# **All C++ Functions**

Bitset Constructors (C++ Bitsets)	create new bitsets
Bitset Operators (C++ Bitsets)	compare and assign bitsets
<u>Vector constructors</u>	create vectors and initialize them with some data
Container constructors (C++ Double-ended Queues)	create containers and initialize them with some data
Container constructors (C++ Lists)	create containers and initialize them with some data
Container constructors & destructors (C++ Sets)	default methods to allocate, copy, and deallocate containers
Container constructors & destructors (C++ Multisets)	default methods to allocate, copy, and deallocate multisets
Map constructors & destructors (C++ Maps)	default methods to allocate, copy, and deallocate maps
Multimap constructors & destructors (C++ Multimaps)	default methods to allocate, copy, and deallocate containers
Container operators (C++ Lists)	assign and compare containers
Container operators (C++ Sets)	assign and compare containers
Container operators (C++ Multisets)	assign and compare containers
Multimap operators (C++ Multimaps)	assign and compare containers
<u>Vector operators</u>	compare, assign, and access elements of a vector
Container operators (C++ Double-ended Queues)	compare, assign, and access elements of a container
I/O Constructors (C++ I/O)	constructors
Map operators (C++ Maps)	assign, compare, and access elements of a map

Priority queue constructors (C++ Priority Queues)	construct a new priority queue
Queue constructor (C++ Queues)	construct a new queue
Stack constructors (C++ Stacks)	construct a new stack
String constructors (C++ Strings)	create strings from arrays of characte and other strings
String operators (C++ Strings)	concatenate strings, assign strings, us strings for I/O, compare strings
accumulate (C++ Algorithms)	sum up a range of elements
adjacent_difference (C++ Algorithms)	compute the differences between adjacent elements in a range
adjacent_find (C++ Algorithms)	finds two items that are adjacent to eachother
any (C++ Bitsets)	true if any bits are set
append (C++ Strings)	append characters and strings onto a string
assign (C++ Vectors)	assign elements to a container
assign (C++ Double-ended Queues)	assign elements to a container
assign (C++ Lists)	assign elements to a container
assign (C++ Strings)	give a string values from strings of characters and other C++ strings
at (C++ Vectors)	returns an element at a specific location
at (C++ Double-ended Queues)	returns an element at a specific location
at (C++ Strings)	returns an element at a specific location
auto ptr (Miscellaneous C++)	create pointers that automatically destroy objects
back (C++ Vectors)	returns a reference to last element of container
back (C++ Double-ended Queues)	returns a reference to last element of container

back (C++ Lists)	returns a reference to last element of a container
back (C++ Queues)	returns a reference to last element of a container
<u>bad</u> (C++ I/O)	true if an error occurred
begin (C++ Strings)	returns an iterator to the beginning of the container
begin (C++ Vectors)	returns an iterator to the beginning of the container
begin (C++ Double-ended Queues)	returns an iterator to the beginning of the container
begin (C++ Lists)	returns an iterator to the beginning of the container
begin (C++ Sets)	returns an iterator to the beginning of the container
begin (C++ Multisets)	returns an iterator to the beginning of the container
begin (C++ Maps)	returns an iterator to the beginning of the container
begin (C++ Multimaps)	returns an iterator to the beginning of the container
binary_search (C++ Algorithms)	determine if an element exists in a certain range
c_str (C++ Strings)	returns a standard C character array version of the string
capacity (C++ Vectors)	returns the number of elements that the container can hold
capacity (C++ Strings)	returns the number of elements that the container can hold
clear (C++ I/O)	clear and set status flags
clear (C++ Strings)	removes all elements from the container
clear (C++ Vectors)	removes all elements from the container
clear (C++ Double-ended Queues)	removes all elements from the container

clear (C++ Lists)	removes all elements from the container
clear (C++ Sets)	removes all elements from the container
clear (C++ Multisets)	removes all elements from the container
clear (C++ Maps)	removes all elements from the container
clear (C++ Multimaps)	removes all elements from the container
close (C++ I/O)	close a stream
compare (C++ Strings)	compares two strings
copy (C++ Strings)	copies characters from a string into an array
copy (C++ Algorithms)	copy some range of elements to a new location
copy_backward (C++ Algorithms)	copy a range of elements in backwards order
copy n (C++ Algorithms)	copy N elements
count (C++ Sets)	returns the number of elements matching a certain key
count (C++ Multisets)	returns the number of elements matching a certain key
count (C++ Maps)	returns the number of elements matching a certain key
count (C++ Multimaps)	returns the number of elements matching a certain key
count (C++ Bitsets)	returns the number of set bits
count (C++ Algorithms)	return the number of elements matching a given value
count_if (C++ Algorithms)	return the number of elements for which a predicate is true
data (C++ Strings)	returns a pointer to the first character of a string
empty (C++ Strings)	true if the container has no elements
empty (C++ Vectors)	true if the container has no elements

empty (C++ Double-ended Queues)	true if the container has no elements
empty (C++ Lists)	true if the container has no elements
empty (C++ Sets)	true if the container has no elements
empty (C++ Multisets)	true if the container has no elements
empty (C++ Maps)	true if the container has no elements
empty (C++ Multimaps)	true if the container has no elements
empty (C++ Stacks)	true if the container has no elements
empty (C++ Queues)	true if the container has no elements
<pre>empty (C++ Priority Queues)</pre>	true if the container has no elements
end (C++ Strings)	returns an iterator just past the last element of a container
end (C++ Vectors)	returns an iterator just past the last element of a container
end (C++ Double-ended Queues)	returns an iterator just past the last element of a container
end (C++ Lists)	returns an iterator just past the last element of a container
end (C++ Sets)	returns an iterator just past the last element of a container
end (C++ Multisets)	returns an iterator just past the last element of a container
end (C++ Maps)	returns an iterator just past the last element of a container
end (C++ Multimaps)	returns an iterator just past the last element of a container
<u>eof</u> (C++ I/O)	true if at the end-of-file
equal (C++ Algorithms)	determine if two sets of elements are the same
equal_range (C++ Sets)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Multisets)	returns iterators to the first and just past the last elements matching a specific key
	specific ney

equal_range (C++ Maps)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Multimaps)	returns iterators to the first and just past the last elements matching a specific key
equal_range (C++ Algorithms)	search for a range of elements that are all equal to a certain element
erase (C++ Strings)	removes elements from a string
erase (C++ Vectors)	removes elements from a container
erase (C++ Double-ended Queues)	removes elements from a container
erase (C++ Lists)	removes elements from a container
erase (C++ Sets)	removes elements from a container
erase (C++ Multisets)	removes elements from a container
erase (C++ Maps)	removes elements from a container
erase (C++ Multimaps)	removes elements from a container
<u>fail</u> (C++ I/O)	true if an error occurred
fill (C++ I/O)	manipulate the default fill character
fill (C++ Algorithms)	assign a range of elements a certain value
fill_n (C++ Algorithms)	assign a value to some number of elements
find (C++ Algorithms)	find a value in a given range
find (C++ Sets)	returns an iterator to specific elements
find (C++ Multisets)	returns an iterator to specific elements
find (C++ Maps)	returns an iterator to specific elements
find (C++ Multimaps)	returns an iterator to specific elements
find (C++ Strings)	find characters in the string
find_end (C++ Algorithms)	find the last sequence of elements in a certain range
find first not of (C++ Strings)	find first absence of characters
find_first_of (C++ Strings)	find first occurrence of characters
find_first_of (C++ Algorithms)	search for any one of a set of elements
find_if (C++ Algorithms)	find the first element for which a

	certain predicate is true
find last not of (C++ Strings)	find last absence of characters
find last of (C++ Strings)	find last occurrence of characters
flags (C++ I/O)	access or manipulate <u>io stream format</u> <u>flag</u> s
<u>flip</u> (C++ Bitsets)	reverses the bitset
flush (C++ I/O)	empty the buffer
<pre>for each (C++ Algorithms)</pre>	apply a function to a range of elements
front (C++ Vectors)	returns a reference to the first element of a container
front (C++ Double-ended	returns a reference to the first element
Queues)	of a container
front (C++ Lists)	returns a reference to the first element of a container
<pre>front (C++ Queues)</pre>	returns a reference to the first element of a container
gcount (C++ I/O)	number of characters read during last input
generate (C++ Algorithms)	saves the result of a function in a range
<pre>generate_n (C++ Algorithms)</pre>	saves the result of N applications of a function
<u>get</u> (C++ I/O)	read characters
getline (C++ I/O)	read a line of characters
getline (C++ Strings)	read data from an I/O stream into a string
good (C++ I/O)	true if no errors have occurred
ignore (C++ I/O)	read and discard characters
includes (C++ Algorithms)	returns true if one set is a subset of another
<pre>inner_product (C++ Algorithms)</pre>	compute the inner product of two ranges of elements
inplace_merge (C++ Algorithms)	merge two ordered ranges in-place
insert (C++ Strings)	insert characters into a string

insert (C++ Vectors)	inserts elements into the container
insert (C++ Double-ended	inserts elements into the container
Queues)	
insert (C++ Lists)	inserts elements into the container
insert (C++ Sets)	insert items into a container
insert (C++ Multisets)	inserts items into a container
insert (C++ Multimaps)	inserts items into a container
insert (C++ Maps)	insert items into a container
is_heap (C++ Algorithms)	returns true if a given range is a heap
is_sorted (C++ Algorithms)	returns true if a range is sorted in ascending order
iter_swap (C++ Algorithms)	swaps the elements pointed to by two iterators
key_comp (C++ Sets)	returns the function that compares keys
key_comp (C++ Multisets)	returns the function that compares keys
key_comp (C++ Maps)	returns the function that compares keys
key_comp (C++ Multimaps)	returns the function that compares keys
<u>length</u> (C++ Strings)	returns the length of the string
lexicographical_compare (C++	returns true if one range is
Algorithms)	lexicographically less than another
lexicographical_compare_3way (C++ Algorithms)	determines if one range is lexicographically less than or greater than another
<u>lower_bound</u> (C++ Sets)	returns an iterator to the first element greater than or equal to a certain value
lower_bound (C++ Multisets)	returns an iterator to the first element greater than or equal to a certain value
lower_bound (C++ Maps)	returns an iterator to the first element greater than or equal to a certain value
lower_bound (C++ Multimaps)	returns an iterator to the first element greater than or equal to a certain value
	search for the first place that a value

<u>lower_bound</u> (C++ Algorithms)	can be inserted while preserving order
<pre>make_heap (C++ Algorithms)</pre>	creates a heap out of a range of elements
max (C++ Algorithms)	returns the larger of two elements
<pre>max_element (C++ Algorithms)</pre>	returns the largest element in a range
max_size (C++ Strings)	returns the maximum number of elements that the container can hold
max_size (C++ Vectors)	returns the maximum number of elements that the container can hold
<pre>max_size (C++ Double-ended Queues)</pre>	returns the maximum number of elements that the container can hold
max_size (C++ Lists)	returns the maximum number of elements that the container can hold
max_size (C++ Sets)	returns the maximum number of elements that the container can hold
max_size (C++ Multisets)	returns the maximum number of elements that the container can hold
max_size (C++ Maps)	returns the maximum number of elements that the container can hold
max_size (C++ Multimaps)	returns the maximum number of elements that the container can hold
merge (C++ Lists)	merge two lists
merge (C++ Algorithms)	merge two sorted ranges
min (C++ Algorithms)	returns the smaller of two elements
min_element (C++ Algorithms)	returns the smallest element in a range
mismatch (C++ Algorithms)	finds the first position where two ranges differ
next_permutation (C++ Algorithms)	generates the next greater lexicographic permutation of a range of elements
none (C++ Bitsets)	true if no bits are set
nth_element (C++ Algorithms)	put one element in its sorted location and make sure that no elements to its left are greater than any elements to its

	right
open (C++ I/O)	create an input stream
<pre>partial sort (C++ Algorithms)</pre>	sort the first N elements of a range
partial_sort_copy (C++	copy and partially sort a range of
Algorithms)	elements
partial_sum (C++ Algorithms)	compute the partial sum of a range of elements
partition (C++ Algorithms)	divide a range of elements into two groups
peek (C++ I/O)	check the next input character
pop (C++ Stacks)	removes the top element of a container
pop (C++ Queues)	removes the top element of a container
pop (C++ Priority Queues)	removes the top element of a container
pop_back (C++ Vectors)	removes the last element of a container
pop_back (C++ Double-ended	removes the last element of a
Queues)	container
pop_back (C++ Lists)	removes the last element of a
	container
pop_front (C++ Double-ended	removes the first element of the
Queues)	container
pop_front (C++ Lists)	removes the first element of the container
pop_heap (C++ Algorithms)	remove the largest element from a heap
precision (C++ I/O)	manipulate the precision of a stream
<pre>prev_permutation (C++ Algorithms)</pre>	generates the next smaller lexicographic permutation of a range of elements
push (C++ Stacks)	adds an element to the top of the container
push (C++ Queues)	adds an element to the end of the container
push (C++ Priority Queues)	adds an element to the end of the container
	add an element to the end of the

<pre>push_back (C++ Vectors)</pre>	container
<pre>push_back (C++ Double-ended Queues)</pre>	add an element to the end of the container
push_back (C++ Lists)	add an element to the end of the container
<pre>push_back (C++ Strings)</pre>	add an element to the end of the container
<pre>push_front (C++ Double-ended Queues)</pre>	add an element to the front of the container
<pre>push_front (C++ Lists)</pre>	add an element to the front of the container
<pre>push_heap (C++ Algorithms)</pre>	add an element to a heap
<u>put</u> (C++ I/O)	write characters
putback (C++ I/O)	return characters to a stream
random_sample (C++ Algorithms)	randomly copy elements from one range to another
random_sample_n (C++ Algorithms)	sample N random elements from a range
<u>random_shuffle</u> (C++ Algorithms)	randomly re-order elements in some range
rbegin (C++ Vectors)	returns a <u>reverse_iterator</u> to the end of the container
rbegin (C++ Strings)	returns a <u>reverse_iterator</u> to the end of the container
rbegin (C++ Double-ended Queues)	returns a <u>reverse_iterator</u> to the end of the container
rbegin (C++ Lists)	returns a <u>reverse_iterator</u> to the end of the container
rbegin (C++ Sets)	returns a <u>reverse_iterator</u> to the end of the container
rbegin (C++ Multisets)	returns a <u>reverse_iterator</u> to the end of the container
rbegin (C++ Maps)	returns a <u>reverse_iterator</u> to the end of the container
	returns a <u>reverse_iterator</u> to the end of

<u>rbegin</u> (C++ Multimaps)	the container
rdstate (C++ I/O)	returns the state flags of the stream
read (C++ I/O)	read data into a buffer
remove (C++ Lists)	removes elements from a list
remove (C++ Algorithms)	remove elements equal to certain value
remove_copy (C++	copy a range of elements omitting
Algorithms)	those that match a certian value
remove_copy_if (C++ Algorithms)	create a copy of a range of elements, omitting any for which a predicate is true
remove_if (C++ Lists)	removes elements conditionally
remove_if (C++ Algorithms)	remove all elements for which a predicate is true
rend (C++ Vectors)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Strings)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Double-ended Queues)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Lists)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Sets)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Multisets)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Maps)	returns a <u>reverse_iterator</u> to the beginning of the container
rend (C++ Multimaps)	returns a <u>reverse_iterator</u> to the beginning of the container
replace (C++ Strings)	replace characters in the string
replace (C++ Algorithms)	replace every occurrence of some value in a range with another value
replace_copy (C++ Algorithms)	copy a range, replacing certain elements with new ones

replace_copy_if (C++ Algorithms)	copy a range of elements, replacing those for which a predicate is true
replace_if (C++ Algorithms)	change the values of elements for which a predicate is true
reserve (C++ Vectors)	sets the minimum capacity of the container
reserve (C++ Strings)	sets the minimum capacity of the container
reset (C++ Bitsets)	sets bits to zero
resize (C++ Vectors)	change the size of the container
resize (C++ Double-ended Queues)	change the size of the container
resize (C++ Lists)	change the size of the container
resize (C++ Strings)	change the size of the container
reverse (C++ Lists)	reverse the list
reverse (C++ Algorithms)	reverse elements in some range
reverse_copy (C++ Algorithms)	create a copy of a range that is reversed
rfind (C++ Strings)	find the last occurrence of a substring
rotate (C++ Algorithms)	move the elements in some range to the left by some amount
<u>rotate_copy</u> (C++ Algorithms)	copy and rotate a range of elements
search (C++ Algorithms)	search for a range of elements
search_n (C++ Algorithms)	search for N consecutive copies of an element in some range
seekg (C++ I/O)	perform random access on an input stream
	perform random access on output
seekp (C++ I/O)	streams
seekp (C++ I/O) set (C++ Bitsets)	1
* ` ,	streams
set (C++ Bitsets)	streams sets bits
<pre>set (C++ Bitsets) set_difference (C++</pre>	streams sets bits computes the difference between two

Algorithms)	between two sets
set union (C++ Algorithms)	computes the union of two sets
setf (C++ I/O)	set format flags
size (C++ Strings)	returns the number of items in the container
size (C++ Vectors)	returns the number of items in the container
<pre>size (C++ Double-ended Queues)</pre>	returns the number of items in the container
size (C++ Lists)	returns the number of items in the container
size (C++ Sets)	returns the number of items in the container
size (C++ Multisets)	returns the number of items in the container
size (C++ Maps)	returns the number of items in the container
size (C++ Multimaps)	returns the number of items in the container
size (C++ Stacks)	returns the number of items in the container
size (C++ Queues)	returns the number of items in the container
size (C++ Priority Queues)	returns the number of items in the container
size (C++ Bitsets)	number of bits that the bitset can hold
sort (C++ Lists)	sorts a list into ascending order
sort (C++ Algorithms)	sort a range into ascending order
sort_heap (C++ Algorithms)	turns a heap into a sorted range of elements
splice (C++ Lists)	merge two lists in constant time
stable_partition (C++ Algorithms)	divide elements into two groups while preserving their relative order
<pre>stable_sort (C++ Algorithms)</pre>	sort a range of elements while preserving order between equal

	elements
substr (C++ Strings)	returns a certain substring
swap (C++ Strings)	swap the contents of this container with another
swap (C++ Vectors)	swap the contents of this container with another
<pre>swap (C++ Double-ended Queues)</pre>	swap the contents of this container with another
swap (C++ Lists)	swap the contents of this container with another
swap (C++ Sets)	swap the contents of this container with another
swap (C++ Multisets)	swap the contents of this container with another
swap (C++ Maps)	swap the contents of this container with another
swap (C++ Multimaps)	swap the contents of this container with another
swap (C++ Algorithms)	swap the values of two objects
<pre>swap_ranges (C++ Algorithms)</pre>	swaps two ranges of elements
<pre>sync_with_stdio (C++ I/O)</pre>	synchronize with standard I/O
tellg (C++ I/O)	read input stream pointers
tellp (C++ I/O)	read output stream pointers
test (C++ Bitsets)	returns the value of a given bit
<pre>to_string (C++ Bitsets)</pre>	string representation of the bitset
to_ulong (C++ Bitsets)	returns an integer representation of the bitset
top (C++ Stacks)	returns the top element of the container
top (C++ Priority Queues)	returns the top element of the container
transform (C++ Algorithms)	applies a function to a range of elements
unique (C++ Lists)	removes consecutive duplicate elements
	remove consecutive duplicate

unique (C++ Algorithms)	elements in a range
unique_copy (C++ Algorithms)	create a copy of some range of elements that contains no consecutive duplicates
unsetf (C++ I/O)	clear <u>io stream format flag</u> s
upper_bound (C++ Sets)	returns an iterator to the first element greater than a certain value
<pre>upper_bound (C++ Multisets)</pre>	returns an iterator to the first element greater than a certain value
upper_bound (C++ Maps)	returns an iterator to the first element greater than a certain value
<pre>upper_bound (C++ Multimaps)</pre>	returns an iterator to the first element greater than a certain value
upper_bound (C++ Algorithms)	searches for the last possible location to insert an element into an ordered range
value_comp (C++ Sets)	returns the function that compares values
<pre>value_comp (C++ Multisets)</pre>	returns the function that compares values
value_comp (C++ Maps)	returns the function that compares values
<pre>value_comp (C++ Multimaps)</pre>	returns the function that compares values
width (C++ I/O)	access and manipulate the minimum field width
write (C++ I/O)	write characters

#### **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;</pre>
```

## **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;</pre>
```

When the above code is run, it displays:

```
bs2 is 10000011
now bs2 is 00110000
```

#### **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:

  cout << "new vector: ";
  vector<int> v2( iter1, iter2 );
  for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
  }
  cout << endl;
}</pre>
```

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 <u>linear time</u> except the first, which runs in constant time.

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

#### **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: " <

cout << "new vector: ";
vector<int> v2( iter1, iter2 );
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;</pre>
```

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 <u>linear time</u> except the first, which runs in <u>constant time</u>.

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

#### **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: " <

cout << "new vector: ";
vector<int> v2( iter1, iter2 );
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
</pre>
```

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 <u>linear time</u> except the first, which runs in <u>constant time</u>.

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

#### **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

# 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 <u>constant time</u>. The default copy constructor runs in <u>linear time</u> 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

## **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["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

cout << "Bart is " << ages["Bart"] << " years old" << endl;</pre>
```

Related topics:
Map Operators

# Multimap constructors & destructors

Syntax:

Multimaps have several constructors:

- The default constructor takes no arguments, creates a new instance of that multimap, and runs in <u>constant time</u>.
- The default copy constructor runs in <u>linear time</u> 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 cout << "Number of employees named 'Bob Thompson': " << m.count("E cout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Employee list: " << endl;
for( multimap<string, int>::iterator iter = m.begin(); iter != m.cout << "Name: " << iter->first << ", ID #" << iter->second << "}</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>List operators</u>
```

## **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 <u>linear time</u>.

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 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
```

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>Set operators</u>
```

### **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 <u>linear time</u>.

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 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
```

# **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 <u>linear time</u>.

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 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
```

# **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 <u>linear time</u>.

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 member in location i in the other multimap.

Comparisons among multimaps are done lexicographically.

Related topics:

**Multimap Constructors** 

## **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 <u>linear time</u>. The [] operator runs in <u>constant time</u>.

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 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;
}</pre>
```

Related topics:

## **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 <u>linear time</u>. The [] operator runs in <u>constant time</u>.

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 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;
}</pre>
```

Related topics:

```
cppreference.com > C++ I/O > I/O Constructors
```

#### I/O Constructors

#### Syntax:

```
#include <fstream>
fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

The fstream, ifstream, and ofstream objects are used to do file I/O. The optional *mode* defines how the file is to be opened, according to the <u>io stream mode flags</u>. The optional *filename* specifies the file to be opened and associated with the stream.

Input and output file streams can be used in a similar manner to C++ predefined I/O streams, cin and cout.

#### Example code:

The following code reads input data and appends the result to an output file.

```
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
  fout << temp + 2 << endl;
fin.close();
fout.close();</pre>
```

#### Related topics:

<u>close</u> <u>open</u>

### 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 <u>linear time</u>.

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 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();
   cout << (*iter).first << " is " << (*iter).second << " years old
}</pre>
```

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** 

<u>cppreference.com</u> > <u>C++ Priority Queues</u> > <u>Priority queue constructors</u>

## **Priority queue constructors**

Syntax:

```
#include <queue>
priority_queue( const Compare& cmp = Compare(), const Container&
priority_queue( input iterator start, input iterator end, const C
```

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*.

### **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();
}</pre>
```

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
```

cppreference.com > C++ Stacks > Stack constructors

# **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.

## **String constructors**

Syntax:

```
#include <string>
  string();
  string( const string& s );
  string( size_type length, const char& ch );
  string( const char* str );
  string( const char* str, size_type length );
  string( const string& str, size_type index, size_type length );
  string( input iterator start, input iterator end );
  ~string();
```

The string constructors create a new string containing:

- nothing; an empty string,
- a copy of the given string *s*,
- *length* copies of *ch*,
- a duplicate of *str* (optionally up to *length* characters long),
- a substring of *str* starting at *index* and *length* characters long
- a string of characters denoted by the *start* and *end* iterators

For example,

```
string str1( 5, 'c' );
string str2( "Now is the time..." );
string str3( str2, 11, 4 );
cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;</pre>
```

displays

```
ccccc
Now is the time...
time
```

The string constructors usually run in linear time, except the empty constructor,

which runs in constant time.

# **String operators**

Syntax:

```
#include <string>
bool operator==(const string& c1, const string& c2);
bool operator!=(const string& c1, const string& c2);
bool operator<(const string& c1, const string& c2);</pre>
bool operator>(const string& c1, const string& c2);
bool operator<=(const string& c1, const string& c2);</pre>
bool operator>=(const string& c1, const string& c2);
string operator+(const string& s1, const string& s2);
string operator+(const char* s, const string& s2 );
string operator+( char c, const string& s2 );
string operator+( const string& s1, const char* s );
string operator+( const string& s1, char c );
ostream& operator<<( ostream& os, const string& s );
istream& operator>>( istream& is, string& s );
string& operator=( const string& s );
string& operator=( const char* s );
string& operator=( char ch );
char& operator[]( size_type index );
```

C++ strings can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one string to another takes <u>linear time</u>.

Two strings are equal if:

```
    Their size is the same, and
    Each member in location i in one string is equal to the the mer
```

Comparisons among strings are done lexicographically.

In addition to the normal container operators, strings can also be concatenated with the + operator and fed to the C++ I/O stream classes with the << and >> operators.

For example, the following code concatenates two strings and displays the result:

```
string s1 = "Now is the time...";
string s2 = "for all good men...";
string s3 = s1 + s2;
cout << "s3 is " << s3 << endl;</pre>
```

Futhermore, strings can be assigned values that are other strings, character arrays, or even single characters. The following code is perfectly valid:

```
char ch = 'N';
string s;
s = ch;
```

Individual characters of a string can be examined with the [] operator, which runs in <u>constant time</u>.

Related topics:

c\_str compare
data

### accumulate

#### Syntax:

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

The accumulate() 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, itera
iterator adjacent_difference( iterator start, iterator end, itera
```

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}</pre>
```

Related topics:

find find\_end find\_first\_of find\_if unique unique\_copy cppreference.com > C++ Bitsets > any

### 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** 

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>append</u>
```

## append

#### Syntax:

```
#include <string>
string& append( const string& str );
string& append( const char* str );
string& append( const string& str, size_type index, size_type len
string& append( const char* str, size_type num );
string& append( size_type num, char ch );
string& append( input iterator start, input iterator end );
```

The append() function either:

- appends *str* on to the end of the current string,
- appends a substring of *str* starting at *index* that is *len* characters long on to the end of the current string,
- appends *num* characters of *str* on to the end of the current string,
- appends *num* repititions of *ch* on to the end of the current string,
- or appends the sequence denoted by *start* and *end* on to the end of the current string.

