Standard Template Library Quick Reference

Containers

Containers are general-purpose template classes that are designed to store objects of almost any type. They are useful by themselves, but become even more powerful when combined with other concepts such as iterators and algorithms.

Standard Containers

Name	***Header***	***Description***
vector<T, Alloc>	<vector> or <vector.h>	Acts very similar to a standard array that can grow to accommodate additional elements.
deque<T, Alloc>	<deque> or <deque.h>	This is a double-ended queue which is efficient at adding or removing elements from either the beginning or end of the queue.
list<T, Alloc>	<list> or <list.h>	A doubly-link list container that uses pointers to nodes. One pointer stores the location of the next node and the second pointer stores the location of the previous node. Faster than the vector and deque containers at some operations, notably adding or removing elements from the middle of the container.

NOTE: The Alloc parameter allows you to define your own custom memory allocator if needed. A custom memory allocator is useful is some situations such as when working with embedded systems which do not have general-purpose malloc/free or new/delete operators.

Container Operation Costs

Operation	***C-Array***	***vector***	***deque***	***list***
Insert/erase at start	N/A	linear	constant	constant
Insert/erase at end	N/A	constant	constant	constant
Insert/erase in middle	N/A	linear	linear	constant
Access first element	constant	constant	constant	constant
Access last element	constant	constant	constant	constant
Access middle element	constant	constant	constant	linear

Operation	***C-Array***	***vector***	***deque***	***list***
Overhead	none	low	medium	high

Vector Advantages

- Vectors can be dynamically resized; when you run out of space it automatically grows
- Elements of a vector can be added or removed from the interior without needing to write custom code
- You can quickly access the start the or end of the vector, without knowing it size in advance
- You can iterate forward or backward through a vector
- It is a simple matter to add bounds checking for both operator[] and pointer dereferencing
- Objects in a vector can be stored in any kind of memory with the help of a custom allocator
- Unlike standard arrays, vector have usable assignment and comparison operators.

Vector Disadvantages

- Most implementations have to store a total of 3 memory pointers, compared to one pointer for a standard C-style dynamically allocated array. This does use up very much extra memory, so it is usually not a great burden.
- A vector will never release memory, even when the number of elements in the vector is reduced.

Deque Advantages

- Since a deque acts a lot like a vector, it has the same advantages as using a vector when compared to standard C-style arrays
- It can grow in either direction (front or back) equally well
- It is often faster than a vector when the container needs to grow to hold new elements

Deque Disadvantages

- The operator[] is not as fast as vector's operator[], although it is still pretty fast
- Iterating over a deque is also slower than iterating over a vector

List Advantages

- Very fast element insertion/removal in the middle of the list
- Implements its own memory management system which actually can be helpful

on some platforms

List Disadvantages

- No random access iterator (which means no operator[])
- Uses extra memory to track next/previous node pointers (lists are best used for large structure, not small data elements list a character)

General Guideline

Use a *vector*<> whenever possible since it has the lowest overhead and best overall performance. However, if you are going to add and removing items from the middle of the collection often, then consider using a *list*<>. Use a *deque*<> whenever you will be inserting elements at the head or end most of the time, but very seldom from the middle of the collection.

Container Adapters

The following containers are specialized containers that use one of the standard containers to actually store the elements they manage. Basically they are wrappers around one of the standard container templates that provide a restricted set of operations.

Container	***Header***	***Description***
stack<T, Sequence>	<stack> or <stack.h>	Implements a standard LIFO (Last-In, First-Out) container. You will probably use the push() and pop() members most often.
Queue<T, Sequence>	<queue> or <queue.h>	Implements a standard FIFO (First-In, First-Out) container. This container does not allow iteration. You will probably use the push() and top()/pop() members most often.
priority_queue	<queue> or <stack.h>	This container implements a FIFO, with one small difference. The largest element is always the first item returned by the top() and pop() methods.

Associative Containers

An associative containers stores objects based on key values.

Container	***Header***	***Description***
set<Key, Compare, Alloc>	<set> or <set.h>	This container holds a unique collection of objects in sorted sequence. The Compare parameter defines the function/functor to use for comparing the objects (default is less<Key>) and the Alloc parameter is for a custom memory allocator.
multiset<Key, Compare, Alloc>	<set> or <multiset.h>	This container holds a collection of objects in sorted sequence. Unlike a standard set<>, this type of container allows duplicate keys.
map<Key, Data, Compare, Alloc>	<map> or <map.h>	Similar to a set<> container, except the key is distinct from the data being stored. Internally a map stores pair<const Key, Data> elements, organized by Key values. The pair<> is a helper template. All Key values must be unique.
map<Key, Data, Compare, Alloc>	<map> or <multimap.h>	Works like the standard map<> template, except duplicate Key values are allowed.

Iterators

The standard template library makes heavy use of iterators, which basically acts like pointers. They are used to indicate a position within a collection of elements and are most often used to process a range of elements.

