The C++ Standard Template Library (STL) contains templates for useful algorithms and data structures (ADTs). The two types of STL classes we're interested in are containers and iterators. A *container* stores data elements organized is some fashion with *iterators* being generalized pointers to access individual data elements and walk through data elements within a container. Iterators for most container class have member functions:

- `begin()` Return iterator to beginning (public member function)
- `end()` Return iterator to just past the end (public member function)
- `rbegin()` Return reverse iterator to reverse beginning (public member function)
- `rend()` Return reverse iterator to reverse end (public member function)

Iterators can be moved through the container via the ++ and -- operators.

Container Type	Container	Description	Iterator Type
sequence - elements access by their linear position	vector (pp. 427-38)	Expandable array capable of quickly adding values to its end (i.e., push_back member)	Random-access (operators: [] ++ --)
	deque	Expandable array capable of quickly adding values to its front or end	Random-access (operators: [] ++ --)
	list	Doubly-linked list capable of quickly adding values at any location, but do not provide random access.	Bidirectional (operators: ++ --)
associative - keys used to rapidly access elements	set	Stores a set of keys with no duplicate keys allowed	Bidirectional (operators: ++ --)
	multiset	Stores a set of keys with duplicate keys allowed	Bidirectional (operators: ++ --)
	map	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates not allowed.	Bidirectional (operators: ++ --)
	multimap	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed.	Bidirectional (operators: ++ --)

In addition to the above containers, container adaptors exist for stacks, queues, and priority queues. *Container adaptors* are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements. In general these container adaptors behave as:

Container Adaptor	Description	Member Functions
stack	LIFO (Last-In-First-Out) stack with access restricted to one end (the "top")	empty, pop, push, size, top
queue	FIFO (First-In-First-Out) queue	back, empty, front, pop, push, size
priority_queue	Returns the highest element	empty, pop, push, size, top

The STL includes template functions of common algorithms to manipulate elements within a container. The following table lists some algorithms.

Example	Description
sort(iter1, iter2)	Sorts the elements (ascending order) in the range [iter1, iter2).
random_shuffle(iter1, iter2)	Randomly reorders the elements in the range [iter1, iter2)
binary_search(iter1, iter2, target)	Returns `true` if the target is found in the range [iter1, iter2); otherwise it returns `false`. (The range from iter1 to iter2 must be sorted previously)
iter3 = find(iter1, iter2, target)	Returns an iterator to the first object matching target in the range [iter1, iter2); otherwise it returns `false`.
iter3 = count(iter1, iter2, value)	Returns number of times that value occurs in the range [iter1, iter2).
iter3 = max_element(iter1, iter2)	Returns an iterator to the largest element in the range [iter1, iter2).
iter3 = min_element(iter1, iter2)	Returns an iterator to the smallest element in the range [iter1, iter2).

```
// Experiment with STL containers, iterators, and algorithms
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
using namespace std;

int main () {
  vector<int> myVector;
  vector<int>::iterator left, right;
  set<int> mySet;
  set<int>::iterator setIter1, setIter2;

  // set some content in the vector:
  for (int i=0; i<10; i++) {
        int value = rand()%100;
        myVector.push_back(value);
        mySet.insert(value);
  } // end for

  cout << "vector values: ";
  for (left = myVector.begin(); left < myVector.end(); left++) {
        cout << *left << " ";
  } // end for
  cout << endl << endl;

  cout << "set values: ";
  for (setIter1 = mySet.begin(); setIter1 != mySet.end(); setIter1++) {
        cout << *setIter1 << " ";
  } // end for
  cout << endl << endl;

  left = myVector.begin();
  right = myVector.end();
  left++;
  left++;
  right--;
  sort(left, right);

  cout << "vector values after sort: ";
  for (left = myVector.begin(); left < myVector.end(); left++) {
        cout << *left << " ";
  } // end for
  cout << endl;

  return 0;
} // end main
```

Output for above program:

```
vector values: 83 86 77 15 93 35 86 92 49 21

set values: 15 21 35 49 77 83 86 92 93

vector values after sort: 83 86 15 35 49 77 86 92 93 21
```

a) Why do you suppose the vector has the expected 10 values, but the set only has 9 values?

b) In the above code, the for-loop condition why do you suppose that "!=" must be used instead of "<" when walks setIter1 down mySet?