For example, the following code uses append() to add 10 copies of the '!' character to a string:

```
string str = "Hello World";
str.append( 10, '!' );
cout << str << endl;</pre>
```

That code displays:

```
Hello World!!!!!!!!!
```

In the next example, append() is used to concatenate a substring of one string onto another string:

```
string str1 = "Eventually I stopped caring...";
string str2 = "but that was the '80s so nobody noticed.";
```

```
str1.append( str2, 25, 15 );
cout << "str1 is " << str1 << endl;
```

When run, the above code displays:

```
str1 is Eventually I stopped caring...nobody noticed.
```

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>assign</u>
```

## 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;</pre>
```

The above code displays the following output:

```
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] << " ";
}</pre>
```

```
cout << endl;</pre>
```

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

## 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;</pre>
```

The above code displays the following output:

```
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] << " ";
}</pre>
```

```
cout << endl;</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>assign</u>
```

## 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;</pre>
```

The above code displays the following output:

```
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] << " ";
}</pre>
```

```
cout << endl;</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>assign</u>
```

# assign

Syntax:

```
#include <string>
void assign( size_type num, const char& val );
void assign( input iterator start, input iterator end );
string& assign( const string& str );
string& assign( const char* str );
string& assign( const char* str, size_type num );
string& assign( const string& str, size_type index, size_type len string& assign( size_type num, const char& ch );
```

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

In addition to the normal assign functionality that all C++ containers have, strings possess an assign() function that also allows them to:

- assign *str* to the current string,
- assign the first *num* characters of *str* to the current string,
- assign a substring of *str* starting at *index* that is *len* characters long to the current string,

For example, the following code:

```
string str1, str2 = "War and Peace";
str1.assign( str2, 4, 3 );
cout << str1 << endl;</pre>
```

displays

```
and
```

This function will destroy the previous contents of the string.

Related topics:

```
(C++ Lists) assign
```

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>at</u>
```

#### 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;
}</pre>
```

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;
}</pre>
```

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** 

```
cppreference.com > C++ Double-ended Queues > at
```

#### 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;
}</pre>
```

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;
}</pre>
```

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) <u>Multimap operators</u>

<u>Deque operators</u>

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>at</u>
```

#### at

Syntax:

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

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

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;
}</pre>
```

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;
}</pre>
```

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) <u>Multimap operators</u>
(C++ Double-ended Queues) <u>Container operators</u>
```

### auto\_ptr

Syntax:

```
#include <memory>
auto_ptr<class TYPE> name
```

The auto\_ptr class allows the programmer to create pointers that point to other objects. When auto\_ptr pointers are destroyed, the objects to which they point are also destroyed.

The auto\_ptr class supports normal pointer operations like =, \*, and ->, as well as two functions TYPE\* get() and TYPE\* release(). The get() function returns a pointer to the object that the auto\_ptr points to. The release() function acts similarly to the get() function, but also relieves the auto\_ptr of its memory destruction duties. When an auto\_ptr that has been released goes out of scope, it will not call the destructor of the object that it points to.

**Warning:** It is generally a **bad idea** to put auto\_ptr objects inside C++ STL containers. C++ containers can do funny things with the data inside them, including frequent reallocation (when being copied, for instance). Since calling the destructor of an auto\_ptr object will free up the memory associated with that object, any C++ container reallocation will cause any auto\_ptr objects to become invalid.

#### Example code:

```
#include <memory>
using namespace std;

class MyClass {
public:
   MyClass() {} // nothing
   ~MyClass() {} // nothing
   void myFunc() {} // nothing
};

int main() {
   auto_ptr<MyClass> ptr1(new MyClass), ptr2;
```

```
ptr2 = ptr1;
ptr2->myFunc();

MyClass* ptr = ptr2.get();

ptr->myFunc();

return 0;
}
```

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>back</u>
```

### 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;</pre>
```

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

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>back</u>

### 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;</pre>
```

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

```
cppreference.com > C++ Lists > back
```

### 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;</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Queues</u> > <u>back</u>
```

### 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;</pre>
```

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
```

<u>cppreference.com</u> > <u>C++ I/O</u> > <u>bad</u>

### bad

Syntax:

```
#include <fstream>
bool bad();
```

The bad() function returns true if a fatal error with the current stream has occurred, false otherwise.

Related topics:

eof fail good rdstate

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>begin</u>
```

*Syntax:* 

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

The function begin() returns an iterator to the first element of the string. 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
   cout << *theIterator;
}</pre>
```

Related topics:

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>begin</u>
```

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;
}</pre>
```

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

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>begin</u>

# 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
   cout << *theIterator;
}</pre>
```

Related topics:

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>begin</u>
```

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 <u>constant time</u>.

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
   cout << *theIterator;
}</pre>
```

Related topics:

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>begin</u>
```

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
   cout << *theIterator;
}</pre>
```

Related topics:

```
<u>cppreference.com</u> > <u>C++ Multisets</u> > <u>begin</u>
```

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 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
   cout << *theIterator;
}</pre>
```

Related topics:

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>begin</u>
```

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
```

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
```

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>begin</u>
```

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
   cout << *theIterator;
}</pre>
```

Related topics:

# 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
```

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;
   }
}</pre>
```

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 is\_sorted lower\_bound partial\_sort partial\_sort copy sort stable\_sort upper\_bound

cppreference.com > C++ Strings > c\_str

#### c\_str

Syntax:

```
#include <string>
const char* c_str();
```

The function c\_str() returns a const pointer to a regular C string, identical to the current string. The returned string is null-terminated.

Note that since the returned pointer is of type <u>const</u>, the character data that c\_str() returns **cannot be modified**. Furthermore, you do not need to call <u>free()</u> or <u>delete</u> on this pointer.

Related topics:

**String operators data** 

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>capacity</u>
```

# 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;</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>capacity</u>
```

# capacity

Syntax:

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

The capacity() function returns the number of elements that the string 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;</pre>
```

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

<u>cppreference.com</u> > <u>C++ I/O</u> > <u>clear</u>

#### clear

Syntax:

```
#include <fstream>
void clear( iostate flags = ios::goodbit );
```

The function clear() does two things:

- it clears all io stream state flags associated with the current stream,
- and sets the flags denoted by *flags*

The *flags* argument defaults to ios::goodbit, which means that by default, all flags will be cleared and ios::goodbit will be set.

Example code:

For example, the following code uses the clear() function to reset the flags of an output file stream, after an attempt is made to read from that output stream:

```
fstream outputFile( "output.txt", fstream::out );

// try to read from the output stream; this shouldn't work
int val;
outputFile >> val;
if( outputFile.fail() ) {
  cout << "Error reading from the output stream" << endl;
  // reset the flags associated with the stream
  outputFile.clear();
}

for( int i = 0; i < 10; i++ ) {
  outputFile << i << " ";
}
outputFile << endl;</pre>
```

Related topics:

<u>eof fail</u> good

#### <u>rdstate</u>

cppreference.com > C++ Strings > clear

#### clear

Syntax:

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

The function clear() deletes all of the elements in the string. clear() runs in <a href="lineartime">linear</a> time.

Related topics: (C++ Lists) erase

<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>clear</u>

#### clear

Syntax:

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

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

clear() runs in <u>linear time</u>.

Related topics:

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>clear</u>

#### clear

Syntax:

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

The function clear() deletes all of the elements in the dequeue. clear() runs in <a href="linear time">linear time</a>.

Related topics:

cppreference.com > C++ Lists > clear

# clear

Syntax:

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

The function clear() deletes all of the elements in the list. clear() runs in <a href="lineartime">linear</a>() runs in <a href="lineartime">lineartime</a>() runs in <a href="lineartime">lineartim

Related topics:

cppreference.com > C++ Sets > clear

#### clear

Syntax:

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

The function clear() deletes all of the elements in the set. clear() runs in <a href="lineartime">linear</a> time.

Related topics: (C++ Lists) erase

cppreference.com > C++ Multisets > clear

#### clear

Syntax:

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

The function clear() deletes all of the elements in the multiset. clear() runs in <a href="linear time">linear time</a>.

Related topics: (C++ Lists) erase

<u>cppreference.com</u> > <u>C++ Maps</u> > <u>clear</u>

# clear

Syntax:

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

The function clear() deletes all of the elements in the map. clear() runs in <a href="lineartime">linear</a> time.

Related topics:

<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>clear</u>

#### clear

Syntax:

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

The function clear() deletes all of the elements in the multimap. clear() runs in <a href="mailto:lineartime">lineartime</a>.

Related topics: (C++ Lists) erase

 $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{close}}$ 

#### close

Syntax:

```
#include <fstream>
void close();
```

The close() function closes the associated file stream.

Related topics:

**I/O Constructors open** 

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>compare</u>
```

#### compare

#### Syntax:

```
#include <string>
int compare( const string& str );
int compare( const char* str );
int compare( size_type index, size_type length, const string& str
int compare( size_type index, size_type length, const string& str
size_type length2 );
int compare( size_type index, size_type length, const char* str,
```

The compare() function either compares *str* to the current string in a variety of ways, returning

Return Value	Case	
less than zero	this < str	
zero	this == str	
greater than zero	this > str	

The various functions either:

- compare *str* to the current string,
- compare *str* to a substring of the current string, starting at *index* for *length* characters,
- compare a substring of *str* to a substring of the current string, where *index2* and *length2* refer to *str* and *index* and *length* refer to the current string,
- or compare a substring of *str* to a substring of the current string, where the substring of *str* begins at zero and is *length2* characters long, and the substring of the current string begins at *index* and is *length* characters long.

For example, the following code uses compare() to compare four strings with eachother:

```
string names[] = {"Homer", "Marge", "3-eyed fish", "inanimate carbo
```

```
for( int i = 0; i < 4; i++ ) {
  for( int j = 0; j < 4; j++ ) {
    cout << names[i].compare( names[j] ) << " ";
  }
  cout << endl;
}</pre>
```

Data from the above code was used to generate this table, which shows how the various strings compare to eachother:

	Homer	Marge	3-eyed fish	inanimate carbon rod
"Homer".compare( x )	0	-1	1	-1
"Marge".compare( x )	1	0	1	-1
"3-eyed fish".compare(x)	-1	-1	0	-1
"inanimate carbon rod".compare( x )	1	1	1	0

Related topics:

**String operators** 

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>copy</u>
```

#### copy

Syntax:

```
#include <string>
size_type copy( char* str, size_type num, size_type index = 0 );
```

The copy() function copies *num* characters of the current string (starting at *index* if it's specified, 0 otherwise) into *str*.

The return value of copy() is the number of characters copied.

For example, the following code uses copy() to extract a substring of a string into an array of characters:

```
char buf[30];
memset( buf, '\0', 30 );
string str = "Trying is the first step towards failure.";
str.copy( buf, 24 );
cout << buf << endl;</pre>
```

When run, this code displays:

```
Trying is the first step
```

Note that before calling copy(), we first call (Standard C String and Character) memset() to fill the destination array with copies of the **NULL** character. This step is included to make sure that the resulting array of characters is **NULL**-terminated.

Related topics:

substr

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>copy</u>
```

#### 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 <u>linear</u> 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;</pre>
```

Related topics:

copy\_backward copy\_n
generate

remove\_copy swap transform

# copy\_backward

*Syntax:* 

```
#include <algorithm>
iterator copy_backward( iterator start, iterator end, iterator de
```

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;</pre>
```

The above code produces the following output:

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

Related topics:

copy copy\_n
swap

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>copy\_n</u>

#### 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 <u>linear time</u>.

Related topics:
<a href="mailto:copy">copy</a> backward
<a href="mailto:swap">swap</a>

```
cppreference.com > C++ Sets > count
```

#### 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 <u>logarithmic time</u>.

cppreference.com > C++ Multisets > count

#### 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 <u>logarithmic time</u>.

<u>cppreference.com</u> > <u>C++ Maps</u> > <u>count</u>

#### 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 <u>logarithmic time</u>.

cppreference.com > C++ Multimaps > count

#### 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 <u>logarithmic time</u>.

```
cppreference.com > C++ Bitsets > count
```

#### 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

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>count</u>
```

#### 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_</pre>
```

The above code displays the following output:

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

#### Related topics:

accumulate adjacent\_difference count\_if inner\_product partial\_sum

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>count_if</u>
```

## 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:

When run, the above code displays the following output:

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

Related topics:

count

cppreference.com > C++ Strings > data

## data

Syntax:

```
#include <string>
const char *data();
```

The function data() returns a pointer to the first character in the current string.

Related topics:

**String operators c\_str** 

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>empty</u>
```

Syntax:

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

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

For example:

```
string s1;
string s2("");
string s3("This is a string");
cout.setf(ios::boolalpha);
cout << s1.empty() << endl;
cout << s2.empty() << endl;
cout << s3.empty() << endl;</pre>
```

When run, this code produces the following output:

```
true
true
false
```

Related topics:

<u>size</u>

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>empty</u>
```

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();
}</pre>
```

Related topics:

size

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>empty</u>

### 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();
}</pre>
```

Related topics:

size

```
cppreference.com > C++ Lists > 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();
}</pre>
```

Related topics:

<u>size</u>

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>empty</u>
```

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();
}</pre>
```

Related topics:

<u>size</u>

```
<u>cppreference.com</u> > <u>C++ Multisets</u> > <u>empty</u>
```

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();
}</pre>
```

Related topics:

size

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>empty</u>
```

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());
}</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>empty</u>
```

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();
}</pre>
```

Related topics:

size

```
cppreference.com > C++ Stacks > 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();
}</pre>
```

Related topics:

<u>size</u>

```
<u>cppreference.com</u> > <u>C++ Queues</u> > <u>empty</u>
```

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();
}</pre>
```

Related topics:

size

```
<u>cppreference.com</u> > <u>C++ Priority Queues</u> > <u>empty</u>
```

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();
}</pre>
```

Related topics:

size

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>end</u>
```

Syntax:

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

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

Note that before you can access the last element of the string 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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>end</u>
```

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;</pre>
```

The next example shows how <u>begin()</u> 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 <a href="begin">begin</a>(). 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.

cppreference.com > C++ Double-ended Queues > end

#### 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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>end</u>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>end</u>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
<u>cppreference.com</u> > <u>C++ Multisets</u> > <u>end</u>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>end</u>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>end</u>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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.

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{eof}}
```

#### eof

Syntax:

```
#include <fstream>
bool eof();
```

The function eof() returns true if the end of the associated input file has been reached, false otherwise.

For example, the following code reads data from an input stream *in* and writes it to an output stream *out*, using eof() at the end to check if an error occurred:

```
char buf[BUFSIZE];
do {
  in.read( buf, BUFSIZE );
  std::streamsize n = in.gcount();
  out.write( buf, n );
} while( in.good() );
if( in.bad() || !in.eof() ) {
  // fatal error occurred
}
in.close();
```

Related topics:

bad clear fail good rdstate

## equal

Syntax:

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

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;
}</pre>
```

Related topics:

find\_if lexicographical\_compare mismatch search cppreference.com > C++ Sets > equal\_range

# 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*.

cppreference.com > C++ Multisets > equal\_range

# 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*.

cppreference.com > C++ Maps > equal\_range

# 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*.

# 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,pai</pre>
multimap<string,pair<int,int> >::iterator i;
ii = input_config.equal_range("key");
                                               // keyboard key-bind
// 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
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);
}
```

```
cppreference.com > C++ Algorithms > equal range
```

## equal\_range

Syntax:

```
#include <algorithm>
pair<iterator,iterator> equal_range( iterator first, iterator las
pair<iterator,iterator> equal_range( iterator first, iterator las
```

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 <a href="lower\_bound">lower\_bound</a>() and `upper\_bound1`() functions, since the first of the pair of iterators that it returns is what <a href="lower\_bound">lower\_bound</a>() 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:

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:
<a href="mailto:binary\_search">binary\_search</a> lower\_bound
<a href="mailto:upper\_bound">upper\_bound</a>

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>erase</u>
```

#### *Syntax:*

```
#include <string>
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
string& erase( size_type index = 0, size_type num = npos );
```

The erase() function either:

- removes the character pointed to by *loc*, returning an iterator to the next character,
- removes the characters between *start* and *end* (including the one at *start* but not the one at *end*), returning an iterator to the character after the last character removed,
- or removes *num* characters from the current string, starting at *index*, and returns \*this.

The parameters *index* and *num* have default values, which means that erase() can be called with just *index* to erase all characters after *index* or with no arguments to erase all characters.

### For example:

```
string s("So, you like donuts, eh? Well, have all the donuts in to cout << "The original string is '" << s << "'" << endl;

s.erase( 50, 14 );
cout << "Now the string is '" << s << "'" << endl;
s.erase( 24 );
cout << "Now the string is '" << s << "'" << endl;
s.erase();
cout << "Now the string is '" << s << "'" << endl;</pre>
```

### will display

```
The original string is 'So, you like donuts, eh? Well, have all to Now the string is 'So, you like donuts, eh? Well, have all the do
```

```
Now the string is 'So, you like donuts, eh?' Now the string is ''
```

erase() runs in <u>linear time</u>.

Related topics:

<u>insert</u>

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>erase</u>
```

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 <u>constant time</u> for lists and <u>linear time</u> for vectors, dequeues, and strings. The multiple-element version of erase always takes <u>linear time</u>.

For example:

```
// Create a vector, load it with the first ten characters of the a
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 != alphaVecout << *tempIterator;
    }
    cout << endl;
}</pre>
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
```

```
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
```

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 {
  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 elemen
  // 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;</pre>
```

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
```

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 <u>constant time</u> for lists and <u>linear time</u> for vectors, dequeues, and strings. The multiple-element version of erase always takes <u>linear time</u>.

For example:

```
// Create a vector, load it with the first ten characters of the al
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 != alphaVec
        cout << *tempIterator;
    }
    cout << endl;
}</pre>
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
```

```
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
```

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 al
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 element
// 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;</pre>
```

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

### Related topics:

```
clear insert
pop_back
pop_front
(C++ Lists) remove
(C++ Lists) remove if
```

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>erase</u>
```

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 <u>constant time</u> for lists and <u>linear time</u> for vectors, dequeues, and strings. The multiple-element version of erase always takes <u>linear time</u>.

For example:

```
// Create a vector, load it with the first ten characters of the al
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 != alphaVec
        cout << *tempIterator;
    }
    cout << endl;
}</pre>
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
```

```
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
```

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 alvector<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 element
// 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;</pre>
```

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

### Related topics:

```
clear insert
pop_back
pop_front
remove
remove if
```

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>erase</u>
```

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*.

```
cppreference.com > C++ Multisets > 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*.

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>erase</u>
```

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 <u>while</u> 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());
}</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>erase</u>
```

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*.

### $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{fail}}$

# fail

Syntax:

```
#include <fstream>
bool fail();
```

The fail() function returns true if an error has occurred with the current stream, false otherwise.

Related topics:

bad clear eof good rdstate

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{fill}}
```

### fill

Syntax:

```
#include <fstream>
char fill();
char fill( char ch );
```

The function fill() either returns the current fill character, or sets the current fill character to ch.

The fill character is defined as the character that is used for padding when a number is smaller than the specified width(). The default fill character is the space character.

Related topics: precision width

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>fill</u>
```

### 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;</pre>
```

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
```

Related topics:

fill\_n generate transform

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>fill_n</u>
```

# 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;</pre>
```

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 5 6 7 8 9
```

Related topics:

### 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 << en
}

else {
   cout << "Found a matching element: " << *result << endl;
}</pre>
```

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 << end } else {
  cout << "Found a matching number: " << *result << endl;
}</pre>
```

### Related topics:

adjacent find find end find first of find\_if mismatch search cppreference.com > C++ Sets > find

# 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.

cppreference.com > C++ Multisets > find

# 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.

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>find</u>
```

### 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 << }
}</pre>
```

When run with this input:

```
my spoon is too big. my spoon is TOO big! my SPOON is TOO big! I
```

...the above code produces this output:

```
You typed 'spoon' 2 time(s)
```

cppreference.com > C++ Multimaps > find

# 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.

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>find</u>
```

### find

#### Syntax:

```
#include <string>
size_type find( const string& str, size_type index );
size_type find( const char* str, size_type index );
size_type find( const char* str, size_type index, size_type lengt
size_type find( char ch, size_type index );
```

#### The function find() either:

- returns the first occurrence of *str* within the current string, starting at *index*, string::npos if nothing is found,
- if the *length* parameter is given, then find() returns the first occurrence of the first *length* characters of *str* within the current string, starting at *index*, string::npos if nothing is found,
- or returns the index of the first occurrence *ch* within the current string, starting at *index*, string::npos if nothing is found.

#### For example:

```
string str1( "Alpha Beta Gamma Delta" );
string::size_type loc = str1.find( "Omega", 0 );
if( loc != string::npos ) {
  cout << "Found Omega at " << loc << endl;
} else {
  cout << "Didn't find Omega" << endl;
}</pre>
```

### Related topics:

```
find_first_not_of find_first_of
find_last_not_of
find_last_of
rfind
```

## find\_end

Syntax:

```
#include <algorithm>
iterator find_end( iterator start, iterator end, iterator seq_sta
iterator find_end( iterator start, iterator end, iterator seq_sta
```

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
if( *result == nums[end] ) {
 cout << "Did not find any subsequence matching { 1, 2, 3 }" << er</pre>
} else {
  cout << "The last matching subsequence is at: " << *result << en</pre>
int target2[] = \{ 3, 2, 3 \};
result = find_end( nums + start, nums + end, target2 + 0, target2
if( *result == nums[end] ) {
  cout << "Did not find any subsequence matching { 3, 2, 3 }" << er
} else {
  cout << "The last matching subsequence is at: " << *result << en</pre>
}
```

Related topics:
adjacent\_find find
find\_first\_of
find\_if
search\_n

# find\_first\_not\_of

Syntax:

```
#include <string>
size_type find_first_not_of( const string& str, size_type index =
size_type find_first_not_of( const char* str, size_type index = 0
size_type find_first_not_of( const char* str, size_type index, si
size_type find_first_not_of( char ch, size_type index = 0 );
```

The find first not of() function either:

- returns the index of the first character within the current string that does not match any character in *str*, beginning the search at *index*, string::npos if nothing is found,
- searches the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning string::npos,
- or returns the index of the first occurrence of a character that does not match *ch* in the current string, starting the search at *index*, string::npos if nothing is found.

For example, the following code searches a string of text for the first character that is not a lower-case character, space, comma, or hypen:

```
string lower_case = "abcdefghijklmnopqrstuvwxyz ,-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-0
cout << "first non-lower-case letter in str at: " << str.find_fire
```

When run, find\_first\_not\_of() finds the first upper-case letter in *str* at index 29 and displays this output:

```
first non-lower-case letter in str at: 29
```

Related topics:

find find first not of find first of

find\_last\_not\_of find\_last\_of rfind

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>find_first_of</u>
```

# find\_first\_of

#### Syntax:

```
#include <string>
size_type find_first_of( const string &str, size_type index = 0 )
size_type find_first_of( const char* str, size_type index = 0 );
size_type find_first_of( const char* str, size_type index, size_t
size_type find_first_of( char ch, size_type index = 0 );
```

The find\_first\_of() function either:

- returns the index of the first character within the current string that matches any character in *str*, beginning the search at *index*, string::npos if nothing is found,
- searches the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or string::npos if no characters match,
- or returns the index of the first occurrence of *ch* in the current string, starting the search at *index*, string::npos if nothing is found.

#### Related topics:

find find first not of find last not of find last of rfind

# find\_first\_of

Syntax:

```
#include <algorithm>
iterator find_first_of( iterator start, iterator end, iterator fi
iterator find_first_of( iterator start, iterator end, iterator fi
```

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 + 0;
if( *result == nums[end] ) {
   cout << "Did not find any of { 9, 4, 7 }" << endl;
} else {
   cout << "Found a matching target: " << *result << endl;
}</pre>
```

Related topics:

adjacent\_find 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>(),
if( *result == nums[end] ) {
  cout << "Did not find any number greater than zero" << endl;
} else {
  cout << "Found a positive non-zero number: " << *result << endl;
}</pre>
```

### Related topics:

adjacent\_find equal find find\_end find\_first\_of search\_n

## find\_last\_not\_of

Syntax:

```
#include <string>
size_type find_last_not_of( const string& str, size_type index =
size_type find_last_not_of( const char* str, size_type index = np
size_type find_last_not_of( const char* str, size_type index, siz
size_type find_last_not_of( char ch, size_type index = npos );
```

The find\_last\_not\_of() function either:

- returns the index of the last character within the current string that does not match any character in *str*, doing a reverse search from *index*, string::npos if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning string::npos,
- or returns the index of the last occurrence of a character that does not match *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, the following code searches for the last non-lower-case character in a mixed string of characters:

```
string lower_case = "abcdefghijklmnopqrstuvwxyz";
string str = "abcdefgABCDEFGhijklmnop";
cout << "last non-lower-case letter in str at: " << str.find_last_</pre>
```

This code displays the following output:

```
last non-lower-case letter in str at: 13
```

Related topics:

find find first not of find first of find last of

### <u>rfind</u>

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>find_last_of</u>
```

## find\_last\_of

#### Syntax:

```
#include <string>
    size_type find_last_of( const string& str, size_type index = npos
    size_type find_last_of( const char* str, size_type index = npos )
    size_type find_last_of( const char* str, size_type index, size_ty
    size_type find_last_of( char ch, size_type index = npos );
```

The find\_last\_of() function either:

- does a reverse search from *index*, returning the index of the first character within the current string that matches any character in *str*, or string::npos if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or string::npos if no characters match,
- or does a reverse search from *index*, returning the index of the first occurrence of *ch* in the current string, string::npos if nothing is found.

### Related topics:

find find\_first\_not\_of find\_first\_of find\_last\_not\_of rfind

```
cppreference.com > C++ I/O > flags
```

# flags

Syntax:

```
#include <fstream>
fmtflags flags();
fmtflags flags( fmtflags f );
```

The flags() function either returns the <u>io stream format flag</u>s for the current stream, or sets the flags for the current stream to be f.

Related topics:

setf unsetf

```
cppreference.com > C++ Bitsets > flip
```

# 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.

 $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{flush}}$ 

## flush

*Syntax:* 

```
#include <fstream>
ostream& flush();
```

The flush() function causes the buffer for the current output stream to be actually written out to the attached device.

This function is useful for printing out debugging information, because sometimes programs abort before they have a chance to write their output buffers to the screen. Judicious use of flush() can ensure that all of your debugging statements actually get printed.

Related topics:

put write

# for\_each

Syntax:

```
#include <algorithm>
UnaryFunction for_each( iterator start, iterator end, UnaryFuncti
```

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 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;</pre>
```

The above code displays the following output:

```
Before, nums[] is: 3 4 2 9 15 267
```

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>front</u>
```

### 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 <u>sort() algorithm</u> 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.from</pre>
```

When provided with this input:

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

...the above code displays:

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

Related topics:

```
back (C++ Lists) pop_front
(C++ Lists) push_front
```

cppreference.com > C++ Double-ended Queues > front

## 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:
<a href="mailto:back">back</a> pop front
<a href="push\_front">push\_front</a>

```
cppreference.com > C++ Lists > front
```

## 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:
<a href="mailto:back">back</a> pop front
<a href="push\_front">push\_front</a>

```
cppreference.com > C++ Queues > front
```

## 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 <u>constant time</u>.

```
Related topics:
<a href="mailto:back">back</a> (C++ Lists) <a href="mailto:pop_front">pop_front</a>
(C++ Lists) <a href="mailto:push_front">push_front</a>
```

 $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{gcount}}$ 

# gcount

Syntax:

```
#include <fstream>
streamsize gcount();
```

The function gcount() is used with input streams, and returns the number of characters read by the last input operation.

Related topics:

get getline read <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>generate</u>

# 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 fill
generate\_n
transform

cppreference.com > C++ Algorithms > generate\_n

## 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** 

```
<u>cppreference.com</u> > <u>C++ I/O</u> > <u>get</u>
```

## get

Syntax:

```
#include <fstream>
int get();
istream& get( char& ch );
istream& get( char* buffer, streamsize num );
istream& get( char* buffer, streamsize num, char delim );
istream& get( streambuf& buffer );
istream& get( streambuf& buffer, char delim );
```

The get() function is used with input streams, and either:

- reads a character and returns that value,
- reads a character and stores it as *ch*,
- reads characters into *buffer* until *num* 1 characters have been read, or **EOF** or newline encountered,
- reads characters into *buffer* until *num* 1 characters have been read, or **EOF** or the *delim* character encountered (*delim* is not read until next time),
- reads characters into buffer until a newline or **EOF** is encountered,
- or reads characters into buffer until a newline, **EOF**, or *delim* character is encountered (again, *delim* isn't read until the next get()).