Iterator Categories

Iterator Category	***Description***
Input Iterator	This type of iterator allows you to read the element it references. It does not allow you to change the element.
Output Iterator	This type of iterator grants permission to write an element, but does not guarantee read access is available (although it may allow reading the element also.)
Forward Iterator	A forward iterator generally supports both Input and Output operations, unless the iterator is constant, which restricts its usage to reading only. The difference between a Forward iterator and an Input or Output iterator is that Forward iterators can usually be used with multi-pass algorithms.
Bidirectional Iterator	This is very similar to a Forward iterator, except it can be both incremented and decremented. Not all of the container templates support Bidirectional iterators.
Random Access Iterators	This type of iterator supports incrementing, decrementing and also adding and subtracting arbitrary offsets, subscripting and more. They act much more like traditional pointers than the other iterators.

Concrete Iterator Template Classes

Iterator Class	***Description***
istream_iterator<T,Distance>	Reads objects of type T from an input stream (such as cin). Stops when it reaches the end of the stream. Used often with the **copy()** algorithm.
ostream_iterator<T>	Writes objects of type T to an output streams (such as cout). Used often with the **copy()** algorithm.

Iterator Class	***Description***
reverse_iterator<RandomAccessIterator, T, Reference, Distance>	This iterator reverses the meaning of the increment (++) and decrement (--) operators.
insert_iterator<Container>	This iterator is used to insert objects into a container. It will track of the next point of insertion and advance automatically whenever a new element is inserted.
front_insert_iterator<FronInsertionSequence>	This iterator class is an output iterator that always inserts new elements at the front of the container. The *FrontInsertSequence* means you can only use this type of iterator with containers that have the **front()**, **push_front()** and **pop_front()** methods. Primarily this includes the *deque*<> and *list*<> template classes.
back_insert_iterator<BackInsertionSequence>	This iterator class is an output iterator that always appends new elements at the end of a container. The *BackInsertionSequence* means you can only use this type of iterator with containers that have the **back()**, **push_back()** and **pop_back()** methods. Primarily this includes the *vector*<>, *list*<> and *deque*<> template classes.

Algorithms

The Standard Template Library also includes a large number of template functions collectively referred to as algorithms. The combination of the container classes with the algorithm template functions provides C++ with many advanced and powerful constructs.

Algorithm Types

Algorithm Type	***Description***
Non-mutating	Algorithms in this category are used to process and/or search a container, but never modify the container's elements.
Mutating	Algorithms in this category are used to modify containers in some way.
Sorting	There are a number of algorithms available for sorting, searching and merging containers and their elements.
Generalized Numeric	These types of algorithms are used to perform some kind of mathematical operation against elements in a container.

Non-Mutating Algorithms

Remember, the non-mutating algorithms never modify the containers they are working on.

Algorithm	***Description***
UnaryFunction **for_each**(InputIterator first, InputIterator last, UnaryFunction f)	Iterates all of the elements from *first* to *last* and calls function *f* once per element. The return value is sometimes useful when dealing with functors.
InputIterator **find**(InputIterator first, InputIterator last, const EqualityComparable& value)	Attempts to locate *value* in the elements pointed to by *first* and *last*. The *value* is usually the same type as the elements in the container, but conversions are performed if needed. Returns an iterator that points to the first match if found. Returns *last* if the item cannot be found.

Algorithm	*Description*
InputIterator **find_if**(InputIterator first, InputIterator last, Predicate pred)	Similar to the **find()** algorithm, except instead of comparing values directly, it passes each element in the range to a helper function (or functor) that tests the object in some way and returns a boolean value. If the helper function returns *true*, the search stops and an iterator to the element is returned, otherwise *last* is returned.
ForwardIterator **adjacent_find**(ForwardIterator first, ForwardIterator last) ForwardIterator **adjacent_find**(ForwardIterator first, ForwardIterator last, BinaryPredicate binary_pred)	This function iterates over the container's elements to locate the first of 2 iterators that are valid. The first version is not very useful, the second works like **find_if()** so you can call a custom function (or functor) that tests adjacent elements and returns a boolean.
InputIterator **find_first_of**(InputIterator first1, InputIterator last1, ForwardIterator first2, ForwardIterator last2) InputIterator **find_first_of**(InputIterator first1, InputIterator last1, ForwardIterator first2, ForwardIterator last2, BinaryPredicate comp)	Similar to using **find()**, except this algorithm searches the first sequence to find any element that is also in a second sequence. The second version of the function allows you to use a custom function (or functor) instead of the standard **operator==()**.

