The Standard Template Library

- This topic introduces the C++ Standard Template Library containers, iterators, and algorithms
- Over the next series of topics, how to develop software using the
  - Vector, list, and deque sequence containers
  - Set, multiset, map, and multimap associative containers
  - Stack, queue, and priority queue adapters
  - Iterators to access container elements
  - Use the copy algorithm and ostream_iterator to output a container
  - Use the bitset container

- The Standard Library
  - Defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.

- Three key components of the Standard Library
  - containers (*templatized* data structures)
    - capable of storing objects of *almost* any data type
  - Iterators
  - Algorithms

- Three styles of container classes
  - *first-class containers*
  - *container adapters*
  - *near containers*

- Iterators
  - Have properties similar to those of *pointers*
  - Manipulate the container elements
- *Built-in arrays* can be manipulated by Standard Library algorithms
  - Pointers as iterators
- Manipulating containers with iterators
  - Convenient
  - Provides tremendous expressive power when combined with Standard Library algorithms
  - Can reduce many lines of code to a single statement

- Standard Library algorithms
  - Function templates
  - Perform *searching*, *sorting* and *comparing elements (or entire containers)*
  - Most use iterators to access container elements
  - Each has *minimum requirements* for the types of iterators that can be used with it
- Containers support specific iterator types, some more powerful than others.
  - The supported iterator type determines whether the container can be used with a specific algorithm

- Iterators encapsulate the mechanisms used to access container elements.
  - Encapsulation enables many of the algorithms to be applied to various containers *independently* of the underlying container implementation
  - Also enables you to create new algorithms that can process the elements of *multiple* container types

Software Engineering Observation 15.1

Avoid reinventing the wheel; program with the components of the C++ Standard Library.

Error-Prevention Tip 15.1

The prepackaged, templatized Standard Library containers are sufficient for most applications. Using the Standard Library helps you reduce testing and debugging time.

formance Tip 15.1

e Standard Library was conceived and designed for
formance and flexibility.

- The containers are divided into four major categories
  - sequence containers
  - ordered associative containers
  - unordered associative containers
  - container adapters

Sequence containers

`array`	Fixed size. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`forward_list`	Singly linked list, rapid insertion and deletion anywhere. New in C++11.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
`vector`	Rapid insertions and deletions at back. Direct access to any element.

Ordered associative containers—keys are maintained in sorted order

`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.

Unordered associative containers

`unordered_set`	Rapid lookup, no duplicates allowed.
`unordered_multiset`	Rapid lookup, duplicates allowed.
`unordered_map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`unordered_multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.

Container adapters

`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

- *Sequence containers*
  - Represent *linear* data structures
  - All of their elements are conceptually "lined up in a row"
  - `Array`s, `vector`s and linked lists

- *Associative containers*
  - *nonlinear* data structures
  - Can locate elements stored in the containers quickly
  - Can store sets of values or key-value pairs
  - As of C++11, the keys in associative containers are *immutable* (they cannot be modified)

- First-class containers
  - Sequence containers and associative containers
- Container adapters
  - `stack`, `queue` and `priority_queue` enable a program to view a sequence container in a constrained manner
- Class `string` supports the same functionality as a *sequence container*, but stores only character data

near containers

- built-in arrays
- `bitset`s for maintaining sets of flag values
- `valarrays` for performing high-speed *mathematical vector* (not to be confused with the `vector` container) operations

Considered *near containers* because they exhibit some, but not all, capabilities of the *first-class containers*

- Figure 15.2 describes the many functions that are commonly available in most Standard Library containers

- Overloaded operators <, <=, >, >= are *not* provided for the *unordered associative containers*
- Member functions `rbegin`, `rend`, `crbegin` and `crend` are not available in a `forward_list`
- Before using any container, you should study its capabilities

Member function	Description
default constructor	A constructor that *initializes an empty container*. Normally, each container has several constructors that provide different ways to initialize the container.
copy constructor	A constructor that initializes the container to be a *copy of an existing container* of the same type.
move constructor	A move constructor (new in C++11 and discussed in Chapter 24) moves the contents of an existing container of the same type into a new container. This avoids the overhead of copying each element of the argument container.
destructor	Destructor function for cleanup after a container is no longer needed.
`empty`	Returns `true` if there are *no* elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
copy `operator=`	Copies the elements of one container into another.