For example, the following code displays the contents of a file called temp.txt, character by character:

```
char ch;
ifstream fin( "temp.txt" );
while( fin.get(ch) )
  cout << ch;
fin.close();</pre>
```

### Related topics:

```
gcount getline
(C++ Strings) getline
ignore
peek
```

<u>put</u> <u>read</u>

```
<u>cppreference.com</u> > <u>C++ I/O</u> > <u>getline</u>
```

# getline

Syntax:

```
#include <fstream>
istream& getline( char* buffer, streamsize num );
istream& getline( char* buffer, streamsize num, char delim );
```

The getline() function is used with input streams, and reads characters into *buffer* until either:

- *num* 1 characters have been read,
- a newline is encountered,
- an **EOF** is encountered,
- or, optionally, until the character *delim* is read. The *delim* character is not put into buffer.

For example, the following code uses the getline function to display the first 100 characters from each line of a text file:

```
ifstream fin("tmp.dat");
int MAX_LENGTH = 100;
char line[MAX_LENGTH];
while( fin.getline(line, MAX_LENGTH) ) {
  cout << "read line: " << line << endl;
}</pre>
```

If you'd like to read lines from a file into <u>strings</u> instead of character arrays, consider using the <u>string getline</u> function.

Those using a Microsoft compiler may find that getline() reads an extra character, and should consult the documentation on the Microsoft getline bug.

```
Related topics:
```

```
gcount get
(C++ Strings) getline
```

### ignore read

# getline

Syntax:

```
#include <string>
istream& getline( istream& is, string& s, char delimiter = '\n' )
```

The C++ string class defines the global function getline() to read strings from an I/O stream. The getline() function, which is not part of the string class, reads a line from *is* and stores it into *s*. If a character *delimiter* is specified, then getline() will use *delimiter* to decide when to stop reading data.

For example, the following code reads a line of text from **stdin** and displays it to **stdout**:

```
string s;
getline( cin, s );
cout << "You entered " << s << endl;</pre>
```

After getting a line of data in a string, you may find that <u>string streams</u> are useful in extracting data from that string. For example, the following code reads numbers from standard input, ignoring any "commented" lines that begin with double slashes:

```
// expects either space-delimited numbers or lines that start with
// two forward slashes (//)
string s;
while( getline(cin,s) ) {
   if( s.size() >= 2 && s[0] == '/' && s[1] == '/' ) {
     cout << " ignoring comment: " << s << endl;
} else {
   istringstream ss(s);
   double d;
   while( ss >> d ) {
     cout << " got a number: " << d << endl;
   }
}
}</pre>
```

When run with a user supplying input, the above code might produce this output:

```
// test
  ignoring comment: // test
23.3 -1 3.14159
  got a number: -1
  got a number: 3.14159
// next batch
  ignoring comment: // next batch
1 2 3 4 5
  got a number: 1
  got a number: 2
  got a number: 3
  got a number: 4
  got a number: 5

50
  got a number: 50
```

### Related topics:

(C++ I/O) <u>get</u> (C++ I/O) <u>getline</u> <u>string streams</u>

### $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{good}}$

# good

Syntax:

```
#include <fstream>
bool good();
```

The function good() returns true if no errors have occurred with the current stream, false otherwise.

Related topics:

bad clear eof fail rdstate

```
<u>cppreference.com</u> > <u>C++ I/O</u> > <u>ignore</u>
```

# ignore

Syntax:

```
#include <fstream>
istream& ignore( streamsize num=1, int delim=EOF );
```

The ignore() function is used with input streams. It reads and throws away characters until *num* characters have been read (where *num* defaults to 1) or until the character *delim* is read (where *delim* defaults to **EOF**).

The ignore() function can sometimes be useful when using the getline() function together with the >> operator. For example, if you read some input that is followed by a newline using the >> operator, the newline will remain in the input as the next thing to be read. Since getline() will by default stop reading input when it reaches a newline, a subsequent call to getline() will return an empty string. In this case, the ignore() function could be called before getline() to "throw away" the newline.

Related topics:

get getline

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>includes</u>

## includes

#### Syntax:

```
#include <algorithm>
bool includes( iterator start1, iterator end1, iterator start2, i
bool includes( iterator start1, iterator end1, iterator start2, i
```

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 cppreference.com > C++ Algorithms > inner\_product

# inner\_product

Syntax:

```
#include <numeric>
    TYPE inner_product( iterator start1, iterator end1, iterator star
    TYPE inner_product( iterator start1, iterator end1, iterator star
```

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 <u>linear time</u>.

Related topics:

accumulate adjacent\_difference

count
partial\_sum

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>inplace\_merge</u>

# inplace\_merge

Syntax:

```
#include <algorithm>
inline void inplace_merge( iterator start, iterator middle, itera
inline void inplace_merge( iterator start, iterator middle, itera
```

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

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>insert</u>
```

#### Syntax:

```
#include <string>
iterator insert( iterator i, const char& ch );
string& insert( size_type index, const string& str );
string& insert( size_type index, const char* str );
string& insert( size_type index1, const string& str, size_type in
string& insert( size_type index, const char* str, size_type num )
string& insert( size_type index, size_type num, char ch );
void insert( iterator i, size_type num, const char& ch );
void insert( iterator i, iterator start, iterator end );
```

The very multi-purpose insert() function either:

- inserts *ch* before the character denoted by *i*,
- inserts *str* into the current string, at location *index*,
- inserts a substring of *str* (starting at *index2* and *num* characters long) into the current string, at location *index1*,
- inserts *num* characters of *str* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, before the character denoted by *i*,
- or inserts the characters denoted by *start* and *end* into the current string, before the character specified by *i*.

### Related topics:

erase replace

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>insert</u>
```

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 e
```

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 <u>linear time</u>. 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 <u>List</u> or a <u>Deque</u>).

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 algorithm 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
    cout << *theIterator;
}</pre>
```

This code would display:

```
CCCCABCDEFGHIJ
```

Here 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: ";</pre>
for( int i = 0; i < v2.size(); i++ ) {</pre>
  cout << v2[i] << " ";
cout << endl;</pre>
v2.insert( v2.end(), v1.begin(), v1.end() );
cout << "After, v2 is: ";</pre>
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

cppreference.com > C++ Double-ended Queues > insert

### 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, i
```

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 algorithm vector 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 cout << *theIterator; }</pre>
```

This code would display:

```
CCCCABCDEFGHIJ
```

### Related topics:

```
assign erase
(C++ Lists) merge
push back
```

push\_front
(C++ Lists) splice

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>insert</u>
```

#### 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, i
```

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 alg
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
    cout << *theIterator;
}</pre>
```

This code would display:

```
CCCCABCDEFGHIJ
```

### Related topics:

```
assign erase
merge
push_back
```

push\_front
splice

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>insert</u>
```

#### 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* before 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

```
<u>cppreference.com</u> > <u>C++ Multisets</u> > <u>insert</u>
```

#### *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*.

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>insert</u>
```

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 cout << "Number of employees named 'Bob Thompson': " << m.count("Ecout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Employee list: " << endl;
for( multimap<string, int>::iterator iter = m.begin(); iter != m.ecout << " Name: " << iter->first << ", ID #" << iter->second << }</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>insert</u>
```

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
}
```

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 <u>string</u>, the map is sorted alphabetically by key.

Related topics:
<a href="Map operators">Map operators</a>

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>is_heap</u>
```

# is\_heap

Syntax:

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

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 <a href="linear time">linear time</a>.

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
```

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 <u>linear time</u>.

Related topics:
<a href="mailto:binary\_search">binary\_search</a> partial sort copy

<u>sort</u>

stable\_sort

cppreference.com > C++ Algorithms > iter\_swap

# 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

cppreference.com > C++ Sets > key\_comp

# 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:

cppreference.com > C++ Multisets > key\_comp

# 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:

cppreference.com > C++ Maps > key\_comp

# 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:

cppreference.com > C++ Multimaps > key\_comp

# 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:

cppreference.com > C++ Strings > length

# length

Syntax:

```
#include <string>
size_type length() const;
```

The length() function returns the number of elements in the current string, performing the same role as the <a href="size">size</a>() function.

Related topics:

# lexicographical\_compare

#### Syntax:

```
#include <algorithm>
bool lexicographical_compare( iterator start1, iterator end1, ite
bool lexicographical_compare( iterator start1, iterator end1, ite
```

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 <a href="mailto:linear time">linear time</a>.

#### Related topics:

equal lexicographical\_compare\_3way mismatch search

# lexicographical\_compare\_3way

Syntax:

```
#include <algorithm>
int lexicographical_compare_3way( iterator start1, iterator end1,
```

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:

<u>lexicographical\_compare</u>

cppreference.com > C++ Sets > lower\_bound

# lower\_bound

Syntax:

```
#include <set>
iterator lower_bound( const <u>key type</u>& 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.

cppreference.com > C++ Multisets > lower\_bound

# 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.

cppreference.com > C++ Maps > lower\_bound

# lower\_bound

Syntax:

```
#include <map>
iterator lower_bound( const <u>key type</u>& 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.

cppreference.com > C++ Multimaps > lower\_bound

# lower\_bound

Syntax:

```
#include <map>
iterator lower_bound( const <u>key type</u>& 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.

### lower\_bound

Syntax:

```
#include <algorithm>
iterator lower_bound( iterator first, iterator last, const TYPE&
iterator lower_bound( iterator first, iterator last, const TYPE&
```

The lower\_bound() function is a type of <a href="mailto:binary\_search">binary\_search</a>(). 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: ";</pre>
for( unsigned int i = 0; i < nums.size(); i++ ) {</pre>
  cout << nums[i] << " ";
}
cout << endl;</pre>
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;</pre>
```

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
```

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 <a href="linear time">linear time</a>.

Related topics:

is\_heap pop\_heap

push\_heap

sort\_heap

```
cppreference.com > C++ Algorithms > max
```

### 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)
```

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+array);
vector<char> v;
v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back(
cout << "Max element in the vector v is " << *max_element( v.beging)</pre>
```

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 cppreference.com > C++ Strings > max\_size

### max\_size

Syntax:

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

The max\_size() function returns the maximum number of elements that the string 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 string and the number of elements that the string will be able to hold before more memory will have to be allocated, respectively.

Related topics:

cppreference.com > C++ Vectors > max\_size

### 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 number of elements that the vector will be able to hold before more memory will have to be allocated, respectively.

Related topics:

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>max\_size</u>

### 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 number of elements that the dequeue will be able to hold before more memory will have to be allocated, respectively.

Related topics:

<u>cppreference.com</u> > <u>C++ Lists</u> > <u>max\_size</u>

### 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 number of elements that the list will be able to hold before more memory will have to be allocated, respectively.

Related topics:

 $\underline{\text{cppreference.com}} > \underline{\text{C++ Sets}} > \underline{\text{max size}}$ 

### 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 <u>size()</u> or (C++ Strings) <u>capacity()</u> functions, which return the number of elements currently in the set and the number of elements that the set will be able to hold before more memory will have to be allocated, respectively.

Related topics:

cppreference.com > C++ Multisets > max\_size

### 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 number of elements that the multiset will be able to hold before more memory will have to be allocated, respectively.

Related topics:

cppreference.com > C++ Maps > max\_size

### 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 <a href="size">size</a>() or (C++ Strings) <a href="capacity">capacity</a>() 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:

cppreference.com > C++ Multimaps > max\_size

### 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 number of elements that the multimap will be able to hold before more memory will have to be allocated, respectively.

Related topics:

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>merge</u>
```

### 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 <u>linear time</u>.

Related topics:

**Container operators insert splice** 

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>merge</u>

### merge

Syntax:

```
#include <algorithm>
iterator merge( iterator start1, iterator end1, iterator start2,
iterator merge( iterator start1, iterator end1, iterator start2,
```

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: <a href="mailto:inplace\_merge">inplace\_merge</a> set union sort

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>min</u>
```

### 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 cppreference.com > C++ Algorithms > 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

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>mismatch</u>

### mismatch

#### Syntax:

```
#include <algorithm>
pair <iterator1,iterator2> mismatch( iterator start1, iterator en
pair <iterator1,iterator2> mismatch( iterator start1, iterator en
```

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 <u>linear time</u>.

#### 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, StrictWeakOr
```

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 <u>cppreference.com</u> > <u>C++ Bitsets</u> > <u>none</u>

### 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

## nth\_element

Syntax:

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

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 <u>linear time</u> on average.

Related topics:

partial\_sort

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{open}}
```

### open

Syntax:

```
#include <fstream>
void open( const char *filename );
void open( const char *filename, openmode mode = default_mode );
```

The function open() is used with file streams. It opens *filename* and associates it with the current stream. The optional <u>io stream mode flag mode</u> defaults to ios::in for ifstream, ios::out for ofstream, and ios::in|ios::out for fstream.

If open() fails, the resulting stream will evaluate to false when used in a Boolean expression. For example:

```
ifstream inputStream;
inputStream.open("file.txt");
if( !inputStream ) {
  cerr << "Error opening input stream" << endl;
  return;
}</pre>
```

Related topics:

**I/O Constructors close** 

## partial\_sort

#### Syntax:

```
#include <algorithm>
void partial_sort( iterator start, iterator middle, iterator end
void partial_sort( iterator start, iterator middle, iterator end,
```

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, iterato
iterator partial_sort_copy( iterator start, iterator end, iterato
```

The partial\_sort\_copy() algorithm behaves like <u>partial\_sort(</u>), 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:
<a href="mailto:binary search">binary search is sorted</a>
<a href="mailto:partial\_sort">partial\_sort</a>
<a href="mailto:sort">sort</a>
<a href="mailto:stable\_sort">stable\_sort</a>

#### partial\_sum

#### Syntax:

```
#include <numeric>
iterator partial_sum( iterator start, iterator end, iterator resu
iterator partial_sum( iterator start, iterator end, iterator resu
```

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 <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>partition</u>

#### 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 <u>linear time</u>.

Related topics: <a href="mailto:stable\_partition">stable\_partition</a>

 $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{peek}}$ 

### peek

Syntax:

```
#include <fstream>
int peek();
```

The function peek() is used with input streams, and returns the next character in the stream or **EOF** if the end of file is read. peek() does not remove the character from the stream.

Related topics: **get putback** 

```
cppreference.com > C++ Stacks > pop
```

#### pop

Syntax:

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

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

Related topics:

push top

```
<u>cppreference.com</u> > <u>C++ Queues</u> > <u>pop</u>
```

#### 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
```

cppreference.com > C++ Priority Queues > pop

#### pop

Syntax:

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

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

Related topics:

push top

cppreference.com > C++ Vectors > pop\_back

# 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 cppreference.com > C++ Double-ended Queues > pop\_back

# 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 erase pop\_front push\_back cppreference.com > C++ Lists > pop\_back

# 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 erase pop\_front push\_back cppreference.com > C++ Double-ended Queues > pop\_front

## 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 front pop\_back push\_front cppreference.com > C++ Lists > pop\_front

# 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

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>pop_heap</u>
```

## pop\_heap

#### Syntax:

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

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

```
<u>cppreference.com</u> > <u>C++ I/O</u> > <u>precision</u>
```

### precision

Syntax:

```
#include <fstream>
streamsize precision();
streamsize precision( streamsize p );
```

The precision() function either sets or returns the current number of digits that is displayed for floating-point variables.

For example, the following code sets the precision of the cout stream to 5:

```
float num = 314.15926535;
cout.precision( 5 );
cout << num;</pre>
```

This code displays the following output:

```
314.16
```

Related topics:

fill width

### prev\_permutation

#### Syntax:

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

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

```
cppreference.com > C++ Stacks > push
```

# 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);</pre>
```

Related topics:

pop

```
cppreference.com > C++ Queues > push
```

# 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);
}</pre>
```

Related topics:

pop

```
cppreference.com > C++ Priority Queues > push
```

# 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);</pre>
```

```
cppreference.com > C++ Vectors > 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 );
}</pre>
```

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

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>push\_back</u>

### 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 );</pre>
```

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

```
cppreference.com > C++ Lists > push_back
```

## 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

```
cppreference.com > C++ Strings > push_back
```

### push\_back

Syntax:

```
#include <string>
void push_back( char c );
```

The push\_back() function appends *c* to the end of the string.

For example, the following code adds 10 characters to a string:

```
string the_string;
for( int i = 0; i < 10; i++ )
  the_string.push_back( i+'a' );</pre>
```

When displayed, the resulting string would look like this:

```
abcdefghij
```

push\_back() runs in constant time.

Related topics: assign insert

cppreference.com > C++ Double-ended Queues > 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

#### cppreference.com > C++ Lists > 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

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>push_heap</u>
```

### push\_heap

#### Syntax:

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

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 make heap pop heap sort heap

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{put}}
```

#### put

Syntax:

```
#include <fstream>
ostream& put( char ch );
```

The function put() is used with output streams, and writes the character *ch* to the stream.

Related topics:

flush get write  $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{putback}}$ 

## putback

Syntax:

```
#include <fstream>
istream& putback( char ch );
```

The putback() function is used with input streams, and returns the previously-read character *ch* to the input stream.

Related topics:

peek (Standard C I/O) ungetc

## random\_sample

#### Syntax:

```
#include <algorithm>
iterator random_sample( iterator start1, iterator end1, iterator
iterator random_sample( iterator start1, iterator end1, iterator
```

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 prev\_permutation random\_sample\_n random\_shuffle

## random\_sample\_n

#### Syntax:

```
#include <algorithm>
iterator random_sample_n( iterator start, iterator end, iterator
iterator random_sample_n( iterator start, iterator end, iterator
```

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 <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>random\_shuffle</u>

#### random\_shuffle

#### Syntax:

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

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

```
cppreference.com > C++ Vectors > rbegin
```

## rbegin

Syntax:

```
#include <vector>
  reverse iterator rbegin();
  const_reverse iterator rbegin() const;
```

The rbegin() function returns a <a href="reverse\_iterator">reverse\_iterator</a> to the end of the current vector.

rbegin() runs in constant time.

Related topics:

```
cppreference.com > C++ Strings > rbegin
```

## rbegin

Syntax:

```
#include <string>
  reverse iterator rbegin();
  const_reverse iterator rbegin() const;
```

The rbegin() function returns a <a href="reverse\_iterator">reverse\_iterator</a> to the end of the current string.

rbegin() runs in constant time.

Related topics:

cppreference.com > C++ Double-ended Queues > rbegin

## rbegin

Syntax:

```
#include <deque>
  reverse iterator rbegin();
const_reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse\_iterator</u> to the end of the current dequeue.

rbegin() runs in constant time.

Related topics:

```
cppreference.com > C++ Lists > rbegin
```

## 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:

```
cppreference.com > C++ Sets > rbegin
```

## rbegin

Syntax:

```
#include <set>
  reverse iterator rbegin();
  const_reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse\_iterator</u> to the end of the current set.

rbegin() runs in constant time.

Related topics:

cppreference.com > C++ Multisets > rbegin

## 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:

cppreference.com > C++ Maps > rbegin

# rbegin

Syntax:

```
#include <map>
  reverse iterator rbegin();
  const reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse\_iterator</u> to the end of the current map.

rbegin() runs in constant time.

Related topics:

begin end rend cppreference.com > C++ Multimaps > rbegin

# rbegin

Syntax:

```
#include <map>
  reverse iterator rbegin();
  const_reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse\_iterator</u> to the end of the current multimap.

rbegin() runs in constant time.

Related topics:

begin end rend  $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{rdstate}}$ 

# rdstate

Syntax:

```
#include <fstream>
iostate rdstate();
```

The rdstate() function returns the io stream state flags of the current stream.

Related topics:

bad clear eof fail good

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{read}}
```

### read

*Syntax:* 

```
#include <fstream>
istream& read( char* buffer, streamsize num );
```

The function read() is used with input streams, and reads *num* bytes from the stream before placing them in *buffer*. If **EOF** is encountered, read() stops, leaving however many bytes it put into *buffer* as they are.

For example:

```
struct {
   int height;
   int width;
} rectangle;

input_file.read( (char *)(&rectangle), sizeof(rectangle) );
if( input_file.bad() ) {
   cerr << "Error reading data" << endl;
   exit( 0 );
}</pre>
```

Related topics:

gcount get getline write

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>remove</u>
```

#### 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' );</pre>
```

Remove runs in linear time.

Related topics:

erase remove\_if unique

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>remove</u>

#### 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 <u>linear time</u>.

#### Related topics:

remove\_copy remove\_copy\_if
remove\_if
unique
unique\_copy

cppreference.com > C++ Algorithms > remove\_copy

### remove\_copy

Syntax:

```
#include <algorithm>
iterator remove_copy( iterator start, iterator end, iterator resu
```

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 <a href="lineartime">lineartime</a>.

Related topics:

copy remove
remove\_copy\_if
remove\_if

# remove\_copy\_if

Syntax:

```
#include <algorithm>
iterator remove_copy_if( iterator start, iterator end, iterator r
```

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 <a href="linear-time">linear time</a>.

*Related topics:* 

remove remove\_copy remove\_if

cppreference.com > C++ Lists > remove\_if

# 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 <a href="linear time">linear time</a>.

Related topics:

erase remove unique

#### cppreference.com > C++ Algorithms > 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 <a href="linear time">linear time</a>.

Related topics:

remove remove\_copy remove\_copy\_if

```
cppreference.com > C++ Vectors > rend
```

Syntax:

```
#include <vector>
  reverse iterator rend();
  const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current vector.

rend() runs in constant time.

Related topics:

```
cppreference.com > C++ Strings > rend
```

Syntax:

```
#include <string>
  reverse iterator rend();
const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current string.

rend() runs in constant time.

Related topics:

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>rend</u>

### rend

Syntax:

```
#include <deque>
  reverse iterator rend();
const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current dequeue.

rend() runs in constant time.

Related topics:

```
cppreference.com > C++ Lists > rend
```

Syntax:

```
#include <list>
  reverse iterator rend();
  const reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current list. rend() runs in <u>constant time</u>.

Related topics: begin end rbegin

```
cppreference.com > C++ Sets > rend
```

Syntax:

```
#include <set>
  reverse iterator rend();
  const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current set. rend() runs in <u>constant time</u>.

Related topics: begin end rbegin

```
cppreference.com > C++ Multisets > rend
```

Syntax:

```
#include <set>
  reverse iterator rend();
  const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current multiset.

rend() runs in constant time.

Related topics:

```
cppreference.com > C++ Maps > rend
```

Syntax:

```
#include <map>
  reverse iterator rend();
  const reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current map.

rend() runs in constant time.

Related topics:

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>rend</u>
```

Syntax:

```
#include <map>
  reverse iterator rend();
  const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current multimap.

rend() runs in constant time.

Related topics:

# replace

#### Syntax:

```
#include <string>
string& replace( size_type index, size_type num, const string& st
string& replace( size_type index1, size_type num1, const string&
string& replace( size_type index, size_type num, const char* str
string& replace( size_type index, size_type num1, const char* str
string& replace( size_type index, size_type num1, size_type num2,
string& replace( iterator start, iterator end, const string& str
string& replace( iterator start, iterator end, const char* str);
string& replace( iterator start, iterator end, const char* str, s
string& replace( iterator start, iterator end, size_type num, cha
```

#### The function replace() either:

- replaces characters of the current string with up to *num* characters from *str*, beginning at *index*,
- replaces up to *num1* characters of the current string (starting at *index1*) with up to *num2* characters from *str* beginning at *index2*,
- replaces up to *num* characters of the current string with characters from *str*, beginning at *index* in *str*,
- replaces up to *num1* characters in the current string (beginning at *index1*) with *num2* characters from *str* beginning at *index2*,
- replaces up to *num1* characters in the current string (beginning at *index*) with *num2* copies of *ch*,
- replaces the characters in the current string from *start* to *end* with *str*,
- replaces characters in the current string from *start* to *end* with *num* characters from *str*,
- or replaces the characters in the current string from *start* to *end* with *num* copies of *ch*.

For example, the following code displays the string "They say he carved it himself...find your soul-mate, Homer."

```
string s = "They say he carved it himself...from a BIGGER spoon", string s2 = "find your soul-mate, Homer.";
```

```
s.replace( 32, s2.length(), s2 );
cout << s << endl;</pre>
```

Related topics:

<u>insert</u>

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>replace</u>

# replace

Syntax:

```
#include <algorithm>
void replace( iterator start, iterator end, const TYPE& old_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 <a href="linear time">linear time</a>.

Related topics:

replace\_copy replace\_copy\_if replace\_if <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>replace\_copy</u>

# replace\_copy

Syntax:

```
#include <algorithm>
iterator replace_copy( iterator start, iterator end, iterator res
```

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:

<u>replace</u>

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>replace\_copy\_if</u>

# replace\_copy\_if

Syntax:

```
#include <algorithm>
iterator replace_copy_if( iterator start, iterator end, iterator
```

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:

<u>replace</u>

cppreference.com > C++ Algorithms > replace\_if

# replace\_if

Syntax:

```
#include <algorithm>
void replace_if( iterator start, iterator end, Predicate p, const
```

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:

<u>replace</u>

<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>reserve</u>

### 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 <u>linear time</u>.

Related topics:

**capacity** 

<u>cppreference.com</u> > <u>C++ Strings</u> > <u>reserve</u>

### reserve

*Syntax:* 

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

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

reserve() runs in <u>linear time</u>.

Related topics:

**capacity** 

```
<u>cppreference.com</u> > <u>C++ Bitsets</u> > <u>reset</u>
```

### 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.

<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>resize</u>

### 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:

<u>Vector constructors & destructors capacity</u> <u>size</u> <u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>resize</u>

### 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) <u>Multimap constructors</u> & <u>destructors</u> (C++ Strings) <u>capacity</u> <u>size</u>

<u>cppreference.com</u> > <u>C++ Lists</u> > <u>resize</u>

### 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) <u>Multimap constructors & destructors</u> (C++ Strings) <u>capacity</u> <u>size</u>

<u>cppreference.com</u> > <u>C++ Strings</u> > <u>resize</u>

### resize

Syntax:

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

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

This function runs in <u>linear time</u>.

Related topics:

(C++ Multimaps) <u>Multimap constructors</u> & <u>destructors</u> capacity <u>size</u>

cppreference.com > C++ Lists > reverse

### reverse

Syntax:

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

The function reverse() reverses the list, and takes <u>linear time</u>.

Related topics:

<u>sort</u>

cppreference.com > C++ Algorithms > reverse

### 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

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>reverse\_copy</u>

## reverse\_copy

Syntax:

```
#include <algorithm>
iterator reverse_copy( iterator start, iterator end, iterator res
```

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

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>rfind</u>
```

### rfind

#### Syntax:

```
#include <string>
size_type rfind( const string& str, size_type index );
size_type rfind( const char* str, size_type index );
size_type rfind( const char* str, size_type index, size_type num
size_type rfind( char ch, size_type index );
```

The rfind() function either:

- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, string::npos if nothing is found,
- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, searching at most *num* characters, string::npos if nothing is found,
- or returns the location of the first occurrence of *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, in the following code, the first call to rfind() returns string::npos, because the target word is not within the first 8 characters of the string. However, the second call returns 9, because the target word is within 20 characters of the beginning of the string.