Algorithm	***Description***
iterator_traits<InputIterator>::difference_type **count**(InputIterator first, InputIterator last, const EqualityComparable& value) void **count**(InputIterator first, InputIterator last, const EqualityComparable& value, Size& n)	This algorithm iterators over a sequence and counts the number of elements that match the given value. The first version returns the number of matches found, while the second version adds the number of matches to the value referenced by *n*. The second version is deprecated and may be removed later.
iterator_traits<InputIterator>::difference_type **count_if**(InputIterator first, InputIterator last, Predicate pred) void **count_if**(InputIterator first, InputIterator last, Predicate pred, Size& n)	Similar to the **count()** algorithm, but instead of comparing the elements to some value, passes each element to a helper function (or functor) and only counts elements where the helper function returns *true*. The second version is deprecated and may be removed later.
pair<InputIterator1, InputIterator2> **mismatch**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2) pair<InputIterator1, InputIterator2> **mismatch**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate binary_pred)	Searches the sequence references by *first1* and *last1* to find an element that does not match the elements in *first2*. In other words, finds the first difference between 2 containers. The second versions of the algorithm allows you to use a custom function (or functor) to compare the elements in the containers.

Algorithm	***Description***
bool **equals**(InputIterator first1, InputIterator last1, InputIterator2 first2) bool **equals**(InputIterator first1, InputIterator last1, InputIterator2 first2, BinaryPredicate binary_pred)	Similar to the **mismatch()** algorithm, except this algorithm simply returns *true* or *false* to indicate whether the 2 sequences are equal or not. Can also be done using: mismatch(f1,l1,f2).first == l1
ForwardIterator1 **search**(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2) ForwardIterator1 **search**(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate binary_pred)	These algorithms attempt to find the sequence *first2->last2* somewhere instead the sequence *first1->last1*. This works a bit like searching for a substring within a larger string, except of course the elements can be of any type, not just characters.
ForwardIterator **search_n**(ForwardIterator first, ForwardIterator last, Integer count, const T& value) ForwardIterator **search_n**(ForwardIterator first, ForwardIterator last, Integer count, const T& value, BinaryPredicate binary_pred)	Attempts to find the position in the sequence where *value* is repeated *count* times in a row. Useful for testing for repeated elements. NOTE: Using 0 for *count* will always succeed, no matter the *value*, since there will be no comparisons performed.

Algorithm	*Description*
ForwardIterator1 **find_end**(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first1, ForwardIterator2 last2) ForwardIterator1 **find_end**(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate binary_pred)	Works similar to **search()**. Probably should be named ***search_end()***. Instead of returning the first match in the search, this algorithm returns an iterator that points to the last match instead.

Mutating Algorithms

The mutating algorithms are used to make changes to containers or the elements inside a container.

Algorithm	*Description*
OutputIterator **copy**(InputIterator first, InputIterator last, OutputIterator result)	This algorithm copies the elements referenced by *first->last* by overwriting the elements in *result*. NOTE; The output container must be large enough to hold all the copied elements, since this algorithm assigns the copied elements, it does not push them.
BidirectionalIterator2 **copy_backward**(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result)	Also copies the elements from *first->last* into the container at *result*, but copies from *last* to *first* in backward sequence. NOTE: *result* must point to the end of the sequence, not the beginning.
void **swap**(Assignable& a, Assignable& b)	Assigns *a* to *b* and *b* to *a*.

Algorithm	*Description*
void **swap_iter**(ForwardIterator1 a, ForwardIterator2 b)	Same as **swap(*a, *b)**.
ForwardIterator2 **swap_ranges**(ForwardIterator1 first1, ForwardIterator2 last1, ForwardIterator2 first2)	Swaps all the elements pointed to by *first1->last1* with the elements pointed to by *first2*.
OutputIterator **transform**(InputIterator first, InputIterator last, OutputIterator result, UnaryFunction op) OutputIterator **transform**(InputIterator1 first1, InputIterator1 last1, InputIterator1 first2, OutputIterator result, BinaryFunction binary_op)	This is similar to the **copy()** algorithm, except before the elements are copied into the new container, a helper function (or functor) is called that can change (transform) the elements in some way. The second version allows you to do that same thing, except in this case you are extracting elements from 2 different containers and combining them into one result container, by calling a helper function that receives one element from each input container.
void **replace**(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value)	Replaces every element with value *old_value*, with the value *new_value* in the sequence *first->last*.
void **replace_if**(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value)	Similar to **replace()**, except instead of comparing each input element against a value, calls a helper function (or functor). If the helper function returns *true*, the element will be replaced by *new_value*.

Algorithm	***Description***
void **replace_copy**(InputIterator first, InputIterator last, OutputIterator result, const T& old_value, const T& new_value)	Copies all the elements into a new container, but replaces any elements with *old_value* with *new_value* if found during the copy process.
void **replace_copy_if**(InputIterator first, InputIterator last, OutputIterator result, Predicate pred, const T& new_value)	Works like **replace_copy()**, except only replaces elements with *new_value* if the helper function (or functor) returns *true*.
void **fill**(ForwardIterator first, ForwardIterator last, const T& value)	Use this algorithm to quickly assign *value* to the elements referenced by *first->last*.
OutputIterator **fill_n**(OutputIterator first, Size n, const T& value)	This algorithm also assigns *value* to the elements referenced by *first*. It does this *n* times.
void **generate**(ForwardIterator first, ForwardIterator last, Generator gen)	This algorithm calls the helper function (or functor) *gen* and stores the results in each element referenced by *first->last*.
void **generate_n**(OutputIterator first, Size n, Generator gen)	Calls the helper function (or functor) *gen* and assigns the results to the iterator *first*, exactly *n* times.

