STL Containers – Supplementary Notes

Jim Fawcett
CSE 687 - Spring 2002

1. **Every container allocates and manages its own storage.**

2. **Type definitions common to all containers:**

– C::value_type	type of values held in container
– C::reference	value_type&
– C::const_reference	
– C::iterator	
– C::const_iterator	
– C::reverse_iterator	
– C::const_reverse_iterator	
– C::difference_type	difference between iterators
– C::size_type	size of container

3. Member functions common to all containers:

– C( )	default constructor
– C(c), C c2(c1)	copy constructor
– ~C( )	destructor
– c.begin()	returns an iterator to first element
– c.end()	returns an iterator after last element
– c.rbegin()	returns a reverse iterator to last elem.
– c.rend()	returns a reverse iterator before first elem.
– c1 == c2	equality comparison for same type cont.
– c1 != c2	“
– c.size()	returns number of elements. in cont.
– c.max_size()	returns size() of largest number of elements.
– c.empty()	returns true if cont. is empty
– c1 < c2	lexicographic comparison
– c1 > c2	“
– c1 <= c2	“
– c1 >= c2	“
– c1 = c2	assignment operation
– c1.swap(c2)	swaps two containers

4. Sequence containers

- vector — simulates an expandable array, occupying contiguous memory
- list — based on doubly linked list
- deque — a double ended queue, which uses a directory managing blocks of contiguousmemory

5. Member functions common to all sequence containers:

– C(n,t)	constructs a sequence of n copies of t
– C(iter1,iter2)	constructs a sequence equal to the range [iter1,iter2)
– c.insert(iter,t)	inserts a copy of t before iter. Returns an iter to t.
– c.insert(iter,n,t)	inserts n copies of t before iter.
– c.insert(iter1,iter2,iter3)	inserts the sequence [iter2,iter3) before iter1
– c.erase(iter)	erases the element pointed to by iter
– c.erase(iter1,iter2)	erases elements in range [iter1,iter2)

6. Invalidation of iterators

- **Invalidation of iterators into vectors:**
  - insertion in a vector invalidates iterators from the point of insertion to the end of the vector.
  - if insertion causes reallocation to provide more memory then all iterators become invalid.
  - erase invalidates all iterators at and past the point of erasure.
  - a safe strategy is to assume that any iterator into a vector becomes invalid after either insertion or erasure.
- **Invalidation of iterators into deques:**
  - insertion and erasure in the interior invalidates all iterators.
- **Invalidation of iterators into lists:**
  - list insertions never invalidate iterators and erase invalidates only iterators pointing to the erased items.
- **Use of invalid iterators:**
  - The only safe things you can do with an invalid iterator is to reinitialize it by assigning a new iterator value to it or destroy it.

7. Sorted associative containers (all are based on balanced red-black tree):

– set	set of elements sorted by value with no duplicates
– multi-set	set of elements sorted by value with duplicates
– map	set of <key,value> pairs sorted on key with no duplicates
– multi-map	set of <key,value> pairs sorted on key with duplicates

8. Types common to all sorted associative containers:

– C::key_type	type of keys used to instantiate C
– C::key_compare	type of the comparison type used to instantiate C
– C::value_compare	type for comparing objects of C::value_type

9. Invalidation of iterators with associative containers:

– insertion does not invalidate any iterators referring to container elements.
– erasure invalidates only iterators pointing to erased elements.

10. Member functions common to all sorted associative containers:

– C( )	void constructor
– C(comp)	constructs empty container using comp for comparisons
– C(iter1,iter21)	constructs empty container and inserts elements from [iter1,iter2) into it.
– C(iter1,iter2,comp)	same as above except that comp is used for comparisons.
– c.key_comp()	returns c's key comparison object
– c.value_comp()	returns c's value comparison object
– c.insert(t)	for sets and maps inserts t if and only if there is no equivalent key stored, returns pair<iterator,bool>. The bool indicates if insertion succeeded and iterator points to the element equivalent to t. for multi-sets and multi-maps inserts t and returns an iterator pointing to the inserted t
– c.insert(iter,t)	same as above except that iter is a hint about where to start search
– c.insert(iter1,iter2)	inserts elements from the sequence [iter1,iter2)
– c.erase(k1)	erases all elements in the container with key equal to k1. Returns the number of elements erased.
– c.erase(iter)	erases the element pointed to.
– c.erase(iter1,iter2)	erases all elements in the range [iter1,iter2).
– c.find(k1)	returns an iterator pointing to an element with key equal to k1 or to c.end( ) if no such element is found.
– c.count(k1)	returns the number of elements with key equivalent to k1
– c.lower_bound(k1)	returns an iterator pointing to first element with key not less than k1.
– c.upper_bound(k1)	returns an iterator pointing to first element with key greater than k1.
– c.equal_range(k1)	returns a pair of iterators with first lower_bound and second upper_bound

STL Iterators

11. Iterators extend the functionality of native pointers.

– Any container, c, defines valid iterators pointing to the first element, returned by c.begin() and one past the last element, returned by c.end().
– an iterator range is a pair of iterators that serve as the beginning and end markers of some operation on container values. Range [iter1, iter2) includes the values pointed to by iter1 through the value pointed to by the predecessor of iter2.
– iterators can be dereferenced, e.g., if iter is an iterator for some container c, *iter returns value_type whenever it is in the range [c.begin(), c.end())
– if iter is in the range [c.begin(), c.end()) then either iter++ stays in the range or is equivalent to c.end().
– iterators can be mutable or constant depending on whether the result of operator* acts like a reference or a reference to a const.

