Table of Contents

C++ Standard Template Library

About the C++ Standard Template Library

The C++ STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists, and stacks.

The C++ STL provides programmers with the following constructs, grouped into three categories:

- Sequences
    - C++ Vectors
    - C++ Lists
    - C++ Double-Ended Queues
- Container Adapters
    - C++ Stacks
    - C++ Queues
    - C++ Priority Queues
- Associative Containers
    - C++ Bitsets
    - C++ Maps
    - C++ Multimaps
    - C++ Sets
    - C++ Multisets

The idea behind the C++ STL is that the hard part of using complex data structures has already been completed. If a programmer would like to use a stack of integers, all that she has to do is use this code:

```
stack<int> myStack;
```

With minimal effort, she can now push() and pop() integers onto this stack. Through the magic of C++ Templates, she could specify any data type, not just integers. The STL Stack class will provide generic functionality of a stack, regardless of the data in the stack.

In addition, the STL also provides a bunch of useful algorithms -- like searching, sorting, and general-purpose iterating algorithms -- that can be used on a variety of data structures.

C++ Iterators

Iterators are used to access members of the container classes, and can be used in a similar manner to pointers. For example, one might use an iterator to step through the elements of a vector. There are several different types of iterators:

Iterator	Description
input_iterator	Read values with forward movement. These can be incremented, compared, and dereferenced.
output_iterator	Write values with forward movement. These can be incremented and dereferenced.
forward_iterator	Read or write values with forward movement. These combine the functionality of input and output iterators with the ability to store the iterators value.
bidirectional_iterator	Read and write values with forward and backward movement. These are like the forward iterators, but you can increment and decrement them.
random_iterator	Read and write values with random access. These are the most powerful iterators, combining the functionality of bidirectional iterators with the ability to do pointer arithmetic and pointer comparisons.
reverse_iterator	Either a random iterator or a bidirectional iterator that moves in reverse direction.

Each of the container classes is associated with a type of iterator, and each of the STL algorithms uses a certain type of iterator. For example, vectors are associated with **random-access iterators**, which means that they can use algorithms that require random access. Since random-access iterators encompass all of the characteristics of the other iterators, vectors can use algorithms designed for other iterators as well.

The following code creates and uses an iterator with a vector:

```
vector<int> the_vector;
vector<int>::iterator the_iterator;
for( int i=0; i < 10; i++ )
  the_vector.push_back(i);
int total = 0;
the_iterator = the_vector.begin();
while( the_iterator != the_vector.end() ) {
  total += *the_iterator;
  the_iterator++;
}
cout << "Total=" << total << endl;
```

Notice that you can access the elements of the container by dereferencing the iterator.

C++ Algorithms

accumulate	sum up a range of elements
adjacent_difference	compute the differences between adjacent elements in a range
adjacent_find	finds two items that are adjacent to eachother
binary_search	determine if an element exists in a certain range
copy	copy some range of elements to a new location
copy_backward	copy a range of elements in backwards order
copy_n	copy N elements
count	return the number of elements matching a given value
count_if	return the number of elements for which a predicate is true
equal	determine if two sets of elements are the same
equal_range	search for a range of elements that are all equal to a certain element
fill	assign a range of elements a certain value
fill_n	assign a value to some number of elements
find	find a value in a given range
find_end	find the last sequence of elements in a certain range
find_first_of	search for any one of a set of elements
find_if	find the first element for which a certain predicate is true
for_each	apply a function to a range of elements
generate	saves the result of a function in a range
generate_n	saves the result of N applications of a function
includes	returns true if one set is a subset of another
inner_product	compute the inner product of two ranges of elements
inplace_merge	merge two ordered ranges in-place
is_heap	returns true if a given range is a heap
is_sorted	returns true if a range is sorted in ascending order
iter_swap	swaps the elements pointed to by two iterators
lexicographical_compare	returns true if one range is lexicographically less than another
lexicographical_compare_3way	determines if one range is lexicographically less than or greater than another
lower_bound	search for the first place that a value can be inserted while preserving order
make_heap	creates a heap out of a range of elements
max	returns the larger of two elements
max_element	returns the largest element in a range
merge	merge two sorted ranges
min	returns the smaller of two elements
min_element	returns the smallest element in a range
mismatch	finds the first position where two ranges differ
next_permutation	generates the next greater lexicographic permutation of a range of elements
nth_element	put one element in its sorted location and make sure that no elements to its left are greater than any elements to its right
partial_sort	sort the first N elements of a range
partial_sort_copy	copy and partially sort a range of elements
partial_sum	compute the partial sum of a range of elements
partition	divide a range of elements into two groups

pop_heap	remove the largest element from a heap
prev_permutation	generates the next smaller lexicographic permutation of a range of elements
push_heap	add an element to a heap
random_sample	randomly copy elements from one range to another
random_sample_n	sample N random elements from a range
random_shuffle	randomly re-order elements in some range
remove	remove elements equal to certain value
remove_copy	copy a range of elements omitting those that match a certian value
remove_copy_if	create a copy of a range of elements, omitting any for which a predicate is true
remove_if	remove all elements for which a predicate is true
replace	replace every occurrence of some value in a range with another value
replace_copy	copy a range, replacing certain elements with new ones
replace_copy_if	copy a range of elements, replacing those for which a predicate is true
replace_if	change the values of elements for which a predicate is true
reverse	reverse elements in some range
reverse_copy	create a copy of a range that is reversed
rotate	move the elements in some range to the left by some amount
rotate_copy	copy and rotate a range of elements
search	search for a range of elements
search_n	search for N consecutive copies of an element in some range
set_difference	computes the difference between two sets
set_intersection	computes the intersection of two sets
set_symmetric_difference	computes the symmetric difference between two sets
set_union	computes the union of two sets
sort	sort a range into ascending order
sort_heap	turns a heap into a sorted range of elements
stable_partition	divide elements into two groups while preserving their relative order
stable_sort	sort a range of elements while preserving order between equal elements
swap	swap the values of two objects
swap_ranges	swaps two ranges of elements
transform	applies a function to a range of elements
unique	remove consecutive duplicate elements in a range
unique_copy	create a copy of some range of elements that contains no consecutive duplicates
upper_bound	searches for the last possible location to insert an element into an ordered range

accumulate

Syntax:

```
  #include <numeric>
  TYPE accumulate( iterator start, iterator end, TYPE val );
  TYPE accumulate( iterator start, iterator end, TYPE val, BinaryFunction f );
```

The accummulate() function computes the sum of *val* and all of the elements in the range [*start*,*end*).

If the binary function *f* if specified, it is used instead of the + operator to perform the summation.

accumulate() runs in linear time.

Related topics:
adjacent_difference
count
inner_product
partial_sum

adjacent_difference

Syntax:

```
  #include <numeric>
  iterator adjacent_difference( iterator start, iterator end, iterator
result );
  iterator adjacent_difference( iterator start, iterator end, iterator
result, BinaryFunction f );
```

The adjacent_difference() function calculates the differences between adjacent elements in the range [*start*,*end*) and stores the result starting at *result*.

If a binary function *f* is given, it is used instead of the - operator to compute the differences.

adjacent_difference() runs in linear time.

Related topics:
accumulate
count
inner_product
partial_sum

adjacent_find

Syntax:

```
  #include <algorithm>
  iterator adjacent_find( iterator start, iterator end );
  iterator adjacent_find( iterator start, iterator end, BinPred pr );
```

The adjacent_find() function searches between *start* and *end* for two consecutive identical elements. If the binary predicate *pr* is specified, then it is used to test whether two elements are the same or not.

The return value is an iterator that points to the first of the two elements that are found. If no matching elements are found, the returned iterator points to *end*.

For example, the following code creates a vector containing the integers between 0 and 10 with 7 appearing twice in a row. adjacent_find() is then used to find the location of the pair of 7's:

```
 vector<int> v1;
 for( int i = 0; i < 10; i++ ) {
   v1.push_back(i);
   // add a duplicate 7 into v1
   if( i == 7 ) {
     v1.push_back(i);
   }
 }
 vector<int>::iterator result;
 result = adjacent_find( v1.begin(), v1.end() );
 if( result == v1.end() ) {
   cout << "Did not find adjacent elements in v1" << endl;
 }
 else {
   cout << "Found matching adjacent elements starting at " << *result
<< endl;
 }
```

Related topics:
find
find_end
find_first_of
find_if
unique
unique_copy

binary_search

Syntax:

```
  #include <algorithm>
  bool binary_search( iterator start, iterator end, const TYPE& val );
  bool binary_search( iterator start, iterator end, const TYPE& val,
Comp f );
```

The binary_search() function searches from *start* to *end* for *val*. The elements between *start* and *end* that are searched should be in ascending order as defined by the < operator. Note that a binary search **will not work** unless the elements being searched are in order.

If *val* is found, binary_search() returns true, otherwise false.

If the function *f* is specified, then it is used to compare elements.

For example, the following code uses binary_search() to determine if the integers 0-9 are in an array of integers:

```
int nums[] = { -242, -1, 0, 5, 8, 9, 11 };
int start = 0;
int end = 7;
for( int i = 0; i < 10; i++ ) {
  if( binary_search( nums+start, nums+end, i ) ) {
    cout << "nums[] contains " << i << endl;
  } else {
    cout << "nums[] DOES NOT contain " << i << endl;
  }
}
```

When run, this code displays the following output:

```
nums[] contains 0
nums[] DOES NOT contain 1
nums[] DOES NOT contain 2
nums[] DOES NOT contain 3
nums[] DOES NOT contain 4
nums[] contains 5
nums[] DOES NOT contain 6
nums[] DOES NOT contain 7
nums[] contains 8
nums[] contains 9
```

Related topics:

equal_range	partial_sort	stable_sort
is_sorted	partial_sort_copy	upper_bound
lower_bound	sort	

copy

Syntax:

```
  #include <algorithm>
  iterator copy( iterator start, iterator end, iterator dest );
```

The copy() function copies the elements between *start* and *end* to *dest*. In other words, after copy() has run,

```
*dest == *start
*(dest+1) == *(start+1)
*(dest+2) == *(start+2)
...
*(dest+N) == *(start+N)
```

The return value is an iterator to the last element copied. copy() runs in linear time.

For example, the following code uses copy() to copy the contents of one vector to another:

```
vector<int> from_vector;
for( int i = 0; i < 10; i++ ) {
  from_vector.push_back( i );
}
vector<int> to_vector(10);
copy( from_vector.begin(), from_vector.end(), to_vector.begin() );

cout << "to_vector contains: ";
for( unsigned int i = 0; i < to_vector.size(); i++ ) {
  cout << to_vector[i] << " ";
}
cout << endl;
```

Related topics:
copy_backward
copy_n
generate
remove_copy
swap
transform

copy_backward

Syntax:

```
  #include <algorithm>
  iterator copy_backward( iterator start, iterator end, iterator dest );
```

copy_backward() is similar to (C++ Strings) copy(), in that both functions copy elements from *start* to *end* to *dest*. The copy_backward() function , however, starts depositing elements at *dest* and then works backwards, such that:

```
 *(dest-1) == *(end-1)
 *(dest-2) == *(end-2)
 *(dest-3) == *(end-3)
 ...
 *(dest-N) == *(end-N)
```

The following code uses copy_backward() to copy 10 integers into the end of an empty vector:

```
 vector<int> from_vector;
 for( int i = 0; i < 10; i++ ) {
   from_vector.push_back( i );
 }
 vector<int> to_vector(15);
 copy_backward( from_vector.begin(), from_vector.end(), to_vector.end()
);
 cout << "to_vector contains: ";
 for( unsigned int i = 0; i < to_vector.size(); i++ ) {
   cout << to_vector[i] << " ";
 }
 cout << endl;
```

The above code produces the following output:

```
 to_vector contains: 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9
```

Related topics:
copy
copy_n
swap

copy_n

Syntax:

```
  #include <algorithm>
  iterator copy_n( iterator from, size_t num, iterator to );
```

The copy_n() function copies *num* elements starting at *from* to the destination pointed at by *to*. To put it another way, copy_n() performs *num* assignments and duplicates a subrange.

The return value of copy_n() is an iterator that points to the last element that was copied, i.e. (to + num).

This function runs in linear time.

Related topics:

copy	copy_backward	swap

---

count
Syntax:

```
  #include <algorithm>
  size_t count( iterator start, iterator end, const TYPE& val );
```

The count() function returns the number of elements between *start* and *end* that match *val*.

For example, the following code uses count() to determine how many integers in a vector match a target value:

```
 vector<int> v;
 for( int i = 0; i < 10; i++ ) {
   v.push_back( i );
 }
 int target_value = 3;
 int num_items = count( v.begin(), v.end(), target_value );
 cout << "v contains " << num_items << " items matching " <<
target_value << endl;
```

The above code displays the following output:

```
 v contains 1 items matching 3
```

Related topics:

accumulate	count_if	partial_sum
adjacent_difference	inner_product	

---

count_if

Syntax:

```
  #include <algorithm>
  size_t count_if( iterator start, iterator end, UnaryPred p );
```

The count_if() function returns the number of elements between *start* and *end* for which the predicate *p* returns true.

For example, the following code uses count_if() with a predicate that returns true for the integer 3 to count the number of items in an array that are equal to 3:

```
 int nums[] = { 0, 1, 2, 3, 4, 5, 9, 3, 13 };
 int start = 0;
 int end = 9;
 int target_value = 3;
 int num_items = count_if( nums+start,
                    nums+end,
                    bind2nd(equal_to<int>(), target_value) );

 cout << "nums[] contains " << num_items << " items matching " <<
target_value << endl;
```

When run, the above code displays the following output:

```
 nums[] contains 2 items matching 3
```

Related topics:
count

equal

Syntax:

```
  #include <algorithm>
  bool equal( iterator start1, iterator end1, iterator start2 );
  bool equal( iterator start1, iterator end1, iterator start2, BinPred
p );
```

The equal() function returns true if the elements in two ranges are the same. The first range of elements are those between *start1* and *end1*. The second range of elements has the same size as the first range but starts at *start2*.

If the binary predicate *p* is specified, then it is used instead of == to compare each pair of elements.

For example, the following code uses equal() to compare two vectors of integers:

```
 vector<int> v1;
 for( int i = 0; i < 10; i++ ) {
   v1.push_back( i );
 }
 vector<int> v2;
 for( int i = 0; i < 10; i++ ) {
   v2.push_back( i );
 }
 if( equal( v1.begin(), v1.end(), v2.begin() ) ) {
   cout << "v1 and v2 are equal" << endl;
 } else {
   cout << "v1 and v2 are NOT equal" << endl;
 }
```

Related topics:
find_if
lexicographical_compare
mismatch
search

equal_range

Syntax:

```
  #include <algorithm>
  pair<iterator,iterator> equal_range( iterator first, iterator last,
const TYPE& val );
  pair<iterator,iterator> equal_range( iterator first, iterator last,
const TYPE& val, CompFn comp );
```

The equal_range() function returns the range of elements between *first* and *last* that are equal to *val*. This function assumes that the elements between *first* and *last* are in order according to *comp*, if it is specified, or the < operator otherwise.

equal_range() can be thought of as a combination of the lower_bound() and \`upper_bound1\`() functions, since the first of the pair of iterators that it returns is what lower_bound() returns and the second iterator in the pair is what \`upper_bound1\`() returns.

For example, the following code uses equal_range() to determine all of the possible places that the number 8 can be inserted into an ordered vector of integers such that the existing ordering is preserved:

```
 vector<int> nums;
 nums.push_back( -242 );
 nums.push_back( -1 );
 nums.push_back( 0 );
 nums.push_back( 5 );
 nums.push_back( 8 );
 nums.push_back( 8 );
 nums.push_back( 11 );
 pair<vector<int>::iterator, vector<int>::iterator> result;
 int new_val = 8;
 result = equal_range( nums.begin(), nums.end(), new_val );
 cout << "The first place that " << new_val << " could be inserted is
before "
      << *result.first << ", and the last place that it could be
inserted is before "
      << *result.second << endl;
```

The above code produces the following output:

```
 The first place that 8 could be inserted is before 8,
 and the last place that it could be inserted is before 11
```

Related topics:
binary_search
lower_bound
upper_bound

fill

Syntax:

```
  #include <algorithm>
  #include <algorithm>
  void fill( iterator start, iterator end, const TYPE& val );
```

The function fill() assigns *val* to all of the elements between *start* and *end*.