```
int loc;
string s = "My cat's breath smells like cat food.";
loc = s.rfind( "breath", 8 );
cout << "The word breath is at index " << loc << endl;
loc = s.rfind( "breath", 20 );
cout << "The word breath is at index " << loc << endl;</pre>
```

#### Related topics:

```
find find_first_not_of
find_first_of
find_last_not_of
find_last_of
```

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>rotate</u>

### rotate

Syntax:

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

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

cppreference.com > C++ Algorithms > rotate\_copy

# rotate\_copy

Syntax:

```
#include <algorithm>
iterator rotate_copy( iterator start, iterator middle, iterator e
```

The rotate\_copy() algorithm is similar to the <u>rotate()</u> algorithm, except that the range of elements is copied to *result* before being rotated.

Related topics:

**rotate** 

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>search</u>

# search

#### Syntax:

```
#include <algorithm>
iterator search( iterator start1, iterator end1, iterator start2,
iterator search( iterator start1, iterator end1, iterator start2,
```

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 <u>linear</u> time.

### Related topics:

equal find lexicographical\_compare mismatch search\_n <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>search\_n</u>

# search\_n

#### Syntax:

```
#include <algorithm>
iterator search_n( iterator start, iterator end, size_t num, cons
iterator search_n( iterator start, iterator end, size_t num, cons
```

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 <u>linear time</u>.

Related topics: find\_end find\_if search

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{seekg}}
```

# seekg

Syntax:

```
#include <fstream>
istream& seekg( off_type offset, ios::seekdir origin );
istream& seekg( pos_type position );
```

The function seekg() is used with input streams, and it repositions the "get" pointer for the current stream to *offset* bytes away from *origin*, or places the "get" pointer at *position*.

Related topics: seekp tellg tellp

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{seekp}}
```

# seekp

Syntax:

```
#include <fstream>
ostream& seekp( off_type offset, ios::seekdir origin );
ostream& seekp( pos_type position );
```

The seekp() function is used with output streams, but is otherwise very similar to seekg().

Related topics: seekg tellg

<u>tellp</u>

```
<u>cppreference.com</u> > <u>C++ Bitsets</u> > <u>set</u>
```

### 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.

# set\_difference

#### Syntax:

```
#include <algorithm>
iterator set_difference( iterator start1, iterator end1, iterator
iterator set_difference( iterator start1, iterator end1, iterator
```

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 intersection set symmetric difference set union

# set\_intersection

#### Syntax:

```
#include <algorithm>
iterator set_intersection( iterator start1, iterator end1, iterat
iterator set_intersection( iterator start1, iterator end1, iterat
```

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\_difference set\_symmetric\_difference set\_union

# set\_symmetric\_difference

#### Syntax:

```
#include <algorithm>
iterator set_symmetric_difference( iterator start1, iterator end1
iterator set_symmetric_difference( iterator start1, iterator end1
```

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\_difference set\_intersection set\_union

#### cppreference.com > C++ Algorithms > set\_union

# set\_union

#### Syntax:

```
#include <algorithm>
iterator set_union( iterator start1, iterator end1, iterator star
iterator set_union( iterator start1, iterator end1, iterator star
```

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 <a href="linear-time">linear time</a>.

#### Related topics:

includes merge
set\_difference
set\_intersection
set\_symmetric\_difference

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{setf}}
```

## setf

Syntax:

```
#include <fstream>
fmtflags setf( fmtflags flags );
fmtflags setf( fmtflags flags, fmtflags needed );
```

The function setf() sets the <u>io stream format flags</u> of the current stream to *flags*. The optional *needed* argument specifies that only the flags that are in both *flags* and *needed* should be set. The return value is the previous configuration of <u>io stream format flags</u>.

For example:

```
int number = 0x3FF;
cout.setf( ios::dec );
cout << "Decimal: " << number << endl;
cout.unsetf( ios::dec );
cout.setf( ios::hex );
cout << "Hexadecimal: " << number << endl;</pre>
```

Note that the preceding code is functionally identical to:

```
int number = 0x3FF;
cout << "Decimal: " << number << endl << hex << "Hexadecimal: " <</pre>
```

thanks to io stream manipulators.

Related topics:

flags unsetf

```
cppreference.com > C++ Strings > size
```

## size

Syntax:

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

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

Related topics: capacity empty

length max\_size resize cppreference.com > C++ Vectors > 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

<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>size</u>

## 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
```

#### <u>cppreference.com</u> > <u>C++ Lists</u> > <u>size</u>

## 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
(C++ Strings) length
max_size
resize
```

```
cppreference.com > C++ Sets > size
```

## 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
```

<u>cppreference.com</u> > <u>C++ Multisets</u> > <u>size</u>

## 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
```

<u>cppreference.com</u> > <u>C++ Maps</u> > <u>size</u>

# 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

cppreference.com > C++ Multimaps > size

## 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
```

cppreference.com > C++ Stacks > size

## 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
```

cppreference.com > C++ Queues > size

## 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
```

cppreference.com > C++ Priority Queues > size

## 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

```
<u>cppreference.com</u> > <u>C++ Bitsets</u> > <u>size</u>
```

# size

Syntax:

```
#include <bitset>
size_t size();
```

The size() function returns the number of bits that the bitset can hold.

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>sort</u>
```

### 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** 

### 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 << "Interpretation of the cout in t
```

```
}
cout << endl;</pre>
```

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;</pre>
```

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++ ) {</pre>
```

```
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;</pre>
```

### Related topics:

binary search is sorted
merge
partial\_sort
partial\_sort\_copy
stable\_sort
(Other Standard C Functions) qsort

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>sort\_heap</u>

# sort\_heap

#### Syntax:

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

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

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>splice</u>
```

# 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 en
```

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 <u>constant time</u>.

Related topics:

<u>insert merge</u> <u>swap</u> <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>stable\_partition</u>

# stable\_partition

Syntax:

```
#include <algorithm>
iterator stable_partition( iterator start, iterator end, Predicat
```

The stable\_partition() function behaves similarly to <u>partition()</u>. 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, StrictWeakOrderin
```

The stable\_sort() algorithm is like the <u>sort()</u> algorithm, in that it sorts a range of elements into ascending order. Unlike <u>sort()</u>, 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 copy sort

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>substr</u>
```

### substr

#### Syntax:

```
#include <string>
string substr( size_type index, size_type length = npos );
```

The substr() function returns a substring of the current string, starting at *index*, and *length* characters long. If *length* is omitted, it will default to string::npos, and the substr() function will simply return the remainder of the string starting at *index*.

#### For example:

```
string s("What we have here is a failure to communicate");
string sub = s.substr(21);
cout << "The original string is " << s << endl;
cout << "The substring is " << sub << endl;</pre>
```

#### displays

The original string is What we have here is a failure to communicate

### Related topics:

**copy** 

```
<u>cppreference.com</u> > <u>C++ Strings</u> > <u>swap</u>
```

Syntax:

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

The swap() function exchanges the elements of the current string with those of *from*. This function operates in <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

(C++ Lists) splice

```
<u>cppreference.com</u> > <u>C++ Vectors</u> > <u>swap</u>
```

*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 <u>constant time</u>.

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;</pre>
```

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
```

```
<u>cppreference.com</u> > <u>C++ Double-ended Queues</u> > <u>swap</u>
```

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

(C++ Lists) splice

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>swap</u>
```

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

**splice** 

```
<u>cppreference.com</u> > <u>C++ Sets</u> > <u>swap</u>
```

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

```
<u>cppreference.com</u> > <u>C++ Multisets</u> > <u>swap</u>
```

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

```
<u>cppreference.com</u> > <u>C++ Maps</u> > <u>swap</u>
```

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

```
<u>cppreference.com</u> > <u>C++ Multimaps</u> > <u>swap</u>
```

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

#### <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>swap</u>

### 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 <u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>swap\_ranges</u>

### swap\_ranges

Syntax:

```
#include <algorithm>
iterator swap_ranges( iterator start1, iterator end1, iterator st
```

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

cppreference.com > C++ I/O > sync\_with\_stdio

# sync\_with\_stdio

Syntax:

```
#include <fstream>
static bool sync_with_stdio( bool sync=true );
```

The sync\_with\_stdio() function allows you to turn on and off the ability for the C++ I/O system to work with the C I/O system.

 $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{tellg}}$ 

# tellg

Syntax:

```
#include <fstream>
pos_type tellg();
```

The tellg() function is used with input streams, and returns the current "get" position of the pointer in the stream.

Related topics: seekg seekp tellp

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{tellp}}
```

## tellp

Syntax:

```
#include <fstream>
pos_type tellp();
```

The tellp() function is used with output streams, and returns the current "put" position of the pointer in the stream.

For example, the following code displays the file pointer as it writes to a stream:

```
string s("In Xanadu did Kubla Khan...");
ofstream fout("output.txt");
for( int i=0; i < s.length(); i++ ) {
  cout << "File pointer: " << fout.tellp();
  fout.put( s[i] );
  cout << " " << s[i] << endl;
}
fout.close();</pre>
```

Related topics: seekg seekp

**tellg** 

```
<u>cppreference.com</u> > <u>C++ Bitsets</u> > <u>test</u>
```

## test

Syntax:

```
#include <bitset>
bool test( size_t pos );
```

The function test() returns the value of the bit at position *pos*.

cppreference.com > C++ Bitsets > to\_string

# to\_string

Syntax:

```
#include <bitset>
string to_string();
```

The to\_string() function returns a string representation of the bitset.

Related topics:

to\_ulong

cppreference.com > C++ Bitsets > 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

```
cppreference.com > C++ Stacks > top
```

## 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();
}</pre>
```

Related topics:

pop

```
<u>cppreference.com</u> > <u>C++ Priority Queues</u> > <u>top</u>
```

## 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();
}</pre>
```

Related topics:

pop

### transform

#### Syntax:

```
#include <algorithm>
iterator transform( iterator start, iterator end, iterator result
iterator transform( iterator start1, iterator end1, iterator star
```

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

```
<u>cppreference.com</u> > <u>C++ Lists</u> > <u>unique</u>
```

## 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

```
<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>unique</u>
```

## 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 resu
iterator unique_copy( iterator start, iterator end, iterator resu
```

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 <u>linear time</u>.

Related topics:
<a href="mailto:adjacent\_find">adjacent\_find</a> remove
<a href="mailto:unique">unique</a>

<u>cppreference.com</u> > <u>C++ I/O</u> > <u>unsetf</u>

## unsetf

Syntax:

```
#include <fstream>
void unsetf( fmtflags flags );
```

The function unsetf() uses *flags* to clear the <u>io stream format flag</u>s associated with the current stream.

Related topics:

**flags** setf

cppreference.com > C++ Sets > upper\_bound

# 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 value greater than *key*.

Related topics:

cppreference.com > C++ Multisets > upper\_bound

# upper\_bound

Syntax:

```
#include <set>
iterator upper_bound( const <u>key type</u>& key );
```

The function upper\_bound() returns an iterator to the first element in the multiset with a key greater than *key*.

Related topics:

cppreference.com > C++ Maps > upper\_bound

# upper\_bound

Syntax:

```
#include <map>
iterator upper_bound( const <u>key type</u>& key );
```

The function upper\_bound() returns an iterator to the first element in the map with a key greater than *key*.

Related topics:

cppreference.com > C++ Multimaps > upper\_bound

# upper\_bound

Syntax:

```
#include <map>
iterator upper_bound( const <u>key type</u>& key );
```

The function upper\_bound() returns an iterator to the first element in the multimap with a key greater than *key*.

Related topics:

<u>cppreference.com</u> > <u>C++ Algorithms</u> > <u>upper\_bound</u>

## upper\_bound

Syntax:

```
#include <algorithm>
iterator upper_bound( iterator start, iterator end, const TYPE& v
iterator upper_bound( iterator start, iterator end, const TYPE& v
```

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

cppreference.com > C++ Sets > value\_comp

# 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:

cppreference.com > C++ Multisets > value\_comp

# 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:

cppreference.com > C++ Maps > value\_comp

# 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:

cppreference.com > C++ Multimaps > value\_comp

# 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:

```
\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{width}}
```

## width

Syntax:

```
#include <fstream>
int width();
int width( int w );
```

The function width() returns the current width, which is defined as the minimum number of characters to display with each output. The optional argument *w* can be used to set the width.

For example:

```
cout.width( 5 );
cout << "2";</pre>
```

displays

2

(that's four spaces followed by a '2')

Related topics:

**fill precision** 

cppreference.com > C++ I/O > write

## write

Syntax:

```
#include <fstream>
ostream& write( const char* buffer, streamsize num );
```

The write() function is used with output streams, and writes *num* bytes from *buffer* to the current output stream.

Related topics:

flush put read

# **All C Functions**

#, ##	manipulate strings
<u>#define</u>	define variables
<u>#error</u>	display an error message
#if, #ifdef, #ifndef, #else, #elif, #endif	conditional operators
#include	insert the contents of another file
#line	set line and file information
#pragma	implementation specific command
#undef	used to undefine variables
Predefined preprocessor variables	miscellaneous preprocessor variables
<u>abort</u>	stops the program
<u>abs</u>	absolute value
<u>acos</u>	arc cosine
<u>asctime</u>	a textual version of the time
<u>asin</u>	arc sine
<u>assert</u>	stops the program if an expression isn't true
<u>atan</u>	arc tangent
atan2	arc tangent, using signs to determine quadrants
atexit	sets a function to be called when the program exits
<u>atof</u>	converts a string to a double
<u>atoi</u>	converts a string to an integer
<u>atol</u>	converts a string to a long
<u>bsearch</u>	perform a binary search
calloc	allocates and clears a two-dimensional chunk of memory
<u>ceil</u>	the smallest integer not less than a certain value
<u>clearerr</u>	clears errors

clock	returns the amount of time that the program has been running
COS	cosine
<u>cosh</u>	hyperbolic cosine
<u>ctime</u>	returns a specifically formatted version of the time
<u>difftime</u>	the difference between two times
div	returns the quotient and remainder of a division
exit	stop the program
exp	returns "e" raised to a given power
<u>fabs</u>	absolute value for floating-point numbers
<u>fclose</u>	close a file
<u>feof</u>	true if at the end-of-file
<u>ferror</u>	checks for a file error
fflush	writes the contents of the output buffer
fgetc	get a character from a stream
fgetpos	get the file position indicator
<u>fgets</u>	get a string of characters from a stream
floor	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
fopen	open a file
<u>fprintf</u>	print formatted output to a file
<u>fputc</u>	write a character to a file
<u>fputs</u>	write a string to a file
<u>fread</u>	read from a file
<u>free</u>	returns previously allocated memory to the operating system
<u>freopen</u>	open an existing stream with a different name
<u>frexp</u>	decomposes a number into scientific notation
<u>fscanf</u>	read formatted input from a file
<u>fseek</u>	move to a specific location in a file
<u>fsetpos</u>	move to a specific location in a file
<u>ftell</u>	returns the current file position indicator

C •.	
<u>fwrite</u>	write to a file
getc	read a character from a file
getchar	read a character from <b>STDIN</b>
getenv	get enviornment information about a variable
<u>gets</u>	read a string from <b>STDIN</b>
<u>gmtime</u>	returns a pointer to the current Greenwich Mean Time
<u>isalnum</u>	true if a character is alphanumeric
<u>isalpha</u>	true if a character is alphabetic
<u>iscntrl</u>	true if a character is a control character
<u>isdigit</u>	true if a character is a digit
<u>isgraph</u>	true if a character is a graphical character
islower	true if a character is lowercase
<u>isprint</u>	true if a character is a printing character
ispunct	true if a character is punctuation
<u>isspace</u>	true if a character is a space character
<u>isupper</u>	true if a character is an uppercase character
isxdigit	true if a character is a hexidecimal character
<u>labs</u>	absolute value for long integers
ldexp	computes a number in scientific notation
ldiv	returns the quotient and remainder of a division, in long integer form
<u>localtime</u>	returns a pointer to the current time
<u>log</u>	natural logarithm
<u>log10</u>	natural logarithm, in base 10
longjmp	start execution at a certain point in the program
malloc	allocates memory
memchr	searches an array for the first occurance of a character
тетстр	compares two buffers
memcpy	copies one buffer to another
memmove	moves one buffer to another
memset	fills a buffer with a character

<u>mktime</u>	returns the calendar version of a given time
modf	decomposes a number into integer and fractional parts
perror	displays a string version of the current error to <b>STDERR</b>
pow	returns a given number raised to another number
<u>printf</u>	write formatted output to <b>STDOUT</b>
putc	write a character to a stream
<u>putchar</u>	write a character to <b>STDOUT</b>
<u>puts</u>	write a string to <b>STDOUT</b>
<u>qsort</u>	perform a quicksort
<u>raise</u>	send a signal to the program
<u>rand</u>	returns a pseudorandom number
realloc	changes the size of previously allocated memory
remove	erase a file
rename	rename a file
rewind	move the file position indicator to the beginning of a file
<u>scanf</u>	read formatted input from <b>STDIN</b>
<u>setbuf</u>	set the buffer for a specific stream
<u>setjmp</u>	set execution to start at a certain point
<u>setlocale</u>	sets the current locale
<u>setvbuf</u>	set the buffer and size for a specific stream
<u>signal</u>	register a function as a signal handler
<u>sin</u>	sine
<u>sinh</u>	hyperbolic sine
<u>sprintf</u>	write formatted output to a buffer
<u>sqrt</u>	square root
srand	initialize the random number generator
sscanf	read formatted input from a buffer
strcat	concatenates two strings
strchr	finds the first occurance of a character in a string

strcmp	compares two strings
strcoll	compares two strings in accordance to the current locale
strcpy	copies one string to another
strcspn	searches one string for any characters in another
<u>strerror</u>	returns a text version of a given error code
strftime	returns individual elements of the date and time
<u>strlen</u>	returns the length of a given string
strncat	concatenates a certain amount of characters of two strings
strncmp	compares a certain amount of characters of two strings
strncpy	copies a certain amount of characters from one string to another
strpbrk	finds the first location of any character in one string, in another string
strrchr	finds the last occurance of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurance of a substring of characters
strtod	converts a string to a double
<u>strtok</u>	finds the next token in a string
<u>strtol</u>	converts a string to a long
<u>strtoul</u>	converts a string to an unsigned long
<u>strxfrm</u>	converts a substring so that it can be used by string comparison functions
system	perform a system call
<u>tan</u>	tangent
<u>tanh</u>	hyperbolic tangent
<u>time</u>	returns the current calendar time of the system
<u>tmpfile</u>	return a pointer to a temporary file
<u>tmpnam</u>	return a unique filename

tolower	converts a character to lowercase
<u>toupper</u>	converts a character to uppercase
ungetc	puts a character back into a stream
<u>va_arg</u>	use variable length parameter lists
vprintf, vfprintf, and	write formatted output with variable argument
<u>vsprintf</u>	lists

#### <u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u> > #, ##

# #, ##

The # and ## operators are used with the <u>#define</u> macro. Using # causes the first argument after the # to be returned as a string in quotes. Using ## concatenates what's before the ## with what's after it.

Example code:

For example, the command

```
#define to_string( s ) # s
```

will make the compiler turn this command

```
cout << to_string( Hello World! ) << endl;</pre>
```

into

```
cout << "Hello World!" << endl;
```

Here is an example of the ## command:

```
#define concatenate( x, y ) x ## y
...
int xy = 10;
...
```

This code will make the compiler turn

```
cout << concatenate( x, y ) << endl;
```

into

```
cout << xy << endl;
```

which will, of course, display '10' to standard output.

Related topics: #define

### #define

Syntax:

```
#define macro-name replacement-string
```

The #define command is used to make substitutions throughout the file in which it is located. In other words, #define causes the compiler to go through the file, replacing every occurrence of *macro-name* with *replacement-string*. The replacement string stops at the end of the line.

Example code:

Here's a typical use for a #define (at least in C):

```
#define TRUE 1
#define FALSE 0
...
int done = 0;
while( done != TRUE ) {
    ...
}
```

Another feature of the #define command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define absolute_value( x ) ( ((x) < 0) ? -(x) : (x) )
...
int num = -1;
while( absolute_value( num ) ) {
    ...
}</pre>
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within it's own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run

the risk of having the compiler misinterpret your code.

Here is an example of how to use the #define command to create a general purpose incrementing for loop that prints out the integers 1 through 20:

```
#define count_up( v, low, high ) \
  for( (v) = (low); (v) <= (high); (v)++ )

...
int i;
count_up( i, 1, 20 ) {
  printf( "i is %d\n", i );
}</pre>
```

#### Related topics:

#, ## #if, #ifdef, #ifndef, #else, #elif, #endif #undef <u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u> > <u>#error</u>

### #error

Syntax:

#error message

The #error command simply causes the compiler to stop when it is encountered. When an #error is encountered, the compiler spits out the line number and whatever *message* is. This command is mostly used for debugging.

<u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u> > <u>#if, #ifdef, #ifndef,</u> #else, #elif, #endif

# #if, #ifdef, #ifndef, #else, #elif, #endif

These commands give simple logic control to the compiler. As a file is being compiled, you can use these commands to cause certain lines of code to be included or not included.

```
#if expression
```

If the value of expression is true, then the code that immediately follows the command will be compiled.

```
#ifdef macro
```

If the *macro* has been defined by a <u>#define</u> statement, then the code immediately following the command will be compiled.

```
#ifndef macro
```

If the *macro* has not been defined by a <u>#define</u> statement, then the code immediately following the command will be compiled.

A few side notes: The command #elif is simply a horribly truncated way to say "elseif" and works like you think it would. You can also throw in a "defined" or "!defined" after an #if to get added functionality.

### Example code:

Here's an example of all these:

```
#ifdef DEBUG
  cout << "This is the test version, i=" << i << endl;
#else
  cout << "This is the production version!" << endl;</pre>
```

You might notice how that second example could make debugging a lot easier than inserting and removing a million "cout"s in your code.

Related topics:

**#define** 

<u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u> > <u>#include</u>

### #include

Syntax:

```
#include <filename>
#include "filename"
```

This command slurps in a file and inserts it at the current location. The main difference between the syntax of the two items is that if *filename* is enclosed in angled brackets, then the compiler searches for it somehow. If it is enclosed in quotes, then the compiler doesn't search very hard for the file.

While the behavior of these two searches is up to the compiler, usually the angled brackets means to search through the standard library directories, while the quotes indicate a search in the current directory. The spiffy new C++ #include commands don't need to map directly to filenames, at least not for the standard libraries. That's why you can get away with

```
#include <iostream>
```

and not have the compiler choke on you.

#### <u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u> > #line

# #line

Syntax:

```
#line line_number "filename"
```

The #line command is simply used to change the value of the \_\_LINE\_\_ and \_\_FILE\_\_ variables. The filename is optional. The \_\_LINE\_\_ and \_\_FILE\_\_ variables represent the current file and which line is being read. The command

```
#line 10 "main.cpp"
```

changes the current line number to 10, and the current file to "main.cpp".

# #pragma

The #pragma command gives the programmer the ability to tell the compiler to do certain things. Since the #pragma command is implementation specific, uses vary from compiler to compiler. One option might be to trace program execution.

cppreference.com > C/C++ Pre-processor Commands > #undef

# #undef

The #undef command undefines a previously defined macro variable, such as a variable defined by a <u>#define</u>.

Related topics:

#define

<u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u> > <u>Predefined</u> <u>preprocessor variables</u>

# **Predefined preprocessor variables**

Syntax:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

The following variables can vary by compiler, but generally work:

- The \_\_LINE\_\_ and \_\_FILE\_\_ variables represent the current line and current file being processed.
- The \_\_DATE\_\_ variable contains the current date, in the form month/day/year. This is the date that the file was compiled, not necessarily the current date.
- The \_\_TIME\_\_ variable represents the current time, in the form hour:minute:second. This is the time that the file was compiled, not necessarily the current time.
- The \_\_cplusplus variable is only defined when compiling a C++ program. In some older compilers, this is also called c\_plusplus.
- The \_\_STDC\_\_ variable is defined when compiling a C program, and may also be defined when compiling C++.

cppreference.com > Other Standard C Functions > abort

# abort

*Syntax:* 

```
#include <cstdlib>
void abort( void );
```

The function abort() terminates the current program. Depending on the implementation, the return value can indicate failure.

Related topics:
assert atexit
exit

```
cppreference.com > Standard C Math > abs
```

# abs

Syntax:

```
#include <cstdlib>
int abs( int num );
```

The abs() function returns the absolute value of *num*. For example:

```
int magic_number = 10;
cout << "Enter a guess: ";
cin >> x;
cout << "Your guess was " << abs( magic_number - x ) << " away for the second country."</pre>
```

Related topics:

fabs labs

#### cppreference.com > Standard C Math > acos

### acos

#### Syntax:

```
#include <cmath>
double acos( double arg );
```

The acos() function returns the arc cosine of *arg*, which will be in the range [0, pi]. *arg* should be between -1 and 1. If *arg* is outside this range, acos() returns NAN and raises a floating-point exception.

### Related topics:

asin atan

atan2

**COS** 

cosh

<u>sin</u>

<u>sinh</u>

<u>tan</u>

**tanh** 

### <u>cppreference.com</u> > <u>Standard C Date & Time</u> > <u>asctime</u>

# asctime

#### Syntax:

```
#include <ctime>
char *asctime( const struct tm *ptr );
```

The function asctime() converts the time in the struct 'ptr' to a character string of the following format:

```
day month date hours:minutes:seconds year
```

### An example:

```
Mon Jun 26 12:03:53 2000
```

#### Related topics:

clock ctime difftime gmtime localtime mktime time

### cppreference.com > Standard C Math > asin

# asin

#### Syntax:

```
#include <cmath>
double asin( double arg );
```

The asin() function returns the arc sine of *arg*, which will be in the range [-pi/2, +pi/2]. *arg* should be between -1 and 1. If *arg* is outside this range, asin() returns NAN and raises a floating-point exception.

### Related topics:

acos atan

atan2

<u>**COS**</u>

**cosh** 

<u>sin</u>

<u>sinh</u>

<u>tan</u>

tanh

cppreference.com > Other Standard C Functions > assert

### assert

Syntax:

```
#include <cassert>
assert( exp );
```

The assert() macro is used to test for errors. If *exp* evaluates to zero, assert() writes information to **stderr** and exits the program. If the macro NDEBUG is defined, the assert() macros will be ignored.

Related topics:

<u>abort</u>

### cppreference.com > Standard C Math > atan

# atan

#### Syntax:

```
#include <cmath>
double atan( double arg );
```

The function atan() returns the arc tangent of *arg*, which will be in the range [-pi/2, +pi/2].

### Related topics:

acos asin

atan2

<u>**COS**</u>

**cosh** 

<u>sin</u>

<u>sinh</u>

<u>tan</u>

<u>tanh</u>

### cppreference.com > Standard C Math > atan2

# atan2

#### Syntax:

```
#include <cmath>
double atan2( double y, double x );
```

The atan2() function computes the arc tangent of y/x, using the signs of the arguments to compute the quadrant of the return value.

Note the order of the arguments passed to this function.

#### Related topics:

acos asin

<u>atan</u>

COS

cosh

<u>sin</u>

<u>sinh</u>

<u>tan</u>

tanh

cppreference.com > Other Standard C Functions > atexit

### atexit

Syntax:

```
#include <cstdlib>
int atexit( void (*func)(void) );
```

The function atexit() causes the function pointed to by *func* to be called when the program terminates. You can make multiple calls to atexit() (at least 32, depending on your compiler) and those functions will be called in reverse order of their establishment. The return value of atexit() is zero upon success, and non-zero on failure.