Algorithm	***Description***
ForwardIterator **remove**(ForwardIterator first, ForwardIterator last, const T& value)	Removes all elements with *value* from the sequence referenced by *first->last*.
ForwardIterator **remove_if**(ForwardIterator first, ForwardIterator last, Predicate pred)	Same as **remove()**, except calls the helper function (or functor) *pred* to determine whether or not to remove the item. The helper function returns *true* when the element should be removed.
OutputIterator **remove_copy**(InputIterator first, InputIterator last, OutputIterator result, const T& value)	Copies elements in *first->last* to *result* if they do not equal *value*.
OutputIterator **remove_copy_if**(InputIterator first, InputIterator last, OuputIterator result, Predicate pred)	Calls the helper function (or functor) *pred* for each element in *first->last* and copies the element to *result* if *pred* returns *false*. In other words, *true* elements are not copied.
ForwardIterator **unique**(ForwardIterator first, ForwardIterator last) ForwardIterator **unique**(ForwardIterator first, ForwardIterator last, BinaryPredicate binary_pred)	Removes duplicate elements from the sequence *first->last* so when completed, only unique elements remain. The second flavor uses a helper function (or functor) to decide if elements are unique or not.

Algorithm	***Description***
OutputIterator **unique_copy**(InputIterator first, InputIterator last, OutputIterator result) OutputIterator **unique_copy**(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate binary_pred)	Copies only unique elements (skips consecutive duplicates) from *first->last* into *result*. Works best when the input container is sorted.
void **reverse**(BidirectionalIterator first, BidirectionalIterator last)	Reverses a container so the last element becomes the first and the first element becomes the last and so on.
OutputIterator **reverse_copy**(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result)	Copies elements *first->last* into container *result*, in reverse sequence.
ForwardIterator **rotate**(ForwardIterator first, ForwardIterator last, ForwardIterator middle)	Rearranges the container so *middle* becomes *first* and *last* becomes *middle*. This means *first* also becomes *middle*+1. Acts like rotating a circle.
OutputIterator **rotate_copy**(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result)	Works like **rotate()**, except copies the rotated elements into *result*. Faster then doing a **copy()** followed by a **rotate()**.
void **random_shuffle**(RandomAccessIterator first, RandomAccessIterator last) void **random_shuffle**(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand)	Randomly rearranges all the elements in *first->last*. The second version allows you to use a custom random number generator functor.

Algorithm	***Description***
RandomAccessIterator **random_sample**(InputIterator first, InputIterator last, RandomAccessIterator ofirst, RandomAccessIterator olast) RandomAccessIterator **random_sample**(InputIterator first, InputIterator last, RandomAccessIterator ofirst, RandomAccessIterator olast, RandomNumberGenerator& rand)	Works like the **random_shuffle()** algorithm, except instead of rearranging the input container, copies the elements randomly into another container. Each input element will only appear once in the output, in random sequence. Again a custom random number generator can be used if needed.
OutputIterator **random_sample_n**(ForwardIterator first, ForwardIterator last, OutputIterator out, Distance n) OutputIterator **random_sample_n**(ForwardIterator first, ForwardIterator last, OutputIterator out, Distance n, RandomNumberGenerator& rand)	Similar to **random_sample()**, except this algorithm stops after copying *n* elements. This algorithm preserves the relative order of the copied elements.
ForwardIterator **partition**(ForwardIterator first, ForwardIterator last, Predicate pred)	This algorithm rearranges the elements in *first->last* by calling the helper function (or functor) **pred** against each element. All elements where **pred** returns *true* are placed before the elements that return *false*. The returned iterator will point to the middle. (i.e. The first *false* element.)

Algorithm	***Description***
ForwardIterator **stable_partition**(ForwardIterator first, ForwardIterator last, Predicate pred)	Same as **partition()**, except the elements will maintain their relative order within the sequence.

Sorting Algorithms

This group of algorithm are used to fully or partially sort the elements in a container.

Algorithm	***Description***
void **sort**(RandomAccessIterator first, RandomAccessIterator last) void **sort**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	Rearranges the container so the elements are in sorted sequence. By default, it uses **operator<()** to compare elements. The second version allows you to use a helper function (or function) to get custom sort sequences.
void **stable_sort**(RandomAccessIterator first, RandomAccessIterator last) void **stable_sort**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	Also sorts elements in a container, but unlike the standard **sort()**, this one preserves the relative order of duplicate elements. This means that **stable_sort()** is less efficient than standard **sort()**.

