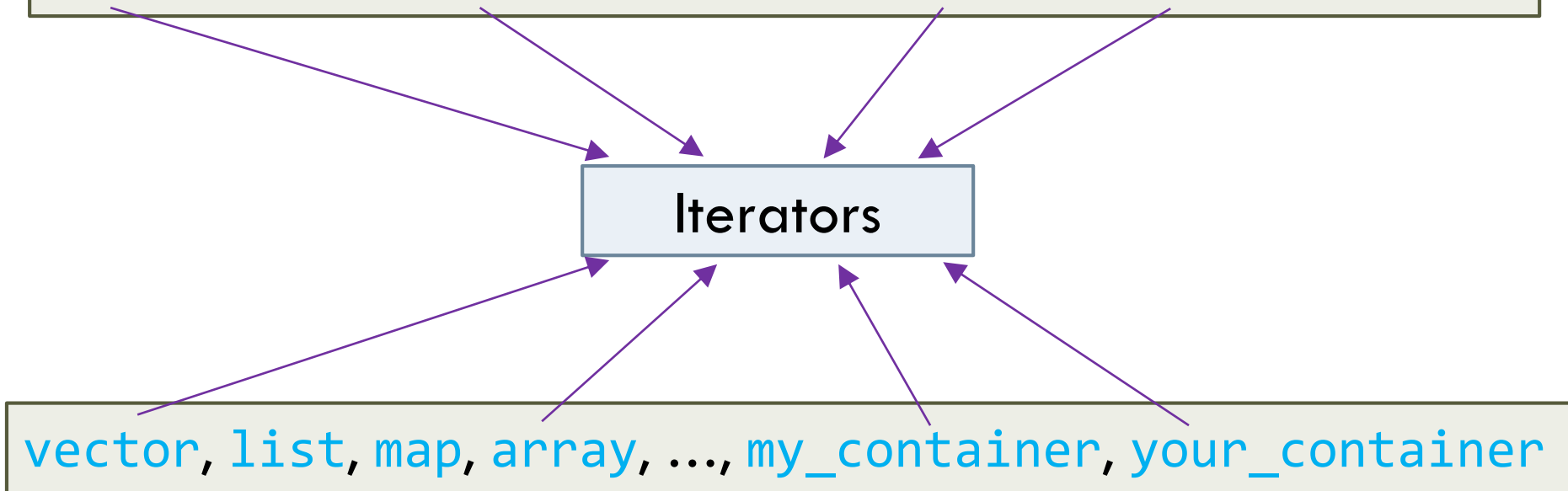
Iterators: Key Idea of STL (2/3)

☐ Mechanism that minimizes algorithm's dependence on data structures on which it operates

sort, find, search, copy, ..., my_algorithm, your_algorithm

Iterators

vector, list, map, array, ..., my_container, your_container

Iterators: Key Idea of STL (3/3)

☐ Find element with largest value in a sequence

```cpp
// return an iterator to the element in
// [first, last) that has highest value
template <typename Iterator>
Iterator high(Iterator first, Iterator last) {
  Iterator high = first;
  while (first != last) {
    high = (*high < *first) ? first : high;
    ++first;
  }
  return high;
}
```

Iterator Model: Fundamental Operations

- Following fundamental operations define behavior of iterator:
  - Operator * returns value of element pointed to by iterator
  - Operator ++ lets iterator step to *next* element
  - Operators == and != return whether two iterators represent same element
  - Operator = copy assigns an iterator

Standard Iterator Operations

□ If **p** and **q** are iterators to elements of same sequence:

Basic standard iterator operations	
p==q	true if and only if p and q point to same element or both point to one beyond last element
p!=q	!(p==q)
*p	Refers to element pointed to by p
*p=val	Writes to element pointed to by p
val=*p	Reads from element pointed to by p
++p	Makes p refer to next element in sequence or to one beyond last element

Containers and Iterators

☐ Every container implements its own iterator

☐ Every container defines two iterator types:

    ▫ *container*::iterator for read/write mode

    ▫ *container*::const_iterator for read-only mode

☐ Every container provides same member functions for using iterators:

    ▫ begin() and end() specify range for read/write mode

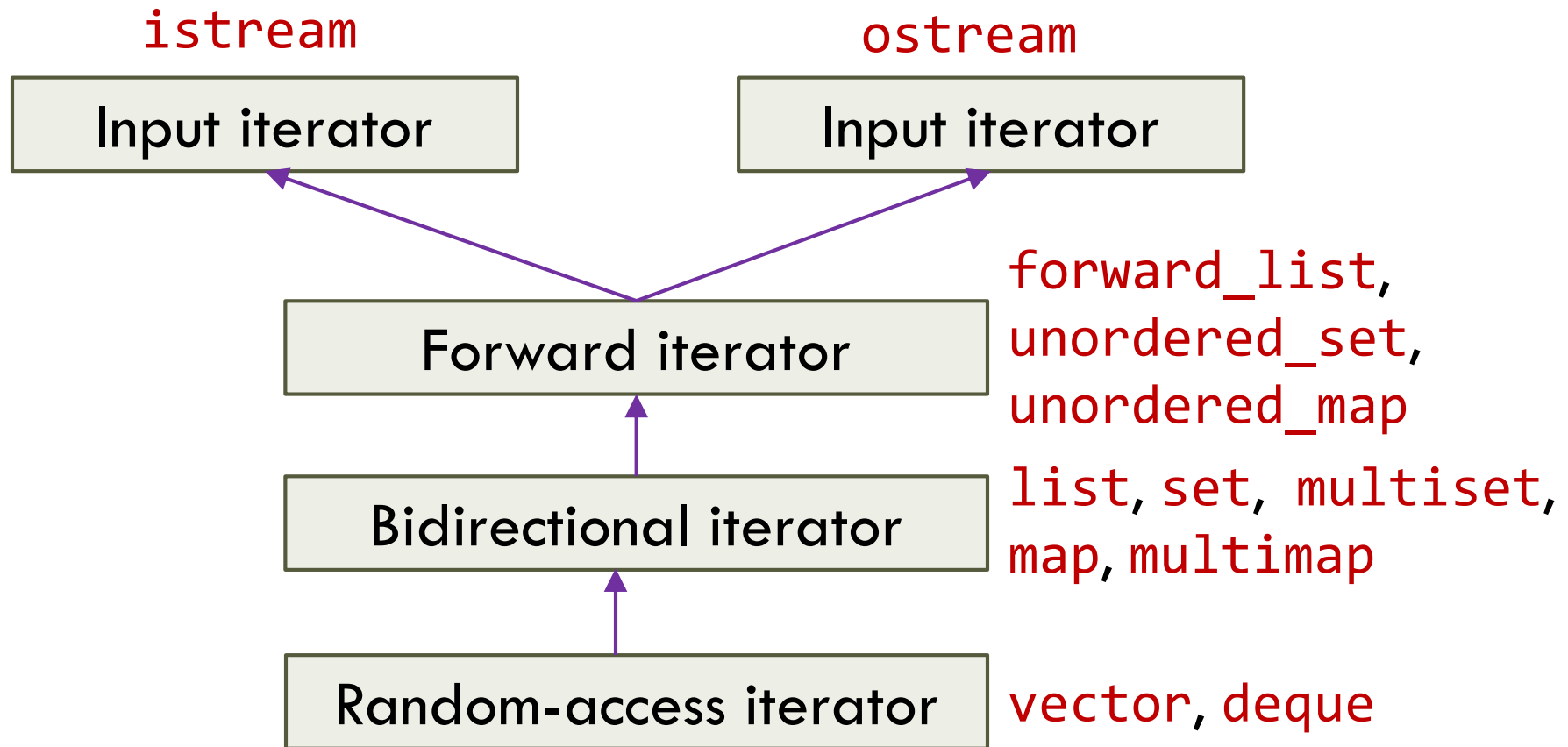   ▫ cbegin() and cend() specify range for read-only mode

Iterator Categories

□ Standard library provides five kinds of iterator categories:

Iterator categories	
Input iterator	Iterate forward using **++** and read each element *once only* using *****
Output iterator	Iterate forward using **++** and write each element *once only* using *****
Forward iterator	Iterate forward repeatedly using **++** and read and write [unless elements are **const**] elements using *****
Bidirectional iterator	Iterate forward [using **++**] and backward [using **--**] and read and write [unless elements are **const**] elements using *****
Random-access iterator	Iterate forward [using **++** and **+=**] and backward [using **--** or **-=**] and read and write [unless elements are **const**] elements using ***** or **[ ]**. Can subscript, add an integer to iterator using **+**, and subtract an integer using **-**. Can find distance between two iterator to same sequence by subtracting one from the other. Can compare iterators using **<**, **<=**, **>**, and **>=**.

