Section 4
C++ Library

1. Standard Template Library
2. Files and streams
3. C++11 features

Section 4.1
Standard Template Library (STL)

1. Overview
2. Iterators
3. Containers
4. Algorithms

4.1.1 Overview

- What is the STL?
  - ➢ library of classes, and algorithms that operate on these classes

- The Good
  - ➢ it provides useful container classes and member functions

- The Bad
  - ➢ it can be non-intuitive to use

- The Ugly
  - ➢ it can severely degrade the performance of your program

©2018 Christine Laurendeau

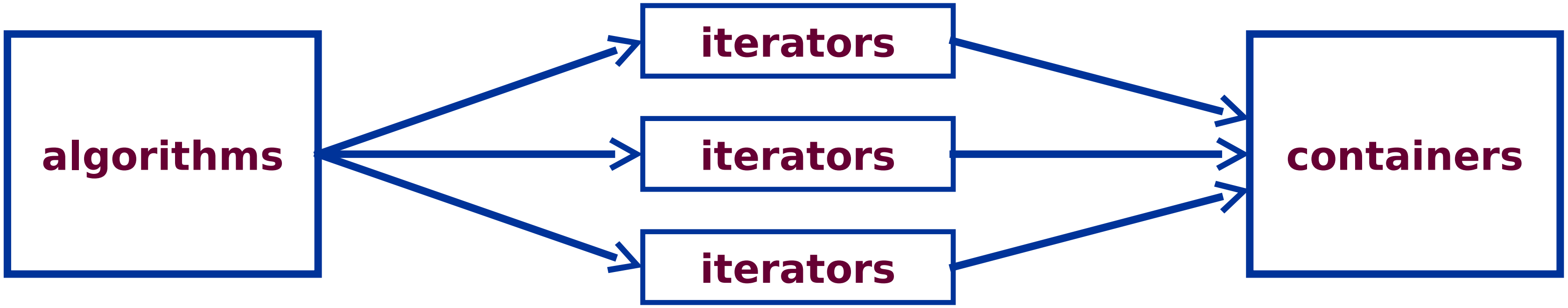
Overview (cont.)

- Main components:
  - containers
    - sequence containers
    - associative containers
    - container adapters
  - iterators
    - allow access to container elements
  - algorithms
    - global functions that perform operations on containers
    - typically use iterators to do this

©2018 Christine Laurendeau

Overview (cont.)

- Interactions between STL components



  - algorithms do not access containers directly
  - algorithms are independent from the underlying container
  - **`coding example <p1>`**

©2018 Christine Laurendeau

4.1.2 Iterators

- What is an iterator?
  - it is *conceptually* similar to a pointer
    - not syntactically or semantically similar
  - it allows access to the elements of an STL container
- Uses of an iterator
  - it can be used to traverse an STL container
    - sequence containers and associative containers only
  - it can be used as a parameter to many STL algorithms
  - **`coding example <p2>`**

Types of Iterators

- Forward iterators
  - they traverse a container from the first element to the last
  - types
    - **`iterator`**
    - **`const_iterator`**

- Reverse iterators
  - they traverse a container from the last element to the first
  - types
    - **`reverse_iterator`**
    - **`const_reverse_iterator`**

©2018 Christine Laurendeau

Categories of Iterators

- Categories:
  - input/output
    - only work on I/O streams
  - forward
    - only work on containers in the forward direction
  - bidirectional
    - work on containers in the forward and reverse directions
  - random access
    - allow direct access to any element in the container

- Each category includes operations in categories above

©2018 Christine Laurendeau

Categories of Iterators (cont.)

- Characteristics of iterator categories
  - the category is determined by the type of container
  - the category determines what algorithms can be used
- Examples:
  - I/O streams only support input/output iterators
  - **`list`** supports bidirectional iterators
  - **`vector`** supports random access iterators

Operations on Iterators

- All iterators support:
  - dereferencing: `*`
  - increment: `++`
  - assignment: `=`
  - equality/inequality: `==` `!=`

©2018 Christine Laurendeau

Operations on Iterators (cont.)

- Forward, bidirectional, random access iterators support:
  - container member function: **`begin()`**
    - points to the first element in the container
  - container member function: **`end()`**
    - points to just **past** the last element in the container

©2018 Christine Laurendeau

Operations on Iterators (cont.)

- Bidirectional, and random access iterators support:
  - container member function: **`rbegin()`**
    - points to the last element in the container
  - container member function: **`rend()`**
    - points to just **past** the first element in the container
  - decrement operator: **`--`**

Operations on Iterators (cont.)

- Random access iterators support operators:
  - subscript: `[ ]`
  - relational: `<` `>` `<=` `>=`
  - addition: `+` `+=`
  - subtraction: `–` `–=`

©2018 Christine Laurendeau

Operations on Iterators (cont.)