Algorithm	***Description***
void **partial_sort**(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last) void **partial_sort**(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp)	This algorithm also sorts elements in containers, but in this case only elements from *first->middle* are sorted and placed at the beginning of the container. The elements from *middle->last* are unsorted (and probably rearranged).
RandomAccessIterator **partial_sort_copy**(InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last) RandomAccessIterator **partial_sort_copy**(InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last, Compare comp)	This algorithm combines partial sorting with copying of the elements. It will stop whenever it processes (*last-first)* or *(result_last-result_first)* elements, whichever is smaller. Useful for extracting *X* number of items (perhaps the smallest or largest values) from a large container into a smaller one.
bool **is_sorted**(ForwardIterator first, ForwardIterator last) bool **is_sorted**(ForwardIterator first, ForwardIterator last, Compare comp)	Returns *true* if the range is already sorted, *false* otherwise.

Algorithm	***Description***
void **nth_element**(RandomAccessIterator first, RanomAccessIterator nth, RandomAccessIterator last) void **nth_element**(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp)	This is a special kind of partial sort that ensures elements to the left of *nth* are less than all elements to the right of *nth*. The left side may or may not be sorted. The same applies to the right side. However, all items to the left will be less than the items to the right, with *nth* used as a split point.

Searching Algorithms

Algorithm	***Description***
ForwardIterator **lower_bound**(ForwardIterator first, ForwardIterator last, const T& value) ForwardIterator **lower_bound**(ForwardIterator first, ForwardIterator last, const T& value, Compare comp)	This algorithm performs a fast binary search of a sorted container and returns the iterator where a new element of *value* can be inserted to maintain the proper order of elements. Uses **operator<()** by default, but the second version can be used to customize the comparison. NOTE: The first version requires *value* to be comparable with a *T*.
ForwardIterator **upper_bound**(ForwardIterator first, ForwardIterator last, const T& value) ForwardIterator **upper_bound**(ForwardIterator first, ForwardIterator last, const T& value, Compare comp)	This algorithm also performs a fast binary search of a sorted container. The difference is in the returned iterator. This one returns a reference to the first element greater than *value*. By comparison the **lower_bound()** algorithm returns a reference to the first element greater than or equal to *value*.

Algorithm	*Description*
pair<ForwardIterator, ForwardIterator> **equal_range**(ForwardIterator first, ForwardIterator last, const T& value) pair<ForwardIterator, ForwardIterator> **equal_range**(ForwardIterator first, ForwardIterator last, const T& value, Compare comp)	Combines using **lower_bound()** and **upper_bound()** into a single algorithm that returns both iterators in a single function call. NOTE: The first version only requires that *value* be comparable to elements of type *T*.
bool **binary_search**(ForwardIterator first, ForwardIterator last, const T& value) bool **binary_search**(ForwardIterator first, ForwardIterator last, const T& value, Compare comp)	Compares each element in *first->last* against *value* using either the default comparison operator or a custom comparison function (or functor) *comp* and returns *true* if found, *false* otherwise. NOTE: Since you will often need to know the position of the element, most of the time you should consider using **lower_bound()**, **upper_bound()**, or **equal_range()** instead.

Merging Algorithms

There are a couple of algorithms you can use for combining and merging sorted containers together.

Algorithm	***Description***
OutputIterator **merge**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result)	Combines 2 sorted containers (*first1*->last1 and *first2*->last2) into another container (*result*) so that the output container is also sorted. The merge is stable, which means the relative order of duplicate elements is preserved.
OutputIterator **merge**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)	
void **inplace_merge**(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last)	This algorithm takes a container that has been partially sorted (split around *middle*) and completes the sort so the entire container is now sorted.
void **inplace_merge**(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp)	

Set Algorithms

There are also a set of algorithms designed specifically for performing set operations. Most of these algorithms do not require a **set<>** container, but they may be used to implement the **set<>** template class.

Algorithm	*Description*
bool **includes**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2) bool **includes**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp)	Tests 2 sorted ranges to determine if all of the elements in *first2->last2* are also found in *first1->last1*. Returns *true* only if all of the elements in the second container can be found in the first container. Both input containers must be sorted for this algorithm to work properly.
OutputIterator **set_union**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result) OutputIterator **set_union**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)	Copies all the sorted elements that are in either *first1->last1* or *first2->last2* into a new container *(result),* while preserving the sort sequence. Both input containers must be sorted for this algorithm to work properly. NOTE: If the same value elements appear in both containers, then this algorithm copies the elements from the container where the value is repeated the most often.