c) Why is the vector not sorted completely?

```
// Experiment with STL map
#include <iostream>
#include <map>
#include <utility>  // to get pair class
#include <string>
#include <algorithm>
using namespace std;

int main () {
  map<string, int> myPhoneBook;
  map<string,int>::iterator mapIter1, mapIter2;

  string person;
  int phoneNumber;

  // Examples of assigning map container some content

  // 1) Assignment using array index notation
  myPhoneBook["Diesburg"] = 2737395;
  myPhoneBook["Schafer"] = 2732187;

  // 2) Assignment using member function insert() and STL pair
  person = "East";
  phoneNumber = 2732939;
  myPhoneBook.insert(pair<string, int>::pair(person,phoneNumber));

  // 3) Assignment using member function insert() and "value_type()"
  person = "Fienup";
  phoneNumber = 2735918;
  myPhoneBook.insert(map<string, int>::value_type(person,phoneNumber));

  myPhoneBook.insert(make_pair("Gray",2735917));

  cout << "myPhoneBook values: " << endl;
  // NOTE:  Interators are pointers to pair values with
  // "first" attribute being the key and "second" being the value
  for (mapIter1 = myPhoneBook.begin(); mapIter1 != myPhoneBook.end(); mapIter1++) {
    cout << mapIter1->first << ":  " << mapIter1->second << endl;
  } // end for
  cout << endl;

  cout << "Enter the name to look up a phone #: ";
  cin >> person;

  cout << person << ": " <<  myPhoneBook[person] << endl;

  return 0;
} // end main
```

Output for above program:

```
myPhoneBook values:
Diesburg:  2737395
East:  2732939
Fienup:  2735918
Gray:  2735917
Schafer:  2732187

Enter the name to look up a phone #: Fienup
Fienup: 2735918
```

d) How would you use a vector of to determine if a string contains a palindrome ("Never odd or even" or "race car")?

			Sequence containers			Associative containers				
#include Headers			<vector>	<deque>	<list>	<set>		<map>		<bitset>
Members		big-oh	vector	deque	list	set	multiset	map	multimap	bitset
	constructor	*	constructor	constructor	constructor	constructor	constructor	constructor	constructor	constructor
	destructor	O(n)	destructor	destructor	destructor	destructor	destructor	destructor	destructor	
	operator=	O(n)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operators
iterators	begin	O(1)	begin	begin	begin	begin	begin	begin		
	end	O(1)	end	end	end	end	end	end		
	rbegin	O(1)	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	begin	
	rend	O(1)	rend	rend	rend	rend	rend	rend		
capacity	size	*	size	size	size	size	size	size	size	size
	max_size	*	max_size	max_size	max_size	max_size	max_size	max_size		
	empty	O(1)	empty	empty	empty	empty	empty	empty		
	resize	O(n)	resize	resize	resize					
element access	front	O(1)	front	front	front					
	back	O(1)	back	back	back					
	operator[]	*	operator[]	operator[]				operator[]		operator[]
	at	O(1)	at	at						
modifiers	assign	O(n)	assign	assign	assign					
	insert	*	insert	insert	insert	insert	insert	insert	insert	
	erase	*	erase	erase	erase	erase	erase	erase	erase	
	swap	O(1)	swap	swap	swap	swap	swap	swap	swap	
	clear	O(n)	clear	clear	clear	clear	clear	clear	clear	
	push_front	O(1)		push_front	push_front					
	pop_front	O(1)		pop_front	pop_front					
	push_back	O(1)	push_back	push_back	push_back					
	pop_back	O(1)	pop_back	pop_back	pop_back					
observers	key_comp	O(1)				key_comp	key_comp	key_comp	key_comp	
	value_comp	O(1)				value_comp	value_comp	value_comp	value_comp	
operations	find	O(log n)				find	find	find	find	
	count	O(log n)				count	count	count	count	count
	lower_bound	O(log n)				lower_bound	lower_bound	lower_bound	lower_bound	
	upper_bound	O(log n)				upper_bound	upper_bound	upper_bound	upper_bound	
	equal_range	O(log n)				equal_range	equal_range	equal_range	equal_range	
unique members			capacity reserve		splice remove remove_if unique merge sort reverse					set reset flip to_ulong to_string test any none

Amortized complexity shown. O(1) constant < O(log n) logarithmic < O(n) linear; *=depends on container (http://www.cplusplus.com/reference/stl/)