Related topics:

abort exit

cppreference.com > Standard C String and Character > atof

# atof

Syntax:

```
#include <cstdlib>
double atof( const char *str );
```

The function atof() converts *str* into a double, then returns that value. *str* must start with a valid number, but can be terminated with any non-numerical character, other than "E" or "e". For example,

```
x = atof( "42.0is_the_answer" );
```

results in x being set to 42.0.

Related topics:

atoi atol (Standard C I/O) sprintf strtod cppreference.com > Standard C String and Character > atoi

### atoi

*Syntax:* 

```
#include <cstdlib>
int atoi( const char *str );
```

The atoi() function converts *str* into an integer, and returns that integer. *str* should start with whitespace or some sort of number, and atoi() will stop reading from *str* as soon as a non-numerical character has been read. For example:

```
int i;
i = atoi( "512" );
i = atoi( "512.035" );
i = atoi( " 512.035" );
i = atoi( " 512+34" );
i = atoi( " 512 bottles of beer on the wall" );
```

All five of the above assignments to the variable *i* would result in it being set to 512.

If the conversion cannot be performed, then atoi() will return zero:

```
int i = atoi( " does not work: 512" ); // results in i == 0
```

You can use sprintf() to convert a number into a string.

*Related topics:* 

atof atol

(Standard C I/O) sprintf

cppreference.com > Standard C String and Character > atol

### atol

Syntax:

```
#include <cstdlib>
long atol( const char *str );
```

The function atol() converts *str* into a long, then returns that value. atol() will read from *str* until it finds any character that should not be in a long. The resulting truncated value is then converted and returned. For example,

```
x = atol( "1024.0001" );
```

results in x being set to 1024L.

Related topics:

atof atoi (Standard C I/O) sprintf strtol cppreference.com > Other Standard C Functions > bsearch

### bsearch

Syntax:

```
#include <cstdlib>
void *bsearch( const void *key, const void *buf, size_t num, size
```

The bsearch() function searches *buf*[0] to *buf*[num-1] for an item that matches *key*, using a binary search. The function *compare* should return negative if its first argument is less than its second, zero if equal, and positive if greater. The items in the array *buf* should be in ascending order. The return value of bsearch() is a pointer to the matching item, or **NULL** if none is found.

*Related topics:* 

**qsort** 

#### cppreference.com > Standard C Memory > calloc

# calloc

Syntax:

```
#include <cstdlib>
void* calloc( size_t num, size_t size );
```

The calloc() function returns a pointer to space for an array of *num* objects, each of size *size*. The newly allocated memory is initialized to zero.

calloc() returns **NULL** if there is an error.

Related topics:

free malloc realloc cppreference.com > Standard C Math > ceil

# ceil

Syntax:

```
#include <cmath>
double ceil( double num );
```

The ceil() function returns the smallest integer no less than *num*. For example,

```
y = 6.04;
x = ceil( y );
```

would set x to 7.0.

Related topics:

**floor fmod** 

cppreference.com > Standard C I/O > clearerr

# clearerr

Syntax:

```
#include <cstdio>
void clearerr( FILE *stream );
```

The clearerr function resets the error flags and **EOF** indicator for the given *stream*. When an error occurs, you can use <u>perror</u>() to figure out which error actually occurred.

Related topics:

feof ferror perror cppreference.com > Standard C Date & Time > clock

# clock

Syntax:

```
#include <ctime>
clock_t clock( void );
```

The clock() function returns the processor time since the program started, or -1 if that information is unavailable. To convert the return value to seconds, divide it by CLOCKS\_PER\_SEC. (Note: if your compiler is POSIX compliant, then CLOCKS\_PER\_SEC is always defined as 1000000.)

Related topics: asctime ctime time

### cppreference.com > Standard C Math > cos

### COS

#### Syntax:

```
#include <cmath>
double cos( double arg );
```

The cos() function returns the cosine of *arg*, where *arg* is expressed in radians. The return value of cos() is in the range [-1,1]. If *arg* is infinite, cos() will return NAN and raise a floating-point exception.

### Related topics:

acos asin

atan

atan2

cosh

<u>sin</u>

<u>sinh</u>

<u>tan</u>

tanh

### cppreference.com > Standard C Math > cosh

# cosh

### Syntax:

```
#include <cmath>
double cosh( double arg );
```

The function cosh() returns the hyperbolic cosine of *arg*.

### Related topics:

acos asin

<u>atan</u>

atan2

<u>**COS**</u>

<u>sin</u>

<u>sinh</u>

<u>tan</u>

<u>tanh</u>

### cppreference.com > Standard C Date & Time > ctime

# ctime

Syntax:

```
#include <ctime>
char *ctime( const time_t *time );
```

The ctime() function converts the calendar time to local time of the format:

```
day month date hours:minutes:seconds year
```

using ctime() is equivalent to

```
asctime( localtime( tp ) );
```

*Related topics:* 

asctime clock gmtime localtime mktime time

### cppreference.com > Standard C Date & Time > difftime

# difftime

Syntax:

```
#include <ctime>
double difftime( time_t time2, time_t time1 );
```

The function difftime() returns *time2 - time1*, in seconds.

Related topics:
asctime gmtime
localtime
time

<u>cppreference.com</u> > <u>Standard C Math</u> > <u>div</u>

# div

Syntax:

```
#include <cstdlib>
div_t div( int numerator, int denominator );
```

The function div() returns the quotient and remainder of the operation *numerator* / *denominator*. The **div\_t** structure is defined in cstdlib, and has at least:

```
int quot; // The quotient
int rem; // The remainder
```

For example, the following code displays the quotient and remainder of x/y:

Related topics:

<u>ldiv</u>

### cppreference.com > Other Standard C Functions > exit

# exit

Syntax:

```
#include <cstdlib>
void exit( int exit_code );
```

The exit() function stops the program. *exit\_code* is passed on to be the return value of the program, where usually zero indicates success and non-zero indicates an error.

Related topics: <a href="mailto:abort atexit">abort atexit</a> <a href="mailto:system">system</a>

### cppreference.com > Standard C Math > exp

# exp

Syntax:

```
#include <cmath>
double exp( double arg );
```

The exp() function returns e (2.7182818) raised to the *arg*th power.

Related topics:

log pow sqrt cppreference.com > Standard C Math > fabs

### fabs

Syntax:

```
#include <cmath>
double fabs( double arg );
```

The function fabs() returns the absolute value of *arg*.

Related topics: <a href="mailto:abs.green">abs.green</a>

labs

cppreference.com > Standard C I/O > fclose

### **fclose**

Syntax:

```
#include <cstdio>
int fclose( FILE *stream );
```

The function fclose() closes the given file stream, deallocating any buffers associated with that stream. fclose() returns 0 upon success, and **EOF** otherwise.

Related topics: fflush fopen freopen

<u>setbuf</u>

### cppreference.com > Standard C I/O > feof

### feof

Syntax:

```
#include <cstdio>
int feof( FILE *stream );
```

The function feof() returns a nonzero value if the end of the given file *stream* has been reached.

Related topics:
clearerr ferror
getc
perror
putc

cppreference.com > Standard C I/O > ferror

### ferror

Syntax:

```
#include <cstdio>
int ferror( FILE *stream );
```

The ferror() function looks for errors with *stream*, returning zero if no errors have occured, and non-zero if there is an error. In case of an error, use <u>perror()</u> to determine which error has occured.

Related topics:

clearerr feof perror

#### cppreference.com > Standard C I/O > fflush

### fflush

*Syntax:* 

```
#include <cstdio>
int fflush( FILE *stream );
```

If the given file *stream* is an output stream, then fflush() causes the output buffer to be written to the file. If the given *stream* is of the input type, then fflush() causes the input buffer to be cleared. fflush() is useful when debugging, if a program segfaults before it has a chance to write output to the screen. Calling fflush( **stdout** ) directly after debugging output will ensure that your output is displayed at the correct time.

```
printf( "Before first call\n" );
fflush( stdout );
shady_function();
printf( "Before second call\n" );
fflush( stdout );
dangerous_dereference();
```

Related topics:

fclose fopen fread fwrite getc putc

### <u>cppreference.com</u> > <u>Standard C I/O</u> > <u>fgetc</u>

## fgetc

Syntax:

```
#include <cstdio>
int fgetc( FILE *stream );
```

The fgetc() function returns the next character from *stream*, or **EOF** if the end of file is reached or if there is an error.

### Related topics:

**fopen fputc** 

<u>fread</u>

**fwrite** 

getc

getchar

gets

<u>putc</u>

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>fgetpos</u>

## fgetpos

Syntax:

```
#include <cstdio>
int fgetpos( FILE *stream, fpos_t *position );
```

The fgetpos() function stores the file position indicator of the given file *stream* in the given *position* variable. The position variable is of type fpos\_t (which is defined in cstdio) and is an object that can hold every possible position in a FILE. fgetpos() returns zero upon success, and a non-zero value upon failure.

Related topics: fseek fsetpos ftell

#### cppreference.com > Standard C I/O > fgets

## fgets

Syntax:

```
#include <cstdio>
char *fgets( char *str, int num, FILE *stream );
```

The function fgets() reads up to *num* - 1 characters from the given file *stream* and dumps them into *str*. The string that fgets() produces is always **NULL**-terminated. fgets() will stop when it reaches the end of a line, in which case *str* will contain that newline character. Otherwise, fgets() will stop when it reaches *num* - 1 characters or encounters the **EOF** character. fgets() returns *str* on success, and **NULL** on an error.

Related topics:

fputs fscanf gets scanf cppreference.com > Standard C Math > floor

### floor

Syntax:

```
#include <cmath>
double floor( double arg );
```

The function floor() returns the largest integer not greater than *arg*. For example,

```
y = 6.04;
x = floor( y );
```

would result in x being set to 6.0.

Related topics:

ceil fmod

cppreference.com > Standard C Math > fmod

### fmod

Syntax:

```
#include <cmath>
double fmod( double x, double y );
```

The fmod() function returns the remainder of x/y.

Related topics:

<u>ceil fabs</u> <u>floor</u>

## fopen

Syntax:

```
#include <cstdio>
FILE *fopen( const char *fname, const char *mode );
```

The fopen() function opens a file indicated by *fname* and returns a stream associated with that file. If there is an error, fopen() returns **NULL**. *mode* is used to determine how the file will be treated (i.e. for input, output, etc)

Mode	Meaning
"r"	Open a text file for reading
"w"	Create a text file for writing
"a"	Append to a text file
"rb"	Open a binary file for reading
"wb"	Create a binary file for writing
"ab"	Append to a binary file
"r+"	Open a text file for read/write
"w+"	Create a text file for read/write
"a+"	Open a text file for read/write
"rb+"	Open a binary file for read/write
"wb+"	Create a binary file for read/write
"ab+"	Open a binary file for read/write

#### An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

# Related topics: fclose fflush

**fgetc** 

**fputc** 

<u>fread</u>

<u>freopen</u>

<u>fseek</u>

**fwrite** 

getc

getchar setbuf

```
cppreference.com > Standard C I/O > fprintf
```

## fprintf

*Syntax:* 

```
#include <cstdio>
int fprintf( FILE *stream, const char *format, ... );
```

The fprintf() function sends information (the arguments) according to the specified *format* to the file indicated by *stream*. fprintf() works just like <u>printf()</u> as far as the format goes. The return value of fprintf() is the number of characters outputted, or a negative number if an error occurs. An example:

```
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
  fprintf( out, "Hello %s\n", name );
```

Related topics:

fputc fputs fscanf printf sprintf

### cppreference.com > Standard C I/O > fputc

## **fputc**

Syntax:

```
#include <cstdio>
int fputc( int ch, FILE *stream );
```

The function fputc() writes the given character *ch* to the given output *stream*. The return value is the character, unless there is an error, in which case the return value is **EOF**.

Related topics:

fgetc fopen fprintf fread fwrite getc getchar putc cppreference.com > Standard C I/O > fputs

## **fputs**

Syntax:

```
#include <cstdio>
int fputs( const char *str, FILE *stream );
```

The fputs() function writes an array of characters pointed to by *str* to the given output *stream*. The return value is non-negative on success, and **EOF** on failure.

Related topics:

fgets fprintf fscanf gets puts

### fread

Syntax:

```
#include <cstdio>
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

The function fread() reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by buffer. The data comes from the given input *stream*. The return value of the function is the number of things read. You can use <u>feof()</u> or <u>ferror()</u> to figure out if an error occurs.

Related topics:

**fflush fgetc** 

**fopen** 

**fputc** 

**fscanf** 

**fwrite** 

getc

<u>cppreference.com</u> > <u>Standard C Memory</u> > <u>free</u>

### free

Syntax:

```
#include <cstdlib>
void free( void* ptr );
```

The free() function deallocates the space pointed to by *ptr*, freeing it up for future use. *ptr* must have been used in a previous call to <u>malloc()</u>, <u>calloc()</u>, or <u>realloc()</u>. An example:

```
typedef struct data_type {
   int age;
   char name[20];
} data;

data *willy;
willy = (data*) malloc( sizeof(*willy) );
...
free( willy );
```

Related topics: <a href="mailto:calloc">calloc</a> (C/C++ Keywords) delete

malloc (C/C++ Keywords) new realloc <u>cppreference.com</u> > <u>Standard C I/O</u> > <u>freopen</u>

## freopen

Syntax:

```
#include <cstdio>
FILE *freopen( const char *fname, const char *mode, FILE *stream
```

The freopen() function is used to reassign an existing *stream* to a different file and mode. After a call to this function, the given file *stream* will refer to *fname* with access given by *mode*. The return value of freopen() is the new stream, or **NULL** if there is an error.

Related topics:

**fclose fopen** 

cppreference.com > Standard C Math > frexp

## frexp

Syntax:

```
#include <cmath>
double frexp( double num, int* exp );
```

The function frexp() is used to decompose *num* into two parts: a mantissa between 0.5 and 1 (returned by the function) and an exponent returned as *exp*. Scientific notation works like this:

```
num = mantissa * (2 ^ exp)
```

Related topics:

ldexp modf

#### <u>cppreference.com</u> > <u>Standard C I/O</u> > <u>fscanf</u>

### fscanf

Syntax:

```
#include <cstdio>
int fscanf( FILE *stream, const char *format, ... );
```

The function fscanf() reads data from the given file *stream* in a manner exactly like scanf(). The return value of fscanf() is the number of variables that are actually assigned values, or **EOF** if no assignments could be made.

Related topics:

**fgets fprintf** 

**fputs** 

**fread** 

**fwrite** 

scanf

sscanf

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>fseek</u>

### fseek

#### Syntax:

```
#include <cstdio>
int fseek( FILE *stream, long offset, int origin );
```

The function fseek() sets the file position data for the given *stream*. The origin value should have one of the following values (defined in cstdio):

Name	Explanation
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

fseek() returns zero upon success, non-zero on failure. You can use fseek() to move beyond a file, but not before the beginning. Using fseek() clears the **EOF** flag associated with that stream.

#### Related topics:

fgetpos fopen fsetpos ftell rewind <u>cppreference.com</u> > <u>Standard C I/O</u> > <u>fsetpos</u>

## fsetpos

Syntax:

```
#include <cstdio>
int fsetpos( FILE *stream, const fpos_t *position );
```

The fsetpos() function moves the file position indicator for the given *stream* to a location specified by the *position* object. fpos\_t is defined in cstdio. The return value for fsetpos() is zero upon success, non-zero on failure.

Related topics:

fgetpos fseek ftell

### cppreference.com > Standard C I/O > ftell

### ftell

Syntax:

```
#include <cstdio>
long ftell( FILE *stream );
```

The ftell() function returns the current file position for *stream*, or -1 if an error occurs.

Related topics:

fgetpos fseek fsetpos

#### cppreference.com > Standard C I/O > fwrite

### **fwrite**

#### Syntax:

```
#include <cstdio>
int fwrite( const void *buffer, size_t size, size_t count, FILE *
```

The fwrite() function writes, from the array *buffer*, *count* objects of size *size* to *stream*. The return value is the number of objects written.

#### Related topics:

**fflush fgetc** 

**fopen** 

**fputc** 

**fread** 

**fscanf** 

getc

```
cppreference.com > Standard C I/O > getc
```

### getc

Syntax:

```
#include <cstdio>
int getc( FILE *stream );
```

The getc() function returns the next character from *stream*, or **EOF** if the end of file is reached. getc() is identical to <u>fgetc()</u>. For example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
while( ch != EOF ) {
   printf( "%c", ch );
   ch = getc( input );
}
```

Related topics:

**feof fflush** 

**fgetc** 

<u>fopen</u>

**fputc** 

<u>fread</u>

**fwrite** 

putc
ungetc

cppreference.com > Standard C I/O > getchar

## getchar

Syntax:

```
#include <cstdio>
int getchar( void );
```

The getchar() function returns the next character from **stdin**, or **EOF** if the end of file is reached.

Related topics:

fgetc fopen fputc putc <u>cppreference.com</u> > <u>Other Standard C Functions</u> > <u>getenv</u>

### getenv

Syntax:

```
#include <cstdlib>
char *getenv( const char *name );
```

The function getenv() returns environmental information associated with *name*, and is very implementation dependent. **NULL** is returned if no information about *name* is available.

Related topics:

**system** 

#### cppreference.com > Standard C I/O > gets

### gets

Syntax:

```
#include <cstdio>
char *gets( char *str );
```

The gets() function reads characters from **stdin** and loads them into *str*, until a newline or **EOF** is reached. The newline character is translated into a null termination. The return value of gets() is the read-in string, or **NULL** if there is an error.

Note that gets() does not perform bounds checking, and thus risks overrunning *str*. For a similar (and safer) function that includes bounds checking, see <u>fgets()</u>.

Related topics:

fgetc fgets fputs puts

#### cppreference.com > Standard C Date & Time > gmtime

## gmtime

#### Syntax:

```
#include <ctime>
struct tm *gmtime( const time_t *time );
```

The gmtime() function returns the given *time* in Coordinated Universal Time (usually Greenwich mean time), unless it's not supported by the system, in which case **NULL** is returned. Watch out for <u>static return</u>.

#### Related topics:

asctime ctime difftime localtime mktime strftime time

cppreference.com > Standard C String and Character > isalnum

### isalnum

Syntax:

```
#include <cctype>
int isalnum( int ch );
```

The function isalnum() returns non-zero if its argument is a numeric digit or a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalnum(c) )
  printf( "You entered the alphanumeric character %c\n", c );
```

*Related topics:* 

isalpha iscntrl

**isdigit** 

<u>isgraph</u>

**isprint** 

**ispunct** 

**isspace** 

cppreference.com > Standard C String and Character > isalpha

## isalpha

Syntax:

```
#include <cctype>
int isalpha( int ch );
```

The function isalpha() returns non-zero if its argument is a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalpha(c) )
  printf( "You entered a letter of the alphabet\n" );
```

*Related topics:* 

isalnum iscntrl

**isdigit** 

<u>isgraph</u>

**isprint** 

**ispunct** 

**isspace** 

cppreference.com > Standard C String and Character > iscntrl

### iscntrl

#### Syntax:

```
#include <cctype>
int iscntrl( int ch );
```

The iscntrl() function returns non-zero if its argument is a control character (between 0 and 0x1F or equal to 0x7F). Otherwise, zero is returned.

### Related topics:

<u>isalnum</u> <u>isalpha</u>

<u>isdigit</u>

**isgraph** 

**isprint** 

**ispunct** 

**isspace** 

cppreference.com > Standard C String and Character > isdigit

## isdigit

Syntax:

```
#include <cctype>
int isdigit( int ch );
```

The function isdigit() returns non-zero if its argument is a digit between 0 and 9. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isdigit(c) )
  printf( "You entered the digit %c\n", c );
```

*Related topics:* 

<u>isalnum</u> <u>isalpha</u>

iscntrl

**isgraph** 

**isprint** 

**ispunct** 

**isspace** 

cppreference.com > Standard C String and Character > isgraph

## isgraph

#### Syntax:

```
#include <cctype>
int isgraph( int ch );
```

The function isgraph() returns non-zero if its argument is any printable character other than a space (if you can see the character, then isgraph() will return a non-zero value). Otherwise, zero is returned.

#### Related topics:

<u>isalnum</u> <u>isalpha</u>

**iscntrl** 

<u>isdigit</u>

**isprint** 

**ispunct** 

**isspace** 

<u>isxdigit</u>

cppreference.com > Standard C String and Character > islower

### islower

Syntax:

```
#include <cctype>
int islower( int ch );
```

The islower() function returns non-zero if its argument is a lowercase letter. Otherwise, zero is returned.

Related topics:

<u>isupper</u>

cppreference.com > Standard C String and Character > isprint

## isprint

Syntax:

```
#include <cctype>
int isprint( int ch );
```

The function isprint() returns non-zero if its argument is a printable character (including a space). Otherwise, zero is returned.

Related topics:

<u>isalnum</u> <u>isalpha</u>

**iscntrl** 

<u>isdigit</u>

**isgraph** 

**ispunct** 

**isspace** 

cppreference.com > Standard C String and Character > ispunct

# ispunct

Syntax:

```
#include <cctype>
int ispunct( int ch );
```

The ispunct() function returns non-zero if its argument is a printing character but neither alphanumeric nor a space. Otherwise, zero is returned.

#### Related topics:

isalnum isalpha

**iscntrl** 

<u>isdigit</u>

**isgraph** 

**isprint** 

**isspace** 

isxdigit

cppreference.com > Standard C String and Character > isspace

## isspace

Syntax:

```
#include <cctype>
int isspace( int ch );
```

The isspace() function returns non-zero if its argument is some sort of space (i.e. single space, tab, vertical tab, form feed, carriage return, or newline). Otherwise, zero is returned.

Related topics:

<u>isalnum</u> <u>isalpha</u>

**iscntrl** 

<u>isdigit</u>

<u>isgraph</u>

**isprint** 

**ispunct** 

**isxdigit** 

cppreference.com > Standard C String and Character > isupper

# isupper

Syntax:

```
#include <cctype>
int isupper( int ch );
```

The isupper() function returns non-zero if its argument is an uppercase letter. Otherwise, zero is returned.

Related topics:

**islower tolower** 

cppreference.com > Standard C String and Character > isxdigit

# isxdigit

Syntax:

```
#include <cctype>
int isxdigit( int ch );
```

The function isxdigit() returns non-zero if its argument is a hexidecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics:

<u>isalnum</u> <u>isalpha</u>

<u>iscntrl</u>

**isdigit** 

**isgraph** 

**ispunct** 

**isspace** 

cppreference.com > Standard C Math > labs

# labs

Syntax:

```
#include <cstdlib>
long labs( long num );
```

The function labs() returns the absolute value of *num*.

Related topics:

abs fabs

cppreference.com > Standard C Math > ldexp

# ldexp

Syntax:

```
#include <cmath>
double ldexp( double num, int exp );
```

The ldexp() function returns  $num * (2 \land exp)$ . And get this: if an overflow occurs, **HUGE\_VAL** is returned.

Related topics:

**frexp modf** 

#### <u>cppreference.com</u> > <u>Standard C Math</u> > <u>ldiv</u>

### ldiv

Syntax:

```
#include <cstdlib>
Idiv_t ldiv( long numerator, long denominator );
```

Testing: adiv\_t, div\_t, ldiv\_t.

The ldiv() function returns the quotient and remainder of the operation *numerator* / *denominator*. The ldiv\_t structure is defined in cstdlib and has at least:

```
long quot; // the quotient long rem; // the remainder
```

*Related topics:* 

<u>div</u>

cppreference.com > Standard C Date & Time > localtime

## localtime

Syntax:

```
#include <ctime>
struct tm *localtime( const time_t *time );
```

The function localtime() converts calendar time into local time. Watch out for the <u>static return</u>.

Related topics:

asctime ctime

**difftime** 

**gmtime** 

**strftime** 

<u>time</u>

#### cppreference.com > Standard C Math > log

# log

Syntax:

```
#include <cmath>
double log( double num );
```

The function log() returns the natural (base e) logarithm of *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

In order to calculate the logarithm of *x* to an arbitrary base *b*, you can use:

```
double answer = log(x) / log(b);
```

Related topics:

exp log10 pow sqrt

#### cppreference.com > Standard C Math > log10

# log10

Syntax:

```
#include <cmath>
double log10( double num );
```

The log10() function returns the base 10 (or common) logarithm for *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

Related topics:

log

cppreference.com > Other Standard C Functions > longjmp

# longjmp

Syntax:

```
#include <csetjmp>
void longjmp( jmp_buf envbuf, int status );
```

The function longjmp() causes the program to start executing code at the point of the last call to <a href="setjmp">setjmp</a>(). <a href="setjmp">envbuf</a> is usually set through a call to <a href="setjmp">setjmp</a>(). <a href="setjmp">status</a> becomes the return value of <a href="setjmp">setjmp</a>() and can be used to figure out where longjmp() came from. <a href="setjmp">status</a> should not be set to zero.

Related topics:

<u>setjmp</u>

### malloc

*Syntax:* 

```
#include <cstdlib>
void *malloc( size_t size );
```

The function malloc() returns a pointer to a chunk of memory of size *size*, or **NULL** if there is an error. The memory pointed to will be on the heap, not the stack, so make sure to free it when you are done with it. An example:

```
typedef struct data_type {
   int age;
   char name[20];
} data;

data *bob;
bob = (data*) malloc( sizeof(data) );
if( bob != NULL ) {
   bob->age = 22;
   strcpy( bob->name, "Robert" );
   printf( "%s is %d years old\n", bob->name, bob->age );
}
free( bob );
```

*Related topics:* 

```
calloc (C/C++ Keywords) delete
free
(C/C++ Keywords) new
realloc
```

cppreference.com > Standard C String and Character > memchr

## memchr

Syntax:

```
#include <cstring>
void *memchr( const void *buffer, int ch, size_t count );
```

The memchr() function looks for the first occurrence of *ch* within *count* characters in the array pointed to by *buffer*. The return value points to the location of the first occurrence of *ch*, or **NULL** if *ch* isn't found. For example:

```
char names[] = "Alan Bob Chris X Dave";
if( memchr(names, 'X', strlen(names)) == NULL )
  printf( "Didn't find an X\n" );
else
  printf( "Found an X\n" );
```

*Related topics:* 

memcmp memcpy strstr

cppreference.com > Standard C String and Character > memcmp

### memcmp

Syntax:

```
#include <cstring>
int memcmp( const void *buffer1, const void *buffer2, size_t coun
```

The function memcmp() compares the first *count* characters of *buffer1* and *buffer2*. The return values are as follows:

Value	Explanation
less than 0	buffer1 is less than buffer2
equal to 0	buffer1 is equal to buffer2
greater than 0	buffer1 is greater than buffer2

Related topics:

memchr memcpy

<u>memset</u>

**strcmp** 

cppreference.com > Standard C String and Character > memcpy

### memcpy

Syntax:

```
#include <cstring>
void *memcpy( void *to, const void *from, size_t count );
```

The function memcpy() copies *count* characters from the array *from* to the array *to*. The return value of memcpy() is *to*. The behavior of memcpy() is undefined if *to* and *from* overlap.

#### Related topics:

memchr memcmp memmove memset strcpy strlen strncpy cppreference.com > Standard C String and Character > memmove

### memmove

*Syntax:* 

```
#include <cstring>
void *memmove( void *to, const void *from, size_t count );
```

The memmove() function is identical to <u>memcpy</u>(), except that it works even if *to* and *from* overlap.

Related topics:

memcpy memset

#### memset

Syntax:

```
#include <cstring>
void* memset( void* buffer, int ch, size_t count );
```

The function memset() copies *ch* into the first *count* characters of *buffer*, and returns *buffer*. memset() is useful for intializing a section of memory to some value. For example, this command:

```
const int ARRAY_LENGTH;
char the_array[ARRAY_LENGTH];
...
// zero out the contents of the_array
memset( the_array, '\0', ARRAY_LENGTH );
```

...is a very efficient way to set all values of the\_array to zero.

The table below compares two different methods for initializing an array of characters: a for-loop versus memset(). As the size of the data being initialized increases, memset() clearly gets the job done much more quickly:

Input size	e Initializo	ed with a for-loop Initialized with memset()
1000	0.016	0.017
10000	0.055	0.013
100000	0.443	0.029
1000000	4.337	0.291

Related topics:

memcmp memcpy memmove

#### cppreference.com > Standard C Date & Time > mktime

## mktime

Syntax:

```
#include <ctime>
time_t mktime( struct tm *time );
```

The mktime() function converts the local time in *time* to calendar time, and returns it. If there is an error, -1 is returned.

Related topics:
asctime ctime
gmtime
time

cppreference.com > Standard C Math > modf

## modf

*Syntax:* 

```
#include <cmath>
double modf( double num, double *i );
```

The function modf() splits *num* into its integer and fraction parts. It returns the fractional part and loads the integer part into *i*.

Related topics:

<u>frexp</u> <u>ldexp</u>

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>perror</u>

### perror

Syntax:

```
#include <cstdio>
void perror( const char *str );
```

The perror() function prints *str* and an implementation-defined error message corresponding to the global variable *errno*. For example:

```
char* input_filename = "not_found.txt";
FILE* input = fopen( input_filename, "r" );
if( input == NULL ) {
   char error_msg[255];
   sprintf( error_msg, "Error opening file '%s'", input_filename )
   perror( error_msg );
   exit( -1 );
}
```

The the file called *not\_found.txt* is not found, this code will produce the following output:

```
Error opening file 'not_found.txt': No such file or directory
```

Related topics:

<u>clearerr</u> <u>feof</u> <u>ferror</u>

#### cppreference.com > Standard C Math > pow

### pow

Syntax:

```
#include <cmath>
double pow( double base, double exp );
```

The pow() function returns *base* raised to the *exp*th power. There's a domain error if *base* is zero and *exp* is less than or equal to zero. There's also a domain error if *base* is negative and *exp* is not an integer. There's a range error if an overflow occurs.

Related topics:

exp log sqrt

#### <u>cppreference.com</u> > <u>Standard C I/O</u> > <u>printf</u>

## printf

Syntax:

```
#include <cstdio>
int printf( const char *format, ... );
```

The printf() function prints output to **stdout**, according to *format* and other arguments passed to printf(). The string *format* consists of two types of items characters that will be printed to the screen, and format commands that define how the other arguments to printf() are displayed. Basically, you specify a format string that has text in it, as well as "special" characters that map to the other arguments of printf(). For example, this code

```
char name[20] = "Bob";
int age = 21;
printf( "Hello %s, you are %d years old\n", name, age );
```

displays the following output:

```
Hello Bob, you are 21 years old
```

The %s means, "insert the first argument, a string, right here." The %d indicates that the second argument (an integer) should be placed there. There are different %-codes for different variable types, as well as options to limit the length of the variables and whatnot.

Code	Format
%с	character
%d	signed integers
%i	signed integers
%e	scientific notation, with a lowercase "e"
%E	scientific notation, with a uppercase "E"

%f	floating point
%g	use %e or %f, whichever is shorter
%G	use %E or %f, whichever is shorter
%0	octal
%s	a string of characters
%u	unsigned integer
%x	unsigned hexadecimal, with lowercase letters
%X	unsigned hexadecimal, with uppercase letters
%p	a pointer
%n	the argument shall be a pointer to an integer into which is placed the number of characters written so far
%%	a '%' sign

An integer placed between a % sign and the format command acts as a minimum field width specifier, and pads the output with spaces or zeros to make it long enough. If you want to pad with zeros, place a zero before the minimum field width specifier:

```
%012d
```

You can also include a precision modifier, in the form of a .N where N is some number, before the format command:

```
%012.4d
```

The precision modifier has different meanings depending on the format command being used:

- With %e, %E, and %f, the precision modifier lets you specify the number of decimal places desired. For example, %12.6f will display a floating number at least 12 digits wide, with six decimal places.
- With %g and %G, the precision modifier determines the maximum number of significant digits displayed.
- With %s, the precision modifer simply acts as a maximumfield length, to

complement the minimum field length that precedes the period.

All of printf()'s output is right-justified, unless you place a minus sign right after the % sign. For example,

%-12.4f

will display a floating point number with a minimum of 12 characters, 4 decimal places, and left justified. You may modify the %d, %i, %o, %u, and %x type specifiers with the letter l and the letter h to specify long and short data types (e.g. %hd means a short integer). The %e, %f, and %g type specifiers can have the letter l before them to indicate that a double follows. The %g, %f, and %e type specifiers can be preceded with the character '#' to ensure that the decimal point will be present, even if there are no decimal digits. The use of the '#' character with the %x type specifier indicates that the hexidecimal number should be printed with the '0x' prefix. The use of the '#' character with the %o type specifier indicates that the octal value should be displayed with a 0 prefix.

Inserting a plus sign '+' into the type specifier will force positive values to be preceded by a '+' sign. Putting a space character ' ' there will force positive values to be preceded by a single space character.

You can also include constant escape sequences in the output string.

The return value of printf() is the number of characters printed, or a negative number if an error occurred.

Related topics:
fprintf puts
scanf
sprintf

### putc

Syntax:

```
#include <cstdio>
int putc( int ch, FILE *stream );
```

The putc() function writes the character *ch* to *stream*. The return value is the character written, or **EOF** if there is an error. For example:

```
int ch;
FILE *input, *output;
input = fopen( "tmp.c", "r" );
output = fopen( "tmpCopy.c", "w" );
ch = getc( input );
while( ch != EOF ) {
   putc( ch, output );
   ch = getc( input );
}
fclose( input );
fclose( output );
```

generates a copy of the file tmp.c called tmpCopy.c.

Related topics:

feof fflush fgetc fputc getc getchar putchar puts cppreference.com > Standard C I/O > putchar

# putchar

Syntax:

```
#include <cstdio>
int putchar( int ch );
```

The putchar() function writes *ch* to **stdout**. The code

```
putchar( ch );
```

is the same as

```
putc( ch, stdout );
```

The return value of putchar() is the written character, or **EOF** if there is an error.

Related topics:

**putc** 

cppreference.com > Standard C I/O > puts

## puts

Syntax:

```
#include <cstdio>
int puts( char *str );
```

The function puts() writes *str* to **stdout**. puts() returns non-negative on success, or **EOF** on failure.

Related topics:

fputs gets printf putc

### qsort

*Syntax:* 

```
#include <cstdlib>
void qsort( void *buf, size_t num, size_t size, int (*compare)(co
```

The qsort() function sorts *buf* (which contains *num* items, each of size *size*) using Quicksort. The *compare* function is used to compare the items in *buf. compare* should return negative if the first argument is less than the second, zero if they are equal, and positive if the first argument is greater than the second. qsort() sorts *buf* in ascending order.

#### Example code:

For example, the following bit of code uses qsort() to sort an array of integers:

```
int \overline{\text{compare\_ints(const void* a, const void* b)}}
  int* arg1 = (int*) a;
  int* arg2 = (int*) b;
  if( *arg1 < *arg2 ) return -1;
  else if( *arg1 == *arg2 ) return 0;
  else return 1;
}
int array[] = \{ -2, 99, 0, -743, 2, 3, 4 \};
int array_size = 7;
. . .
printf( "Before sorting: " );
for( int i = 0; i < array_size; i++ ) {
 printf( "%d ", array[i] );
printf( "\n" );
qsort( array, array_size, sizeof(int), compare_ints );
printf( "After sorting: " );
for( int i = 0; i < array_size; i++ ) {
  printf( "%d ", array[i] );
```

```
}
printf( "\n" );
```

When run, this code displays the following output:

```
Before sorting: -2 99 0 -743 2 3 4
After sorting: -743 -2 0 2 3 4 99
```

Related topics:

bsearch (C++ Algorithms) sort

cppreference.com > Other Standard C Functions > raise

## raise

Syntax:

```
#include <csignal>
int raise( int signal );
```

The raise() function sends the specified *signal* to the program. Some signals:

Signal	Meaning
SIGABRT	Termination error
SIGFPE	Floating pointer error
SIGILL	Bad instruction
SIGINT	User presed CTRL-C
SIGSEGV	Illegal memory access
SIGTERM	Terminate program

The return value is zero upon success, nonzero on failure.

Related topics:

<u>signal</u>

cppreference.com > Other Standard C Functions > rand

### rand

Syntax:

```
#include <cstdlib>
int rand( void );
```

The function rand() returns a pseudorandom integer between zero and RAND\_MAX. An example:

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
printf( "Random number #%d: %d\n", i, rand() );
```

Related topics:

**srand** 

cppreference.com > Standard C Memory > realloc

### realloc

Syntax:

```
#include <cstdlib>
void *realloc( void *ptr, size_t size );
```

The realloc() function changes the size of the object pointed to by ptr to the given size. size can be any size, larger or smaller than the original. The return value is a pointer to the new space, or **NULL** if there is an error.

Related topics:

calloc free malloc

cppreference.com > Standard C I/O > remove

### remove

Syntax:

```
#include <cstdio>
int remove( const char *fname );
```

The remove() function erases the file specified by *fname*. The return value of remove() is zero upon success, and non-zero if there is an error.

Related topics:

<u>rename</u>

cppreference.com > Standard C I/O > rename

### rename

Syntax:

```
#include <cstdio>
int rename( const char *oldfname, const char *newfname );
```

The function rename() changes the name of the file *oldfname* to *newfname*. The return value of rename() is zero upon success, non-zero on error.

Related topics:

**remove** 

cppreference.com > Standard C I/O > rewind

## rewind

Syntax:

```
#include <cstdio>
void rewind( FILE *stream );
```

The function rewind() moves the file position indicator to the beginning of the specified *stream*, also clearing the error and **EOF** flags associated with that stream.

Related topics:

**fseek** 

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>scanf</u>

### scanf

Syntax:

```
#include <cstdio>
int scanf( const char *format, ... );
```

The scanf() function reads input from **stdin**, according to the given *format*, and stores the data in the other arguments. It works a lot like <u>printf()</u>. The *format* string consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are as follows:

Control Character	Explanation
%с	a single character
%d	a decimal integer
%i	an integer
%e, %f, %g	a floating-point number
%lf	a double
%0	an octal number
%s	a string
%x	a hexadecimal number
%p	a pointer
%n	an integer equal to the number of characters read so far
%u	an unsigned integer
%[]	a set of characters
%%	a percent sign

scanf() reads the input, matching the characters from format. When a control character is read, it puts the value in the next variable. Whitespace (tabs, spaces,

etc) are skipped. Non-whitespace characters are matched to the input, then discarded. If a number comes between the % sign and the control character, then only that many characters will be converted into the variable. If scanf() encounters a set of characters, denoted by the %[] control character, then any characters found within the brackets are read into the variable. The return value of scanf() is the number of variables that were successfully assigned values, or **EOF** if there is an error.

#### Example code:

This code snippet uses scanf() to read an int, float, and a double from the user. Note that the variable arguments to scanf() are passed in by address, as denoted by the ampersand (&) preceding each variable:

```
int i;
float f;
double d;

printf( "Enter an integer: " );
scanf( "%d", &i );

printf( "Enter a float: " );
scanf( "%f", &f );

printf( "Enter a double: " );
scanf( "%lf", &d );

printf( "You entered %d, %f, and %f\n", i, f, d );
```

#### Related topics:

fgets fscanf printf sscanf cppreference.com > Standard C I/O > setbuf

## setbuf

Syntax:

```
#include <cstdio>
void setbuf( FILE *stream, char *buffer );
```

The setbuf() function sets *stream* to use *buffer*, or, if *buffer* is null, turns off buffering. If a non-standard buffer size is used, it should be BUFSIZ characters long.

Related topics:

fclose fopen setvbuf

#### cppreference.com > Other Standard C Functions > setjmp

# setjmp

Syntax:

```
#include <csetjmp>
int setjmp( jmp_buf envbuf );
```

The setjmp() function saves the system stack in *envbuf* for use by a later call to <a href="longjmp">longjmp</a>(). When you first call setjmp(), its return value is zero. Later, when you call <a href="longjmp">longjmp</a>(), the second argument of <a href="longjmp">longjmp</a>() is what the return value of setjmp() will be. Confused? Read about <a href="longjmp">longjmp</a>().

Related topics:

**longjmp** 

cppreference.com > Standard C Date & Time > setlocale

### setlocale

Syntax:

```
#include <clocale>
char *setlocale( int category, const char * locale );
```

The setlocale() function is used to set and retrieve the current locale. If *locale* is **NULL**, the current locale is returned. Otherwise, *locale* is used to set the locale for the given *category*.

category can have the following values:

Value	Description
LC_ALL	All of the locale
LC_TIME	Date and time formatting
LC_NUMERIC	Number formatting
LC_COLLATE	String collation and regular expression matching
LC_CTYPE	Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.
LC_MONETARY	For monetary formatting
LC_MESSAGES	For natural language messages

*Related topics:* 

(Standard C String and Character) strcoll

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>setvbuf</u>

## setvbuf

Syntax:

```
#include <cstdio>
int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
```

The function setvbuf() sets the buffer for *stream* to be *buffer*, with a size of *size*. *mode* can be:

- \_IOFBF, which indicates full buffering
- \_IOLBF, which means line buffering
- \_IONBF, which means no buffering

Related topics:

**setbuf** 

#### cppreference.com > Other Standard C Functions > signal

# signal

#### Syntax:

```
#include <csignal>
void ( *signal( int signal, void (* func) (int)) ) (int);
```

The signal() function sets *func* to be called when *signal* is recieved by your program. *func* can be a custom signal handler, or one of these macros (defined in the csignal header file):

Macro	Explanation
SIG_DFL	default signal handling
SIG_IGN	ignore the signal

Some basic signals that you can attach a signal handler to are:

Signal	Description
SIGTERM	Generic stop signal that can be caught.
SIGINT	Interrupt program, normally ctrl-c.
SIGQUIT	Interrupt program, similar to SIGINT.
SIGKILL	Stops the program. Cannot be caught.
SIGHUP	Reports a disconnected terminal.

The return value of signal() is the address of the previously defined function for this signal, or SIG\_ERR is there is an error.

#### Example code:

The following example uses the signal() function to call an arbitrary number of functions when the user aborts the program. The functions are stored in a vector,

and a single "clean-up" function calls each function in that vector of functions when the program is aborted:

```
void f1() {
  cout << "calling f1()..." << endl;</pre>
void f2() {
  cout << "calling f2()..." << endl;</pre>
typedef void(*endFunc)(void);
vector<endFunc> endFuncs;
void cleanUp( int dummy ) {
  for( unsigned int i = 0; i < endFuncs.size(); i++ ) {</pre>
    endFunc f = endFuncs.at(i);
    (*f)();
  exit(-1);
int main() {
  // connect various signals to our clean-up function
  signal( SIGTERM, cleanUp );
  signal( SIGINT, cleanUp );
  signal( SIGQUIT, cleanUp');
  signal( SIGHUP, cleanUp );
  // add two specific clean-up functions to a list of functions
  endFuncs.push_back( f1 );
  endFuncs.push_back( f2 );
  // loop until the user breaks
  while( 1 );
  return 0;
```

Related topics:

<u>raise</u>

#### cppreference.com > Standard C Math > sin

### sin

#### Syntax:

```
#include <cmath>
double sin( double arg );
```

The function sin() returns the sine of *arg*, where *arg* is given in radians. The return value of sin() will be in the range [-1,1]. If *arg* is infinite, sin() will return NAN and raise a floating-point exception.

#### Related topics:

acos asin

atan

atan2

**COS** 

cosh

<u>sinh</u>

<u>tan</u>

tanh

### cppreference.com > Standard C Math > sinh

# sinh

#### Syntax:

```
#include <cmath>
double sinh( double arg );
```

The function sinh() returns the hyperbolic sine of *arg*.

#### Related topics:

acos asin

<u>atan</u>

atan2

<u>**COS**</u>

**cosh** 

<u>sin</u>

<u>tan</u>

**tanh** 

```
<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>sprintf</u>
```

# sprintf

Syntax:

```
#include <cstdio>
int sprintf( char *buffer, const char *format, ... );
```

The sprintf() function is just like <u>printf(</u>), except that the output is sent to *buffer*. The return value is the number of characters written. For example:

```
char string[50];
int file_number = 0;
sprintf( string, "file.%d", file_number );
file_number++;
output_file = fopen( string, "w" );
```

Note that sprintf() does the opposite of a function like <u>atoi()</u> -- where <u>atoi()</u> converts a string into a number, sprintf() can be used to convert a number into a string.

For example, the following code uses sprintf() to convert an integer into a string of characters:

```
char result[100];
int num = 24;
sprintf( result, "%d", num );
```

This code is similar, except that it converts a floating-point number into an array of characters:

```
char result[100];
float fnum = 3.14159;
sprintf( result, "%f", fnum );
```

Related topics:

```
(Standard C String and Character) <u>atof</u> (Standard C String and Character) <u>atoi</u>
```

(Standard C String and Character) atol fprintf printf

### cppreference.com > Standard C Math > sqrt

## sqrt

*Syntax:* 

```
#include <cmath>
double sqrt( double num );
```

The sqrt() function returns the square root of *num*. If *num* is negative, a domain error occurs.

Related topics:

exp log pow cppreference.com > Other Standard C Functions > srand

### srand

Syntax:

```
#include <cstdlib>
void srand( unsigned seed );
```

The function srand() is used to seed the random sequence generated by <u>rand()</u>. For any given *seed*, rand() will generate a specific "random" sequence over and over again.

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );</pre>
```

*Related topics:* 

rand (Standard C Date & Time) time

cppreference.com > Standard C I/O > sscanf

## sscanf

Syntax:

```
#include <cstdio>
int sscanf( const char *buffer, const char *format, ... );
```

The function sscanf() is just like <u>scanf()</u>, except that the input is read from *buffer*.

Related topics:

**fscanf scanf** 

<u>cppreference.com</u> > <u>Standard C String and Character</u> > <u>strcat</u>

### strcat

*Syntax:* 

```
#include <cstring>
char *strcat( char *str1, const char *str2 );
```

The strcat() function concatenates str2 onto the end of str1, and returns str1. For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
title = strcat( name, " the Great" );
printf( "Hello, %s\n", title );
```

Note that strcat() does not perform bounds checking, and thus risks overrunning str1 or str2. For a similar (and safer) function that includes bounds checking, see  $\underline{strncat()}$ .

Related topics: <a href="mailto:strchr">strchr</a> <a href="mailto:strchr">strcmp</a>

strcpy

**strncat** 

Another set of related (but non-standard) functions are strlcpy and strlcat.

cppreference.com > Standard C String and Character > strchr

## strchr

#### Syntax:

```
#include <cstring>
char *strchr( const char *str, int ch );
```

The function strchr() returns a pointer to the first occurence of *ch* in *str*, or **NULL** if *ch* is not found.

#### Related topics:

strcat strcmp

**strcpy** 

<u>strlen</u>

**strncat** 

**strncmp** 

**strncpy** 

**strpbrk** 

strspn

<u>strstr</u>

**strtok** 

cppreference.com > Standard C String and Character > strcmp

## strcmp

#### Syntax:

```
#include <cstring>
int strcmp( const char *str1, const char *str2 );
```

The function strcmp() compares *str1* and *str2*, then returns:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than "str2"

#### For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 ) {
   printf( "Hello, Dr. Mary!\n" );
}
```

Note that if str1 or str2 are missing a null-termination character, then strcmp() may not produce valid results. For a similar (and safer) function that includes explicit bounds checking, see strncmp().

#### Related topics:

```
memcmp strcat
strchr
strcoll
strcpy
strlen
strncmp
strxfrm
```

cppreference.com > Standard C String and Character > strcoll

### strcoll

Syntax:

```
#include <cstring>
int strcoll( const char *str1, const char *str2 );
```

The strcoll() function compares *str1* and *str2*, much like <u>strcmp</u>(). However, strcoll() performs the comparison using the locale specified by the (Standard C Date & Time) <u>setlocale()</u> function.

Related topics:

(Standard C Date & Time) setlocale strcmp strxfrm

cppreference.com > Standard C String and Character > strcpy

## strcpy

Syntax:

```
#include <cstring>
char *strcpy( char *to, const char *from );
```

The strcpy() function copies characters in the string *from* to the string *to*, including the null termination. The return value is *to*.

Note that strcpy() does not perform bounds checking, and thus risks overrunning *from* or *to*. For a similar (and safer) function that includes bounds checking, see <a href="strncpy">strncpy()</a>.

Related topics:
memcpy strcat
strchr
strcmp
strncmp
strncpy

Another set of related (but non-standard) functions are strlcpy and strlcat.

cppreference.com > Standard C String and Character > strcspn

## strcspn

Syntax:

```
#include <cstring>
size_t strcspn( const char *str1, const char *str2 );
```

The function strcspn() returns the index of the first character in *str1* that matches any of the characters in *str2*.

Related topics:
strpbrk strrchr
strstr
strtok

cppreference.com > Standard C String and Character > strerror

### strerror

Syntax:

```
#include <cstring>
char *strerror( int num );
```

The function strerror() returns an implementation defined string corresponding to *num*.

### strftime

#### Syntax:

```
#include <ctime>
size_t strftime( char *str, size_t maxsize, const char *fmt, stru
```

The function strftime() formats date and time information from *time* to a format specified by *fmt*, then stores the result in *str* (up to *maxsize* characters). Certain codes may be used in *fmt* to specify different types of time:

Code	Meaning
%a	abbreviated weekday name (e.g. Fri)
%A	full weekday name (e.g. Friday)
%b	abbreviated month name (e.g. Oct)
%B	full month name (e.g. October)
%с	the standard date and time string
%d	day of the month, as a number (1-31)
%Н	hour, 24 hour format (0-23)
%I	hour, 12 hour format (1-12)
%j	day of the year, as a number (1-366)
%m	month as a number (1-12). Note: some versions of Microsoft Visual C++ may use values that range from 0-11.
%M	minute as a number (0-59)
%p	locale's equivalent of AM or PM
%S	second as a number (0-59)
%U	week of the year, (0-53), where week 1 has the first Sunday
%w	weekday as a decimal (0-6), where Sunday is 0

%W	week of the year, (0-53), where week 1 has the first Monday
%x	standard date string
%X	standard time string
%y	year in decimal, without the century (0-99)
%Y	year in decimal, with the century
%Z	time zone name
%%	a percent sign

The strftime() function returns the number of characters put into *str*, or zero if an error occurs.

Related topics:
gmtime localtime
time

cppreference.com > Standard C String and Character > strlen

## strlen

*Syntax:* 

```
#include <cstring>
size_t strlen( char *str );
```

The strlen() function returns the length of *str* (determined by the number of characters before null termination).

Related topics:

memcpy strchr strcmp strncmp cppreference.com > Standard C String and Character > strncat

### strncat

#### Syntax:

```
#include <cstring>
char *strncat( char *str1, const char *str2, size_t count );
```

The function strncat() concatenates at most *count* characters of *str2* onto *str1*, adding a null termination. The resulting string is returned.

Related topics:

strcat strchr strncmp strncpy

Another set of related (but non-standard) functions are strlcpy and strlcat.

cppreference.com > Standard C String and Character > strncmp

## strncmp

#### Syntax:

```
#include <cstring>
int strncmp( const char *str1, const char *str2, size_t count );
```

The strncmp() function compares at most *count* characters of *str1* and *str2*. The return value is as follows:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than str2"

If there are less than *count* characters in either string, then the comparison will stop after the first null termination is encountered.

#### Related topics:

strchr strcmp

**strcpy** 

<u>strlen</u>

<u>strncat</u>

strncpy

cppreference.com > Standard C String and Character > strncpy

## strncpy

#### Syntax:

```
#include <cstring>
char *strncpy( char *to, const char *from, size_t count );
```

The strncpy() function copies at most *count* characters of *from* to the string *to*. If from has less than *count* characters, the remainder is padded with '\0' characters. The return value is the resulting string.

#### *Related topics:*

memcpy strchr strcpy strncat strncmp

Another set of related (but non-standard) functions are strlcpy and strlcat.

cppreference.com > Standard C String and Character > strpbrk

# strpbrk

Syntax:

```
#include <cstring>
char* strpbrk( const char* str1, const char* str2 );
```

The function strpbrk() returns a pointer to the first ocurrence in *str1* of any character in *str2*, or **NULL** if no such characters are present.

#### Related topics:

```
(C++ Algorithms) find_first_of
strchr
strcspn
strrchr
strspn
strstr
strtok
```

cppreference.com > Standard C String and Character > strrchr

## strrchr

Syntax:

```
#include <cstring>
char *strrchr( const char *str, int ch );
```

The function strrchr() returns a pointer to the last occurrence of *ch* in *str*, or **NULL** if no match is found.

Related topics:

strcspn strpbrk

strspn

<u>strstr</u>

**strtok** 

cppreference.com > Standard C String and Character > strspn

# strspn

Syntax:

```
#include <cstring>
size_t strspn( const char *str1, const char *str2 );
```

The strspn() function returns the index of the first character in *str1* that doesn't match any character in *str2*.

Related topics: strchr strpbrk strrchr strstr strtok <u>cppreference.com</u> > <u>Standard C String and Character</u> > <u>strstr</u>

### strstr

Syntax:

```
#include <cstring>
char *strstr( const char *str1, const char *str2 );
```

The function strstr() returns a pointer to the first occurrence of *str2* in *str1*, or **NULL** if no match is found. If the length of *str2* is zero, then strstr() will simply return *str1*.

For example, the following code checks for the existence of one string within another string:

```
char* str1 = "this is a string of characters";
char* str2 = "a string";
char* result = strstr( str1, str2 );
if( result == NULL ) printf( "Could not find '%s' in '%s'\n", str2
else printf( "Found a substring: '%s'\n", result );
```

When run, the above code displays this output:

```
Found a substring: 'a string of characters'
```

Related topics:

memchr strchr

**strcspn** 

**strpbrk** 

strrchr

strspn

strtok

cppreference.com > Standard C String and Character > strtod

### strtod

Syntax:

```
#include <cstdlib>
double strtod( const char *start, char **end );
```

The function strtod() returns whatever it encounters first in *start* as a double. *end* is set to point at whatever is left in *start* after that double. If overflow occurs, strtod() returns either **HUGE\_VAL** or **-HUGE\_VAL**.

Related topics:

atof

### strtok

*Syntax:* 

```
#include <cstring>
char *strtok( char *str1, const char *str2 );
```

The strtok() function returns a pointer to the next "token" in str1, where str2 contains the delimiters that determine the token. strtok() returns **NULL** if no token is found. In order to convert a string to tokens, the first call to strtok() should have str1 point to the string to be tokenized. All calls after this should have str1 be **NULL**.

For example:

```
char str[] = "now # is the time for all # good men to come to the
char delims[] = "#";
char *result = NULL;
result = strtok( str, delims );
while( result != NULL ) {
    printf( "result is \"%s\"\n", result );
    result = strtok( NULL, delims );
}
```

The above code will display the following output:

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

Related topics:

strchr strcspn strpbrk strrchr strspn strstr cppreference.com > Standard C String and Character > strtol

### strtol

Syntax:

```
#include <cstdlib>
long strtol( const char *start, char **end, int base );
```

The strtol() function returns whatever it encounters first in *start* as a long, doing the conversion to *base* if necessary. *end* is set to point to whatever is left in *start* after the long. If the result can not be represented by a long, then strtol() returns either **LONG\_MAX** or **LONG\_MIN**. Zero is returned upon error.

Related topics:

atol strtoul

cppreference.com > Standard C String and Character > strtoul

### strtoul

Syntax:

```
#include <cstdlib>
unsigned long strtoul( const char *start, char **end, int base );
```

The function strtoul() behaves exactly like <u>strtol()</u>, except that it returns an unsigned long rather than a mere long.

Related topics:

<u>strtol</u>

cppreference.com > Standard C String and Character > strxfrm

### strxfrm

Syntax:

```
#include <cstring>
size_t strxfrm( char *str1, const char *str2, size_t num );
```

The strxfrm() function manipulates the first num characters of str2 and stores them in str1. The result is such that if a strcoll() is performed on str1 and the old str2, you will get the same result as with a strcmp().

Related topics:

strcmp strcoll

cppreference.com > Other Standard C Functions > system

## system

Syntax:

```
#include <cstdlib>
int system( const char *command );
```

The system() function runs the given *command* by passing it to the default command interpreter.

The return value is usually zero if the command executed without errors. If *command* is **NULL**, system() will test to see if there is a command interpreter available. Non-zero will be returned if there is a command interpreter available, zero if not.

Related topics: <a href="mailto:exit getenv">exit getenv</a>

### cppreference.com > Standard C Math > tan

## tan

#### *Syntax:*

```
#include <cmath>
double tan( double arg );
```

The tan() function returns the tangent of *arg*, where *arg* is given in radians. If *arg* is infinite, tan() will return NAN and raise a floating-point exception.

### Related topics:

acos asin

atan

atan2

**COS** 

cosh

<u>sin</u>

<u>sinh</u>

tanh

### cppreference.com > Standard C Math > tanh

## tanh

### Syntax:

```
#include <cmath>
double tanh( double arg );
```

The function tanh() returns the hyperbolic tangent of *arg*.

### Related topics:

acos asin

<u>atan</u>

atan2

<u>**COS**</u>

**cosh** 

<u>sin</u>

<u>sinh</u>

<u>tan</u>

### cppreference.com > Standard C Date & Time > time

## time

Syntax:

```
#include <ctime>
time_t time( time_t *time );
```

The function time() returns the current time, or -1 if there is an error. If the argument 'time' is given, then the current time is stored in 'time'.

#### Related topics:

asctime clock
ctime
difftime
gmtime
localtime

mktime
(Other Standard C Functions) srand
strftime

cppreference.com > Standard C I/O > tmpfile

# tmpfile

*Syntax:* 

```
#include <cstdio>
FILE *tmpfile( void );
```

The function tmpfile() opens a temporary file with an unique filename and returns a pointer to that file. If there is an error, null is returned.

Related topics:

**tmpnam** 

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>tmpnam</u>

## tmpnam

*Syntax:* 

```
#include <cstdio>
char *tmpnam( char *name );
```

The tmpnam() function creates an unique filename and stores it in *name*. tmpnam() can be called up to **TMP\_MAX** times.

Related topics:

**tmpfile** 

cppreference.com > Standard C String and Character > tolower

## tolower

Syntax:

```
#include <cctype>
int tolower( int ch );
```

The function tolower() returns the lowercase version of the character *ch*.

Related topics:

isupper toupper

cppreference.com > Standard C String and Character > toupper

## toupper

Syntax:

```
#include <cctype>
int toupper( int ch );
```

The toupper() function returns the uppercase version of the character *ch*.

Related topics:

**tolower** 

<u>cppreference.com</u> > <u>Standard C I/O</u> > <u>ungetc</u>

# ungetc

Syntax:

```
#include <cstdio>
int ungetc( int ch, FILE *stream );
```

The function ungetc() puts the character *ch* back in *stream*.

Related topics:

getc (C++ I/O) putback

## va\_arg

#### Syntax:

```
#include <cstdarg>
type va_arg( va_list argptr, type );
void va_end( va_list argptr );
void va_start( va_list argptr, last_parm );
```

The va\_arg() macros are used to pass a variable number of arguments to a function.

- 1. First, you must have a call to va\_start() passing a valid **va\_list** and the mandatory first argument of the function. This first argument can be anything; one way to use it is to have it be an integer describing the number of parameters being passed.
- 2. Next, you call va\_arg() passing the **va\_list** and the type of the argument to be returned. The return value of va\_arg() is the current parameter.
- 3. Repeat calls to va\_arg() for however many arguments you have.
- 4. Finally, a call to va\_end() passing the **va\_list** is necessary for proper cleanup.

#### For example:

```
int sum( int num, ... ) {
   int answer = 0;
   va_list argptr;

  va_start( argptr, num );

  for( ; num > 0; num-- ) {
     answer += va_arg( argptr, int );
  }

  va_end( argptr );

  return( answer );
}
```

```
int main( void ) {
  int answer = sum( 4, 4, 3, 2, 1 );
  printf( "The answer is %d\n", answer );
  return( 0 );
}
```

This code displays 10, which is 4+3+2+1.

Here is another example of variable argument function, which is a simple printing function:

```
void my_printf( char *format, ... ) {
  va_list argptr;
 va_start( argptr, format );
  while( *format != '\0' ) {
    // string
    if( *format == 's' ) {
      char* s = va_arg( argptr, char * );
      printf( "Printing a string: %s\n", s );
    // character
    else if( *format == 'c' ) {
      char c = (char) va_arg( argptr, int );
      printf( "Printing a character: %c\n", c );
      break;
    }
    // integer
    else if( *format == 'd' ) {
      int d = va_arg( argptr, int );
      printf( "Printing an integer: %d\n", d );
    *format++;
  va_end( argptr );
int main( void ) {
  my_printf( "sdc", "This is a string", 29, 'X' );
  return( 0 );
```

}

This code displays the following output when run:

Printing a string: This is a string Printing an integer: 29 Printing a character: X

# vprintf, vfprintf, and vsprintf

Syntax:

```
#include <cstdarg>
#include <cstdio>
int vprintf( char *format, va_list arg_ptr );
int vfprintf( FILE *stream, const char *format, va_list arg_ptr )
int vsprintf( char *buffer, char *format, va_list arg_ptr );
```

These functions are very much like <a href="mailto:printf">printf</a>(), <a href="mailto:fprintf">fprintf</a>(), and <a href="mailto:sprintf">sprintf</a>(). The difference is that the argument list is a pointer to a list of arguments. <a href="mailto:va\_list">va\_list</a> is defined in cstdarg, and is also used by (Other Standard C Functions) <a href="mailto:va\_arg">va\_arg</a>(). For example:

```
void error( char *fmt, ... ) {
   va_list args;
   va_start( args, fmt );
   fprintf( stderr, "Error: " );
   vfprintf( stderr, fmt, args );
   fprintf( stderr, "\n" );
   va_end( args );
   exit( 1 );
}
```

# C/C++ Reference

### General C/C++

- <u>Pre-processor commands</u>
- Operator Precedence
- Escape Sequences
- ASCII Chart
- Data Types Keywords

## **Standard C Library**

- Standard C I/O
- Standard C String & Character
- Standard C Math
- Standard C Time & Date
- Standard C Memory
- Other standard C functions

All C Functions

### $\mathbb{C}$ ++

- <u>C++ I/O</u>
- <u>C++ Strings</u>
- C++ String StreamsMiscellaneous C++

## C++ Standard Template Library

- About the Standard Template Library
- <u>Iterators</u>
- C++ Algorithms
- C++ Vectors
- C++ Double-Ended Queues
- C++ Lists
- <u>C++ Priority Queues</u>
- C++ Queues
- C++ Stacks
- <u>C++ Sets</u>
- C++ Multisets
- <u>C++ Maps</u>
- <u>C++ Multimaps</u>
- C++ Bitsets

All C++ Functions

Questions? Check out <u>the FAQ</u>, look at these <u>other language references</u>, or contact us.

Last modified on 3/18/2008 by Nate Kohl, with help from a <u>lot of people</u>. Windows compliled help (CHM) file created by James Brown.

# **String Stream Constructors**

Syntax:

```
#include <sstream>
  stringstream()
  stringstream( openmode mode )
  stringstream( string s, openmode mode )
  ostringstream()
  ostringstream( openmode mode )
  ostringstream( string s, openmode mode )
  istringstream()
  istringstream( openmode mode )
  istringstream( string s, openmode mode )
```

The stringstream, ostringstream, and istringstream objects are used for input and output to a string. They behave in a manner similar to fstream, ofstream and ifstream objects.

The optional *mode* parameter defines how the file is to be opened, according to the <u>io stream mode flags</u>.

An ostringstream object can be used to write to a string. This is similar to the C sprintf() function. For example:

```
ostringstream s1;
int i = 22;
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;</pre>
```

An istringstream object can be used to read from a string. This is similar to the C sscanf() function. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25</pre>
```

You can also specify the input string in the istringstream constructor as in this example:

```
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25</pre>
```

A stringstream object can be used for both input and output to a string like an fstream object.

Related topics:

C++ I/O Streams

<u>cppreference.com</u> > <u>C++ String Streams</u> > <u>Operators</u>

# **String Stream Operators**

Syntax:

```
#include <sstream>
  operator<<
  operator>>
```

Like <u>C++ I/O Streams</u>, the simplest way to use string streams is to take advantage of the overloaded << and >> operators.

The << operator inserts data into the stream. For example:

```
stream1 << "hello" << i;
```

This example inserts the string "hello" and the variable i into stream1. In contrast, the >> operator extracts data out of a string stream:

```
stream1 >> i;
```

This code reads a value from stream1 and assigns the variable i that value.

Related topics:

C++ I/O Streams

cppreference.com > C++ String Streams > rdbuf

# rdbuf

Syntax:

```
#include <sstream>
stringbuf* rdbuf();
```

The *rdbuf()* function returns a pointer to the string buffer for the current string stream.

Related topics: <a href="mailto:str()">str()</a> C++ I/O Streams

```
<u>cppreference.com</u> > <u>C++ String Streams</u> > <u>str</u>
```

### str

Syntax:

```
#include <sstream>
void str( string s );
string str();
```

The function *str()* can be used in two ways. First, it can be used to get a copy of the string that is being manipulated by the current stream string. This is most useful with output strings. For example:

```
ostringstream stream1;
stream1 << "Testing!" << endl;
cout << stream1.str();</pre>
```

Second, *str()* can be used to copy a string into the stream. This is most useful with input strings. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
```

*str()*, along with *clear()*, is also handy when you need to clear the stream so that it can be reused:

```
istringstream stream1;
float num;

// use it once
string string1 = "25 1 3.235\n1111111\n222222";
stream1.str(string1);
while( stream1 >> num ) cout << "num: " << num << endl; // displa

// use the same string stream again with clear() and str()
string string2 = "1 2 3 4 5 6 7 8 9 10";
stream1.clear();
stream1.str(string2);

while( stream1 >> num ) cout << "num: " << num << endl; // displa</pre>
```

Related topics:
rdbuf() C++ I/O Streams

```
cppreference.com > C/C++ Keywords > asm
```

### asm

Syntax:

```
asm( "instruction" );
```

The asm command allows you to insert assembly language commands directly into your code. Various different compilers allow differing forms for this command, such as

```
asm {
   instruction-sequence
}
```

or

```
asm( instruction );
```

### cppreference.com > C/C++ Keywords > auto

## auto

The keyword auto is used to declare local variables, and is purely optional.

Related topics:

<u>register</u>

## bool

The keyword bool is used to declare Boolean logic variables; that is, variables which can be either true or false.

For example, the following code declares a boolean variable called *done*, initializes it to false, and then loops until that variable is set to true.

```
bool done = false;
while( !done ) {
   ...
}
```

Also see the data types page.

Related topics:

char double

false

float

int

long

short

signed

true

unsigned

wchar\_t

## break

The break keyword is used to break out of a <u>do</u>, <u>for</u>, or <u>while</u> loop. It is also used to finish each clause of a <u>switch</u> statement, keeping the program from "falling through" to the next case in the code. An example:

```
while( x < 100 ) {
   if( x < 0 )
     break;
   cout << x << endl;
   x++;
}</pre>
```

A given break statement will break out of only the closest loop, no further. If you have a triply-nested for loop, for example, you might want to include extra logic or a <u>goto</u> statement to break out of the loop.

Related topics:

continue do for goto switch while

### cppreference.com > C/C++ Keywords > case

### case

The case keyword is used to test a variable against a certain value in a <u>switch</u> statement.

Related topics: default switch

### cppreference.com > C/C++ Keywords > catch

# catch

The catch statement handles exceptions generated by the **throw** statement.

Related topics:

throw try

### cppreference.com > C/C++ Keywords > char

## char

The char keyword is used to declare character variables. For more information about variable types, see the <u>data types</u> page.

#### Related topics:

**bool** double

float

int

long

<u>short</u>

<u>signed</u>

unsigned

<u>void</u>

wchar\_t

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>class</u>
```

### class

Syntax:

```
class class-name : inheritance-list {
  private-members-list;
  protected:
  protected-members-list;
  public:
  public-members-list;
} object-list;
```

The class keyword allows you to create new classes. *class-name* is the name of the class that you wish to create, and *inheritance-list* is an optional list of classes inherited by the new class. Members of the class are private by default, unless listed under either the protected or public labels. *object-list* can be used to immediately instantiate one or more instances of the class, and is also optional. For example:

```
class Date {
   int Day;
   int Month;
   int Year;
   public:
    void display();
};
```

Related topics:

friend private
protected
public
struct
this
typename
union
virtual

### cppreference.com > C/C++ Keywords > const

## const

The const keyword can be used to tell the compiler that a certain variable should not be modified once it has been initialized.

It can also be used to declare functions of a class that do not alter any class data.

Related topics:

const cast mutable

#### cppreference.com > C/C++ Keywords > const\_cast

## const\_cast

Syntax:

```
const_cast<type> (object);
```

The const\_cast keyword can be used to remove the **const** or **volatile** property from some variable. The target data type must be the same as the source type, except (of course) that the target type doesn't have to be <u>const</u>.

Related topics:

const dynamic\_cast
reinterpret\_cast
static\_cast

## continue

The continue statement can be used to bypass iterations of a given loop.

For example, the following code will display all of the numbers between 0 and 20 except 10:

```
for( int i = 0; i < 21; i++ ) {
   if( i == 10 ) {
     continue;
   }
   cout << i << " ";
}</pre>
```

Related topics:

<u>break</u> <u>do</u>

<u>for</u>

<u>while</u>

### <u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>default</u>

# default

A default <u>case</u> in the <u>switch</u> statement.

Related topics: <a href="mailto:case">case</a> <a href="mailto:switch">switch</a>

<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>delete</u>

### delete

Syntax:

```
delete p;
delete[] pArray;
```

The delete operator frees the memory pointed to by p. The argument should have been previously allocated by a call to  $\underline{\text{new}}$ . The second form of delete should be used to delete an array.

Related topics:

(Standard C Memory) <u>free</u> (Standard C Memory) <u>malloc</u> <u>new</u>

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>do</u>
```

### do

Syntax:

```
do {
  statement-list;
} while( condition );
```

The do construct evaluates the given *statement-list* repeatedly, until *condition* becomes false. Note that every do loop will evaluate its statement list at least once, because the terminating condition is tested at the end of the loop.

Related topics:

break continue for while

#### cppreference.com > C/C++ Keywords > double

## double

The double keyword is used to declare double precision floating-point variables. Also see the <u>data types</u> page.

#### Related topics:

**bool** char

float

int

long

**short** 

**signed** 

unsigned

<u>void</u>

wchar\_t

cppreference.com > C/C++ Keywords > dynamic\_cast

# dynamic\_cast

Syntax:

```
dynamic_cast<type> (object);
```

The dynamic\_cast keyword casts a datum from one type to another, performing a runtime check to ensure the validity of the cast. If you attempt to cast between incompatible types, the result of the cast will be **NULL**.

Related topics:

const cast reinterpret cast
static\_cast

#### <u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>else</u>

# else

The else keyword is used as an alternative case for the **if** statement.

Related topics:

<u>if</u>

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>enum</u>
```

#### enum

Syntax:

```
enum name {name-list} var-list;
```

The enum keyword is used to create an enumerated type named name that consists of the elements in *name-list*. The *var-list* argument is optional, and can be used to create instances of the type along with the declaration. For example, the following code creates an enumerated type for colors:

```
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
...
ColorT c1 = indigo;
if( c1 == indigo ) {
   cout << "c1 is indigo" << endl;
}</pre>
```

In the above example, the effect of the enumeration is to introduce several new constants named *red*, *orange*, *yellow*, etc. By default, these constants are assigned consecutive integer values starting at zero. You can change the values of those constants, as shown by the next example:

```
enum ColorT { red = 10, blue = 15, green };
...
ColorT c = green;
cout << "c is " << c << endl;</pre>
```

When executed, the above code will display the following output:

```
c is 16
```

Note that the above examples will only work with C++ compilers. If you're working in regular C, you will need to specify the *enum* keyword whenever you create an instance of an enumerated type:

```
enum ColorT { red = 10, blue = 15, green };
...
```

```
enum ColorT c = green; // note the aditional enum keyword printf( "c is %d\n", c );
```

## explicit

When a constructor is specified as explicit, no automatic conversion will be used with that constructor -- but parameters passed to the constructor may still be converted. For example:

## export

The export keyword is intended to allow definitions of C++ templates to be separated from their declarations. While officially part of the C++ standard, the export keyword is only supported by a few compilers (such as the Comeau C++ compiler) and is not supported by such mainstream compilers as GCC and Visual C++.

### extern

The extern keyword is used to inform the compiler about variables declared outside of the current scope. Variables described by extern statements will not have any space allocated for them, as they should be properly defined elsewhere.

Extern statements are frequently used to allow data to span the scope of multiple files.

#### cppreference.com > C/C++ Keywords > false

# false

The Boolean value of "false".

Related topics:

**bool** true

#### cppreference.com > C/C++ Keywords > float

## float

The float keyword is used to declare floating-point variables. Also see the <u>data</u> <u>types</u> page.

#### Related topics:

**bool** char

**double** 

<u>int</u>

long

**short** 

**signed** 

unsigned

<u>void</u>

wchar\_t

<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>for</u>

### for

Syntax:

```
for( initialization; test-condition; increment ) {
  statement-list;
}
```

The for construct is a general looping mechanism consisting of 4 parts:

- 1. the initialization, which consists of 0 or more comma-delimited variable initialization statements
- 2. the test-condition, which is evaluated to determine if the execution of the for loop will continue
- 3. the increment, which consists of 0 or more comma-delimited statements that increment variables
- 4. and the statement-list, which consists of 0 or more statements that will be executed each time the loop is executed.

For example:

Related topics:

break continue do

<u>if</u> while

#### cppreference.com > C/C++ Keywords > friend

## friend

The friend keyword allows classes or functions not normally associated with a given class to have access to the private data of that class.

Related topics:

**class** 

<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>goto</u>

### goto

Syntax:

```
goto labelA;
...
labelA:
```

The goto statement causes the current thread of execution to jump to the specified label. While the use of the goto statement is generally <u>considered</u> <u>harmful</u>, it can occasionally be useful. For example, it may be cleaner to use a goto to break out of a deeply-nested <u>for</u> loop, compared to the space and time that extra <u>break</u> logic would consume.

*Related topics:* 

**break** 

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>if</u>
```

### if

Syntax:

```
if( conditionA ) {
   statement-listA;
}
else if( conditionB ) {
   statement-listB;
}
...
else {
   statement-listN;
}
```

The if construct is a branching mechanism that allows different code to execute under different conditions. The conditions are evaluated in order, and the statement-list of the first condition to evaluate to true is executed. If no conditions evaluate to true and an <u>else</u> statement is present, then the statement list within the else block will be executed. All of the <u>else</u> blocks are optional.

#### Related topics:

else for switch while

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>inline</u>
```

### inline

Syntax:

```
inline int functionA( int i ) {
   ...
}
```

The inline keyword requests that the compiler expand a given function in place, as opposed to inserting a call to that function. The inline keyword is a request, not a command, and the compiler is free to ignore it for whatever reason.

When a function declaration is included in a class definition, the compiler should try to automatically inline that function. No inline keyword is necessary in this case.

#### cppreference.com > C/C++ Keywords > int

## int

The int keyword is used to declare integer variables. Also see the <u>data types</u> page.

#### Related topics:

**bool** char

**double** 

**float** 

long

**short** 

**signed** 

unsigned

<u>void</u>

wchar\_t

#### cppreference.com > C/C++ Keywords > long

# long

The long keyword is a data type modifier that is used to declare long integer variables. For more information on long, see the <u>data types</u> page.

#### Related topics:

**bool** char

**double** 

float

int

**short** 

**signed** 

<u>void</u>

#### cppreference.com > C/C++ Keywords > mutable

## mutable

The mutable keyword overrides any enclosing <u>const</u> statement. A mutable member of a <u>const</u> object can be modified.

Related topics:

const

<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>namespace</u>

### namespace

Syntax:

```
namespace name {
declaration-list;
}
```

The namespace keyword allows you to create a new scope. The name is optional, and can be omitted to create an unnamed namespace. Once you create a namespace, you'll have to refer to it explicitly or use the <u>using</u> keyword.

Example code:

```
namespace CartoonNameSpace {
   int HomersAge;
   void incrementHomersAge() {
      HomersAge++;
   }
}
int main() {
   ...
   CartoonNameSpace::HomersAge = 39;
   CartoonNameSpace::incrementHomersAge();
   cout << CartoonNameSpace::HomersAge << endl;
   ...
}</pre>
```

Related topics:

using

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>new</u>
```

#### new

Syntax:

```
pointer = new type;
pointer = new type( initializer );
pointer = new type[size];
pointer = new( arg-list ) type...
```

The new operator (valid only in C++) allocates a new chunk of memory to hold a variable of type *type* and returns a pointer to that memory. An optional initializer can be used to initialize the memory. Allocating arrays can be accomplished by providing a *size* parameter in brackets.

The optional *arg-list* parameter can be used with any of the other formats to pass a variable number of arguments to an overloaded version of new(). For example, the following code shows how the new() function can be overloaded for a class and then passed arbitrary arguments:

```
class Base {
public:
    Base() { }

    void *operator new( unsigned int size, string str ) {
        cout << "Logging an allocation of " << size << " bytes for new
        return malloc( size );
    }

    int var;
    double var2;
};

...

Base* b = new ("Base instance 1") Base;</pre>
```

If an int is 4 bytes and a double is 8 bytes, the above code generates the following output when run:

Logging an allocation of 12 bytes for new object 'Base instance 1

Related topics:
<a href="mailto:delete">delete</a> (Standard C Memory) free
(Standard C Memory) malloc

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>operator</u>
```

### operator

Syntax:

```
return-type class-name::operator#(parameter-list) {
    ...
}
return-type operator#(parameter-list) {
    ...
}
```

The operator keyword is used to overload operators. The sharp sign (#) listed above in the syntax description represents the operator which will be overloaded. If part of a class, the *class-name* should be specified. For unary operators, *parameter-list* should be empty, and for binary operators, *parameter-list* should contain the operand on the right side of the operator (the operand on the left side is passed as <a href="mailto:this">this</a>).

For the non-member operator overload function, the operand on the left side should be passed as the first parameter and the operand on the right side should be passed as the second parameter.

You cannot overload the #, ##, ., :, .\*, or ? tokens.

Related topics:

this

#### cppreference.com > C/C++ Keywords > private

## private

Private data of a class can only be accessed by members of that class, except when <u>friend</u> is used. The <u>private</u> keyword can also be used to inherit a base class privately, which causes all <u>public</u> and <u>protected</u> members of the base class to become private members of the derived class.

Related topics: class protected public

## protected

Protected data are private to their own class but can be inherited by derived classes. The protected keyword can also be used as an inheritance specifier, which causes all <u>public</u> and protected members of the base class to become protected members of the derived class.

Related topics: class private public

#### cppreference.com > C/C++ Keywords > public

# public

Public data in a class are accessible to everyone. The public keyword can also be used as an inheritance specifier, which causes all public and <u>protected</u> members of the base class to become public and protected members of the derived class.

Related topics:

class private protected

#### cppreference.com > C/C++ Keywords > register

# register

The register keyword requests that a variable be optimized for speed, and fell out of common use when computers became better at most code optimizations than humans.

Related topics:

<u>auto</u>

cppreference.com > C/C++ Keywords > reinterpret\_cast

# reinterpret\_cast

Syntax:

```
reinterpret_cast<type> (object);
```

The reinterpret\_cast operator changes one data type into another. It should be used to cast between incompatible pointer types.

Related topics:

const\_cast dynamic\_cast
static\_cast

cppreference.com > C/C++ Keywords > return

### return

Syntax:

```
return;
return( value );
```

The return statement causes execution to jump from the current function to whatever function called the current function. An optional *value* can be returned. A function may have more than one return statement.

#### cppreference.com > C/C++ Keywords > short

## short

The short keyword is a data type modifier that is used to declare short integer variables. See the <u>data types</u> page.

#### Related topics:

**bool** char

**double** 

float

int

long

**signed** 

unsigned

<u>void</u>

wchar\_t

#### cppreference.com > C/C++ Keywords > signed

# signed

The signed keyword is a data type modifier that is usually used to declare signed char variables. See the <u>data types</u> page.

#### Related topics:

**bool** char

**double** 

float

int

long

**short** 

unsigned

<u>void</u>

wchar\_t

### sizeof

The size of operator is a compile-time operator that returns the size of the argument passed to it. The size is a multiple of the size of a char, which on many personal computers is 1 byte (or 8 bits). The number of bits in a char is stored in the CHAR\_BIT constant defined in the <cli>climits> header file.

For example, the following code uses size of to display the sizes of a number of variables:

```
struct EmployeeRecord {
  int ID;
  int age;
  double salary;
  EmployeeRecord* boss;
};
cout << "sizeof(int): " << sizeof(int) << endl</pre>
     << "sizeof(float): " << sizeof(float) << endl
<< "sizeof(double): " << sizeof(double) << endl</pre>
     << "sizeof(char): " << sizeof(char) << endl
     << "sizeof(EmployeeRecord): " << sizeof(EmployeeRecord) << er</pre>
int i;
float f;
double d;
char c;
EmployeeRecord er;
cout << "sizeof(i): " << sizeof(i) << endl</pre>
     << "sizeof(f): " << sizeof(f) << endl
     << "sizeof(d): " << sizeof(d) << endl
     << "sizeof(c): " << sizeof(c) << endl
     << "sizeof(er): " << sizeof(er) << endl;
```

On some machines, the above code displays this output:

```
sizeof(int): 4
sizeof(float): 4
```

```
sizeof(double): 8
sizeof(char): 1
sizeof(EmployeeRecord): 20
sizeof(i): 4
sizeof(f): 4
sizeof(d): 8
sizeof(c): 1
sizeof(er): 20
```

Note that size of can either take a variable type (such as **int**) or a variable name (such as **i** in the example above).

It is also important to note that the sizes of various types of variables can change depending on what system you're on. Check out <u>a description of the C and C++ data types</u> for more information.

The parentheses around the argument are not required if you are using size of with a variable type (e.g. sizeof(int)).