Member function	Description
move `operator=`	The move assignment operator (new in C++11 and discussed in Chapter 24) moves the elements of one container into another. This avoids the overhead of copying each element of the argument container.
`operator<`	Returns `true` if the contents of the first container are *less than* the second; otherwise, returns `false`.
`operator<=`	Returns `true` if the contents of the first container are *less than or equal to* the second; otherwise, returns `false`.
`operator>`	Returns `true` if the contents of the first container are *greater than* the second; otherwise, returns `false`.
`operator>=`	Returns `true` if the contents of the first container are *greater than or equal to* the second; otherwise, returns `false`.
`operator==`	Returns `true` if the contents of the first container are *equal to* the contents of the second; otherwise, returns `false`.
`operator!=`	Returns `true` if the contents of the first container are *not equal to* the contents of the second; otherwise, returns `false`.

Member function	Description
`swap`	Swaps the elements of two containers. As of C++11, there is now a non-member function version of swap that swaps the contents of its two arguments (which must be of the same container type) using move operations rather than copy operations.
`max_size`	Returns the *maximum number of elements* for a container.
`begin`	Overloaded to return either an `iterator` or a `const_iterator` that refers to the *first element* of the container.
`end`	Overloaded to return either an `iterator` or a `const_iterator` that refers to the *next position after the end* of the container.
`cbegin` (C++11)	Returns a `const_iterator` that refers to the container's *first element*.
`cend` (C++11)	Returns a `const_iterator` that refers to the *next position after the end* of the container.
`rbegin`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the *last element* of the container.

Member function	Description
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the *position before the first element* of the container.
`crbegin` (C++11)	Returns a `const_reverse_iterator` that refers to the *last element* of the container.
`crend` (C++11)	Returns a `const_reverse_iterator` that refers to the *position before the first element* of the container.
`erase`	Removes *one or more* elements from the container.
`clear`	Removes *all* elements from the container.

- The first-class container *nested types* (types defined inside each container class definition) are used in template-based declarations of variables, parameters to functions and return values from functions
  - `value_type` in each container
    - Represents the type of elements stored in the container
- `reverse_iterator` and `const_reverse_iterator`
- Are not provided by class `forward_list`

typedef	Description
allocator_type	The type of the object used to allocate the container's memory—not included in class template array.
value_type	The type of element stored in the container.
reference	A reference for the container's element type.
const_reference	A reference for the container's element type that can be used only to *read* elements in the container and to perform const operations.
pointer	A pointer for the container's element type.
const_pointer	A pointer for the container's element type that can be used only to *read* elements and to perform const operations.
iterator	An iterator that points to an element of the container's element type.
const_iterator	An iterator that points to an element of the container's element type. Used only only to *read* elements and to perform const operations.
reverse_iterator	A reverse iterator that points to an element of the container's element type. Used to iterate through a container in reverse.

typedef	Description
`const_reverse_iterator`	A reverse iterator that points to an element of the container's element type and can be used only to *read* elements and to perform `const` operations. Used to iterate through a container in reverse.
`difference_type`	The type of the result of subtracting two iterators that refer to the same container (`operator-` is not defined for iterators of `lists` and associative containers).
`size_type`	The type used to count items in a container and index through a sequence container (cannot index through a `list`).

- Before using a Standard Library container, it's important to ensure that the type of objects being stored in the container supports a *minimum* set of functionality
  - When an object is inserted into a container, a *copy* of the object is made.
  - Object type must provide a *copy constructor* and *copy assignment operator* (custom or default versions, depending on whether the class uses dynamic memory)
    - *Required only if default memberwise copy and default memberwise assignment do not perform proper copy and assignment operations for the element type*

- *Ordered associative containers* and many algorithms require elements to be *compared*
  - The object type must provide *less than (<)* and *equality (==) operators*
- As of C++11, objects can also be *moved* into container elements
  - In which case the object type needs a *move constructor* and *move assignment operator*

This topic introduced the C++ Standard Template Library containers, iterators, and algorithms

-