12. Input iterator requirements:

– I(i)	copy constructor
– i == j	returns true if iterator i is equivalent to iterator j
– i != j	returns true if and only if i == j returns false
– *i	returns value_type if dereferenceable. If i == j then it must be true that *i == *j. Note: don't attempt to write to *i as it may not be an l-value.
– i->m	equivalent to (*i).m
– ++i	returns an iterator pointing to the successor element to *i or to c.end();
– i++ `	returns i then points to the successor of *i or to c.end()

– Algorithms that use input iterators should be single-pass.

13. Output iterator requirements:

– I(i)	copy constructor
– *i = t	t is assigned through the iterator.
– ++i	returns an iterator pointing to the successor element to *i or to c.end()
– i++	returns i then points to the successor of *i or to c.end()

– The only valid use of *i is on the left of an assignement. Algorithms that use output iterators should be single-pass.

14. Forward iterator requirements:

– I()	void constructor, result may be a singular value
– I(i)	result must satisfy i == I(i);
– i == j	true if i is equivalent to j
– i != j	true if i==j is false
– i = j	result must satisfy i == j
– *i	returns value_type if dereferenceable. If i == j then *i == *j must be true. If i is mutable then *i = t is valid.
– i -> m	equivalent to (*i).m
– ++i	returns an iterator pointing to the successor element to *i or to c.end() i == j and i dereferenceable implies that ++i == ++ j.
– i++	returns i then points to the successor of *i or to c.end()

15. Bidirectional iterator requirements:

– meets all requirements of Forward iterators.

– - - i	Assume that there is a j such that ++j = i. Then - - i refers to the same element as j. It must be true that --(++i) = i and if - -i == --j then i == j.
– i- -	returns i then points to the predecessor of i

16. Random access iterator requirements:

- meets the requirements for a bidirectional iterator.
- i += n the result must be equivalent to incrementing i n times.
- i + n returns an iterator equivalent to i += n.
- i -= n the result must be equivalent to decrementing i n times.
- i – n returns an iterator equivalent to i -= n.
- i – j returns a value of type distance. If i + n = j then j – 1 == n
- i[n] equivalent to *(i + n)
- i < j must be a total order relationship returning bool
- i > j must be a total order relationship returning true whenever i < j || i == j is false
- i <= j must be a total order relationship equivalent to !(i > j)
- i >= j must be a total order relationship equivalent to !(i < j)

17. Algorithms – Non modifying (Prata, C++ Primer Plus, Third Edition, Waite Group)

for_each	Applies a non-modifying function object to each element in a range
find	Finds the first occurrence of a value in a range
find_if	finds the first value satisfying a predicate test criterion in a range
find_end	finds the last occurrence of a subsequence whose values match the values of a second sequence. Matching may be by equality or by applying a binary predicate.
find_first_of	Finds the first occurrence of any element of a second sequence that matches a value in the first sequence. Matching may be by equality or be evaluated with a binary predicate.
adjacent_find	Finds the first element that matches the element immediately following it. Matching may be by equality or evaluated with a binary predicate.
count	Returns the number of times a given value occurs in a range.
count_if	Returns the number of times a given value matches values in a range, with a match determined by using a binary predicate.
mismatch	Finds the first element in one range that does not match the corresponding element in a second range and returns iterators to both. Matching may be by equality or be evaluated with a binary predicate.
Equal	Returns true if each element in one range matches the corresponding element in a second range. Matching may be by equality or evaluated with a binary predicate.
search	Finds the first occurrence of a subsequence whose values match the values of a second sequence. Matching may be by equality or by applying a binary predicate.
search_n	Finds the first subsequence of n elements that each match a given value. Matching may be by equality or applying a binary predicate.

Example:

```
template <class T>
class Sum
{
    Sum() : sum_(0) {}
    void operator()(T& t) { sum_ += t; }
    result() { return sum_; }
    private: T sum_;
}

std::list<int> li;

// push on some elements
// foreach is the only algorithm that returns its operation, e.g., Sum()

int sum = foreach(li.begin(),li.end(),Sum()).result();
```