Iterator Categories: Logical Organization

istream

ostream

| Input iterator | | Input iterator |

Forward iterator — forward_list, unordered_set, unordered_map

Bidirectional iterator — list, set, multiset, map, multimap

Random-access iterator — vector, deque

Auxiliary Iterator Functions

☐ Standard library provides auxiliary functions [declared in `<iterator>`] to provide all iterator [categories] some abilities of random-access iterators:

- `advance()`
- `next()`
- `prev()`
- `distance()`
- `iter_swap()`

std::advance

```cpp
std::list<int> li{1,2,3,4,5,6,7,8,9};
std::list<int>::iterator pos = li.begin();
std::cout << *pos << '\n';

// step three elements forward
std::advance(pos, 3);
std::cout << *pos << '\n';

// step two elements backward
std::advance(pos, -2);
std::cout << *pos << '\n';
```

STL Algorithms (1/2)

- Standard library offers about 100 algorithms in `<functional>`
  - All are useful for someone sometimes
  - We'll focus on some that are often useful for many and on some that are occasionally very useful for someone

STL Algorithms (2/2)

- By default
  - Comparison for equality is done using ==
  - Ordering is done based on <
- Input sequence defined by pair of iterators
- Output sequence defined by iterator to its first element
- Typically, algorithm is parameterized by one or more operations that can be defined as function object or functions
- Failure is reported by returning end of input sequence
  - `std::find(b, e, v)` returns e if it doesn't find v

Algorithms: Classification (1/2)

- Nonmodifying [read-only] algorithms
  - Change neither order nor value of elements they process
- Modifying algorithms
  - Change value of elements directly or modify them while they're being copied into another range
- Removing algorithms
  - Remove elements either in single range or while these elements are being copied into another range

Algorithms: Classification (2/2)

- Mutating algorithms
  - Change order of elements but not change their values
- Sorting algorithms
  - Special kind of mutating algorithms but more complex
- Sorted-range algorithms
  - Ranges on which they operate must be sorted
- Numeric algorithms
  - Combine numeric elements in different ways

Function Objects

- Have always existed in C++

- Called functionals or functors

- Objects of class that defines operator()

```
class X {
public:
  // define function call operator
  return-value operator() (parameters) const;
  ...
};
```

```
X func;
...
// a function call
func(arg1, arg2);
```

Why Function Objects?

- Functions with state
- Each function object has its own type
  - This type can be passed as template parameter
- Usually faster than function pointers
- See *wfo.cpp*

Types of Function Objects

- Zero parameter is called *generator*
  - See *gen.cpp*
- One parameter is called *unary function*
  - See *unary.cpp*
- Two parameters is called *binary function*
  - See *binary.cpp*
- *Predicates* are stateless function objects that return Boolean value
  - See *predicate.cpp*

Pass By Value: Advantage

☐ By default, function objects are passed by value rather than by reference

☐ Advantage: You can pass constant and temporary expressions

```cpp
IncreasingNumberGenerator seq(3);
std::list<int> li;
// insert sequence beginning with 3
std::generate_n(std::back_inserter(li), 5, seq);
// insert sequence beginning with 3 again ...
std::generate_n(std::back_inserter(li), 5, seq);
```

Pass By Value: Disadvantage

- By default, function objects are passed by value rather than by reference

- Disadvantage: You can't get back modifications to state of function objects

- Three ways to get result from function objects passed to algorithms:
  - Keep state externally and let function object refer to it
  - Pass function objects by reference
  - Use return value of for_each algorithm

Keeping State Externally …

- Using global objects is never a good idea!!!

Passing By Reference

```cpp
class IncreasingNumberGenerator {
public:
  IncreasingNumberGenerator(int ival) : number{ival} {}
  int operator()() noexcept { return number++; }
private:
  int number {};
};
```

```cpp
// passing fuction objects by reference ...
IncreasingNumberGenerator seq(3);
std::list<int> li;
// insert sequence beginning with 3
std::generate_n<std::back_insert_iterator<std::list<int>>,
                int, IncreasingNumberGenerator&>
          (std::back_inserter(li), 5, seq);
print(li, "li: ");
// insert sequence beginning with 8 again ...
std::generate_n(std::back_inserter(li), 5, seq);
```

Return Value of for_each

- See *foreach.cpp*

Non-Modifying Algorithms

- Counting and searching for elements; check properties on ranges; compare ranges; perform operation for each element
- See *non-modifying.cpp …*

Non-Modifying Algorithms

Name	Effect
count	Returns number of elements
count_if	Returns number of elements that match a criterion
find	Searches for first element with passed value
find_if	Searches for first element that matches criterion
min_element	Returns element with smallest value
max_element	Returns element with largest value
min_max_element	Returns elements with smallest and largest values
equal	Returns whether two ranges are equal
is_sorted	Returns whether elements in range are sorted
for_each	Performs non-modifying operation on each element