Algorithm	***Description***
OutputIterator **set_intersection**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result) OutputIterator **set_intersect**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)	Copies all the sorted elements that are found in both *first1->last1* and *first2->last2* into a new container *(result),* while preserving the sort sequence. Both input containers must be sorted for this algorithm to work properly. NOTE: If the same value elements appear in both containers, then this algorithm copies the elements from the container where the value is repeated the least often.
OutputIterator **set_difference**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result) OutputIterator **set_difference**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)	Copies all the sorted elements that are in *first1->last1* but are not in *first2->last2* into a new container *(result),* preserving the sort sequence. Both input containers must be sorted for this algorithm to work properly.

Algorithm	***Description***
OutputIterator **set_symmetric_difference**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result) OutputIterator **set_symmetric_difference**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)	Copies all the sorted elements that are in *first1->last1* but not in *first2->last2* as well as all the element in *first2->last2* that are not in *first1->last1* into a new container *(result)*, preserving the sort sequence. After using this algorithm the output container will have the set of elements that are not found in both input containers. Both input containers must be sorted for this algorithm to work properly.

Heap Operations

A heap is data structure similar to a tree, but normally stores its elements as an array (including vector and deque). The difference is that in a heap not every element has to be perfectly sorted. Instead the elements have to arranged so the highest value is always *above* the lower values. This is used by the **priority_queue<>** template internally to arrange elements by value.

Algorithm	***Description***
Void **make_heap**(RandomAccessIterator first, RandomAccessIterator last) void **make_heap**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	Turns the container *first->last* into a heap. Typically the underlying container will be a C-style array, **vector<>** or **deque<>** object.

Algorithm	***Description***
void **push_heap**(RandomAccessIterator first, RandomAccessIterator last) void **push_heap**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	This function moves an element that has already been added to the end of a container into its proper location within the heap structure. You must add the element to the underlying container yourself, perhaps by using the **push_back()** function.
void **pop_heap**(RandomAccessIterator first, RandomAccessIterator last) void **pop_heap**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	This method removes the largest element from the heap structure (the largest element is normally the first element). It does not actually remove the element, but instead moves it to the end of the underlying container and reorganizes the remaining elements so the heap is still valid.
void **sort_heap**(RandomAccessIterator first, RandomAccessIterator last) void **sort_heap**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	Returns the heap's underlying heap sequence back into a sorted sequence. The relative order of the elements is not guaranteed to be preserved.
bool **is_heap**(RandomAccessIterator first, RandomAccessIterator last) bool **is_heap**(RandomAccessIterator first, RandomAccessIterator last, Compare comp)	Tests a container to determine if it is already organized into the sequence needed to be treated as a heap structure.

Miscellaneous Algorithms

Here are several general-purpose algorithms.

Algorithm	***Description***
const T& **min**(const T& a, const T& b) const T& **min**(const T& a, const T& b, Compare comp)	Compares *a* to *b* and returns the one with the lesser value (returns *a* if they are equal). Uses **operator<** by default.
const T& **max**(const T& a, const T& b) const T& **max**(const T& a, const T& b, Compare comp)	Compares *a* to *b* and returns the one with the greater value (returns *a* if they are equal). Uses **operator<** by default.
ForwardIterator **min_element**(ForwardIterator first, ForwardIterator1 last) ForwardIterator **min_element**(ForwardIterator first, ForwardIterator last, Compare comp)	Finds the smallest element in the container and returns an iterator that references that element.
ForwardIterator **max_element**(ForwardIterator first, ForwardIterator1 last) ForwardIterator **max_element**(ForwardIterator first, ForwardIterator last, Compare comp)	Finds the largest element in the container and returns an iterator that references that element.

Algorithm	*Description*
bool **lexicographical_compare**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2) bool **lexicographical_compare**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp)	While this algorithm's name is awkward, the job it performs is simple. It compares elements one by one from both containers until it either reaches the end or finds elements that do not match. If both containers stored exactly the same elements in the same sequence, it returns *true*, otherwise it returns *false*.
bool **next_permutation**(BidirectionalIterator first, BidirectionalIterator last) bool **next_permutation**(BidirectionalIterator first, BidirectionalIterator last, Compare comp)	This algorithm is used to rearrange the elements in a sorted container in every other possible sequence (or permutation). Every time you call this algorithm, the elements will be reordered and it will return *true*. Once all the permutations have been generated, the elements are returned to the original sorted sequence and *false* is returned.
bool **prev_permutation**(BidirectionalIterator first, BidirectionalIterator last) bool **prev_permutation**(BidirectionalIterator first, BidirectionalIterator last, Compare comp)	This algorithm is the mirror image of **next_permutation()**.