For example, the following code uses fill() to set all of the elements of a vector of integers to -1:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}
cout << "Before, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;
fill( v1.begin(), v1.end(), -1 );
cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;
```

When run, the above code displays:

```
Before, v1 is: 0 1 2 3 4 5 6 7 8 9
After, v1 is: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Related topics:
fill_n
generate
transform

fill_n

Syntax:

```
  #include <algorithm>
  #include <algorithm>
  iterator fill_n( iterator start, size_t n, const TYPE& val );
```

The fill_n() function is similar to (C++ I/O) fill(). Instead of assigning *val* to a range of elements, however, fill_n() assigns *val* to the first *n* elements starting at *start*.

For example, the following code uses fill_n() to assign -1 to the first half of a vector of integers:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}
cout << "Before, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;
fill_n( v1.begin(), v1.size()/2, -1 );
cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
  cout << v1[i] << " ";
}
cout << endl;
```

When run, this code displays:

```
Before, v1 is: 0 1 2 3 4 5 6 7 8 9
After, v1 is: -1 -1 -1 -1 -1 5 6 7 8 9
```

Related topics:
fill

find

Syntax:

```
  #include <algorithm>
  iterator find( iterator start, iterator end, const TYPE& val );
```

The find() algorithm looks for an element matching *val* between *start* and *end*. If an element matching *val* is found, the return value is an iterator that points to that element. Otherwise, the return value is an iterator that points to *end*.

For example, the following code uses find() to search a vector of integers for the number 3:

```
int num_to_find = 3;
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back(i);
}
vector<int>::iterator result;
result = find( v1.begin(), v1.end(), num_to_find );
if( result == v1.end() ) {
  cout << "Did not find any element matching " << num_to_find << endl;
}
else {
  cout << "Found a matching element: " << *result << endl;
}
```

In the next example, shown below, the find() function is used on an array of integers. This example shows how the C++ Algorithms can be used to manipulate arrays and pointers in the same manner that they manipulate containers and iterators:

```
int nums[] = { 3, 1, 4, 1, 5, 9 };
int num_to_find = 5;
int start = 0;
int end = 2;
int* result = find( nums + start, nums + end, num_to_find );

if( result == nums + end ) {
  cout << "Did not find any number matching " << num_to_find << endl;
} else {
  cout << "Found a matching number: " << *result << endl;
}
```

Related topics:
adjacent_find
find_end
find_first_of
find_if
mismatch
search

find_end

Syntax:

```
  #include <algorithm>
  iterator find_end( iterator start, iterator end, iterator seq_start,
iterator seq_end );
  iterator find_end( iterator start, iterator end, iterator seq_start,
iterator seq_end, BinPred bp );
```

The find_end() function searches for the sequence of elements denoted by *seq_start* and *seq_end*. If such a sequence if found between *start* and *end*, an iterator to the first element of the last found sequence is returned. If no such sequence is found, an iterator pointing to *end* is returned.

If the binary predicate *bp* is specified, then it is used to when elements match.

For example, the following code uses find_end() to search for two different sequences of numbers. The the first chunk of code, the last occurence of "1 2 3" is found. In the second chunk of code, the sequence that is being searched for is not found:

```
 int nums[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4 };
 int* result;
 int start = 0;
 int end = 11;
 int target1[] = { 1, 2, 3 };
 result = find_end( nums + start, nums + end, target1 + 0, target1 +
2 );
 if( *result == nums[end] ) {
   cout << "Did not find any subsequence matching { 1, 2, 3 }" << endl;
 } else {
   cout << "The last matching subsequence is at: " << *result << endl;
 }
 int target2[] = { 3, 2, 3 };
 result = find_end( nums + start, nums + end, target2 + 0, target2 +
2 );
 if( *result == nums[end] ) {
   cout << "Did not find any subsequence matching { 3, 2, 3 }" << endl;
 } else {
   cout << "The last matching subsequence is at: " << *result << endl;
 }
```

Related topics:
adjacent_find
find
find_first_of
find_if
search_n

find_first_of

Syntax:

```
  #include <algorithm>
  iterator find_first_of( iterator start, iterator end, iterator
find_start, iterator find_end );
  iterator find_first_of( iterator start, iterator end, iterator
find_start, iterator find_end, BinPred bp );
```

The find_first_of() function searches for the first occurence of any element between *find_start* and *find_end*. The data that are searched are those between *start* and *end*.

If any element between *find_start* and *find_end* is found, an iterator pointing to that element is returned. Otherwise, an iterator pointing to *end* is returned.

For example, the following code searches for a 9, 4, or 7 in an array of integers:

```
 int nums[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int* result;
 int start = 0;
 int end = 10;
 int targets[] = { 9, 4, 7 };
 result = find_first_of( nums + start, nums + end, targets + 0, targets
+ 2 );
 if( *result == nums[end] ) {
   cout << "Did not find any of { 9, 4, 7 }" << endl;
 } else {
   cout << "Found a matching target: " << *result << endl;
 }
```

Related topics:
adjacent_find
find
find_end
find_if
(Standard C String and Character) strpbrk

find_if

Syntax:

```
  #include <algorithm>
  iterator find_if( iterator start, iterator end, UnPred up );
```

The find_if() function searches for the first element between *start* and *end* for which the unary predicate *up* returns true.

If such an element is found, an iterator pointing to that element is returned. Otherwise, an iterator pointing to *end* is returned.

For example, the following code uses find_if() and a "greater-than-zero" unary predicate to the first positive, non-zero number in a list of numbers:

```
 int nums[] = { 0, -1, -2, -3, -4, 342, -5 };
 int* result;
 int start = 0;
 int end = 7;
 result = find_if( nums + start, nums + end, bind2nd(greater<int>(),
0));
 if( *result == nums[end] ) {
   cout << "Did not find any number greater than zero" << endl;
 } else {
   cout << "Found a positive non-zero number: " << *result << endl;
 }
```

Related topics:
adjacent_find
equal
find
find_end
find_first_of
search_n

for_each

Syntax:

```
  #include <algorithm>
  UnaryFunction for_each( iterator start, iterator end, UnaryFunction f
);
```

The for_each() algorithm applies the function *f* to each of the elements between *start* and *end*. The return value of for_each() is *f*.
For example, the following code snippets define a unary function then use it to increment all of the elements of an array:

```
 template<class TYPE> struct increment : public unary_function<TYPE,
void> {
   void operator() (TYPE& x) {
     x++;
   }
 };
 ...
 int nums[] = {3, 4, 2, 9, 15, 267};
 const int N = 6;
 cout << "Before, nums[] is: ";
 for( int i = 0; i < N; i++ ) {
   cout << nums[i] << " ";
 }
 cout << endl;
 for_each( nums, nums + N, increment<int>() );
 cout << "After, nums[] is: ";
 for( int i = 0; i < N; i++ ) {
   cout << nums[i] << " ";
 }
 cout << endl;
```

The above code displays the following output:

```
 Before, nums[] is: 3 4 2 9 15 267
 After, nums[] is: 4 5 3 10 16 268
```

---

generate

Syntax:

```
  #include <algorithm>
  void generate( iterator start, iterator end, Generator g );
```

The generate() function runs the Generator function object *g* a number of times, saving the result of each execution in the range [*start*,*end*).

Related topics:

copy	generate_n
fill	transform

generate_n

Syntax:

```
  #include <algorithm>
  iterator generate_n( iterator result, size_t num, Generator g );
```

The generate_n() function runs the Generator function object *g* *num* times, saving the result of each execution in *result*, (*result*+1), etc.

Related topics:
generate

---

includes

Syntax:

```
  #include <algorithm>
  bool includes( iterator start1, iterator end1, iterator start2,
iterator end2 );
  bool includes( iterator start1, iterator end1, iterator start2,
iterator end2, StrictWeakOrdering cmp );
```

The includes() algorithm returns true if every element in [*start2*,*end2*) is also in [*start1*,*end1*). Both of the given ranges must be sorted in ascending order.

By default, the < operator is used to compare elements. If the strict weak ordering function object *cmp* is given, then it is used instead.

includes() runs in linear time.

Related topics:
set_difference
set_intersection
set_symmetric_difference
set_union

---

inner_product

Syntax:

```
  #include <numeric>
  TYPE inner_product( iterator start1, iterator end1, iterator start2,
TYPE val );
  TYPE inner_product( iterator start1, iterator end1, iterator start2,
TYPE val, BinaryFunction f1, BinaryFunction f2 );
```

The inner_product() function computes the inner product of [*start1*,*end1*) and a range of the same size starting at *start2*.

inner_product() runs in linear time.

Related topics:
accumulate
adjacent_difference
count
partial_sum

---

inplace_merge

Syntax:

```
  #include <algorithm>
  inline void inplace_merge( iterator start, iterator middle, iterator
end );
  inline void inplace_merge( iterator start, iterator middle, iterator
end, StrictWeakOrdering cmp );
```

The inplace_merge() function is similar to the merge() function, but instead of creating a new sorted range of elements, inplace_merge() alters the existing ranges to perform the merge in-place.

Related topics:
merge

---

is_heap

Syntax:

```
  #include <algorithm>
  bool is_heap( iterator start, iterator end );
  bool is_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The is_heap() function returns true if the given range [*start*,*end*) is a heap.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

is_heap() runs in linear time.

Related topics:
make_heap
pop_heap
push_heap
sort_heap

---

is_sorted

Syntax:

```
  #include <algorithm>
  bool is_sorted( iterator start, iterator end );
  bool is_sorted( iterator start, iterator end, StrictWeakOrdering
cmp );
```

The is_sorted() algorithm returns true if the elements in the range [*start*,*end*) are sorted in ascending order.

By default, the < operator is used to compare elements. If the strict weak order function object *cmp* is given, then it is used instead.

is_sorted() runs in linear time.

Related topics:
binary_search
partial_sort
partial_sort_copy
sort
stable_sort

---

iter_swap

Syntax:

```
  #include <algorithm>
  inline void iter_swap( iterator a, iterator b );
```

A call to iter_swap() exchanges the values of two elements exactly as a call to

```
 swap( *a, *b );
```

would.

Related topics:
swap
swap_ranges

---

lexicographical_compare

Syntax:

```
  #include <algorithm>
  bool lexicographical_compare( iterator start1, iterator end1,
iterator start2, iterator end2 );
  bool lexicographical_compare( iterator start1, iterator end1,
iterator start2, iterator end2, BinPred p );
```

The lexicographical_compare() function returns true if the range of elements [*start1*,*end1*) is lexicographically less than the range of elements [*start2*,*end2*).

If you're confused about what lexicographic means, it might help to know that dictionaries are ordered lexicographically.

lexicographical_compare() runs in linear time.

Related topics:
equal
lexicographical_compare_3way
mismatch
search

---

lexicographical_compare_3way

Syntax:

```
  #include <algorithm>
  int lexicographical_compare_3way( iterator start1, iterator end1,
iterator start2, iterator end2 );
```

The lexicographical_compare_3way() function compares the first range, defined by [*start1*,*end1*) to the second range, defined by [*start2*,*end2*).

If the first range is lexicographically less than the second range, this function returns a negative number. If the first range is lexicographically greater than the second, a positive number is returned. Zero is returned if neither range is lexicographically greater than the other.

lexicographical_compare_3way() runs in linear time.

Related topics:
lexicographical_compare

lower_bound

Syntax:

```
  #include <algorithm>
  iterator lower_bound( iterator first, iterator last,  const TYPE& val
);
  iterator lower_bound( iterator first, iterator last, const TYPE& val,
CompFn f );
```

The lower_bound() function is a type of binary_search(). This function searches for the first place that *val* can be inserted into the ordered range defined by *first* and *last* that will not mess up the existing ordering.

The return value of lower_bound() is an iterator that points to the location where *val* can be safely inserted. Unless the comparison function *f* is specified, the < operator is used for ordering.

For example, the following code uses lower_bound() to insert the number 7 into an ordered vector of integers:

```
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );
cout << "Before nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
  cout << nums[i] << " ";
}
cout << endl;
vector<int>::iterator result;
int new_val = 7;
result = lower_bound( nums.begin(), nums.end(), new_val );
nums.insert( result, new_val );
cout << "After, nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
  cout << nums[i] << " ";
}
cout << endl;
```

The above code produces the following output:

```
Before nums is: -242 -1 0 5 8 8 11
After, nums is: -242 -1 0 5 7 8 8 11
```

Related topics:
binary_search
equal_range

make_heap

Syntax:

```
  #include <algorithm>
  void make_heap( iterator start, iterator end );
  void make_heap( iterator start, iterator end, StrictWeakOrdering
cmp );
```

The make_heap() function turns the given range of elements [*start*,*end*) into a heap.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

make_heap() runs in linear time.

Related topics:
is_heap pop_heap push_heap sort_heap

---

max

Syntax:

```
  #include <algorithm>
  const TYPE& max( const TYPE& x, const TYPE& y );
  const TYPE& max( const TYPE& x, const TYPE& y, BinPred p );
```

The max() function returns the greater of *x* and *y*.

If the binary predicate *p* is given, then it will be used instead of the < operator to compare the two elements.

Example code:

For example, the following code snippet displays various uses of the max() function:

```
 cout << "Max of 1 and 9999 is " << max( 1, 9999) << endl;
 cout << "Max of 'a' and 'b' is " << max( 'a', 'b') << endl;
 cout << "Max of 3.14159 and 2.71828 is " << max( 3.14159, 2.71828) <<
endl;
```

When run, this code displays:

```
 Max of 1 and 9999 is 9999
 Max of 'a' and 'b' is b
 Max of 3.14159 and 2.71828 is 3.14159
```

Related topics:
max_element min min_element

---

max_element

Syntax:

```
  #include <algorithm>
  iterator max_element( iterator start, iterator end );
  iterator max_element( iterator start, iterator end, BinPred p );
```

The max_element() function returns an iterator to the largest element in the range [*start*,*end*).

If the binary predicate *p* is given, then it will be used instead of the < operator to determine the largest element.

Example code:

For example, the following code uses the max_element() function to determine the largest integer in an array and the largest character in a vector of characters:

```
 int array[] = { 3, 1, 4, 1, 5, 9 };
 unsigned int array_size = 6;
 cout << "Max element in array is " << *max_element( array, array
+array_size) << endl;
 vector<char> v;
 v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back('d');
 cout << "Max element in the vector v is " << *max_element( v.begin(),
v.end() ) << endl;
```

When run, the above code displays this output:

```
 Max element in array is 9
 Max element in the vector v is d
```

Related topics:
max
min
min_element

merge

Syntax:

```
  #include <algorithm>
  iterator merge( iterator start1, iterator end1, iterator start2,
iterator end2, iterator result );
  iterator merge( iterator start1, iterator end1, iterator start2,
iterator end2, iterator result, StrictWeakOrdering cmp );
```

The merge() function combines two sorted ranges [*start1*,*end1*) and [*start2*,*end2*) into a single sorted range, stored starting at *result*. The return value of this function is an iterator to the end of the merged range.

If the strict weak ordering function object *cmp* is given, then it is used in place of the < operator to perform comparisons between elements.

merge() runs in linear time.

Related topics:
inplace_merge
set_union
sort

min

Syntax:

```
  #include <algorithm>
  const TYPE& min( const TYPE& x, const TYPE& y );
  const TYPE& min( const TYPE& x, const TYPE& y, BinPred p );
```

The min() function, unsurprisingly, returns the smaller of *x* and *y*.

By default, the < operator is used to compare the two elements. If the binary predicate *p* is given, it will be used instead.

Related topics:
max
max_element
min_element

min_element

Syntax:

```
  #include <algorithm>
  iterator min_element( iterator start, iterator end );
  iterator min_element( iterator start, iterator end, BinPred p );
```

The min_element() function returns an iterator to the smallest element in the range [*start*,*end*).

If the binary predicate *p* is given, then it will be used instead of the < operator to determine the smallest element.

Related topics:
max
max_element
min

---

mismatch

Syntax:

```
  #include <algorithm>
  pair <iterator1,iterator2> mismatch( iterator start1, iterator end1,
iterator start2 );
  pair <iterator1,iterator2> mismatch( iterator start1, iterator end1,
iterator start2, BinPred p );
```

The mismatch() function compares the elements in the range defined by [*start1*,*end1*) to the elements in a range of the same size starting at *start2*. The return value of mismatch() is the first location where the two ranges differ.

If the optional binary predicate *p* is given, then it is used to compare elements from the two ranges.

The mismatch() algorithm runs in linear time.