Modifying Algorithms

- Modify elements of range directly or modify them while they're being copied into another range

- Cannot use associative containers as destination because elements of these containers are considered to be constant

- See *modifying.cpp …*

Modifying Algorithms

Name	Effect
copy	Copies range starting with first element
copy_if	Copies elements that match criterion
transform	Modifies [and copies] elements; combines elements of two ranges
fill	Replaces each element with given value
generate	Replaces each element with result of operation
for_each	Performs modifying operation on each element

Removing Algorithms

- □ Special form of modifying algorithms
- □ Remove elements in single range or while these elements are being copied into another range
- □ Cannot use associative containers as destination because elements of these containers are considered to be constant
- □ See *removing.cpp* …

Removing Algorithms

Name	Effect
remove	Removes elements with given value
remove_if	Removes elements that match given criterion
remove_copy	Copies elements that don't match given value
remove_copy_if	Copies elements that don't match given criterion
unique	Removes adjacent duplicates

Mutating Algorithms

☐ Changes order [not value] of elements by assigning and swapping their values

☐ Cannot use associative containers as destination because elements of these containers are considered to be constant

☐ See *mutating.cpp …*

Name	Effect
reverse	Reverses order of elements
reverse_copy	Copies elements while reversing their order

Sorting Algorithms

☐ Special kind of mutating algorithm because they change order of elements

☐ Separate category because sorting is more complicated than mutating operations

☐ See *sorting.cpp …*

Name	Effect
sort	Sorts all elements
sort_stable	Sorts all elements while preserving order of equal elements

Sorted-Range Algorithms

- Require that ranges on which they operate be sorted

- See *sorted-range.cpp …*

Name	Effect
binary_search	Returns whether the range contains an element
lower_bound	Finds 1st element greater than or equal to given value
upper_bound	Finds 1st element greater than given value
equal_range	Returns range of elements equal to given value
merge	Merges elements of two ranges

Numeric Algorithms

☐ Declared in `<numeric>`

☐ Combine numeric elements in different ways

☐ See *numeric.cpp* …

Name	Effect
`accumulate`	Combines all element values to compute sum, product, …
`inner_product`	Combines all elements of two ranges
`iota`	Replaces each element with sequence of incremented values

Iterator Adapters

☐ Reverse iterators

☐ Insert iterators

☐ Stream iterators

Reverse Iterators

☐ Redefine increment and decrement operators so that they behave in reverse

```cpp
template <typename T>
void print(T const& elem) {
  std::cout << elem << ' ';
}

std::list<int> li{1,2,3,4,5,6,7,8,9};
// print elements in normal order
std::for_each(std::begin(li), std::end(li), print<int>);
std::cout << "\n";

// print elements in reverse order
std::for_each(std::rbegin(li), std::rend(li), print<int>);
std::cout << "\n";
```

Insert Iterators

☐ Transform   assignment   of   new   value   into
   insertion of that value

☐ Allows   algorithms   to   insert   rather   than
   overwrite!!!

Name	Class	Called Function	Creation
Back inserter	back_insert_iterator	push_back(val)	back_inserter(cont)
Front inserter	front_insert_iterator	push_front(val)	front_inserter(cont)
General inserter	insert_iterator	insert(pos, val)	inserter(cont, pos)

Stream Iterators

- Allows stream to be used as source or destination of algorithms

Ostream Iterators

```cpp
// create ostream iterator for stream cout
std::ostream_iterator<int> iw(std::cout, "\n");
// write elements with usual iterator interface
*iw = 42;
iw++;
*iw = 77;
iw++;
*iw = -5;

std::list<int> li{1,2,3,4,5,6,7,8,9};
// write elements with < delimiter
std::copy(std::begin(li), std::end(li),
  std::ostream_iterator<int>(std::cout, " < "));
std::cout << "\n";
```

Istream Iterators (1/2)

```cpp
// create istream iterator to read integers from cin
std::istream_iterator<int> ir(std::cin), eof;

// while able to read tokens with istream iterator
// write them twice
while (ir != eof) {
  std::cout << "once:       " << *ir << "\n";
  std::cout << "once again: " << *ir << "\n";
  ++ir;
}
```

Istream Iterators (2/2)

```cpp
std::istream_iterator<std::string> ir(std::cin), eof;
std::ostream_iterator<std::string> iw(std::cout, " ");

// while input is not at the end of the file
// write every third string
while (ir != eof) {
  // ignore the following two strings
  std::advance(ir, 2);

  // read and write the third string
  if (ir != eof) {
    *iw++ = *ir++;
  }
}
std::cout << "\n";
```