Algorithm	*Description*
T **accumulate**(InputIterator first, InputIterator last, T init) T **accumulate**(InputIterator first, InputIterator last, T init, BinaryFunction binary_op)	Adds all the elements from *first->last* to *init* and returns the sum. The second version allows you to use a function (or functor) that will be called with the previous result (initially the same as *init*) and the next element in the container. The function will return a value of type T that will be added to the next result. NOTE: Defined in the header <numeric>.
T **inner_product**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init) T **inner_product**(InputIterator1 first1, InputIterator1 last1, InputIterator2 first1, T init, BinaryFunction1 binary_op1, BinaryFunction2 binary_op2)	This algorithm takes each element in the first container (*first1->last1)* and multiplies it by each corresponding element in the second container (*first2*) and returns the sum of all of the results + the value in *init*. Think of it as a crude matrix multiply and add operation. NOTE: Defined in the header <numeric>.
OutputIterator **partial_sum**(InputIterator first, InputIterator last, OutputIterator result) OutputIterator **partial_sum**(InputIterator first, InputIterator last, OutputIterator result, BinaryOperator binary_op)	This algorithm visits each element in a container and adds the element's value to the next element's value and stores the result in the output container. The first element is always just copied to the output. Output[0] = Input[0] Output[i] = Input[i-1] + Input[i]

<table>
<tr><th>Algorithm</th><th>Description</th></tr>
<tr><td>OutputIterator
adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result)</td><td rowspan="2">Copies the first element from first to result. Next, subtracts all elements from the previous element and stores the result in result.

Output[0] = Input[0]
Output[i] = Input[i] – Input[i-1]</td></tr>
<tr><td>OutputIterator
adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result)</td></tr>
</table>

Function Objects (aka Functors)

A function object or functor is any object that can be used as if it were a plain old function. A class can used as a functor if it defines **operator()**, which is sometimes referred to as the default operator. So a functor is really either a pointer to a static function, or a pointer to an object that defines **operator()**. The advantages of using a function object should become apparent soon.

Many of the algorithms in the Standard Template Library will accept a functor to use instead of the default functor defined by the template class. This allows the user of the algorithm to adapt the algorithm to their specific needs. You can use the predefined function objects that are included with the STL, or you can roll your own as long as your functors have the required function signatures.

There are 3 major types of function objects and several other less commonly used function objects.

Major Functor Types

Functor Type	***Used By***	***Description***
Predicate (Unary or Binary)	**Unary:** *remove_if*, *find_if*, *count_if*, *replace_if*, *replace_copy_if*, *remove_if*, and *remove_copy_if* **Binary**: *adjacent_find*, *find_first_of*, *mismatch*, *equal*, *search*, *search_n*, *find_end*, *unique*, and *unique_copy*	A predicate function object returns a **bool** value of *true* or *false*. Generally they will receive one argument of type *T*, but some algorithms will require a binary predicate function which takes in two arguments of type **T** and returns a **bool**.
Comparison Functions	*sort*, *stable_sort*, *partial_sort*, *partial_sort_copy*, *is_sorted*, *nth_element*, *lower_bound*, *upper_bound*, *equal_range*, *binary_search*, *merge*, *inplace_merge*, *includes*, *set_union*, *set_intersection*, *set_difference*, *set_symmetric_difference*, *make_heap*, *push_heap*, *pop_heap*, *sort_heap*, *is_heap*, *min*, *max*, *min_element*, *max_element*, *lexicographical_compare*, *next_permutation* and *prev_permutation*	This kind of function object takes two arguments of type T and return *true* or *false* after the items have been compared. The **operator<** is an example of this kind of function and is generally the default used when you do not supply your own function object.

Functor Type	***Used By***	***Description***
Numeric Functions (Unary or Binary)	**Unary**: *for_each* and *transform* **Binary**: *transform*, *accumulate*, and *inner_product*	This kind of function will generally accept either one or two arguments of type *T* and returns the results of some sort of mathematical operation. The **accumulate** algorithm uses **operator+** as its default numeric function.

Here is an example of an algorithm that uses a function.

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

bool failingGrade( int score )
{
   return score < 70;
}

int main( int argc, char *argv[] )
{
   vector<int> scores;

   scores.push_back( 69 );
   scores.push_back( 70 );
   scores.push_back( 85 );
   scores.push_back( 80 );

   cout << “Scores Before: “ << endl;
   copy( scores.begin(), scores.end(), ostream_iterator<int>(cout,
“\n”) );

   vector<int>::iterator new_end;

   new_end = remove_if( scores.begin(), scores.end(), failingGrade );
   scores.remove( new_end, scores.end() );

   cout << “Scores After: “ << endl
   copy( scores.begin(), scores.end(), ostream_iterator<int>(cout,
“\n”) );

   return 0;
}
```

The only problem with this example is that the failingGrade function is not very flexible. It uses a hard-coded cutoff of 70.

Here is a better version that uses a function object (object of a class with an **operator()** defined).

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

class Failing
{
private:
  int cutoff;
public:
  Failing( int below ) : cutoff(below) {}
  bool operator()( int score )
  {
    return score < cutoff;
  }
};

int main( int argc, char *argv[] )
{
   vector<int> scores;

   scores.push_back( 69 );
   scores.push_back( 70 );
   scores.push_back( 85 );
   scores.push_back( 80 );

   cout << "Scores Before: " << endl;
   copy( scores.begin(), scores.end(), ostream_iterator<int>(cout,
"\n") );

   vector<int>::iterator new_end;
   new_end = remove_if( scores.begin(), scores.end(), Failing(75) );
   scores.erase( new_end, scores.end() );

   cout << "Scores After: " << endl;
   copy( scores.begin(), scores.end(), ostream_iterator<int>(cout,
"\n") );

   return 0;
}
```