Related topics:
equal
find
lexicographical_compare
search

---

next_permutation

Syntax:

```
  #include <algorithm>
  bool next_permutation( iterator start, iterator end );
  bool next_permutation( iterator start, iterator end,
StrictWeakOrdering cmp );
```

The next_permutation() function attempts to transform the given range of elements [*start*,*end*) into the next lexicographically greater permutation of elements. If it succeeds, it returns true, otherwise, it returns false.

If a strict weak ordering function object *cmp* is provided, it is used in lieu of the < operator when comparing elements.

Related topics:
prev_permutation
random_sample
random_sample_n
random_shuffle

---

nth_element

Syntax:

```
  #include <algorithm>
  void nth_element( iterator start, iterator middle, iterator end );
  void nth_element( iterator start, iterator middle, iterator end,
StrictWeakOrdering cmp );
```

The nth_element() function semi-sorts the range of elements defined by [*start*,*end*). It puts the element that *middle* points to in the place that it would be if the entire range was sorted, and it makes sure that none of the elements before that element are greater than any of the elements that come after that element.

nth_element() runs in linear time on average.

Related topics:
partial_sort

---

partial_sort

Syntax:

```
  #include <algorithm>
  void partial_sort( iterator start, iterator middle, iterator end );
  void partial_sort( iterator start, iterator middle, iterator end,
StrictWeakOrdering cmp );
```

The partial_sort() function arranges the first N elements of the range [*start*,*end*) in ascending order. N is defined as the number of elements between *start* and *middle*.

By default, the < operator is used to compare two elements. If the strict weak ordering comparison function *cmp* is given, it is used instead.

Related topics:
binary_search
is_sorted
nth_element
partial_sort_copy
sort
stable_sort

---

partial_sort_copy

Syntax:

```
  #include <algorithm>
  iterator partial_sort_copy( iterator start, iterator end, iterator
result_start, iterator result_end );
  iterator partial_sort_copy( iterator start, iterator end, iterator
result_start, iterator result_end, StrictWeakOrdering cmp );
```

The partial_sort_copy() algorithm behaves like partial_sort(), except that instead of partially sorting the range in-place, a copy of the range is created and the sorting takes place in the copy. The initial range is defined by [*start*,*end*) and the location of the copy is defined by [*result_start*,*result_end*).

partial_sort_copy() returns an iterator to the end of the copied, partially-sorted range of elements.

Related topics:
binary_search
is_sorted
partial_sort
sort
stable_sort

---

partial_sum

Syntax:

```
  #include <numeric>
  iterator partial_sum( iterator start, iterator end, iterator result );
  iterator partial_sum( iterator start, iterator end, iterator result,
BinOp p );
```

The partial_sum() function calculates the partial sum of a range defined by [*start*,*end*), storing the output at *result*.

- *start* is assigned to **result*, the sum of **start* and *(*start* + 1) is assigned to *(*result* + 1), etc.

partial_sum() runs in linear time.

Related topics:
accumulate
adjacent_difference
count
inner_product

---

partition

Syntax:

```
  #include <algorithm>
  iterator partition( iterator start, iterator end, Predicate p );
```

The partition() algorithm re-orders the elements in [*start*,*end*) such that the elements for which the predicate *p* returns true come before the elements for which *p* returns false.

In other words, partition() uses *p* to divide the elements into two groups.

The return value of partition() is an iterator to the first element for which *p* returns false.

parition() runs in linear time.

Related topics:
stable_partition

---

pop_heap

Syntax:

```
  #include <algorithm>
  void pop_heap( iterator start, iterator end );
  void pop_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The pop_heap() function removes the larges element (defined as the element at the front of the heap) from the given heap.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

pop_heap() runs in logarithmic time.

Related topics:
is_heap
make_heap
push_heap
sort_heap

---

prev_permutation

Syntax:

```
  #include <algorithm>
  bool prev_permutation( iterator start, iterator end );
  bool prev_permutation( iterator start, iterator end, 
StrictWeakOrdering cmp );
```

The prev_permutation() function attempts to transform the given range of elements [*start*,*end*) into the next lexicographically smaller permutation of elements. If it succeeds, it returns true, otherwise, it returns false.

If a strict weak ordering function object *cmp* is provided, it is used instead of the < operator when comparing elements.

Related topics:
next_permutation
random_sample
random_sample_n
random_shuffle

---

push_heap

Syntax:

```
  #include <algorithm>
  void push_heap( iterator start, iterator end );
  void push_heap( iterator start, iterator end, StrictWeakOrdering
cmp );
```

The push_heap() function adds an element (defined as the last element before *end*) to a heap (defined as the range of elements between [*start*,"end-1).

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

push_heap() runs in logarithmic time.

Related topics:

is_heap	pop_heap
make_heap	sort_heap

---

random_sample

Syntax:

```
  #include <algorithm>
  iterator random_sample( iterator start1, iterator end1, iterator
start2, iterator end2 );
  iterator random_sample( iterator start1, iterator end1, iterator
start2, iterator end2, RandomNumberGenerator& rnd );
```

The random_sample() algorithm randomly copies elements from [*start1*,*end1*) to [*start2*,*end2*). Elements are chosen with uniform probability and elements from the input range will appear at most once in the output range.

If a random number generator function object *rnd* is supplied, then it will be used instead of an internal random number generator.

The return value of random_sample() is an iterator to the end of the output range.

random_sample() runs in linear time.

Related topics:

next_permutation	random_sample_n
prev_permutation	random_shuffle

---

random_sample_n

Syntax:

```
  #include <algorithm>
  iterator random_sample_n( iterator start, iterator end, iterator
result, size_t N );
  iterator random_sample_n( iterator start, iterator end, iterator
result, size_t N, RandomNumberGenerator& rnd );
```

The random_sample_n() algorithm randomly copies *N* elements from [*start*,*end*) to *result*. Elements are chosen with uniform probability and elements from the input range will appear at most once in the output range. **Element order is preserved** from the input range to the output range.

If a random number generator function object *rnd* is supplied, then it will be used instead of an internal random number generator.

The return value of random_sample_n() is an iterator to the end of the output range.

random_sample_n() runs in linear time.

Related topics:
next_permutation
prev_permutation
random_sample
random_shuffle

---

random_shuffle

Syntax:

```
  #include <algorithm>
  void random_shuffle( iterator start, iterator end );
  void random_shuffle( iterator start, iterator end,
RandomNumberGenerator& rnd );
```

The random_shuffle() function randomly re-orders the elements in the range [*start*,*end*). If a random number generator function object *rnd* is supplied, it will be used instead of an internal random number generator.

Related topics:
next_permutation
prev_permutation
random_sample
random_sample_n

---

remove

Syntax:

```
  #include <algorithm>
  iterator remove( iterator start, iterator end, const TYPE& val );
```

The remove() algorithm removes all of the elements in the range [*start*,*end*) that are equal to *val*.

The return value of this function is an iterator to the last element of the new sequence that should contain no elements equal to *val*.

The remove() function runs in linear time.

Related topics:
remove_copy
remove_copy_if
remove_if
unique
unique_copy

---

remove_copy

Syntax:

```
  #include <algorithm>
  iterator remove_copy( iterator start, iterator end, iterator result,
const TYPE& val );
```

The remove_copy() algorithm copies the range [*start*,*end*) to *result* but omits any elements that are equal to *val*.

remove_copy() returns an iterator to the end of the new range, and runs in linear time.

Related topics:
copy
remove
remove_copy_if
remove_if

---

remove_copy_if

Syntax:

```
  #include <algorithm>
  iterator remove_copy_if( iterator start, iterator end, iterator
result, Predicate p );
```

The remove_copy_if() function copies the range of elements [*start*,*end*) to *result*, omitting any elements for which the predicate function *p* returns true.

The return value of remove_copy_if() is an iterator the end of the new range.

remove_copy_if() runs in linear time.

Related topics:
remove
remove_copy
remove_if

---

remove_if

Syntax:

```
  #include <algorithm>
  iterator remove_if( iterator start, iterator end, Predicate p );
```

The remove_if() function removes all elements in the range [*start*,*end*) for which the predicate *p* returns true.

The return value of this function is an iterator to the last element of the pruned range.

remove_if() runs in linear time.

Related topics:
remove
remove_copy
remove_copy_if

---

replace

Syntax:

```
  #include <algorithm>
  void replace( iterator start, iterator end, const TYPE& old_value,
const TYPE& new_value );
```

The replace() function sets every element in the range [*start*,*end*) that is equal to *old_value* to have *new_value* instead.

replace() runs in linear time.

Related topics:
replace_copy
replace_copy_if
replace_if

---

replace_copy

Syntax:

```
  #include <algorithm>
  iterator replace_copy( iterator start, iterator end, iterator result,
const TYPE& old_value, const TYPE& new_value );
```

The replace_copy() function copies the elements in the range [*start*,*end*) to the destination *result*. Any elements in the range that are equal to *old_value* are replaced with *new_value*.

Related topics:
replace

---

replace_copy_if

Syntax:

```
  #include <algorithm>
  iterator replace_copy_if( iterator start, iterator end, iterator
result, Predicate p, const TYPE& new_value );
```

The replace_copy_if() function copies the elements in the range [*start*,*end*) to the destination *result*. Any elements for which the predicate *p* is true are replaced with *new_value*.

Related topics:
replace

---

replace_if

Syntax:

```
  #include <algorithm>
  void replace_if( iterator start, iterator end, Predicate p, const
TYPE& new_value );
```

The replace_if() function assigns every element in the range [*start*,*end*) for which the predicate function *p* returns true the value of *new_value*.

This function runs in linear time.

Related topics:
replace

---

reverse

Syntax:

```
  #include <algorithm>
  void reverse( iterator start, iterator end );
```

The reverse() algorithm reverses the order of elements in the range [*start*,*end*).

Related topics:
reverse_copy

---

reverse_copy

Syntax:

```
  #include <algorithm>
  iterator reverse_copy( iterator start, iterator end, iterator
result );
```

The reverse_copy() algorithm copies the elements in the range [*start*,*end*) to *result* such that the elements in the new range are in reverse order.

The return value of the reverse_copy() function is an iterator the end of the new range.

Related topics:
reverse

---

rotate

Syntax:

```
  #include <algorithm>
  inline iterator rotate( iterator start, iterator middle, iterator end
);
```

The rotate() algorithm moves the elements in the range [*start*,*end*) such that the *middle* element is now where *start* used to be, (*middle*+1) is now at (*start*+1), etc.

The return value of rotate() is an iterator to *start* + (*end*-*middle*).

rotate() runs in linear time.

Related topics:
rotate_copy

---

rotate_copy

Syntax:

```
  #include <algorithm>
  iterator rotate_copy( iterator start, iterator middle, iterator end,
iterator result );
```

The rotate_copy() algorithm is similar to the rotate() algorithm, except that the range of elements is copied to *result* before being rotated.

Related topics:
rotate

---

search

Syntax:

```
  #include <algorithm>
  iterator search( iterator start1, iterator end1, iterator start2,
iterator end2 );
  iterator search( iterator start1, iterator end1, iterator start2,
iterator end2, BinPred p );
```

The search() algorithm looks for the elements [*start2*,*end2*) in the range [*start1*,*end1*). If the optional binary predicate *p* is provided, then it is used to perform comparisons between elements.

If search() finds a matching subrange, then it returns an iterator to the beginning of that matching subrange. If no match is found, an iterator pointing to *end1* is returned.

In the worst case, search() runs in quadratic time, on average, it runs in linear time.

Related topics:

equal	lexicographical_compare	search_n
find	mismatch	

---

search_n

Syntax:

```
  #include <algorithm>
  iterator search_n( iterator start, iterator end, size_t num, const
TYPE& val );
  iterator search_n( iterator start, iterator end, size_t num, const
TYPE& val, BinPred p );
```

The search_n() function looks for *num* occurances of *val* in the range [*start*,*end*).

If *num* consecutive copies of *val* are found, search_n() returns an iterator to the beginning of that sequence. Otherwise it returns an iterator to *end*.

If the optional binary predicate *p* is given, then it is used to perform comparisons between elements.

This function runs in linear time.

Related topics:

find_end	search
find_if	

---

set_difference

Syntax:

```
  #include <algorithm>
  iterator set_difference( iterator start1, iterator end1, iterator
start2, iterator end2, iterator result );
  iterator set_difference( iterator start1, iterator end1, iterator
start2, iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_difference() algorithm computes the difference between two sets defined by [*start1*,*end1*) and [*start2*,*end2*) and stores the difference starting at *result*.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_difference() is an iterator to the end of the result range.

If the strict weak ordering comparison function object *cmp* is not specified, set_difference() will use the < operator to compare elements.

Related topics:

includes	set_symmetric_difference
set_intersection	set_union

---

set_intersection

Syntax:

```
  #include <algorithm>
  iterator set_intersection( iterator start1, iterator end1, iterator
start2, iterator end2, iterator result );
  iterator set_intersection( iterator start1, iterator end1, iterator
start2, iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_intersection() algorithm computes the intersection of the two sets defined by [*start1*,*end1*) and [*start2*,*end2*) and stores the intersection starting at *result*.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_intersection() is an iterator to the end of the intersection range.

If the strict weak ordering comparison function object *cmp* is not specified, set_intersection() will use the < operator to compare elements.

Related topics:

includes	set_symmetric_difference
set_difference	set_union

set_symmetric_difference

Syntax:

```
  #include <algorithm>
  iterator set_symmetric_difference( iterator start1, iterator end1,
iterator start2, iterator end2, iterator result );
  iterator set_symmetric_difference( iterator start1, iterator end1,
iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp
);
```

The set_symmetric_difference() algorithm computes the symmetric difference of the two sets defined by [*start1*,*end1*) and [*start2*,*end2*) and stores the difference starting at *result*.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_symmetric_difference() is an iterator to the end of the result range.

If the strict weak ordering comparison function object *cmp* is not specified, set_symmetric_difference() will use the < operator to compare elements.

Related topics:

includes	set_intersection
set_difference	set_union

---

set_union

Syntax:

```
  #include <algorithm>
  iterator set_union( iterator start1, iterator end1, iterator start2,
iterator end2, iterator result );
  iterator set_union( iterator start1, iterator end1, iterator start2,
iterator end2, iterator result, StrictWeakOrdering cmp );
```

The set_union() algorithm computes the union of the two ranges [*start1*,*end1*) and [*start2*,*end2*) and stores it starting at *result*.

The return value of set_union() is an iterator to the end of the union range.

set_union() runs in linear time.

Related topics:

includes	set_difference	set_symmetric_difference
merge	set_intersection	

---

sort

Syntax:

```
  #include <algorithm>
  void sort( iterator start, iterator end );
  void sort( iterator start, iterator end, StrictWeakOrdering cmp );
```

The sort() algorithm sorts the elements in the range [*start*,*end*) into ascending order. If two elements are equal, there is no guarantee what order they will be in.

If the strict weak ordering function object *cmp* is given, then it will be used to compare two objects instead of the < operator.

The algorithm behind sort() is the *introsort* algorithm. sort() runs in O(N log(N)) time (average and worst case) which is faster than polynomial time but slower than linear time.

Example code:

For example, the following code sorts a vector of integers into ascending order:

```
vector<int> v;
v.push_back( 23 );
v.push_back( -1 );
v.push_back( 9999 );
v.push_back( 0 );
v.push_back( 4 );
cout << "Before sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
sort( v.begin(), v.end() );
cout << "After sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

When run, the above code displays this output:

```
Before sorting: 23 -1 9999 0 4
After sorting: -1 0 4 23 9999
```

Alternatively, the following code uses the sort() function to sort a normal array of integers, and displays the same output as the previous example:

```
int array[] = { 23, -1, 9999, 0, 4 };
unsigned int array_size = 5;
cout << "Before sorting: ";
for( unsigned int i = 0; i < array_size; i++ ) {
  cout << array[i] << " ";
}
```

```
cout << endl;
sort( array, array + array_size );
cout << "After sorting: ";
for( unsigned int i = 0; i < array_size; i++ ) {
  cout << array[i] << " ";
}
cout << endl;
```

This next example shows how to use sort() with a user-specified comparison function. The function **cmp** is defined to do the opposite of the < operator. When sort() is called with **cmp** used as the comparison function, the result is a list sorted in descending, rather than ascending, order:

```
bool cmp( int a, int b ) {
  return a > b;
}
...
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  v.push_back(i);
}
cout << "Before: ";
for( int i = 0; i < 10; i++ ) {
  cout << v[i] << " ";
}
cout << endl;
sort( v.begin(), v.end(), cmp );
cout << "After: ";
for( int i = 0; i < 10; i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

Related topics:
binary_search
is_sorted
merge
partial_sort
partial_sort_copy
stable_sort
(Other Standard C Functions) qsort

sort_heap

Syntax:

```
  #include <algorithm>
  void sort_heap( iterator start, iterator end );
  void sort_heap( iterator start, iterator end, StrictWeakOrdering
cmp );
```

The sort_heap() function turns the heap defined by [*start*,*end*) into a sorted range.

If the strict weak ordering comparison function object *cmp* is given, then it is used instead of the < operator to compare elements.

Related topics:
is_heap
make_heap
pop_heap
push_heap

---

stable_partition

Syntax:

```
  #include <algorithm>
  iterator stable_partition( iterator start, iterator end, Predicate
p );
```

The stable_partition() function behaves similarly to partition(). The difference between the two algorithms is that stable_partition() will preserve the initial ordering of the elements in the two groups.

Related topics:
partition

---

stable_sort

Syntax:

```
  #include <algorithm>
  void stable_sort( iterator start, iterator end );
  void stable_sort( iterator start, iterator end, StrictWeakOrdering
cmp );
```

The stable_sort() algorithm is like the sort() algorithm, in that it sorts a range of elements into ascending order. Unlike sort(), however, stable_sort() will preserve the original ordering of elements that are equal to eachother.

This functionality comes at a small cost, however, as stable_sort() takes a few more comparisons that sort() in the worst case: N (log N)^2 instead of N log N.

Related topics:
binary_search
is_sorted
partial_sort
partial_sort_copy
sort

---

swap

Syntax:

```
  #include <algorithm>
  void swap( Assignable& a, Assignable& b );
```

The swap() function swaps the values of *a* and *b*.

swap() expects that its arguments will conform to the Assignable model; that is, they should have a copy constructor and work with the = operator. This function performs one copy and two assignments.

Related topics:
copy
copy_backward
copy_n
iter_swap
swap_ranges

---

swap_ranges

Syntax:

```
  #include <algorithm>
  iterator swap_ranges( iterator start1, iterator end1, iterator start2
);
```

The swap_ranges() function exchanges the elements in the range [*start1*,*end1*) with the range of the same size starting at *start2*.

The return value of swap_ranges() is an iterator to *start2* + (*end1*-*start1*).

Related topics:
iter_swap
swap

---

transform

Syntax:

```
  #include <algorithm>
  iterator transform( iterator start, iterator end, iterator result,
UnaryFunction f );
  iterator transform( iterator start1, iterator end1, iterator start2,
iterator result, BinaryFunction f );
```

The transform() algorithm applies the function *f* to some range of elements, storing the result of each application of the function in *result*.

The first version of the function applies *f* to each element in [*start*,*end*) and assigns the first output of the function to *result*, the second output to (*result*+1), etc.

The second version of the transform() works in a similar manner, except that it is given two ranges of elements and calls a binary function on a pair of elements.

Related topics:
copy
fill
generate

---

unique

Syntax:

```
  #include <algorithm>
  iterator unique( iterator start, iterator end );
  iterator unique( iterator start, iterator end, BinPred p );
```

The unique() algorithm removes all consecutive duplicate elements from the range [*start*,*end*). If the binary predicate *p* is given, then it is used to test to test two elements to see if they are duplicates.

The return value of unique() is an iterator to the end of the modified range.

unique() runs in linear time.

Related topics:
adjacent_find
remove
unique_copy

---

unique_copy

Syntax:

```
  #include <algorithm>
  iterator unique_copy( iterator start, iterator end, iterator result );
  iterator unique_copy( iterator start, iterator end, iterator result,
BinPred p );
```

The unique_copy() function copies the range [*start*,*end*) to *result*, removing all consecutive duplicate elements. If the binary predicate *p* is provided, then it is used to test two elements to see if they are duplicates.

The return value of unique_copy() is an iterator to the end of the new range.

unique_copy() runs in linear time.

Related topics:
adjacent_find
remove
unique

---

upper_bound

Syntax:

```
  #include <algorithm>
  iterator upper_bound( iterator start, iterator end, const TYPE& val );
  iterator upper_bound( iterator start, iterator end, const TYPE& val,
StrictWeakOrdering cmp );
```

The upper_bound() algorithm searches the ordered range [*start*,*end*) for the last location that *val* could be inserted without disrupting the order of the range.

If the strict weak ordering function object *cmp* is given, it is used to compare elements instead of the < operator.

upper_bound() runs in logarithmic time.

Related topics:
binary_search
equal_range

C++ Vectors

Vectors contain contiguous elements stored as an array. Accessing members of a vector or appending elements can be done in constant time, whereas locating a specific value or inserting elements into the vector takes linear time.

Vector constructors	create vectors and initialize them with some data
Vector operators	compare, assign, and access elements of a vector
assign	assign elements to a vector
at	returns an element at a specific location
back	returns a reference to last element of a vector
begin	returns an iterator to the beginning of the vector
capacity	returns the number of elements that the vector can hold
clear	removes all elements from the vector
empty	true if the vector has no elements
end	returns an iterator just past the last element of a vector
erase	removes elements from a vector
front	returns a reference to the first element of a vector
insert	inserts elements into the vector
max_size	returns the maximum number of elements that the vector can hold
pop_back	removes the last element of a vector
push_back	add an element to the end of the vector
rbegin	returns a reverse_iterator to the end of the vector
rend	returns a reverse_iterator to the beginning of the vector
reserve	sets the minimum capacity of the vector
resize	change the size of the vector
size	returns the number of items in the vector
swap	swap the contents of this vector with another

Vector constructors

Syntax:

```
#include <vector>
vector();
vector( const vector& c );
vector( size_type num, const TYPE& val = TYPE() );
vector( input_iterator start, input_iterator end );
~vector();
```

The default vector constructor takes no arguments, creates a new instance of that vector.

The second constructor is a default copy constructor that can be used to create a new vector that is a copy of the given vector *c*.

The third constructor creates a vector with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
vector<int> v1( 5, 42 );
```

The last constructor creates a vector that is initialized to contain the elements between *start* and *end*. For example:

```
// create a vector of random integers
cout << "original vector: ";
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  int num = (int) rand() % 10;
  cout << num << " ";
  v.push_back( num );
}
cout << endl;
// find the first element of v that is even
vector<int>::iterator iter1 = v.begin();
while( iter1 != v.end() && *iter1 % 2 != 0 ) {
  iter1++;
}
// find the last element of v that is even
vector<int>::iterator iter2 = v.end();
do {
  iter2--;
} while( iter2 != v.begin() && *iter2 % 2 != 0 );
// only proceed if we find both numbers
if( iter1 != v.end() && iter2 != v.begin() ) {
  cout << "first even number: " << *iter1 << ", last even number: " <<
*iter2 << endl;
  cout << "new vector: ";
  vector<int> v2( iter1, iter2 );
  for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
  }
  cout << endl;
}
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the vector should be destroyed.

Vector operators

Syntax:

```
  #include <vector>
  TYPE& operator[]( size_type index );
  const TYPE& operator[]( size_type index ) const;
  vector operator=(const vector& c2);
  bool operator==(const vector& c1, const vector& c2);
  bool operator!=(const vector& c1, const vector& c2);
  bool operator<(const vector& c1, const vector& c2);
  bool operator>(const vector& c1, const vector& c2);
  bool operator<=(const vector& c1, const vector& c2);
  bool operator>=(const vector& c1, const vector& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a vector can be examined with the [] operator.

Performing a comparison or assigning one vector to another takes linear time. The [] operator runs in constant time.

Two vectors are equal if:

1. Their size is the same, and
2. Each member in location i in one vector is equal to the the member in location i in the other vector.

Comparisons among vectors are done lexicographically. For example, the following code uses the [] operator to access all of the elements of a vector:

```
vector<int> v( 5, 1 );
for( int i = 0; i < v.size(); i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
}
```

Related topics:
at

---

assign

Syntax:

```
  #include <vector>
  void assign( size_type num, const TYPE& val );
  void assign( input_iterator start, input_iterator end );
```

The assign() function either gives the current vector the values from *start* to *end*, or gives it *num* copies of *val*.

This function will destroy the previous contents of the vector.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}
vector<int> v2;
v2.assign( v1.begin(), v1.end() );
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
(C++ Lists) push_front

at

Syntax:

```
  #include <vector>
  TYPE& at( size_type loc );
  const TYPE& at( size_type loc ) const;
```

The at() function returns a reference to the element in the vector at index *loc*. The at() function is safer than the [] operator, because it won't let you reference items outside the bounds of the vector.

For example, consider the following code:

```
 vector<int> v( 5, 1 );
 for( int i = 0; i < 10; i++ ) {
   cout << "Element " << i << " is " << v[i] << endl;
 }
```

This code overrunns the end of the vector, producing potentially dangerous results. The following code would be much safer:

```
 vector<int> v( 5, 1 );
 for( int i = 0; i < 10; i++ ) {
   cout << "Element " << i << " is " << v.at(i) << endl;
 }
```

Instead of attempting to read garbage values from memory, the at() function will realize that it is about to overrun the vector and will throw an exception.

Related topics:
Vector operators

back

Syntax:

```
  #include <vector>
  TYPE& back();
  const TYPE& back() const;
```

The back() function returns a reference to the last element in the vector.

For example:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
cout << "The first element is " << v.front()
     << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
pop_back

begin

Syntax:

```
#include <vector>
iterator begin();
const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the vector, and runs in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse the elements of a vector:

```
vector<string> words;
string str;

while( cin >> str ) words.push_back(str);
vector<string>::iterator iter;
for( iter = words.begin(); iter != words.end(); iter++ ) {
  cout << *iter << endl;
}
```

When given this input:

```
hey mickey you're so fine
```

...the above code produces the following output:

```
hey
mickey
you're
so
fine
```

Related topics:
[] operator
at
end
rbegin
rend

capacity

Syntax:

```
#include <vector>
size_type capacity() const;
```

The capacity() function returns the number of elements that the vector can hold before it will need to allocate more space.

For example, the following code uses two different methods to set the capacity of two vectors. One method passes an argument to the constructor that suggests an initial size, the other method calls the reserve function to achieve a similar goal:

```
vector<int> v1(10);
cout << "The capacity of v1 is " << v1.capacity() << endl;
vector<int> v2;
v2.reserve(20);
cout << "The capacity of v2 is " << v2.capacity() << endl;
```

When run, the above code produces the following output:

```
The capacity of v1 is 10
The capacity of v2 is 20
```

C++ containers are designed to grow in size dynamically. This frees the programmer from having to worry about storing an arbitrary number of elements in a container. However, sometimes the programmer can improve the performance of her program by giving hints to the compiler about the size of the containers that the program will use. These hints come in the form of the reserve() function and the constructor used in the above example, which tell the compiler how large the container is expected to get.

The capacity() function runs in constant time.

Related topics:
reserve
resize
size

clear

Syntax:

```
#include <vector>
void clear();
```

The function clear() deletes all of the elements in the vector.

clear() runs in linear time.

Related topics:
erase

---

empty

Syntax:

```
#include <vector>
bool empty() const;
```

The empty() function returns true if the vector has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a vector and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end

Syntax:

```
#include <vector>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the vector.

Note that before you can access the last element of the vector using an iterator that you get from a call to end(), you'll have to decrement the iterator first. This is because end() doesn't point to the end of the vector; it points **just past the end of the vector**.

For example, in the following code, the first "cout" statement will display garbage, whereas the second statement will actually display the last element of the vector:

```
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );
int bad_val = *(v1.end());
cout << "bad_val is " << bad_val << endl;
int good_val = *(v1.end() - 1);
cout << "good_val is " << good_val << endl;
```

The next example shows how begin() and end() can be used to iterate through all of the members of a

```
); vector<int>::iterator it; for( it = v1.begin(); it !=
v1.end(); it++ ) { cout << *it << endl; }
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

erase

Syntax:

```
  #include <vector>
  iterator erase( iterator loc );
  iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
  // Create a vector, load it with the first ten characters of the
alphabet
  vector<char> alphaVector;
  for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
  }
  int size = alphaVector.size();
  vector<char>::iterator startIterator;
  vector<char>::iterator tempIterator;
  for( int i=0; i < size; i++ ) {
    startIterator = alphaVector.begin();
    alphaVector.erase( startIterator );
    // Display the vector
    for( tempIterator = alphaVector.begin(); tempIterator !=
alphaVector.end(); tempIterator++ ) {
      cout << *tempIterator;
    }
    cout << endl;
  }
```

That code would display the following output:

```
  BCDEFGHIJ
  CDEFGHIJ
  DEFGHIJ
  EFGHIJ
  FGHIJ
  GHIJ
  HIJ
  IJ
  J
```

In the next example, erase() is called with two iterators to delete a range of elements from a vector:

```
  // create a vector, load it with the first ten characters of the
```

```
alphabet
  vector<char> alphaVector;
  for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
  }
  // display the complete vector
  for( int i = 0; i < alphaVector.size(); i++ ) {
    cout << alphaVector[i];
  }
  cout << endl;
  // use erase to remove all but the first two and last three elements
  // of the vector
  alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
  // display the modified vector
  for( int i = 0; i < alphaVector.size(); i++ ) {
    cout << alphaVector[i];
  }
  cout << endl;
```

When run, the above code displays:

```
  ABCDEFGHIJ
  ABHIJ
```

Related topics:
clear
insert
pop_back
(C++ Lists) pop_front
(C++ Lists) remove
(C++ Lists) remove_if

front

Syntax:

```
  #include <vector>
  TYPE& front();
  const TYPE& front() const;
```

The front() function returns a reference to the first element of the vector, and runs in constant time.

For example, the following code uses a vector and the sort() algorithm to display the first word (in alphabetical order) entered by a user:

```
  vector<string> words;
  string str;

  while( cin >> str ) words.push_back(str);
  sort( words.begin(), words.end() );
  cout << "In alphabetical order, the first word is '" << words.front()
<< "'." << endl;
```

When provided with this input:

```
  now is the time for all good men to come to the aid of their country
```

...the above code displays:

```
  In alphabetical order, the first word is 'aid'.
```

Related topics:
back
(C++ Lists) pop_front
(C++ Lists) push_front

insert

Syntax:

```
  #include <vector>
  iterator insert( iterator loc, const TYPE& val );
  void insert( iterator loc, size_type num, const TYPE& val );
  void insert( iterator loc, input_iterator start, input_iterator end );
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

Note that inserting elements into a vector can be relatively time-intensive, since the underlying data structure for a vector is an array. In order to insert data into an array, you might need to displace a lot of the elements of that array, and this can take linear time. If you are planning on doing a lot of insertions into your vector and you care about speed, you might be better off using a container that has a linked list as its underlying data structure (such as a List or a Deque).

For example, the following code uses the insert() function to splice four copies of the character 'C' into a vector of characters:

```
 // Create a vector, load it with the first 10 characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 // Insert four C's into the vector
 vector<char>::iterator theIterator = alphaVector.begin();
 alphaVector.insert( theIterator, 4, 'C' );
 // Display the vector
 for( theIterator = alphaVector.begin(); theIterator !=
alphaVector.end(); theIterator++ )    {
   cout << *theIterator;
 }
```

This code would display:

```
 CCCCABCDEFGHIJ
```

On the next page is another example of the insert() function. In this code, insert() is used to append the contents of one vector onto the end of another:

```
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );
vector<int> v2;
v2.push_back( 5 );
v2.push_back( 6 );
v2.push_back( 7 );
v2.push_back( 8 );
cout << "Before, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
v2.insert( v2.end(), v1.begin(), v1.end() );
cout << "After, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, this code displays:

```
Before, v2 is: 5 6 7 8
After, v2 is: 5 6 7 8 0 1 2 3
```

Related topics:
assign
erase
push_back
(C++ Lists) merge
(C++ Lists) push_front
(C++ Lists) splice

---

max_size

Syntax:

```
#include <vector>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the vector can hold. The max_size() function should not be confused with the size() or capacity() functions, which return the number of elements currently in the vector and the the number of elements that the vector will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

pop_back

Syntax:

```
  #include <vector>
  void pop_back();
```

The pop_back() function removes the last element of the vector.

pop_back() runs in constant time.