18. Algorithms – Modifying (Prata, C++ Primer Plus, Third Edition, Waite Group)

copy	Copies elements from a range to a location identified by an iterator.
copy_backward	Copies elements from a range to a location identified by an iterator. Copying begins at the end of the range and proceeds backwards.
Swap	Exchanges two values stored at locations specified by references.
Swap_ranges	Exchanges corresponding values in two ranges.
iter_swap	Exchanges two values stored at locations specified by iterators.
transform	Applies a function object to each element in a range (or to each pair of elements in a pair of ranges), copying the return value to the corresponding location of another range.
replace	Replaces each occurrence of a value in a range with another value.
replace_if	Replaces each occurrence of a value in a range with another value if a predicate function object applied to the original value returns true.
replace_copy	Copies one range to another, replacing each value for which a predicate function object is true with an indicated value.
fill	Sets each value in a range to an indicated value.
fill_n	Sets n consecutive elements to a value.
generate	Sets each value in a range to the return value of a generator, which is a function object that takes no arguments.
generate_n	Sets the first n values in a range to the return value of a generator, which is a function object that takes no arguments.
remove	Removes all occurrences of a value from a range and returns a past-the-end iterator for the resulting range.
remove_if	Removes all occurrences of values for which a predicate object returns true from a range and returns a past-the-end iterator for the resulting range.

remove_copy	Copies elements from one range to another, omitting elements that equal a specified value.
remove_copy_if	Copies elements from one range to another, omitting elements for which a predicate function object returns true.
unique	Reduces each sequence of two or more equivalent elements in a range to a single element.
unique_copy	Copies elements from one range to another, reducing each sequence of two or more equivalent elements to one.
reverse	Reverses the elements in a range.
reverse_copy	Copies a range in reverse order to a second range.
Rotate	Treats a range as a circular ordering and rotates the elements left.
Rotate_copy	Copies one range to another in a rotated order.
Random_shuffle	Randomly rearranges the elements in a range.
partition	Places all the elements that satisfy a predicate function object before all elements that don't.
Stable_partition	Places all the elements that satisfy a predicate function object before all elements that don't. The relative order of elements in each group is preserved.

19. Sorting & Related Operations (Prata, C++ Primer Plus, Third Edition, Waite Group)

sort	Sorts a range.
stable_sort	Sorts a range, preserving the relative order of equivalent elements.
partial_sort	Partially sorts a range, providing the first n elements of a full sort.
partial_sort_copy	Copies a partially sorted range to another range.
nth_element	Given an iterator into a range, finds the element that would be there if the range were sorted, and places that element there.
lower_bound	Given a value, finds the first position in a sorted range before which the value can be inserted while maintaining the ordering.
upper_bound	Given a value, finds the last position in a sorted range before which the value can be inserted while maintaining the ordering.
equal_range	Given a value, finds the largest subrange of a sorted range such that the vlue can be inserted before any element in the subrange without violating the ordering.
binary_search	Returns true if a sorted range contains a value equivalent to a given value, and false otherwise.
merge	Merges two sorted ranges into a third range.
in-place_merge	Merges two consecutive sorted ranges in place.
includes	Returns true if every element in one set also is found in another set.
set_union	Constructs the union of two sets, which is a set containing all elements present in either set.
set_intersection	Constructs the intersection of two sets, which is a set containing only those elements found in both sets.
set_difference	Constructs the difference of two sets, which is a set containing only those elements found in the first set but not the second.

set_symmetric_difference	Constructs a set consisting of elements found in one set or the other, but not both.
make_heap	Converts a range to heap.
push_heap	Adds an element to a heap.
pop_heap	Removes the largest element from a heap.
sort_heap	Sorts a heap.
min	Returns the lesser of two values.
max	Returns the greater of two values.
min_element	Finds the first occurrence of the smallest value in a range.
max_element	Finds the first occurrence of the largest value in a range.
lexicographic_compare	Compares two sequences lexicographically, returning true if the first sequence is lexicographically less than the second, and false otherwise.
next_permutation	Generates the next permutation in a sequence.
previous_permutation	Generates the preceding permutation in a sequence.

20. Predefined Function Objects (Josuttis, C++ Standard Library, Addison-Wesley)

<table>
<tr><th>Expression</th><th>Effect</th></tr>
<tr><td>negate<T>()</td><td>- param</td></tr>
<tr><td>plus<T>()</td><td>param1 + param2</td></tr>
<tr><td>minus<T>()</td><td>param1 – param2</td></tr>
<tr><td>multiplies<T>()</td><td>param1 * param2</td></tr>
<tr><td>divides<T>()</td><td>param1 / param2</td></tr>
<tr><td>modulus<T>()</td><td>param1 % param2</td></tr>
<tr><td>equal_to<T>()</td><td>param1 == param2</td></tr>
<tr><td>not_equal_to<T>()</td><td>param1 != param2</td></tr>
<tr><td>less<T>()</td><td>param1 < param2</td></tr>
<tr><td>greater<T>()</td><td>param1 > param2</td></tr>
<tr><td>less_equal<T>()</td><td>param1 <= param2</td></tr>
<tr><td>greater_equal<T>()</td><td>param1 >= param2</td></tr>
<tr><td>logical_not<T>()</td><td>! param</td></tr>
<tr><td>logical_and<T>()</td><td>param1 && param2</td></tr>
<tr><td>logical_or<T>()</td><td>param1 || param2</td></tr>
</table>

Example:

```
std::list<int> li;
// push on some elements
std::list<int>::iterator itPos;
// find first positive element in list
itPos = find_if(li.begin(),li.end(),bind2nd(greater<int>(),0);
```