This can also be achieved using a class template instead as follows:

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

using namespace std;

template <typename T>
```

```
class Failing
{
private:
  T cutoff;
public:
  Failing( T below ) : cutoff(below) {}
  bool operator()( T const& score )
  {
    return score < cutoff;
  }
};

int main( int argc, char *argv[] )
{
   vector<int> scores;

   scores.push_back( 69 );
   scores.push_back( 70 );
   scores.push_back( 85 );
   scores.push_back( 80 );

   cout << "Scores Before: " << endl;
   copy( scores.begin(), scores.end(), ostream_iterator<int>(cout,
"\n") );

   vector<int>::iterator new_end;
   new_end = remove_if( scores.begin(), scores.end(), Failing<int>
(81) );
   scores.erase( new_end, scores.end() );

   cout << "Scores After: " << endl;
   copy( scores.begin(), scores.end(), ostream_iterator<int>(cout,
"\n") );

   return 0;
}
```

Predefined Function Objects

Since using function objects with algorithms is so common, a number of predefined function objects are available for you to use in your code.

Arithmetic Function Objects

Functor	***Type***	***Description***
plus<T>	Binary	Adds two elements together to calculate a sum.
minus<T>	Binary	Subtracts two elements to calculate the difference.
multiplies<T>* * times<T> in older versions of STL	Binary	Multiples two elements to calculate a product.
divides<T>	Binary	Divides one element by another to calculate a dividend.
modulus<T>	Binary	Performs a modulo operation against two elements and calculates the remainder.
negate<T>	Unary	Negates the element so positive values become negative and vice-versa.

Comparison Function Objects

Functor	***Type***	***Description***
equal_to<T>	Binary	Compares two elements for equality using **operator==**.
not_equal_to<T>	Binary	Compares two elements for inequality using **operator==**.
less<T>	Binary	Compares two elements using **operator<**.
greater<T>	Binary	Compares two elements using **operator>**.
less_equal<T>	Binary	Compares two elements using **operator<=**.
greater_equal<T>	Binary	Compares two elements using **operator>=**.

Logical Function Objects

Functor	***Type***	***Description***
logical_and<T>	Binary	Performs an AND operation with two other conditions.

Functor	***Type***	***Description***
logical_or<T>	Binary	Performs an OR operation with two other conditions.
logical_not<T>	Unary	Inverts boolean logic.

Function Object Adapters

Alas the function objects listed above are quite useful, but limited in scope. There is no apparent way you can use functors like less<T> with any algorithm that requires a unary function, or is there?

This is the concept of a Function Object Adapter. They can be used to convert binary functors into unary functors or to do other conversions such as converting a plain function into a function object or converting a class member function back into a standard function so it can be used with the algorithms.

Adapter	***Description***	***Notes***
binder1st	Adapts a unary function/functor and a constant into a binary functor, where the constant will be used as the first argument to the functor.	Don't use directly. Instead use the **bind1st()** function, which creates a binder1st object internally.
binder2nd	Adapts a unary function/functor and a constant into a binary functor, where the constant will be used as the second argument to the functor.	Don't use directly. Instead, use the **bind2nd()** function, which creates a binder2nd object internally.
ptr_fun	Converts a pointer to a standard function into a function object. Needed when trying to customize the container templates (such as set<>) which require a functor class and do not support function objects.	Can be used to convert both unary and binary functions into function objects.
unary_negate	Converts a unary predicate function object by inverting the logical return value.	Don't use directly. Use the **not1()** function instead.
binary_negate	Converts a binary predicate function object by inverting the logical return value.	Don't use directly. Use the **not2()** helper function instead.

Adapter	***Description***	***Notes***
unary_compose	Combines multiple function objects into a single object by calling the first function and passing the results to the next function. In other words, allows you to chain function calls together.	Don't use directly. Use the **compose1()** helper function instead.
binary_compose	Combines multiple function objects into a single object in the same manner as unary_compose, except works on binary functions.	Don't use directly. Use the **compose2()** helper function instead.
mem_fun	Converts a member function of a class (or struct) into a plain function so it can be used with algorithms. It performs opposite from the same way **ptr_fun()** does.	Use this helper function when you are storing pointers to objects in a container and want to call a member function of the class in an algorithm.
mem_fun_ref	Converts a member function of a class (or template) into a function object just like the **mem_fun()** function adapter does.	Use this helper function when you are storing objects (not pointers) in a container and need to access the object by reference.