Related topics:
back
erase
(C++ Lists) pop_front
push_back

---

push_back

Syntax:

```
  #include <vector>
  void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the vector.

For example, the following code puts 10 integers into a vector:

```
   vector<int> the_vector;
   for( int i = 0; i < 10; i++ ) {
     the_vector.push_back( i );
   }
```

When displayed, the resulting vector would look like this:

```
 0 1 2 3 4 5 6 7 8 9
```

push_back() runs in constant time.

Related topics:
assign
insert
pop_back
(C++ Lists) push_front

---

rbegin

Syntax:

```
  #include <vector>
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current vector.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

rend

Syntax:

```
  #include <vector>
  reverse_iterator rend();
  const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current vector.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

reserve

Syntax:

```
  #include <vector>
  void reserve( size_type size );
```

The reserve() function sets the capacity of the vector to at least *size*.

reserve() runs in linear time.

Related topics:
capacity

---

resize

Syntax:

```
#include <vector>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the vector to *size*. If *val* is specified then any newly-created elements will be initialized to have a value of *val*.

This function runs in linear time.

Related topics:
Vector constructors & destructors
capacity
size

---

size

Syntax:

```
#include <vector>
size_type size() const;
```

The size() function returns the number of elements in the current vector.

Related topics:
capacity
empty
(C++ Strings) length
max_size
resize

---

swap

Syntax:

```
#include <vector>
void swap( container& from );
```

The swap() function exchanges the elements of the current vector with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the contents of two vectors:

```
vector v1;
v1.push_back("I'm in v1!");
vector v2;
v2.push_back("And I'm in v2!");
v1.swap(v2);
cout << "The first element in v1 is " << v1.front() << endl;
cout << "The first element in v2 is " << v2.front() << endl;
```

The above code displays:

```
The first element in v1 is And I'm in v2!
The first element in v2 is I'm in v1!
```

Related topics:
= operator
(C++ Lists) splice

C++ Double-ended Queues

Double-ended queues are like vectors, except that they allow fast insertions and deletions at the beginning (as well as the end) of the container.

Container constructors	create dequeues and initialize them with some data
Container operators	compare, assign, and access elements of a dequeue
assign	assign elements to a dequeue
at	returns an element at a specific location
back	returns a reference to last element of a dequeue
begin	returns an iterator to the beginning of the dequeue
clear	removes all elements from the dequeue
empty	true if the dequeue has no elements
end	returns an iterator just past the last element of a dequeue
erase	removes elements from a dequeue
front	returns a reference to the first element of a dequeue
insert	inserts elements into the dequeue
max_size	returns the maximum number of elements that the dequeue can hold
pop_back	removes the last element of a dequeue
pop_front	removes the first element of the dequeue
push_back	add an element to the end of the dequeue
push_front	add an element to the front of the dequeue
rbegin	returns a reverse_iterator to the end of the dequeue
rend	returns a reverse_iterator to the beginning of the dequeue
resize	change the size of the dequeue
size	returns the number of items in the dequeue
swap	swap the contents of this dequeue with another

Container constructors

Syntax:

```
  #include <deque>
  container();
  container( const container& c );
  container( size_type num, const TYPE& val = TYPE() );
  container( input_iterator start, input_iterator end );
  ~container();
```

The default dequeue constructor takes no arguments, creates a new instance of that dequeue.

The second constructor is a default copy constructor that can be used to create a new dequeue that is a copy of the given dequeue *c*.

The third constructor creates a dequeue with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
 vector<int> v1( 5, 42 );
```

The last constructor creates a dequeue that is initialized to contain the elements between *start* and *end*. For example:

```
 // create a vector of random integers
 cout << "original vector: ";
 vector<int> v;
 for( int i = 0; i < 10; i++ ) {
   int num = (int) rand() % 10;
   cout << num << " ";
   v.push_back( num );
 }
 cout << endl;
 // find the first element of v that is even
 vector<int>::iterator iter1 = v.begin();
 while( iter1 != v.end() && *iter1 % 2 != 0 ) {
   iter1++;
 }
 // find the last element of v that is even
 vector<int>::iterator iter2 = v.end();
 do {
   iter2--;
 } while( iter2 != v.begin() && *iter2 % 2 != 0 );
 cout << "first even number: " << *iter1 << ", last even number: " <<
*iter2 << endl;
 cout << "new vector: ";
 vector<int> v2( iter1, iter2 );
 for( int i = 0; i < v2.size(); i++ ) {
   cout << v2[i] << " ";
 }
 cout << endl;
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the dequeue should be destroyed.

Container operators

Syntax:

```
#include <deque>
TYPE& operator[]( size_type index );
const TYPE& operator[]( size_type index ) const;
container operator=(const container& c2);
bool operator==(const container& c1, const container& c2);
bool operator!=(const container& c1, const container& c2);
bool operator<(const container& c1, const container& c2);
bool operator>(const container& c1, const container& c2);
bool operator<=(const container& c1, const container& c2);
bool operator>=(const container& c1, const container& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a dequeue can be examined with the [] operator.

Performing a comparison or assigning one dequeue to another takes linear time. The [] operator runs in constant time.

Two `containers` are equal if:

1. Their size is the same, and
2. Each member in location i in one dequeue is equal to the the member in location i in the other dequeue.

Comparisons among dequeues are done lexicographically.

For example, the following code uses the [] operator to access all of the elements of a vector:

```
vector<int> v( 5, 1 );
for( int i = 0; i < v.size(); i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
}
```

Related topics:
at

Container [] operator

Syntax:

```
  TYPE& operator[]( size_type index );  const TYPE& operator[]( size_type index ) const;
```

Individual elements of a dequeue can be examined with the [] operator.

For example, the following code uses the [] operator to access all of the elements of a vector:

```
 for( int i = 0; i < v.size(); i++ ) {
   cout << "Element " << i << " is " << v[i] << endl;
 }
```

The [] operator runs in constant time.

Related topics:
at

Container constructors & destructors

Syntax:

```
  container();  container( const container& c );  ~container();
```

Every dequeue has a default constructor, copy constructor, and destructor.

The default constructor takes no arguments, creates a new instance of that dequeue, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new dequeue that is a copy of the given dequeue *c*.

The default destructor is called when the dequeue should be destroyed.

For example, the following code creates a pointer to a vector of integers and then uses the default dequeue constructor to allocate a memory for a new vector:

```
 v = new vector<int>();
```

Related topics:
Special container constructors, resize

---

assign

Syntax:

```
  #include <deque>
  void assign( size_type num, const TYPE& val );
  void assign( input_iterator start, input_iterator end );
```

The assign() function either gives the current dequeue the values from *start* to *end*, or gives it *num* copies of *val*.

This function will destroy the previous contents of the dequeue.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
 vector<int> v;
 v.assign( 10, 42 );
 for( int i = 0; i < v.size(); i++ ) {
   cout << v[i] << " ";
 }
 cout << endl;
```

The above code displays the following output:

```
 42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
 vector<int> v1;
 for( int i = 0; i < 10; i++ ) {
   v1.push_back( i );
 }
 vector<int> v2;
 v2.assign( v1.begin(), v1.end() );
 for( int i = 0; i < v2.size(); i++ ) {
   cout << v2[i] << " ";
 }
 cout << endl;
```

When run, the above code displays the following output:

```
 0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
push_front

at

Syntax:

```
#include <deque>
TYPE& at( size_type loc );
const TYPE& at( size_type loc ) const;
```

The at() function returns a reference to the element in the dequeue at index *loc*. The at() function is safer than the [] operator, because it won't let you reference items outside the bounds of the dequeue.

For example, consider the following code:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
}
```

This code overrunns the end of the vector, producing potentially dangerous results. The following code would be much safer:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
  cout << "Element " << i << " is " << v.at(i) << endl;
}
```

Instead of attempting to read garbage values from memory, the at() function will realize that it is about to overrun the vector and will throw an exception.

Related topics:
(C++ Multimaps) Multimap operators
Deque operators

back

Syntax:

```
  #include <deque>
  TYPE& back();
  const TYPE& back() const;
```

The back() function returns a reference to the last element in the dequeue.

For example:

```
 vector<int> v;
 for( int i = 0; i < 5; i++ ) {
   v.push_back(i);
 }
 cout << "The first element is " << v.front()
      << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
 The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
pop_back

begin

Syntax:

```
  #include <deque>
  iterator begin();
  const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the dequeue. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
   // Create a list of characters
   list<char> charList;
   for( int i=0; i < 10; i++ ) {
     charList.push_front( i + 65 );
   }
   // Display the list
   list<char>::iterator theIterator;
   for( theIterator = charList.begin(); theIterator != charList.end();
theIterator++ ) {
     cout << *theIterator;
   }
```

Related topics:
end
rbegin
rend

clear

Syntax:

```
  #include <deque>
  void clear();
```

The function clear() deletes all of the elements in the dequeue. clear() runs in linear time.

Related topics:
erase

---

empty

Syntax:

```
  #include <deque>
  bool empty() const;
```

The empty() function returns true if the dequeue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a dequeue and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end

Syntax:

```
  #include <deque>
  iterator end();
  const_iterator end() const;
```

The end() function returns an iterator just past the end of the dequeue.

Note that before you can access the last element of the dequeue using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
 vector<int> v1( 5, 789 );
 vector<int>::iterator it;
 for( it = v1.begin(); it != v1.end(); it++ ) {
   cout << *it << endl;
 }
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

erase

Syntax:

```
  #include <deque>
  iterator erase( iterator loc );
  iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
 // Create a vector, load it with the first ten characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 int size = alphaVector.size();
 vector<char>::iterator startIterator;
 vector<char>::iterator tempIterator;
 for( int i=0; i < size; i++ ) {
   startIterator = alphaVector.begin();
   alphaVector.erase( startIterator );
   // Display the vector
   for( tempIterator = alphaVector.begin(); tempIterator !=
alphaVector.end(); tempIterator++ ) {
     cout << *tempIterator;
   }
   cout << endl;
 }
```

That code would display the following output:

```
 BCDEFGHIJ
 CDEFGHIJ
 DEFGHIJ
 EFGHIJ
 FGHIJ
 GHIJ
 HIJ
 IJ
 J
```

In the example on the following page, erase() is called with two iterators to delete a range of elements from a vector:

```
// create a vector, load it with the first ten characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 // display the complete vector
 for( int i = 0; i < alphaVector.size(); i++ ) {
   cout << alphaVector[i];
 }
 cout << endl;
 // use erase to remove all but the first two and last three elements
 // of the vector
 alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
 // display the modified vector
 for( int i = 0; i < alphaVector.size(); i++ ) {
   cout << alphaVector[i];
 }
 cout << endl;
```

When run, the above code displays:

```
 ABCDEFGHIJ
 ABHIJ
```

Related topics:
clear
insert
pop_back
pop_front
(C++ Lists) remove
(C++ Lists) remove_if

---

front

Syntax:

```
  #include <deque>
  TYPE& front();
  const TYPE& front() const;
```

The front() function returns a reference to the first element of the dequeue, and runs in constant time.

Related topics:
back
pop_front
push_front

---

insert

Syntax:

```
  #include <deque>
  iterator insert( iterator loc, const TYPE& val );
  void insert( iterator loc, size_type num, const TYPE& val );
  template<TYPE> void insert( iterator loc, input_iterator start,
input_iterator end );
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

For example:

```
 // Create a vector, load it with the first 10 characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 // Insert four C's into the vector
 vector<char>::iterator theIterator = alphaVector.begin();
 alphaVector.insert( theIterator, 4, 'C' );
 // Display the vector
 for( theIterator = alphaVector.begin(); theIterator !=
alphaVector.end(); theIterator++ )    {
   cout << *theIterator;
 }
```

This code would display:

```
 CCCCABCDEFGHIJ
```

Related topics:
assign
erase
(C++ Lists) merge
push_back
push_front
(C++ Lists) splice

max_size

Syntax:

```
#include <deque>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the dequeue can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the dequeue and the the number of elements that the dequeue will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

pop_back

Syntax:

```
#include <deque>
void pop_back();
```

The pop_back() function removes the last element of the dequeue.

pop_back() runs in constant time.

Related topics:

back	pop_front
erase	push_back

---

pop_front

Syntax:

```
#include <deque>
void pop_front();
```

The function pop_front() removes the first element of the dequeue.

The pop_front() function runs in constant time.

Related topics:

erase	pop_back
front	push_front

---

push_back

Syntax:

```
  #include <deque>
  void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the dequeue.

For example, the following code puts 10 integers into a list:

```
   list<int> the_list;
   for( int i = 0; i < 10; i++ )
     the_list.push_back( i );
```

When displayed, the resulting list would look like this:

```
 0 1 2 3 4 5 6 7 8 9
```

push_back() runs in constant time.

Related topics:
assign
insert
pop_back
push_front

---

push_front

Syntax:

```
  #include <deque>
  void push_front( const TYPE& val );
```

The push_front() function inserts *val* at the beginning of dequeue.

push_front() runs in constant time.

Related topics:
assign
front
insert
pop_front
push_back

---

rbegin

Syntax:

```
#include <deque>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current dequeue.

rbegin() runs in constant time.

Related topics:

begin	end	rend

---

rend

Syntax:

```
#include <deque>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current dequeue.

rend() runs in constant time.

Related topics:

begin	end	rbegin

---

resize

Syntax:

```
#include <deque>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the dequeue to *size*. If *val* is specified then any newly-created elements will be initialized to have a value of *val*.

This function runs in linear time.

Related topics:
(C++ Multimaps) Multimap constructors & destructors
(C++ Strings) capacity
size

size

Syntax:

```
#include <deque>
size_type size() const;
```

The size() function returns the number of elements in the current dequeue.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
max_size
resize

---

swap

Syntax:

```
#include <deque>
void swap( container& from );
```

The swap() function exchanges the elements of the current dequeue with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
(C++ Lists) splice

C++ Lists

Lists are sequences of elements stored in a linked list. Compared to vectors, they allow fast insertions and deletions, but slower random access.

List constructors	create lists and initialize them with some data
List operators	assign and compare lists
assign	assign elements to a list
back	returns a reference to last element of a list
begin	returns an iterator to the beginning of the list
clear	removes all elements from the list
empty	true if the list has no elements
end	returns an iterator just past the last element of a list
erase	removes elements from a list
front	returns a reference to the first element of a list
insert	inserts elements into the list
max_size	returns the maximum number of elements that the list can hold
merge	merge two lists
pop_back	removes the last element of a list
pop_front	removes the first element of the list
push_back	add an element to the end of the list
push_front	add an element to the front of the list
rbegin	returns a reverse_iterator to the end of the list
remove	removes elements from a list
remove_if	removes elements conditionally
rend	returns a reverse_iterator to the beginning of the list
resize	change the size of the list
reverse	reverse the list
size	returns the number of items in the list
sort	sorts a list into ascending order
splice	merge two lists in constant time
swap	swap the contents of this list with another
unique	removes consecutive duplicate elemen

List constructors

Syntax:

```
  #include <list>
  list();
  list( const list& c );
  list( size_type num, const TYPE& val = TYPE() );
  list( input_iterator start, input_iterator end );
  ~list();
```

The default list constructor takes no arguments, creates a new instance of that list.

The second constructor is a default copy constructor that can be used to create a new list that is a copy of the given list *c*.

The third constructor creates a list with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
 vector<int> v1( 5, 42 );
```

The last constructor creates a list that is initialized to contain the elements between *start* and *end*. For example:

```
 // create a vector of random integers
 cout << "original vector: ";
 vector<int> v;
 for( int i = 0; i < 10; i++ ) {
   int num = (int) rand() % 10;
   cout << num << " ";
   v.push_back( num );
 }
 cout << endl;
 // find the first element of v that is even
 vector<int>::iterator iter1 = v.begin();
 while( iter1 != v.end() && *iter1 % 2 != 0 ) {
   iter1++;
 }
 // find the last element of v that is even
 vector<int>::iterator iter2 = v.end();
 do {
   iter2--;
 } while( iter2 != v.begin() && *iter2 % 2 != 0 );
 cout << "first even number: " << *iter1 << ", last even number: " <<
*iter2 << endl;
 cout << "new vector: ";
 vector<int> v2( iter1, iter2 );
 for( int i = 0; i < v2.size(); i++ ) {
   cout << v2[i] << " ";
 }
 cout << endl;
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the list should be destroyed.

