Cpt S 122 – Data Structures

Standard Template Library (STL)

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction to Standard Template Library (STL)
- Introduction to Containers
  - Templated data structure
  - `vector`, `list`, `deque`; `set`, `multiset`, `map`, `multimap`; `stack`, `queue`, `priority_queue`
- Introduction to Iterators
  - Access the elements of STL containers
- Introduction to Algorithms
  - Program with many STL algorithms
  - `equal`, `size`, `find`, `remove`, `replace`, `min`, `max`, `swap`, basic searching, sorting algorithms

Introduction to the Standard Template Library (STL)

- The Standard Template Library (STL) defines powerful, template-based, reusable components.
- Implement many common data structures and algorithms used to process those data structures.
- The STL was conceived and designed for performance and flexibility.

- STL has three key components
  - containers (popular templatized data structures)
  - iterators (to access the elements of STL containers)
  - algorithms (searching, sorting, comparing etc)

Advantage of STL

- Data structures.
  - linked lists, queues, stacks and trees.
  - objects are linked together with pointers.
- Pointer-based code is complex
  - the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors with no compiler complaints.
- Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work.
- An advantage of the STL is that you can reuse the STL *containers, iterators* and *algorithms*
  - implement common data structures and manipulations project-wide.

STL Pillars

Containers

Iterators

Algorithms

STL Containers

- Each STL container has associated member functions.
  - A subset of these member functions is defined in all STL containers.
- Example of STL containers
  - `vector` (a dynamically resizable array)
  - `list` (a doubly linked list)
  - `deque` (a double-ended queue, pronounced “deck”).
    - Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).
    - individual elements are accessed directly through random access iterators

STL Iterators

- STL iterators
  - properties similar to those of pointers
  - used by programs to manipulate the STL-container elements.
- Standard arrays can be manipulated by STL algorithms
  - using standard pointers as iterators.
- Manipulating containers with iterators is convenient
  - provides tremendous expressive power combined with STL algorithms
  - reduce many lines of code to a single statement.
- There are five categories of iterators
  - *input,*
  - *output,*
  - *forward,*
  - *bidirectional,*
  - *random*.

STL Algorithms

- STL algorithms are functions that perform common data manipulations
  - searching, sorting and comparing elements (or entire containers) etc.
- Each algorithm has minimum requirements for the types of iterators that can be used with it.
- Each first-class container supports specific iterator types, some more powerful than others.
- A container’s supported iterator type determines whether the container can be used with a specific algorithm.

Containers

- The STL containers are divided into three major categories
  - sequence containers
  - associative containers
  - container adapters

- There are three styles of container classes
  - first-class containers
  - adapters
  - near containers

Containers Types and Examples

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`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.

Fig. 22.1 | Standard Library container classes. (Part 1 of 2.)

Containers Types and Examples

Standard Library container class	Description
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.

Fig. 22.1 | Standard Library container classes. (Part 2 of 2.)

Containers Types

- The sequence containers represent linear data structures
  - vectors and linked lists.
- The associative containers are nonlinear containers
  - locate elements stored in the containers quickly
  - store sets of values or key/value pairs.
- The sequence containers and associative containers are collectively referred to as the first-class containers.
- Stacks and queues actually are constrained versions of sequential containers.
  - STL implements stacks and queues as container adapters
  - enable a program to view a *sequential container* in a *constrained manner*.
- near containers
  - C-like pointer-based arrays, `bitset`s for maintaining sets of flag values
  - exhibit capabilities similar to those of the first-class containers, but do not support all the first-class-container capabilities.

Containers' Common Member Functions

- Most STL containers provide similar functionality.
- Many generic operations, such as member function `size`, apply to all containers
  - other operations apply to subsets of similar containers.
  - encourages extensibility of the STL with new classes.
- [*Note: Overloaded operators* `<`, `<=`, `>`, `>=`, `==` *and* `!=` *are not provided for* `priority_queue`*s.]*

Containers' Common Member Functions

Member function	Description
default constructor	A constructor that initializes an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
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.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the contents of the first container is less than the second; otherwise, returns `false`.

Fig. 22.2 | Common member functions for most STL containers. (Part 1 of 3.)

Containers' Common Member Functions

Member function	Description
`operator<=`	Returns `true` if the contents of the first container is less than or equal to the second; otherwise, returns `false`.
`operator>`	Returns `true` if the contents of the first container is greater than the second; otherwise, returns `false`.
`operator>=`	Returns `true` if the contents of the first container is greater than or equal to the second; otherwise, returns `false`.
`operator==`	Returns `true` if the contents of the first container is equal to the second; otherwise, returns `false`.
`operator!=`	Returns `true` if the contents of the first container is not equal to the second; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.

Fig. 22.2 | Common member functions for most STL containers. (Part 2 of 3.)

Common Member Functions

Member function	Description
Functions found only in first-class containers	
`max_size`	Returns the maximum number of elements for a container.
`begin`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the first element of the container.
`end`	The two versions of this function return either an `iterator` or 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.
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to next position after the last element of the container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

Fig. 22.2 | Common member functions for most STL containers. (Part 3 of 3.)