- Optimal performance using iterators
  - use *prefix* increment and decrement
    - this avoids creating temporary objects
  - store loop ending value in a variable
    - this avoids repeatedly calling **`end()`** member function

4.1.3 Containers

- What are STL containers?
  - they are collection classes
  - they are data structures that contain a collection of elements
  - all elements are of one type
  - many member functions are provided

- Types of STL containers
  - sequence containers
  - associative containers
  - container adapters

Containers (cont.)

- All STL containers provide:
  - default constructor, copy constructor, destructor, assignment op
  - insertion and deletion member functions
    - examples: **`insert()`**, **`delete()`**, **`clear()`**
    - many overloaded versions of these
  - size related member functions
    - examples: **`size()`**, **`empty()`**, **`max_size()`**
  - relational operators

©2018 Christine Laurendeau

Containers (cont.)

- Sequence and associative containers provide:
  - member functions for iteration
  - examples: **`begin()`**, **`end()`**, **`rbegin()`**, **`rend()`**

Containers (cont.)

- To use STL containers with your classes, you must provide:
  - operators for copying
    - copy constructor
    - assignment operator
  - comparison operators
    - equality
    - less-than

©2018 Christine Laurendeau

Streams as Containers

- What are streams?
  - sequences of bytes
    - files
    - console I/O
    - devices
  - … more on this later …

- These can be used on streams:
  - input/output iterators
  - some STL algorithms
    - example: **`copy`**
  - **`coding examples <p3> and <p4>`**

Sequence Containers

- What are sequence containers?
  - containers that retain the order of their elements

- Types of sequence containers:
  - **`vector`**
  - **`list`**
  - **`deque`**

- Useful member functions
  - **`front()`, `back()`, `push_back()`, `pop_back()`**

©2018 Christine Laurendeau

Vectors

- Characteristics of **`vector`**s:
  - storage
    - elements are stored contiguously in memory
    - **`vector`** grows as needed
    - allows direct access to any element
      - subscript operator or **`at()`** function
  - insertion and deletion
    - at the back: very efficient
    - anywhere else: causes the **`vector`** to be copied ☠
  - iterators
    - supports random access iterators
  - **`coding example <p5>`**

©2018 Christine Laurendeau

Lists

- Characteristics of **`list`**s:

  - storage
    - implemented as doubly-linked list
    - elements are *not* stored contiguously in memory
    - **`list`** grows as needed
    - does *not* allow direct access to elements

  - insertion and deletion
    - efficient anywhere within the **`list`**

  - iterators
    - supports bidirectional iterators
    - does *not* support random access iterators

  - **`coding example <p6>`**

Deques

- Characteristics of **`deque`**s (**d**ouble **e**nded **que**ues):
  - storage
    - elements are *not* stored contiguously
    - **`deque`** grows as needed
    - allows direct access to any element
  - insertion and deletion
    - at the front and back: very efficient
    - anywhere else: more efficient than **`vector`**, less than **`list`**
  - iterators
    - supports random access iterators
  - **`coding example <p7>`**

©2018 Christine Laurendeau

Associative Containers

- What are associative containers?
  - containers that store elements using keys
  - keys are stored in user-specified order
    - default is ascending
    - predicate can be used to specify order
- Types of associative containers:
  - **`set`**
  - **`multiset`**
  - **`map`**
  - **`multimap`**

©2018 Christine Laurendeau

Associative Containers (cont.)

- Characteristics of associative containers

➢ **`set`** and **`multiset`**:	store keys only
➢ **`map`** and **`multimap`**:	store combinations of key and value
➢ **`set`** and **`map`**:	do not allow duplicates
➢ **`multiset`** and **`multimap`**:	do allow duplicates

- Useful member functions
  - ➢ examples: **`insert()`**, **`find()`**, **`lower_bound()`**, **`upper_bound()`**

- Iterators
  - ➢ support bidirectional iterators

©2018 Christine Laurendeau

Container Adapters

- What are container adapters?
    - higher level containers providing restricted access to elements

- Types of container adapters
    - **`stack`**
    - **`queue`**
    - **`priority_queue`**

Container Adapters (cont.)

- Characteristics of container adapters
  - use underlying containers to store elements
    - **`stack`**
      - can be implemented with any sequence container
    - **`queue`**
      - can be implemented with **`deque`** or **`list`**
    - **`priority_queue`**
      - can be implemented with **`vector`** or **`deque`**
  - users can specify the underlying container
  - do *not* support iterators

©2018 Christine Laurendeau

4.1.4 Algorithms

- What are STL algorithms?
  - global function templates that operate on containers
    - they use iterators
    - they may work on non-STL containers, such as primitive arrays
  - indirect access to containers allows for more generic algorithms
    - they work with multiple types of containers

©2018 Christine Laurendeau

Algorithms (cont.)

- Characteristics of STL algorithms
  - often operate on containers using pairs of iterators
  - often return an iterator
  - each algorithm requires a specific category of iterators
    - so each algorithm needs specific type of container
    - also works with higher category iterators

- Useful algorithms
  - **`sort()`, `copy()`, `remove()`, `fill()`**

©2018 Christine Laurendeau