List operators

Syntax:

```
  #include <list>
  list operator=(const list& c2);
  bool operator==(const list& c1, const list& c2);
  bool operator!=(const list& c1, const list& c2);
  bool operator<(const list& c1, const list& c2);
  bool operator>(const list& c1, const list& c2);
  bool operator<=(const list& c1, const list& c2);
  bool operator>=(const list& c1, const list& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one list to another takes linear time.

Two lists are equal if:

1. Their size is the same, and
2. Each member in location i in one list is equal to the the member in location i in the other list.

Comparisons among lists are done lexicographically.

Related topics:
(C++ Strings) String operators
(C++ Strings) at
merge
unique

Container constructors & destructors

Syntax:

```
  container();  container( const container& c );  ~container();
```

Every list has a default constructor, copy constructor, and destructor.

The default constructor takes no arguments, creates a new instance of that list, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new list that is a copy of the given list *c*.

The default destructor is called when the list should be destroyed.

For example, the following code creates a pointer to a vector of integers and then uses the default list constructor to allocate a memory for a new vector:

```
 v = new vector<int>();
```

Related topics:
Special container constructors, resize

assign

Syntax:

```
  #include <list>
  void assign( size_type num, const TYPE& val );
  void assign( input_iterator start, input_iterator end );
```

The assign() function either gives the current list the values from *start* to *end*, or gives it *num* copies of *val*.

This function will destroy the previous contents of the list.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}
vector<int> v2;
v2.assign( v1.begin(), v1.end() );
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
push_front

back

Syntax:

```
  #include <list>
  TYPE& back();
  const TYPE& back() const;
```

The back() function returns a reference to the last element in the list.

For example:

```
 vector<int> v;
 for( int i = 0; i < 5; i++ ) {
   v.push_back(i);
 }
 cout << "The first element is " << v.front()
      << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
 The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
pop_back

begin

Syntax:

```
  #include <list>
  iterator begin();
  const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the list. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
   // Create a list of characters
   list<char> charList;
   for( int i=0; i < 10; i++ ) {
     charList.push_front( i + 65 );
   }
   // Display the list
   list<char>::iterator theIterator;
   for( theIterator = charList.begin(); theIterator != charList.end();
theIterator++ ) {
     cout << *theIterator;
   }
```

Related topics:
end
rbegin
rend

---

clear

Syntax:

```
  #include <list>
  void clear();
```

The function clear() deletes all of the elements in the list. clear() runs in linear time.

Related topics:
erase

---

empty

Syntax:

```
  #include <list>
  bool empty() const;
```

The empty() function returns true if the list has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a list and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end

Syntax:

```
  #include <list>
  iterator end();
  const_iterator end() const;
```

The end() function returns an iterator just past the end of the list.

Note that before you can access the last element of the list using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
vector<int> v1( 5, 789 );
vector<int>::iterator it;
for( it = v1.begin(); it != v1.end(); it++ ) {
  cout << *it << endl;
}
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:

begin	rbegin	rend

erase

Syntax:

```
  #include <list>
  iterator erase( iterator loc );
  iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
 // Create a vector, load it with the first ten characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 int size = alphaVector.size();
 vector<char>::iterator startIterator;
 vector<char>::iterator tempIterator;
 for( int i=0; i < size; i++ ) {
   startIterator = alphaVector.begin();
   alphaVector.erase( startIterator );
   // Display the vector
   for( tempIterator = alphaVector.begin(); tempIterator !=
alphaVector.end(); tempIterator++ ) {
     cout << *tempIterator;
   }
   cout << endl;
 }
```

That code would display the following output:

```
 BCDEFGHIJ
 CDEFGHIJ
 DEFGHIJ
 EFGHIJ
 FGHIJ
 GHIJ
 HIJ
 IJ
 J
```

In the example on the next page, erase() is called with two iterators to delete a range of elements from a vector:

```
 // create a vector, load it with the first ten characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 // display the complete vector
 for( int i = 0; i < alphaVector.size(); i++ ) {
   cout << alphaVector[i];
 }
 cout << endl;
 // use erase to remove all but the first two and last three elements
 // of the vector
 alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
 // display the modified vector
 for( int i = 0; i < alphaVector.size(); i++ ) {
   cout << alphaVector[i];
 }
 cout << endl;
```

When run, the above code displays:

```
 ABCDEFGHIJ
 ABHIJ
```

Related topics:
clear
insert
pop_back
pop_front
remove
remove_if

---

front

Syntax:

```
  #include <list>
  TYPE& front();
  const TYPE& front() const;
```

The front() function returns a reference to the first element of the list, and runs in constant time.

Related topics:
back
pop_front
push_front

---

insert

Syntax:

```
  #include <list>
  iterator insert( iterator loc, const TYPE& val );
  void insert( iterator loc, size_type num, const TYPE& val );
  template<TYPE> void insert( iterator loc, input_iterator start,
input_iterator end );
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

For example:

```
 // Create a vector, load it with the first 10 characters of the
alphabet
 vector<char> alphaVector;
 for( int i=0; i < 10; i++ ) {
   alphaVector.push_back( i + 65 );
 }
 // Insert four C's into the vector
 vector<char>::iterator theIterator = alphaVector.begin();
 alphaVector.insert( theIterator, 4, 'C' );
 // Display the vector
 for( theIterator = alphaVector.begin(); theIterator !=
alphaVector.end(); theIterator++ )    {
   cout << *theIterator;
 }
```

This code would display:

```
 CCCCABCDEFGHIJ
```

Related topics:
assign
erase
merge
push_back
push_front
splice

max_size

Syntax:

```
#include <list>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the list can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the list and the the number of elements that the list will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

merge

Syntax:

```
#include <list>
void merge( list &lst );
void merge( list &lst, BinPred compfunction );
```

The function merge() merges the list with lst, producing a combined list that is ordered with respect to the < operator. If compfunction is specified, then it is used as the comparison function for the lists instead of <.

merge() runs in linear time.

Related topics:

Container operators	insert	splice

---

pop_back

Syntax:

```
#include <list>
void pop_back();
```

The pop_back() function removes the last element of the list.

pop_back() runs in constant time.

Related topics:

back	pop_front	push_back
erase		

---

pop_front

Syntax:

```
  #include <list>
  void pop_front();
```

The function pop_front() removes the first element of the list.

The pop_front() function runs in constant time.

Related topics:
erase
front
pop_back
push_front

---

push_back

Syntax:

```
  #include <list>
  void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the list.

For example, the following code puts 10 integers into a list:

```
   list<int> the_list;
   for( int i = 0; i < 10; i++ )
     the_list.push_back( i );
```

When displayed, the resulting list would look like this:

```
 0 1 2 3 4 5 6 7 8 9
```

push_back() runs in constant time.

Related topics:
assign
insert
pop_back
push_front

---

push_front

Syntax:

```
#include <list>
void push_front( const TYPE& val );
```

The push_front() function inserts *val* at the beginning of list.

push_front() runs in constant time.

Related topics:
assign
front
insert
pop_front
push_back

---

rbegin

Syntax:

```
#include <list>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current list.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

remove

Syntax:

```
#include <list>
void remove( const TYPE &val );
```

The function remove() removes all elements that are equal to val from the list.

For example, the following code creates a list of the first 10 characters of the alphabet, then uses remove() to remove the letter 'E' from the list:

```
// Create a list that has the first 10 letters of the alphabet
list<char> charList;
for( int i=0; i < 10; i++ )
  charList.push_front( i + 65 );
// Remove all instances of 'E'
charList.remove( 'E' );
```

Remove runs in linear time.

Related topics:
erase
remove_if
unique

---

remove_if

Syntax:

```
#include <list>
void remove_if( UnPred pr );
```

The remove_if() function removes all elements from the list for which the unary predicate *pr* is true.

remove_if() runs in linear time.

Related topics:
erase
remove
unique

---

rend

Syntax:

```
#include <list>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current list.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

resize

Syntax:

```
#include <list>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the list to *size*. If *val* is specified then any newly-created elements will be initialized to have a value of *val*.

This function runs in linear time.

Related topics:
(C++ Multimaps) Multimap constructors & destructors
(C++ Strings) capacity
size

---

reverse

Syntax:

```
#include <list>
void reverse();
```

The function reverse() reverses the list, and takes linear time.

Related topics:
sort

---

size

Syntax:

```
#include <list>
size_type size() const;
```

The size() function returns the number of elements in the current list.

Related topics:

(C++ Strings) capacity	empty	resize
(C++ Strings) length	max_size	

---

sort

Syntax:

```
#include <list>
void sort();
void sort( BinPred p );
```

The sort() function is used to sort lists into ascending order. Ordering is done via the < operator, unless *p* is specified, in which case it is used to determine if an element is less than another.

Sorting takes N log N time.

Related topics:
reverse

---

splice

Syntax:

```
#include <list>
void splice( iterator pos, list& lst );
void splice( iterator pos, list& lst, iterator del );
void splice( iterator pos, list& lst, iterator start, iterator end );
```

The splice() function inserts *lst* at location *pos*. If specified, the element(s) at *del* or from *start* to *end* are removed.

splice() simply moves elements from one list to another, and doesn't actually do any copying or deleting. Because of this, splice() runs in constant time.

Related topics:

insert	merge	swap

swap

Syntax:

```
#include <list>
void swap( container& from );
```

The swap() function exchanges the elements of the current list with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
splice

---

unique

Syntax:

```
#include <list>
void unique();
void unique( BinPred pr );
```

The function unique() removes all consecutive duplicate elements from the list. Note that only consecutive duplicates are removed, which may require that you sort() the list first.

Equality is tested using the == operator, unless *pr* is specified as a replacement. The ordering of the elements in a list should not change after a call to unique().

unique() runs in linear time.

Related topics:
Container operators
remove
remove_if

C++ Priority Queues

C++ Priority Queues are like queues, but the elements inside the queue are ordered by some predicate.

Priority queue constructors	construct a new priority queue
empty	true if the priority queue has no elements
pop	removes the top element of a priority queue
push	adds an element to the end of the priority queue
size	returns the number of items in the priority queue
top	returns the top element of the priority queue

Priority queue constructors

Syntax:

```
  #include <queue>
  priority_queue( const Compare& cmp = Compare(), const Container& c =
Container() );
  priority_queue( input_iterator start, input_iterator end, const
Compare& comp = Compare(), const Container& c = Container() );
```

Priority queues can be constructed with an optional compare function *cmp* and an optional container *c*. If *start* and *end* are specified, the priority queue will be constructed with the elements between *start* and *end*.

---

empty

Syntax:

```
  #include <queue>
  bool empty() const;
```

The empty() function returns true if the priority queue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a priority queue and display its contents in reverse order:

```
 vector<int> v;
 for( int i = 0; i < 5; i++ ) {
   v.push_back(i);
 }
 while( !v.empty() ) {
   cout << v.back() << endl;
   v.pop_back();
 }
```

Related topics:
size

---

pop

Syntax:

```
  #include <queue>
  void pop();
```

The function pop() removes the top element of the priority queue and discards it.

Related topics:
push
top

---

push

Syntax:

```
#include <queue>
void push( const TYPE& val );
```

The function push() adds *val* to the end of the current priority queue.

For example, the following code uses the push() function to add ten integers to the end of a queue:

```
queue<int> q;
for( int i=0; i < 10; i++ )
  q.push(i);
```

---

size

Syntax:

```
#include <queue>
size_type size() const;
```

The size() function returns the number of elements in the current priority queue.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
(C++ Multimaps) max_size
(C++ Strings) resize

---

top

Syntax:

```
#include <queue>
TYPE& top();
```

The function top() returns a reference to the top element of the priority queue.

For example, the following code removes all of the elements from a stack and uses top() to display them:

```
 while( !s.empty() ) {
   cout << s.top() << " ";
   s.pop();
 }
```

Related topics:
pop

C++ Queues

The C++ Queue is a container adapter that gives the programmer a FIFO (first-in, first-out) data structure.

Queue constructor	construct a new queue
back	returns a reference to last element of a queue
empty	true if the queue has no elements
front	returns a reference to the first element of a queue
pop	removes the first element of a queue
push	adds an element to the end of the queue
size	returns the number of items in the queue

Queue constructor

Syntax:

```
#include <queue>
queue();
queue( const Container& con );
```

Queues have a default constructor as well as a copy constructor that will create a new queue out of the container *con*.

For example, the following code creates a queue of strings, populates it with input from the user, and then displays it back to the user:

```
queue<string> waiting_line;
while( waiting_line.size() < 5 ) {
  cout << "Welcome to the line, please enter your name: ";
  string s;
  getline( cin, s );
  waiting_line.push(s);
}
while( !waiting_line.empty() ) {
  cout << "Now serving: " << waiting_line.front() << endl;
  waiting_line.pop();
}
```

When run, the above code might produce this output:

```
Welcome to the line, please enter your name: Nate
Welcome to the line, please enter your name: lizzy
Welcome to the line, please enter your name: Robert B. Parker
Welcome to the line, please enter your name: ralph
Welcome to the line, please enter your name: Matthew
Now serving: Nate
Now serving: lizzy
Now serving: Robert B. Parker
Now serving: ralph
Now serving: Matthew
```

back

Syntax:

```
  #include <queue>
  TYPE& back();
  const TYPE& back() const;
```

The back() function returns a reference to the last element in the queue.

For example:

```
queue<int> q;
for( int i = 0; i < 5; i++ ) {
  q.push(i);
}
cout << "The first element is " << q.front()
     << " and the last element is " << q.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
(C++ Lists) pop_back

---

empty

Syntax:

```
  #include <queue>
  bool empty() const;
```

The empty() function returns true if the queue has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a queue while displaying its contents:

```
queue<int> q;
for( int i = 0; i < 5; i++ ) {
  q.push(i);
}
while( !q.empty() ) {
  cout << q.front() << endl;
  q.pop();
}
```

Related topics:
size

front

Syntax:

```
#include <queue>
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the queue, and runs in constant time.

Related topics:
back
(C++ Lists) pop_front
(C++ Lists) push_front

---

pop

Syntax:

```
#include <queue>
void pop();
```

The function pop() removes the first element of the queue and discards it.

Related topics:
push
(C++ Priority Queues) top

---

push

Syntax:

```
#include <queue>
void push( const TYPE& val );
```

The function push() adds *val* to the end of the current queue.

For example, the following code uses the push() function to add ten integers to the end of a queue:

```
queue<int> q;
for( int i=0; i < 10; i++ ) {
  q.push(i);
}
```

Related topics:
pop

---

size

Syntax:

```
#include <queue>
size_type size() const;
```

The size() function returns the number of elements in the current queue.

Related topics:
empty
(C++ Strings) capacity
(C++ Strings) length
(C++ Multimaps) max_size
(C++ Strings) resize

C++ Stacks

The C++ Stack is a container adapter that gives the programmer the functionality of a stack -- specifically, a FILO (first-in, last-out) data structure.

Stack constructors	construct a new stack
empty	true if the stack has no elements
pop	removes the top element of a stack
push	adds an element to the top of the stack
size	returns the number of items in the stack
top	returns the top element of the stack

Stack constructors

Syntax:

```
  #include <stack>
  stack();
  stack( const Container& con );
```

Stacks have an empty constructor and a constructor that can be used to specify a container type.

empty

Syntax:

```
  #include <stack>
  bool empty() const;
```

The empty() function returns true if the stack has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a stack and display its contents in reverse order:

```
 stack<int> s;
 for( int i = 0; i < 5; i++ ) {
   s.push(i);
 }
 while( !s.empty() ) {
   cout << s.top() << endl;
   s.pop();
 }
```

Related topics:
size

---

pop

Syntax:

```
  #include <stack>
  void pop();
```

The function pop() removes the top element of the stack and discards it.

Related topics:
push
top

---

push

Syntax:

```
#include <stack>
void push( const TYPE& val );
```

The function push() adds *val* to the top of the current stack.

For example, the following code uses the push() function to add ten integers to the top of a stack:

```
 stack<int> s;
 for( int i=0; i < 10; i++ )
   s.push(i);
```

Related topics:
pop

---

size

Syntax:

```
#include <stack>
size_type size() const;
```

The size() function returns the number of elements in the current stack.

Related topics:
empty
(C++ Multimaps) max_size
(C++ Strings) capacity
(C++ Strings) length
(C++ Strings) resize

---

top

Syntax:

```
#include <stack>
TYPE& top();
```

The function top() returns a reference to the top element of the stack.

For example, the following code removes all of the elements from a stack and uses top() to display them:

```
while( !s.empty() ) {
  cout << s.top() << " ";
  s.pop();
}
```

Related topics:
pop

C++ Sets

The C++ Set is an associative container that contains a sorted set of unique objects.