Container Headers

Standard Library container headers	
`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	Contains both `queue` and `priority_queue`.
`<stack>`	
`<map>`	Contains both `map` and `multimap`.
`<set>`	Contains both `set` and `multiset`.
`<valarray>`	
`<bitset>`	

Fig. 22.3 | Standard Library container headers.

Container typedefs

`typedef`	**Description**
`allocator_type`	The type of the object used to allocate the container's memory.
`value_type`	The type of element stored in the container.
`reference`	A reference for the container's element type.
`const_reference`	A constant reference for the container's element type. Such a reference can be used only for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer for the container's element type.
`const_pointer`	A pointer for a constant of the container's element type.
`iterator`	An iterator that points to an element of the container's element type.
`const_iterator`	A constant iterator that points to an element of the container's element type and can be used only to *read* elements.
`reverse_iterator`	A reverse iterator that points to an element of the container's element type. This type of iterator is for iterating through a container in reverse.

Fig. 22.4 | `typedef`s found in first-class containers. (Part 1 of 2.)

Container typedefs

typedef	Description
const_reverse_iterator	A constant reverse iterator that points to an element of the container's element type and can be used only to *read* elements. This type of iterator is for iterating 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).

Fig. 22.4 | typedefs found in first-class containers. (Part 2 of 2.)

- These `typedef`s are used in generic declarations of variables, parameters to functions and return values from functions.

Introduction to Iterators

- Iterators have many similarities to pointers
  - point to first-class container elements.
- Certain iterator operations are uniform across containers.
- For example, the dereferencing operator (*) dereferences an iterator
  - get the element to which it points.
- The ++ operation on an iterator moves it to the container's next element

Iterators

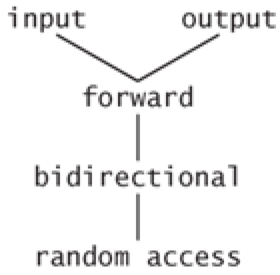
- STL first-class containers provide member functions `begin` and `end`.
- Function begin returns an iterator pointing to the first element of the container.
- Function end returns an iterator pointing to the first element past the end of the container (an element that doesn’t exist).

Iterators

- 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
  - determine whether the "moving iterator" (`i` in this case) has reached the end of the container.
- An object of type `iterator` refers to a container element that can be modified.
- An object of type `const_iterator` refers to a container element that cannot be modified.

Iterators Categories

- Different categories of STL iterators.
  - Each category provides a specific set of functionality.
- The hierarchy of iterator categories.
  - each iterator category supports all the functionality of the categories above it.
  - the “weakest” iterator types are at the top and the most powerful one is at the bottom.
  - this is not an inheritance hierarchy.



Iterators Categories

Category	Description
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.
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.
forward	Combines the capabilities of *input and output iterators* and retains their position in the container (as state information).
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.
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.

Fig. 22.6 | Iterator categories.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional

Fig. 22.8 | Iterator types supported by each container. (Part 1 of 2.)

Container	Type of iterator supported
Container adapters	
stack	no iterators supported
queue	no iterators supported
priority_queue	no iterators supported

Fig. 22.8 | Iterator types supported by each container. (Part 2 of 2.)

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

Fig. 22.9 | Iterator typedefs.

- Predefined iterator `typedef`s
  - found in the class definitions of the STL containers.
- Not every `typedef` is defined for every container.
- Use `const` versions of the iterators for traversing read-only containers.
- Use reverse iterators to traverse containers in the reverse direction.

Introduction to Algorithms

- STL algorithms can be used generically across a variety of containers.
- STL provides many algorithms to manipulate containers.
  - inserting, deleting, searching, sorting etc.
- The algorithms operate on container elements only indirectly through iterators.
- Many algorithms operate on sequences of elements defined by pairs of iterators
  - one pointing to the first element of the sequence
  - one pointing to one element past the last element

Introduction to Algorithms

- Algorithms often return iterators that indicate the results of the algorithms.
- Algorithm `find`
  - locates an element and returns an iterator to that element.
  - If the element is not found, `find` returns the "one past the `end`" iterator.
- The `find` algorithm can be used with any first-class STL container.
- Some algorithms demand powerful iterators; e.g., `sort` demands random-access iterators.

Introduction to Algorithms

- Mutating-sequence algorithms
  - the algorithms that result in modifications of the containers to which the algorithms are applied.
- Non-modifying sequence algorithms
  - the algorithms that do not result in modifications of the containers to which they're applied.

Modifying Algorithms

Mutating-sequence algorithms			
copy	partition	replace_copy	stable_partition
copy_backward	random_shuffle	replace_copy_if	swap
fill	remove	replace_if	swap_ranges
fill_n	remove_copy	reverse	transform
generate	remove_copy_if	reverse_copy	unique
generate_n	remove_if	rotate	unique_copy
iter_swap	replace	rotate_copy	

Fig. 22.11 | Mutating-sequence algorithms.

Non-modifying Algorithms

Nonmodifying sequence algorithms			
adjacent_find	equal	find_end	mismatch
count	find	find_first_of	search
count_if	find_each	find_if	search_n

Fig. 22.12 | Nonmodifying sequence algorithms.