Set constructors & destructors	default methods to allocate, copy, and deallocate sets
Set operators	assign and compare sets
begin	returns an iterator to the beginning of the set
clear	removes all elements from the set
count	returns the number of elements matching a certain key
empty	true if the set has no elements
end	returns an iterator just past the last element of a set
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a set
find	returns an iterator to specific elements
insert	insert items into a set
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the set can hold
rbegin	returns a reverse_iterator to the end of the set
rend	returns a reverse_iterator to the beginning of the set
size	returns the number of items in the set
swap	swap the contents of this set with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

Set constructors & destructors

Syntax:

```
  #include <set>
  set();
  set( const set& c );
  ~set();
```

Every set has a default constructor, copy constructor, and destructor.

The default constructor takes no arguments, creates a new instance of that set, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new set that is a copy of the given set *c*.

The default destructor is called when the set should be destroyed.

For example, the following code creates a pointer to a vector of integers and then uses the default set constructor to allocate a memory for a new vector:

```
 vector<int>* v;
 v = new vector<int>();
```

Related topics:
(C++ Strings) resize

Set operators

Syntax:

```
#include <set>
set operator=(const set& c2);
bool operator==(const set& c1, const set& c2);
bool operator!=(const set& c1, const set& c2);
bool operator<(const set& c1, const set& c2);
bool operator>(const set& c1, const set& c2);
bool operator<=(const set& c1, const set& c2);
bool operator>=(const set& c1, const set& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one set to another takes linear time.

Two sets are equal if:

1. Their size is the same, and
2. Each member in location i in one set is equal to the the member in location i in the other set.

Comparisons among sets are done lexicographically.

Related topics:
(C++ Strings) String operators
(C++ Strings) at
(C++ Lists) merge
(C++ Lists) unique

begin

Syntax:

```
  #include <set>
  iterator begin();
  const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the set. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
   // Create a list of characters
   list<char> charList;
   for( int i=0; i < 10; i++ ) {
     charList.push_front( i + 65 );
   }
   // Display the list
   list<char>::iterator theIterator;
   for( theIterator = charList.begin(); theIterator != charList.end();
theIterator++ ) {
     cout << *theIterator;
   }
```

Related topics:
end
rbegin
rend

---

clear

Syntax:

```
  #include <set>
  void clear();
```

The function clear() deletes all of the elements in the set. clear() runs in linear time.

Related topics:
(C++ Lists) erase

---

count

Syntax:

```
#include <set>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the set.

count() should run in logarithmic time.

---

empty

Syntax:

```
#include <set>
bool empty() const;
```

The empty() function returns true if the set has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a set and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end

Syntax:

```
#include <set>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the set.

Note that before you can access the last element of the set using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
vector<int> v1( 5, 789 );
vector<int>::iterator it;
for( it = v1.begin(); it != v1.end(); it++ ) {
  cout << *it << endl;
}
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

---

equal_range

Syntax:

```
#include <set>
pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

---

erase

Syntax:

```
#include <set>
void erase( iterator pos );
void erase( iterator start, iterator end );
size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

---

find

Syntax:

```
#include <set>
iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the set if *key* is not found.

find() runs in logarithmic time.

---

insert

Syntax:

```
#include <set>
iterator insert( iterator i, const TYPE& val );
void insert( input_iterator start, input_iterator end );
pair<iterator,bool> insert( const TYPE& val );
```

The function insert() either:

- inserts *val* after the element at *pos* (where *pos* is really just a suggestion as to where *val* should go, since sets and maps are ordered), and returns an iterator to that element.
- inserts a range of elements from *start* to *end*.
- inserts *val*, but only if *val* doesn't already exist. The return value is an iterator to the element inserted, and a boolean describing whether an insertion took place.

Related topics:
(C++ Maps) Map operators

---

key_comp

Syntax:

```
#include <set>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:
value_comp

---

lower_bound

Syntax:

```
#include <set>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

---

max_size

Syntax:

```
#include <set>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the set can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the set and the the number of elements that the set will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

rbegin

Syntax:

```
#include <set>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current set.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

rend

Syntax:

```
#include <set>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current set.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

size

Syntax:

```
#include <set>
size_type size() const;
```

The size() function returns the number of elements in the current set.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
max_size
(C++ Strings) resize

---

swap

Syntax:

```
#include <set>
void swap( container& from );
```

The swap() function exchanges the elements of the current set with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
(C++ Lists) splice

---

upper_bound

Syntax:

```
#include <set>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the set with a key greater than *key*.

Related topics:
lower_bound

---

value_comp

Syntax:

```
#include <set>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.

value_comp() runs in constant time.

Related topics:
key_comp

C++ Multisets

C++ Multisets are like sets, in that they are associative containers containing a sorted set of objects, but differ in that they allow duplicate objects.

Container constructors & destructors	default methods to allocate, copy, and deallocate multisets
Container operators	assign and compare multisets
begin	returns an iterator to the beginning of the multiset
clear	removes all elements from the multiset
count	returns the number of elements matching a certain key
empty	true if the multiset has no elements
end	returns an iterator just past the last element of a multiset
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a multiset
find	returns an iterator to specific elements
insert	inserts items into a multiset
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the multiset can hold
rbegin	returns a reverse_iterator to the end of the multiset
rend	returns a reverse_iterator to the beginning of the multiset
size	returns the number of items in the multiset
swap	swap the contents of this multiset with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

Container constructors & destructors

Syntax:

```
  #include <set>
  container();
  container( const container& c );
  ~container();
```

Every multiset has a default constructor, copy constructor, and destructor.

The default constructor takes no arguments, creates a new instance of that multiset, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new multiset that is a copy of the given multiset *c*.

The default destructor is called when the multiset should be destroyed.

For example, the following code creates a pointer to a vector of integers and then uses the default multiset constructor to allocate a memory for a new vector:

```
 vector<int>* v;
 v = new vector<int>();
```

Related topics:
(C++ Strings) resize

Container operators

Syntax:

```
#include <set>
container operator=(const container& c2);
bool operator==(const container& c1, const container& c2);
bool operator!=(const container& c1, const container& c2);
bool operator<(const container& c1, const container& c2);
bool operator>(const container& c1, const container& c2);
bool operator<=(const container& c1, const container& c2);
bool operator>=(const container& c1, const container& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one multiset to another takes linear time.

Two multisets are equal if:

1. Their size is the same, and
2. Each member in location i in one multiset is equal to the the member in location i in the other multiset.

Comparisons among multisets are done lexicographically.

Related topics:
(C++ Strings) String operators
(C++ Strings) at
(C++ Lists) merge
(C++ Lists) unique

begin

Syntax:

```
#include <set>
iterator begin();
const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the multiset. begin() should run in constant tim

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
 // Create a list of characters
 list<char> charList;
 for( int i=0; i < 10; i++ ) {
   charList.push_front( i + 65 );
 }
 // Display the list
 list<char>::iterator theIterator;
 for( theIterator = charList.begin(); theIterator != charList.end(); theIterator++ )
   cout << *theIterator;
 }
```

Related topics:
end
rbegin
rend

---

clear

Syntax:

```
#include <set>
void clear();
```

The function clear() deletes all of the elements in the multiset. clear() runs in linear time.

Related topics:
(C++ Lists) erase

---

count

Syntax:

```
#include <set>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the multiset.

count() should run in logarithmic time.

---

empty

Syntax:

```
#include <set>
bool empty() const;
```

The empty() function returns true if the multiset has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a multiset and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end

Syntax:

```
  #include <set>
  iterator end();
  const_iterator end() const;
```

The end() function returns an iterator just past the end of the multiset.

Note that before you can access the last element of the multiset using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
 vector<int> v1( 5, 789 );
 vector<int>::iterator it;
 for( it = v1.begin(); it != v1.end(); it++ ) {
   cout << *it << endl;
 }
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

---

equal_range

Syntax:

```
  #include <set>
  pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

---

erase

Syntax:

```
#include <set>
void erase( iterator pos );
void erase( iterator start, iterator end );
size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

---

find

Syntax:

```
#include <set>
iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the multiset if *key* is not found.

find() runs in logarithmic time.

---

insert

Syntax:

```
#include <set>
iterator insert( iterator pos, const TYPE& val );
iterator insert( const TYPE& val );
void insert( input_iterator start, input_iterator end );
```

The function insert() either:

- inserts *val* after the element at *pos* (where *pos* is really just a suggestion as to where *val* should go, since multisets and multimaps are ordered), and returns an iterator to that element.
- inserts *val* into the multiset, returning an iterator to the element inserted.
- inserts a range of elements from *start* to *end*.

---

key_comp

Syntax:

```
  #include <set>
  key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:
value_comp

---

lower_bound

Syntax:

```
  #include <set>
  iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

---

max_size

Syntax:

```
  #include <set>
  size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the multiset can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the multiset and the the number of elements that the multiset will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

rbegin

Syntax:

```
#include <set>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current multiset.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

rend

Syntax:

```
#include <set>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current multiset.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

size
Syntax:

```
#include <set>
size_type size() const;
```

The size() function returns the number of elements in the current multiset.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
max_size
(C++ Strings) resize

swap

Syntax:

```
#include <set>
void swap( container& from );
```

The swap() function exchanges the elements of the current multiset with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
 string first( "This comes first" );
 string second( "And this is second" );
 first.swap( second );
 cout << first << endl;
 cout << second << endl;
```

The above code displays:

```
 And this is second
 This comes first
```

Related topics:
(C++ Lists) splice

---

upper_bound

Syntax:

```
#include <set>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the multiset with a key greater than *key*.

Related topics:
lower_bound

---

value_comp

Syntax:

```
#include <set>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.
value_comp() runs in constant time.

Related topics:
key_comp

C++ Maps

C++ Maps are sorted associative containers that contain unique key/value pairs. For example, you could create a map that associates a string with an integer, and then use that map to associate the number of days in each month with the name of each month.

Map constructors & destructors	default methods to allocate, copy, and deallocate maps
Map operators	assign, compare, and access elements of a map
begin	returns an iterator to the beginning of the map
clear	removes all elements from the map
count	returns the number of elements matching a certain key
empty	true if the map has no elements
end	returns an iterator just past the last element of a map
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a map
find	returns an iterator to specific elements
insert	insert items into a map
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the map can hold
rbegin	returns a reverse_iterator to the end of the map
rend	returns a reverse_iterator to the beginning of the map
size	returns the number of items in the map
swap	swap the contents of this map with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

Map Constructors & Destructors

Syntax:

```
#include <map>
map();
map( const map& m );
map( iterator start, iterator end );
map( iterator start, iterator end, const key_compare& cmp );
map( const key_compare& cmp );
~map();
```

The default constructor takes no arguments, creates a new instance of that map, and runs in constant time. The default copy constructor runs in linear time and can be used to create a new map that is a copy of the given map *m*.

You can also create a map that will contain a copy of the elements between *start* and *end*, or specify a comparison function *cmp*.

The default destructor is called when the map should be destroyed.

For example, the following code creates a map that associates a string with an integer:

```
struct strCmp {
  bool operator()( const char* s1, const char* s2 ) const {
    return strcmp( s1, s2 ) < 0;
  }
};
...
map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;
cout << "Bart is " << ages["Bart"] << " years old" << endl;
```

Related topics:
Map Operators

Map operators

Syntax:

```
#include <map>
TYPE& operator[]( const key_type& key );
map operator=(const map& c2);
bool operator==(const map& c1, const map& c2);
bool operator!=(const map& c1, const map& c2);
bool operator<(const map& c1, const map& c2);
bool operator>(const map& c1, const map& c2);
bool operator<=(const map& c1, const map& c2);
bool operator>=(const map& c1, const map& c2);
```

Maps can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a map can be examined with the [] operator.

Performing a comparison or assigning one map to another takes linear time.

Two maps are equal if:

1. Their size is the same, and
2. Each member in location *i* in one map is equal to the the member in location *i* in the other map.

Comparisons among maps are done lexicographically. For example, the following code defines a map between strings and integers and loads values into the map using the [] operator:

```
  struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
      return strcmp( s1, s2 ) < 0;
    }
  };
  map<const char*, int, strCmp> ages;
  ages["Homer"] = 38;
  ages["Marge"] = 37;
  ages["Lisa"] = 8;
  ages["Maggie"] = 1;
  ages["Bart"] = 11;
  cout << "Bart is " << ages["Bart"] << " years old" << endl;
  cout << "In alphabetical order: " << endl;
  for( map<const char*, int, strCmp>::iterator iter = ages.begin(); iter !=
ages.end(); iter++ ) {
    cout << (*iter).first << " is " << (*iter).second << " years old" << endl;
  }
```

When run, the above code displays this output:

```
Bart is 11 years old
In alphabetical order:
Bart is 11 years old
Homer is 38 years old
Lisa is 8 years old
Maggie is 1 years old
Marge is 37 years old
```

Related topics:
insert
Map Constructors & Destructors

begin

Syntax:

```
  #include <map>
  iterator begin();
  const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the map. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
  map<string,int> stringCounts;
  string str;
  while( cin >> str ) stringCounts[str]++;
  map<string,int>::iterator iter;
  for( iter = stringCounts.begin(); iter != stringCounts.end(); iter+
+ ) {
    cout << "word: " << iter->first << ", count: " << iter->second <<
endl;
  }
```

When given this input:

```
  here are some words and here are some more words
```

...the above code generates this output:

```
  word: and, count: 1
  word: are, count: 2
  word: here, count: 2
  word: more, count: 1
  word: some, count: 2
  word: words, count: 2
```

Related topics:
end
rbegin
rend

clear

Syntax:

```
#include <map>
void clear();
```

The function clear() deletes all of the elements in the map. clear() runs in linear time.

Related topics:
erase

---

count

Syntax:

```
#include <map>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the map.

count() should run in logarithmic time.

---

empty

Syntax:

```
  #include <map>
  bool empty() const;
```

The empty() function returns true if the map has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a while loop to clear a map and display its contents in order:

```
  struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
      return strcmp( s1, s2 ) < 0;
    }
  };
  ...
  map<const char*, int, strCmp> ages;
  ages["Homer"] = 38;
  ages["Marge"] = 37;
  ages["Lisa"] = 8;
  ages["Maggie"] = 1;
  ages["Bart"] = 11;
  while( !ages.empty() ) {
    cout << "Erasing: " << (*ages.begin()).first << ", " <<
(*ages.begin()).second << endl;
    ages.erase( ages.begin() );
  }
```

When run, the above code displays:

```
  Erasing: Bart, 11
  Erasing: Homer, 38
  Erasing: Lisa, 8
  Erasing: Maggie, 1
  Erasing: Marge, 37
```

Related topics:
begin
erase
size

end

Syntax:

```
  #include <map>
  iterator end();
  const_iterator end() const;
```

The end() function returns an iterator just past the end of the map.

Note that before you can access the last element of the map using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
 vector<int> v1( 5, 789 );
 vector<int>::iterator it;
 for( it = v1.begin(); it != v1.end(); it++ ) {
   cout << *it << endl;
 }
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

equal_range

Syntax:

```
  #include <map>
  pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

erase

Syntax:

```
  #include <map>
  void erase( iterator pos );
  void erase( iterator start, iterator end );
  size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

For example, the following code uses erase() in a while loop to incrementally clear a map and display its contents in order:

```
  struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
      return strcmp( s1, s2 ) < 0;
    }
  };
  ...
  map<const char*, int, strCmp> ages;
  ages["Homer"] = 38;
  ages["Marge"] = 37;
  ages["Lisa"] = 8;
  ages["Maggie"] = 1;
  ages["Bart"] = 11;
  while( !ages.empty() ) {
    cout << "Erasing: " << (*ages.begin()).first << ", " <<
(*ages.begin()).second << endl;
    ages.erase( ages.begin() );
  }
```

When run, the above code displays:

```
  Erasing: Bart, 11
  Erasing: Homer, 38
  Erasing: Lisa, 8
  Erasing: Maggie, 1
  Erasing: Marge, 37
```

Related topics:
begin
clear
empty
size

find

Syntax:

```
  #include <map>
  iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the map if *key* is not found.

find() runs in logarithmic time.

For example, the following code uses the find() function to determine how many times a user entered a certain word:

```
  map<string,int> stringCounts;
  string str;
  while( cin >> str ) stringCounts[str]++;
  map<string,int>::iterator iter = stringCounts.find("spoon");
  if( iter != stringCounts.end() ) {
    cout << "You typed '" << iter->first << "' " << iter->second << "
time(s)" << endl;
  }
```

When run with this input:

```
my spoon is too big.  my spoon is TOO big!  my SPOON is TOO big!  I am
a BANANA!
```

...the above code produces this output:

```
You typed 'spoon' 2 time(s)
```

insert

Syntax:

```
  #include <map>
  iterator insert( iterator i, const TYPE& pair );
  void insert( input_iterator start, input_iterator end );
  pair<iterator,bool> insert( const TYPE& pair );
```

The function insert() either:

- inserts *pair* after the element at *pos* (where *pos* is really just a suggestion as to where *pair* should go, since sets and maps are ordered), and returns an iterator to that element.
- inserts a range of elements from *start* to *end*.
- inserts *pair<key*,*val>*, but only if no element with key *key* already exists. The return value is an iterator to the element inserted (or an existing pair with key *key*), and a boolean which is true if an insertion took place.

For example, the following code uses the insert() function (along with the make_pair() function) to insert some data into a map and then displays that data:

```
  map<string,int> theMap;
  theMap.insert( make_pair( "Key 1", -1 ) );
  theMap.insert( make_pair( "Another key!", 32 ) );
  theMap.insert( make_pair( "Key the Three", 66667 ) );
  map<string,int>::iterator iter;
  for( iter = theMap.begin(); iter != theMap.end(); ++iter ) {
    cout << "Key: '" << iter->first << "', Value: " << iter->second <<
endl;
  }
```

When run, the above code displays this output:

```
  Key: 'Another key!', Value: 32
  Key: 'Key 1', Value: -1
  Key: 'Key the Three', Value: 66667
```

Note that because maps are sorted containers, the output is sorted by the key value. In this case, since the map key data type is string, the map is sorted alphabetically by key.

Related topics:
Map operators

key_comp

Syntax:

```
#include <map>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:
value_comp

lower_bound

Syntax:

```
#include <map>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

max_size

Syntax:

```
#include <map>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the map can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the map and the the number of elements that the map will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

rbegin

Syntax:

```
#include <map>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current map.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

rend

Syntax:

```
#include <map>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current map.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

size

Syntax:

```
#include <map>
size_type size() const;
```

The size() function returns the number of elements in the current map.

Related topics:
empty
max_size

---

swap

Syntax:

```
#include <map>
void swap( container& from );
```

The swap() function exchanges the elements of the current map with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
(C++ Lists) splice

---

upper_bound

Syntax:

```
#include <map>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the map with a key greater than *key*.

Related topics:
lower_bound

---

value_comp

Syntax:

```
#include <map>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.value_comp() runs in constant time.

Related topics:
key_comp

C++ Multimaps

C++ Multimaps are like maps, in that they are sorted associative containers, but differ from maps in that they allow duplicate keys.

Multimap constructors & destructors	default methods to allocate, copy, and deallocate multimaps
Multimap operators	assign and compare multimaps
begin	returns an iterator to the beginning of the multimap
clear	removes all elements from the multimap
count	returns the number of elements matching a certain key
empty	true if the multimap has no elements
end	returns an iterator just past the last element of a multimap
equal_range	returns iterators to the first and just past the last elements matching a specific key
erase	removes elements from a multimap
find	returns an iterator to specific elements
insert	inserts items into a multimap
key_comp	returns the function that compares keys
lower_bound	returns an iterator to the first element greater than or equal to a certain value
max_size	returns the maximum number of elements that the multimap can hold
rbegin	returns a reverse_iterator to the end of the multimap
rend	returns a reverse_iterator to the beginning of the multimap
size	returns the number of items in the multimap
swap	swap the contents of this multimap with another
upper_bound	returns an iterator to the first element greater than a certain value
value_comp	returns the function that compares values

Multimap constructors & destructors

Syntax:

```
  #include <map>
  multimap();
  multimap( const multimap& c );
  multimap( iterator begin, iterator end,
            const key_compare& cmp = Compare(), const allocator& alloc
= Allocator() );
  ~multimap();
```

Multimaps have several constructors:

- The default constructor takes no arguments, creates a new instance of that multimap, and runs in constant time.
- The default copy constructor runs in linear time and can be used to create a new multimap that is a copy of the given multimap *c*.
- Multimaps can also be created from a range of elements defined by *begin* and *end*. When using this constructor, an optional comparison function *cmp* and allocator *alloc* can also be provided.

The default destructor is called when the multimap should be destroyed.

The template definition of multimaps requires that both a key type and value type be supplied. For example, you can instantiate a multimap that maps strings to integers with this statement:

```
  multimap<string,int> m;
```

You can also supply a comparison function and an allocator in the template:

```
  multimap<string,int,myComp,myAlloc> m;
```

For example, the following code uses a multimap to associate a series of employee names with numerical IDs:

```
  multimap<string,int> m;
  int employeeID = 0;
  m.insert( pair<string,int>("Bob Smith",employeeID++) );
  m.insert( pair<string,int>("Bob Thompson",employeeID++) );
  m.insert( pair<string,int>("Bob Smithey",employeeID++) );
  m.insert( pair<string,int>("Bob Smith",employeeID++) );
  cout << "Number of employees named 'Bob Smith': " << m.count("Bob
Smith") << endl;
  cout << "Number of employees named 'Bob Thompson': " << m.count("Bob
Thompson") << endl;
  cout << "Number of employees named 'Bob Smithey': " << m.count("Bob
Smithey") << endl;
  cout << "Employee list: " << endl;
  for( multimap<string, int>::iterator iter = m.begin(); iter !=
m.end(); ++iter ) {
    cout << " Name: " << iter->first << ", ID #" << iter->second <<
endl;
  }
```

When run, the above code produces the following output. Note that the employee list is displayed in alphabetical order, because multimaps are sorted associative containers:

```
Number of employees named 'Bob Smith': 2
Number of employees named 'Bob Thompson': 1
Number of employees named 'Bob Smithey': 1
Employee list:
 Name: Bob Smith, ID #0
 Name: Bob Smith, ID #3
 Name: Bob Smithey, ID #2
 Name: Bob Thompson, ID #1
```

Related topics:
count
insert

Multimap operators

Syntax:

```
#include <map>
multimap operator=(const multimap& c2);
bool operator==(const multimap& c1, const multimap& c2);
bool operator!=(const multimap& c1, const multimap& c2);
bool operator<(const multimap& c1, const multimap& c2);
bool operator>(const multimap& c1, const multimap& c2);
bool operator<=(const multimap& c1, const multimap& c2);
bool operator>=(const multimap& c1, const multimap& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one multimap to another takes linear time.

Two multimaps are equal if:

1. Their size is the same, and
2. Each member in location i in one multimap is equal to the the member in location i in the other multimap.

Comparisons among multimaps are done lexicographically.

Related topics:
Multimap Constructors

begin

Syntax:

```
  #include <map>
  iterator begin();
  const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the multimap. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
   // Create a list of characters
   list<char> charList;
   for( int i=0; i < 10; i++ ) {
     charList.push_front( i + 65 );
   }
   // Display the list
   list<char>::iterator theIterator;
   for( theIterator = charList.begin(); theIterator != charList.end();
theIterator++ ) {
     cout << *theIterator;
   }
```

Related topics:
end
rbegin
rend

---

clear

Syntax:

```
  #include <map>
  void clear();
```

The function clear() deletes all of the elements in the multimap. clear() runs in linear time.

Related topics:
(C++ Lists) erase

---

count

Syntax:

```
#include <map>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of *key* in the multimap.

count() should run in logarithmic time.

---

empty

Syntax:

```
#include <map>
bool empty() const;
```

The empty() function returns true if the multimap has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a multimap and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end

Syntax:

```
  #include <map>
  iterator end();
  const_iterator end() const;
```

The end() function returns an iterator just past the end of the multimap.

Note that before you can access the last element of the multimap using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
 vector<int> v1( 5, 789 );
 vector<int>::iterator it;
 for( it = v1.begin(); it != v1.end(); it++ ) {
   cout << *it << endl;
 }
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

---

find

Syntax:

```
  #include <map>
  iterator find( const key_type& key );
```

The find() function returns an iterator to *key*, or an iterator to the end of the multimap if *key* is not found.

find() runs in logarithmic time.

---

equal_range

Syntax:

```
  #include <map>
  pair<iterator, iterator> equal_range( const key_type& key );
```

The function equal_range() returns two iterators - one to the first element that contains *key*, another to a point just after the last element that contains *key*.

For example, here is a hypothetical input-configuration loader using multimaps, strings and equal_range():

```
  multimap<string,pair<int,int> > input_config;

  // read configuration from file "input.conf" to input_config
  readConfigFile( input_config, "input.conf" );

  pair<multimap<string,pair<int,int>
>::iterator,multimap<string,pair<int,int> >::iterator> ii;
  multimap<string,pair<int,int> >::iterator i;

  ii = input_config.equal_range("key");         // keyboard key-bindings
  // we can iterate over a range just like with begin() and end()
  for( i = ii.first; i != ii.second; ++i ) {
    // add a key binding with this key and output
    bindkey(i->second.first, i->second.second);
  }
  ii = input_config.equal_range("joyb");        // joystick button key-
bindings
  for( i = ii.first; i != ii.second; ++i ) {
    // add a key binding with this joystick button and output
    bindjoyb(i->second.first, i->second.second);
  }
```

---

erase

Syntax:

```
  #include <map>
  void erase( iterator pos );
  void erase( iterator start, iterator end );
  size_type erase( const key_type& key );
```

The erase function() either erases the element at *pos*, erases the elements between *start* and *end*, or erases all elements that have the value of *key*.

---

insert

Syntax:

```
  #include <map>
  iterator insert( iterator pos, const TYPE& val );
  iterator insert( const TYPE& val );
  void insert( input_iterator start, input_iterator end );
```

The function insert() either:

- inserts *val* after the element at *pos* (where *pos* is really just a suggestion as to where *val* should go, since multimaps are ordered), and returns an iterator to that element.
- inserts *val* into the multimap, returning an iterator to the element inserted.
- inserts a range of elements from *start* to *end*.

For example, the following code uses the insert() function to add several <name,ID> pairs to a employee multimap:

```
  multimap<string,int> m;
  int employeeID = 0;
  m.insert( pair<string,int>("Bob Smith",employeeID++) );
  m.insert( pair<string,int>("Bob Thompson",employeeID++) );
  m.insert( pair<string,int>("Bob Smithey",employeeID++) );
  m.insert( pair<string,int>("Bob Smith",employeeID++) );
  cout << "Number of employees named 'Bob Smith': " << m.count("Bob
Smith") << endl;
  cout << "Number of employees named 'Bob Thompson': " << m.count("Bob
Thompson") << endl;
  cout << "Number of employees named 'Bob Smithey': " << m.count("Bob
Smithey") << endl;
  cout << "Employee list: " << endl;
  for( multimap<string, int>::iterator iter = m.begin(); iter !=
m.end(); ++iter ) {
    cout << " Name: " << iter->first << ", ID #" << iter->second <<
endl;
  }
```

When run, the above code produces the following output:

```
  Number of employees named 'Bob Smith': 2
  Number of employees named 'Bob Thompson': 1
  Number of employees named 'Bob Smithey': 1
  Employee list:
   Name: Bob Smith, ID #0
   Name: Bob Smith, ID #3
   Name: Bob Smithey, ID #2
   Name: Bob Thompson, ID #1
```

key_comp

Syntax:

```
#include <map>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in constant time.

Related topics:
value_comp

---

lower_bound

Syntax:

```
#include <map>
iterator lower_bound( const key_type& key );
```

The lower_bound() function returns an iterator to the first element which has a value greater than or equal to key.

lower_bound() runs in logarithmic time.

Related topics:
upper_bound

---

max_size

Syntax:

```
#include <map>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the multimap can hold. The max_size() function should not be confused with the size() or (C++ Strings) capacity() functions, which return the number of elements currently in the multimap and the the number of elements that the multimap will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

rbegin

Syntax:

```
#include <map>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current multimap.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

rend

Syntax:

```
#include <map>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current multimap.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

size

Syntax:

```
#include <map>
size_type size() const;
```

The size() function returns the number of elements in the current multimap.

Related topics:
(C++ Strings) capacity
empty
(C++ Strings) length
max_size
(C++ Strings) resize

---

swap

Syntax:

```
#include <map>
void swap( container& from );
```

The swap() function exchanges the elements of the current multimap with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
(C++ Lists) splice

---

upper_bound

Syntax:

```
#include <map>
iterator upper_bound( const key_type& key );
```

The function upper_bound() returns an iterator to the first element in the multimap with a key greater than *key*.

Related topics:
lower_bound

---

value_comp

Syntax:

```
#include <map>
value_compare value_comp() const;
```

The value_comp() function returns the function that compares values.

value_comp() runs in constant time.

Related topics:
key_comp

C++ Bitsets

C++ Bitsets give the programmer a set of bits as a data structure. Bitsets can be manipulated by various binary operators such as logical AND, OR, and so on.

Bitset Constructors	create new bitsets
Bitset Operators	compare and assign bitsets
any	true if any bits are set
count	returns the number of set bits
flip	reverses the bitset
none	true if no bits are set
reset	sets bits to zero
set	sets bits
size	number of bits that the bitset can hold
test	returns the value of a given bit
to_string	string representation of the bitset
to_ulong	returns an integer representation of the bitset

Bitset Operators

Syntax:

```
  #include <bitset>
  !=, ==, &=, ^=, |=, ~, <<=, >>=, []
```

These operators all work with bitsets. They can be described as follows:

- != returns true if the two bitsets are not equal.
- == returns true if the two bitsets are equal.
- &= performs the AND operation on the two bitsets.
- ^= performs the XOR operation on the two bitsets.
- |= performs the OR operation on the two bitsets.
- ~ reverses the bitset (same as calling flip())
- <<= shifts the bitset to the left
- >>= shifts the bitset to the right
- [x] returns a reference to the xth bit in the bitset.

For example, the following code creates a bitset and shifts it to the left 4 places:

```
// create a bitset out of a number
bitset<8> bs2( (long) 131 );
cout << "bs2 is " << bs2 << endl;
// shift the bitset to the left by 4 digits
bs2 <<= 4;
cout << "now bs2 is " << bs2 << endl;
```

When the above code is run, it displays:

```
bs2 is 10000011
now bs2 is 00110000
```

Bitset Constructors

Syntax:

```
 #include <bitset>
 bitset();
 bitset( unsigned long val );
```

Bitsets can either be constructed with no arguments or with an unsigned long number val that will be converted into binary and inserted into the bitset. When creating bitsets, the number given in the place of the template determines how long the bitset is.

For example, the following code creates two bitsets and displays them:

```
// create a bitset that is 8 bits long
bitset<8> bs;
// display that bitset
for( int i = (int) bs.size()-1; i >= 0; i-- ) {
  cout << bs[i] << " ";
}
cout << endl;
// create a bitset out of a number
bitset<8> bs2( (long) 131 );
// display that bitset, too
for( int i = (int) bs2.size()-1; i >= 0; i-- ) {
  cout << bs2[i] << " ";
}
cout << endl;
```

any

Syntax:

```
  #include <bitset>
  bool any();
```

The any() function returns true if any bit of the bitset is 1, otherwise, it returns false.

Related topics:
count
none

---

count

Syntax:

```
  #include <bitset>
  size_type count();
```

The function count() returns the number of bits that are set to 1 in the bitset.

Related topics:
any

---

flip

Syntax:

```
  #include <bitset>
  bitset<N>& flip();
  bitset<N>& flip( size_t pos );
```

The flip() function inverts all of the bits in the bitset, and returns the bitset. If *pos* is specified, only the bit at position *pos* is flipped.

---

none

Syntax:

```
#include <bitset>
bool none();
```

The none() function only returns true if none of the bits in the bitset are set to 1.

Related topics:
any

---

reset

Syntax:

```
#include <bitset>
bitset<N>& reset();
bitset<N>& reset( size_t pos );
```

The reset() function clears all of the bits in the bitset, and returns the bitset. If *pos* is specified, then only the bit at position *pos* is cleared.

---

set

Syntax:

```
#include <bitset>
bitset<N>& set();
bitset<N>& set( size_t pos, int val=1 );
```

The set() function sets all of the bits in the bitset, and returns the bitset. If *pos* is specified, then only the bit at position *pos* is set.

---

size

Syntax:

```
#include <bitset>
size_t size();
```

The size() function returns the number of bits that the bitset can hold.

---

test

Syntax:

```
#include <bitset>
bool test( size_t pos );
```

The function test() returns the value of the bit at position *pos*.

---

to_string

Syntax:

```
#include <bitset>
string to_string();
```

The to_string() function returns a string representation of the bitset.

Related topics:
to_ulong

---

to_ulong

Syntax:

```
#include <bitset>
unsigned long to_ulong();
```

The function to_ulong() returns the bitset, converted into an unsigned long integer.

Related topics:
to_string