Related topics:

**C/C++ Data Types** 

cppreference.com > C/C++ Keywords > static

## static

The static data type modifier is used to create permanent storage for variables. Static variables keep their value between function calls. When used in a <u>class</u>, all instantiations of that class share one copy of the variable.

#### cppreference.com > C/C++ Keywords > static\_cast

## static\_cast

Syntax:

```
static_cast<type> (object);
```

The static\_cast keyword can be used for any normal conversion between types. No runtime checks are performed.

Related topics:

const\_cast dynamic\_cast
reinterpret\_cast

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>struct</u>
```

### struct

Syntax:

```
struct struct-name : inheritance-list {
  public-members-list;
  protected:
  protected-members-list;
  private:
  private-members-list;
} object-list;
```

Structs are like `classes`, except that by default members of a struct are <u>public</u> rather than <u>private</u>. In C, structs can only contain data and are not permitted to have inheritance lists. For example:

```
struct Date {
  int Day;
  int Month;
  int Year;
};
```

Related topics:

class union

### switch

Syntax:

```
switch( expression ) {
  case A:
  statement list;
  break;
  case B:
  statement list;
  break;
  ...
  case N:
  statement list;
  break;
  default:
  statement list;
  break;
}
```

The switch statement allows you to test an expression for many values, and is commonly used as a replacement for multiple <u>if()...else if()...else if()</u>... statements. <u>break</u> statements are required between each <u>case</u> statement, otherwise execution will "fall-through" to the next <u>case</u> statement. The <u>default</u> case is optional. If provided, it will match any case not explicitly covered by the preceding cases in the switch statement. For example:

```
char keystroke = getch();
switch( keystroke ) {
   case 'a':
   case 'b':
   case 'c':
   case 'd':
     KeyABCDPressed();
     break;
   case 'e':
     KeyEPressed();
     break;
   default:
     UnknownKeyPressed();
     break;
}
```

Related topics:
break case
default
if

# template

#### Syntax:

```
template <class data-type> return-type name( parameter-list ) {
  statement-list;
}
```

Templates are used to create generic functions and can operate on data without knowing the nature of that data. They accomplish this by using a placeholder data-type for which many other <u>data types</u> can be substituted.

#### Example code:

For example, the following code uses a template to define a generic swap function that can swap two variables of any type:

```
template<class X> void genericSwap( X &a, X &b ) {
  X tmp;
  tmp = a;
  a = b;
  b = tmp;
int main(void) {
  int num1 = 5;
  int num2 = 21;
  cout << "Before, num1 is " << num1 << " and num2 is " << num2</pre>
  genericSwap( num1, num2 );
 cout << "After, num1 is " << num1 << " and num2 is " << num2 <
  char c1 = 'a';
  char c2 = 'z';
 cout << "Before, c1 is " << c1 << " and c2 is " << c2 << endl;
  genericSwap( c1, c2 );
  cout << "After, c1 is " << c1 << " and c2 is " << c2 << endl;
  return( 0 );
```

Related topics:

#### **typename**

### cppreference.com > C/C++ Keywords > this

# this

The this keyword is a pointer to the current object. All member functions of a <u>class</u> have a this pointer.

Related topics: <a href="mailto:class">class</a> <a href="mailto:operator">operator</a>

### throw

*Syntax:* 

```
try {
  statement list;
  }
  catch( typeA arg ) {
   statement list;
  }
  catch( typeB arg ) {
   statement list;
  }
  ...
  catch( typeN arg ) {
   statement list;
  }
}
```

The throw statement is part of the C++ mechanism for exception handling. This statement, together with the <u>try</u> and <u>catch</u> statements, the C++ exception handling system gives programmers an elegant mechanism for error recovery.

You will generally use a <u>try</u> block to execute potentially error-prone code. Somewhere in this code, a throw statement can be executed, which will cause execution to jump out of the <u>try</u> block and into one of the <u>catch</u> blocks. For example:

```
try {
   cout << "Before throwing exception" << endl;
   throw 42;
   cout << "Shouldn't ever see this" << endl;
}
catch( int error ) {
   cout << "Error: caught exception " << error << endl;
}</pre>
```

Related topics:

catch try

### cppreference.com > C/C++ Keywords > true

## true

The Boolean value of "true".

Related topics: bool false

### <u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>try</u>

# try

The try statement attempts to execute exception-generating code. See the <u>throw</u> statement for more details.

Related topics: <a href="mailto:catch">catch</a> throw

<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>typedef</u>

# typedef

Syntax:

```
typedef existing-type new-type;
```

The typedef keyword allows you to create a new alias for an existing data type.

This is often useful if you find yourself using a unwieldy data type -- you can use typedef to create a shorter, easier-to-use name for that data type. For example:

```
typedef unsigned int* pui_t;

// data1 and data2 have the same type
piu_t data1;
unsigned int* data2;
```

cppreference.com > C/C++ Keywords > typeid

# typeid

Syntax:

```
typeid( object );
```

The typeid operator returns a reference to a type\_info object that describes `object`.

### <u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>typename</u>

# typename

The typename keyword can be used to describe an undefined type or in place of the <u>class</u> keyword in a <u>template</u> declaration.

Related topics: <a href="mailto:class">class</a> <a href="template">template</a>

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>union</u>
```

## union

Syntax:

```
union union-name {
  public-members-list;
  private:
  private-members-list;
  } object-list;
```

A union is like a <u>class</u>, except that all members of a union share the same memory location and are by default <u>public</u> rather than <u>private</u>. For example:

```
union Data {
   int i;
   char c;
};
```

Related topics:

class struct

#### cppreference.com > C/C++ Keywords > unsigned

# unsigned

The unsigned keyword is a data type modifier that is usually used to declare unsigned <u>int</u> variables. See the <u>data types</u> page.

#### Related topics:

**bool** char

**double** 

**float** 

int

**short** 

**signed** 

<u>void</u>

wchar\_t

#### <u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>using</u>

# using

The using keyword is used to import a <u>namespace</u> (or parts of a namespace) into the current scope.

#### Example code:

For example, the following code imports the entire *std* namespace into the current scope so that items within that namespace can be used without a preceding "std::".

```
using namespace std;
```

Alternatively, the next code snippet just imports a single element of the *std* namespace into the current namespace:

using std::cout;

Related topics:

**namespace** 

### virtual

#### Syntax:

```
virtual return-type name( parameter-list );
virtual return-type name( parameter-list ) = 0;
```

The virtual keyword can be used to create virtual functions, which can be overridden by derived classes.

- A virtual function indicates that a function can be overridden in a subclass, and that the overridden function will actually be used.
- When a base object pointer points to a derived object that contains a virtual function, the decision about which version of that function to call is based on the type of object pointed to by the pointer, and this process happens at runtime.
- A base object can point to different derived objects and have different versions of the virtual function run.

If the function is specified as a pure virtual function (denoted by the = 0), it must be overridden by a derived class.

#### Example code:

For example, the following code snippet shows how a child class can override a virtual method of its parent, and how a non-virtual method in the parent cannot be overridden:

```
class Base {
public:
  void nonVirtualFunc() {
    cout << "Base: non-virtual function" << endl;
  }
  virtual void virtualFunc() {
    cout << "Base: virtual function" << endl;
  }
};</pre>
```

```
class Child : public Base {
public:
  void nonVirtualFunc() {
    cout << "Child: non-virtual function" << endl;
}
  void virtualFunc() {
    cout << "Child: virtual function" << endl;
}
};

int main() {
  Base* basePointer = new Child();
  basePointer->nonVirtualFunc();
  basePointer->virtualFunc();
  return 0;
}
```

When run, the above code displays:

```
Base: non-virtual function
Child: virtual function
```

Related topics:

**class** 

#### cppreference.com > C/C++ Keywords > void

# void

The void keyword is used to denote functions that return no value, or generic variables which can point to any type of data. Void can also be used to declare an empty parameter list. Also see the <u>data types</u> page.

#### Related topics:

char double
float
int
long
short
signed
unsigned
wchar\_t

## volatile

The volatile keyword is an implementation-dependent modifier, used when declaring variables, which prevents the compiler from optimizing those variables. Volatile should be used with variables whose value can change in unexpected ways (i.e. through an interrupt), which could conflict with optimizations that the compiler might perform.

#### cppreference.com > C/C++ Keywords > wchar\_t

# wchar\_t

The keyword wchar\_t is used to declare wide character variables. Also see the <u>data types</u> page.

#### Related topics:

**bool** char

**double** 

**float** 

int

**short** 

**signed** 

unsigned

<u>void</u>

<u>cppreference.com</u> > <u>C/C++ Keywords</u> > <u>while</u>

### while

Syntax:

```
while( condition ) {
  statement-list;
}
```

The while keyword is used as a looping construct that will evaluate the *statement-list* as long as *condition* is true. Note that if the *condition* starts off as false, the *statement-list* will never be executed. (You can use a <u>do</u> loop to guarantee that the statement-list will be executed at least once.) For example:

```
bool done = false;
while( !done ) {
   ProcessData();
   if( StopLooping() ) {
      done = true;
   }
}
```

Related topics:

break continue do

<u>for</u> if cppreference.com > I/O Flags

# C++ I/O Flags

### **Format flags**

C++ defines some format flags for standard input and output, which can be manipulated with the <a href="mailto:flags()">flags()</a>, <a href="mailto:setf()">setf()</a>, and <a href="mailto:unsetf()">unsetf()</a> functions. For example,

```
cout.setf(ios::left);
```

turns on left justification for all output directed to **cout**.

Flag	Meaning
boolalpha	Boolean values can be input/output using the words "true" and "false".
dec	Numeric values are displayed in decimal.
fixed	Display floating point values using normal notation (as opposed to scientific).
hex	Numeric values are displayed in hexidecimal.
internal	If a numeric value is padded to fill a field, spaces are inserted between the sign and base character.
left	Output is left justified.
oct	Numeric values are displayed in octal.
right	Output is right justified.
scientific	Display floating point values using scientific notation.
showbase	Display the base of all numeric values.
showpoint	Display a decimal and extra zeros, even when not needed.
showpos	Display a leading plus sign before positive numeric values.
skipws	Discard whitespace characters (spaces, tabs, newlines) when reading from a stream.
unitbuf	Flush the buffer after each insertion.
uppercase	Display the "e" of scientific notation and the "x" of hexidecimal notation as capital letters.

### **Manipulators**

You can also manipulate flags indirectly, using the following *manipulators*. Most programmers are familiar with the **endl** manipulator, which might give you an idea of how manipulators are used. For example, to set the *dec* flag, you might use the following command:

cout << dec;</pre>

Manipulato	rs defined in <iostream></iostream>		
Manipulator Description Input Outpu			
boolalpha	Turns on the boolalpha flag	X	X
dec	Turns on the dec flag	X	X
endl	Output a newline character, flush the stream		X
ends	Output a null character		X
fixed	Turns on the fixed flag		X
flush	Flushes the stream		X
hex	Turns on the hex flag	X	X
internal	Turns on the internal flag		X
left	Turns on the left flag		X
noboolalpha	Turns off the boolalpha flag	X	X
noshowbase	Turns off the showbase flag		X
noshowpoint	Turns off the showpoint flag		X
noshowpos	Turns off the showpos flag		X
noskipws	Turns off the skipws flag	X	
nounitbuf	Turns off the unitbuf flag		X
nouppercase	Turns off the uppercase flag		X
oct	Turns on the oct flag	X	X
right	Turns on the right flag		X
scientific	Turns on the scientific flag		X
showbase	Turns on the showbase flag		X
showpoint	Turns on the showpoint flag		X
showpos	Turns on the showpos flag		X
skipws	Turns on the skipws flag	X	

unitbuf T	Turns on the unitbuf flag		X
uppercase T	Turns on the uppercase flag		X
ws S	kip any leading whitespace	X	
Manipulators	defined in <iomanip></iomanip>		
Manipulator	Description	Inp	ut Output
resetiosflags( lo	ng f ) Turn off the flags specified by $f$	X	X
setbase( int base	e ) Sets the number base to <i>base</i>		X
setfill( int ch )	Sets the fill character to <i>ch</i>		X
setiosflags( long	g(f) Turn on the flags specified by $f$	X	X
setprecision( int	p) Sets the number of digits of precision	1	X
setw( int w )	Sets the field width to <i>w</i>		X

# **State flags**

The I/O stream state flags tell you the current state of an I/O stream. The flags are:

Flag	Meaning	
badbit	a fatal error has occurred	
eofbit	EOF has been found	
failbit	a nonfatal error has occurred	
goodbit no errors have occurred		

# **Mode flags**

The I/O stream mode flags allow you to access files in different ways. The flags are:

Mode	Meaning
ios::app	append output
ios::ate	seek to EOF when opened
ios::binary	open the file in binary mode
ios::in	open the file for reading
ios::out	open the file for writing
ios::trunc	overwrite the existing file

#### 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 <u>vector</u>. 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
output_iterator	incremented and dereferenced.
	Read or write values with forward movement. These
forward_iterator	combine the functionality of input and output iterators with
	the ability to store the iterators value.
	Read and write values with forward and backward
bidirectional_iterator	movement. These are like the forward iterators, but you
	can increment and decrement them.
	Read and write values with random access. These are the
random_iterator	most powerful iterators, combining the functionality of
ומוועטווו_וופומוטו	bidirectional iterators with the ability to do pointer
	arithmetic and pointer comparisons.
rovorco itorator	Either a random iterator or a bidirectional iterator that
reverse_iterator	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++ )</pre>
```

```
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;</pre>
```

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

### cppreference.com > Complexity

### Complexity

There are different measurements of the speed of any given algorithm. Given an input size of  ${\bf N}$ , they can be described as follows:

Name	Speed	Description	Formula
exponential time	slow	takes an amount of time proportional to a constant raised to the <b>N</b> th power	K^ <b>N</b>
polynomial time	fast	takes an amount of time proportional to ${f N}$ raised to some constant power	<b>N</b> ^K
linear time	faster	takes an amount of time directly proportional to ${f N}$	K * <b>N</b>
logarithmic time	much faster	takes an amount of time proportional to the logarithm of ${f N}$	K * log( <b>N</b> )
constant time	fastest	takes a fixed amount of time, no matter how large the input is	K

## 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.

<u>Display all entries</u> for C++ Bitsets on one page, or view entries individually:

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
<u>reset</u>	sets bits to zero
set	sets bits
<u>size</u>	number of bits that the bitset can hold
<u>test</u>	returns the value of a given bit
to_string	string representation of the bitset
to_ulong	returns an integer representation of the bitset

# C++ Vectors

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

<u>Display all entries</u> for C++ Vectors on one page, or view entries individually:

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
Vector operatorscompare, assign, and access elements of a vectorassignassign elements to a vectoratreturns an element at a specific location
compare, assign, and access elements of a vector  assign assign elements to a vector  at returns an element at a specific location
assign assign elements to a vector at returns an element at a specific location
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
<u>capacity</u> returns the number of elements that the vector can hold
<u>clear</u> removes all elements from the vector
<u>empty</u> true if the vector has no elements
end returns an iterator just past the last element of a vector
<u>erase</u> removes elements from a vector
front returns a reference to the first element of a vector
<u>insert</u> inserts elements into the vector
returns the maximum number of elements that the
max_size vector can hold
pop_back removes the last element of a vector
push_back add an element to the end of the vector
<u>rbegin</u> returns a <u>reverse_iterator</u> to the end of the vector
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

<u>size</u>	returns the number of items in the vector
swap	swap the contents of this vector with another

cppreference.com > Containers

### C++ Containers

The C++ Containers (vectors, lists, etc.) are generic vessels capable of holding many different types of data. For example, the following statement creates a <u>vector</u> of integers:

vector<int> v;

Containers can hold standard objects (like the **int** in the above example) as well as custom objects, as long as the objects in the container meet a few requirements:

- The object must have a default constructor,
- an accessible destructor, and
- an accessible assignment operator.

When describing the functions associated with these various containers, this website defines the word **TYPE** to be the object type that the container holds. For example, in the above statement, **TYPE** would be **int**. Similarly, when referring to containers associated with pairs of data (map for example) **key\_type** and **value\_type** are used to refer to the key and value types for that container.

## 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.

<u>Display all entries</u> for C++ Double-ended Queues on one page, or view entries individually:

Container constructors	create dequeues and initialize them with some data
Container operators	compare, assign, and access elements of a dequeue
<u>assign</u>	assign elements to a dequeue
<u>at</u>	returns an element at a specific location
<u>back</u>	returns a reference to last element of a dequeue
<u>begin</u>	returns an iterator to the beginning of the dequeue
<u>clear</u>	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
<u>erase</u>	removes elements from a dequeue
<u>front</u>	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
<u>rbegin</u>	returns a <u>reverse_iterator</u> to the end of the dequeue
rend	returns a <u>reverse_iterator</u> to the beginning of the dequeue

<u>resize</u>	change the size of the dequeue
<u>size</u>	returns the number of items in the dequeue
swap	swap the contents of this dequeue with another

### 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.

<u>Display all entries</u> for C++ Lists on one page, or view entries individually:

<u>List</u> <u>constructors</u>	create lists and initialize them with some data
List operators	assign and compare lists
<u>assign</u>	assign elements to a list
<u>back</u>	returns a reference to last element of a list
<u>begin</u>	returns an iterator to the beginning of the list
<u>clear</u>	removes all elements from the list
<u>empty</u>	true if the list has no elements
<u>end</u>	returns an iterator just past the last element of a list
<u>erase</u>	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
<u>merge</u>	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
<u>rbegin</u>	returns a <u>reverse_iterator</u> to the end of the list
<u>remove</u>	removes elements from a list
remove_if	removes elements conditionally
<u>rend</u>	returns a <u>reverse_iterator</u> to the beginning of the list
<u>resize</u>	change the size of the list
<u>reverse</u>	reverse the list

<u>size</u>	returns the number of items in the list
<u>sort</u>	sorts a list into ascending order
<u>splice</u>	merge two lists in constant time
swap	swap the contents of this list with another
unique	removes consecutive duplicate elements

### C++ Sets

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

<u>Display all entries</u> for C++ Sets on one page, or view entries individually:

Set constructors	default methods to allocate, copy, and deallocate
& destructors	sets
	assign and compare sets
Set operators	
<u>begin</u>	returns an iterator to the beginning of the set
<u>clear</u>	removes all elements from the set
count	returns the number of elements matching a certain key
<u>empty</u>	true if the set has no elements
<u>end</u>	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
<u>erase</u>	removes elements from a set
<u>find</u>	returns an iterator to specific elements
insert	insert items into a set
<u>key_comp</u>	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
<u>rbegin</u>	returns a <u>reverse_iterator</u> to the end of the set
rend	returns a <u>reverse_iterator</u> 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

## 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.

<u>Display all entries</u> for C++ Multisets on one page, or view entries individually:

Container constructors & destructors	default methods to allocate, copy, and deallocate multisets
Container operators	assign and compare multisets
<u>begin</u>	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
<u>erase</u>	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 <u>reverse</u> <u>iterator</u> to the end of the multiset
<u>rend</u>	returns a <u>reverse_iterator</u> to the beginning of the multiset

<u>size</u>	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

## C++ Maps

C++ Maps are sorted associative containers that contain unique key/value pairs. For example, you could create a map that associates a <u>string</u> with an integer, and then use that map to associate the number of days in each month with the name of each month.

<u>Display all entries</u> for C++ Maps on one page, or view entries individually:

Map constructors	default methods to allocate, copy, and deallocate
<u>&amp; destructors</u>	maps
Map operators	assign, compare, and access elements of a map
<u>begin</u>	returns an iterator to the beginning of the map
<u>clear</u>	removes all elements from the map
count	returns the number of elements matching a certain key
<u>empty</u>	true if the map has no elements
<u>end</u>	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
<u>erase</u>	removes elements from a map
<u>find</u>	returns an iterator to specific elements
<u>insert</u>	insert items into a map
<u>key_comp</u>	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
<u>rbegin</u>	returns a <u>reverse_iterator</u> to the end of the map
rend	returns a <u>reverse_iterator</u> to the beginning of the map

<u>size</u>	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

# C++ Multimaps

C++ Multimaps are like maps, in that they are sorted associative containers, but differ from maps in that they allow duplicate keys.

Display all entries for C++ Multimaps on one page, or view entries individually:

<u></u>	
Multimap constructors & destructors	default methods to allocate, copy, and deallocate multimaps
Multimap operators	assign and compare multimaps
<u>begin</u>	returns an iterator to the beginning of the multimap
<u>clear</u>	removes all elements from the multimap
count	returns the number of elements matching a certain key
<u>empty</u>	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
<u>erase</u>	removes elements from a multimap
find	returns an iterator to specific elements
<u>insert</u>	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 <u>reverse_iterator</u> to the end of the multimap
rend	returns a <u>reverse_iterator</u> to the beginning of the multimap

<u>size</u>	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

#### <u>cppreference.com</u> > <u>C++ I/O</u>

### C++I/O

The <iostream> library automatically defines a few standard objects:

- cout, an object of the ostream class, which displays data to the standard output device.
- cerr, another object of the ostream class that writes unbuffered output to the standard error device.
- clog, like cerr, but uses buffered output.
- cin, an object of the istream class that reads data from the standard input device.

The <fstream> library allows programmers to do file input and output with the ifstream and ofstream classes.

C++ programmers can also do input and output from strings by using the <u>String Stream</u> class.

Some of the behavior of the C++ I/O streams (precision, justification, etc) may be modified by manipulating various <u>io stream format flags</u>.

Here are some examples of what you can do with C++ I/O.

<u>Display all entries</u> for C++ I/O on one page, or view entries individually:

I/O Constructors	constructors
<u>bad</u>	true if an error occurred
<u>clear</u>	clear and set status flags
<u>close</u>	close a stream
<u>eof</u>	true if at the end-of-file
<u>fail</u>	true if an error occurred
fill	manipulate the default fill character
flags	access or manipulate io stream format flags
flush	empty the buffer

gcount	number of characters read during last input
<u>get</u>	read characters
<u>getline</u>	read a line of characters
good	true if no errors have occurred
<u>ignore</u>	read and discard characters
<u>open</u>	open a new stream
<u>peek</u>	check the next input character
precision	manipulate the precision of a stream
<u>put</u>	write characters
<u>putback</u>	return characters to a stream
<u>rdstate</u>	returns the state flags of the stream
<u>read</u>	read data into a buffer
<u>seekg</u>	perform random access on an input stream
<u>seekp</u>	perform random access on output streams
<u>setf</u>	set format flags
sync with stdio	synchronize with standard I/O
<u>tellg</u>	read input stream pointers
<u>tellp</u>	read output stream pointers
unsetf	clear <u>io stream format flags</u>
width	access and manipulate the minimum field width
<u>write</u>	write characters

# **C++ Priority Queues**

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

<u>Display all entries</u> for C++ Priority Queues on one page, or view entries individually:

Priority queue constructors	construct a new priority queue
<u>empty</u>	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
<u>512C</u>	queue
<u>top</u>	returns the top element of the priority queue

#### cppreference.com > C++ Queues

## C++ Queues

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

<u>Display all entries</u> for C++ Queues on one page, or view entries individually:

Queue constructor	construct a new queue
<u>back</u>	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

#### cppreference.com > C++ Stacks

## 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.

<u>Display all entries</u> for C++ Stacks on one page, or view entries individually:

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
<u>size</u>	returns the number of items in the stack
top	returns the top element of the stack

# C++ Strings

<u>Display all entries</u> for C++ Strings on one page, or view entries individually:

String	create strings from arrays of characters and other
<u>constructors</u>	strings
String operators	concatenate strings, assign strings, use strings for I/O,
otring operators	compare strings
<u>append</u>	append characters and strings onto a string
<u>assign</u>	give a string values from strings of characters and
<u> </u>	other C++ strings
<u>at</u>	returns the character at a specific location
<u>begin</u>	returns an iterator to the beginning of the string
<u>c_str</u>	returns a non-modifiable standard C character array
<u></u>	version of the string
<u>capacity</u>	returns the number of characters that the string can
	hold
<u>clear</u>	removes all characters from the string
<u>compare</u>	compares two strings
copy	copies characters from a string into an array
<u>data</u>	returns a pointer to the first character of a string
empty	true if the string has no characters
end	returns an iterator just past the last character of a
<u> </u>	string
<u>erase</u>	removes characters from a string
find	find characters in the string
find_first_not_of	find first absence of characters
<u>find_first_of</u>	find first occurrence of characters
find_last_not_of	find last absence of characters
<u>find_last_of</u>	find last occurrence of characters
getline	read data from an I/O stream into a string
insert	insert characters into a string

<u>length</u>	returns the length of the string
max_size	returns the maximum number of characters that the string can hold
push_back	add a character to the end of the string
<u>rbegin</u>	returns a <u>reverse_iterator</u> to the end of the string
rend	returns a <u>reverse</u> <u>iterator</u> to the beginning of the string
<u>replace</u>	replace characters in the string
<u>reserve</u>	sets the minimum capacity of the string
<u>resize</u>	change the size of the string
rfind	find the last occurrence of a substring
<u>size</u>	returns the number of items in the string
substr	returns a certain substring
swap	swap the contents of this string with another

# C++ Algorithms

<u>Display all entries</u> for C++ Algorithms on one page, or view entries individually:

<u>accumulate</u>	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
<u>copy_n</u>	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
<u>find_first_of</u>	search for any one of a set of elements

find_if	find the first element for which a certain predicate is true
<u>for_each</u>	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
<u>includes</u>	returns true if one set is a subset of another
inner_product	compute the inner product of two ranges of elements
<u>inplace_merge</u>	merge two ordered ranges in-place
<u>is_heap</u>	returns true if a given range is a heap
<u>is_sorted</u>	returns true if a range is sorted in ascending order
<u>iter_swap</u>	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
<u>lower_bound</u>	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
<u>max</u>	returns the larger of two elements
max_element	returns the largest element in a range
merge	merge two sorted ranges
<u>min</u>	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
<u>replace</u>	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
<u>reverse</u>	reverse elements in some range
reverse_copy	create a copy of a range that is reversed
<u>rotate</u>	move the elements in some range to the left by some amount
rotate_copy	copy and rotate a range of elements
<u>search</u>	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
<u>set_union</u>	computes the union of two sets
<u>sort</u>	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
<u>transform</u>	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
-------------	--

## **Miscellaneous C++**

<u>Display all entries</u> for Miscellaneous C++ on one page, or view entries individually:

## C++ String Streams

String streams are similar to the <iostream> and <fstream> libraries, except that string streams allow you to perform I/O on strings instead of streams. The <sstream> library provides functionality similar to sscanf() and sprintf() in the standard C library. Three main classes are available in <sstream>:

- stringstream allows input and output
- istringstream allows input only
- ostringstream allows output only

String streams are actually subclasses of iostreams, so **all of the functions available for iostreams are also available for stringstream**. See the  $\underline{C++\ I/O}$  functions for more information.

<u>Display all entries</u> for C++ String Streams on one page, or view entries individually:

Constructors	create new string streams
<u>Operators</u>	read from and write to string streams
<u>rdbuf</u>	get the buffer for a string stream
str	get or set the stream's string

# **C/C++ Pre-processor Commands**

<u>Display all entries</u> for C/C++ Pre-processor Commands on one page, or view entries individually:

<u>#, ##</u>	manipulate strings
<u>#define</u>	define variables
<u>#error</u>	display an error message
#if, #ifdef, #ifndef, #else, #elif, #endif	conditional operators
<u>#include</u>	insert the contents of another file
<u>#line</u>	set line and file information
<u>#pragma</u>	implementation specific command
<u>#undef</u>	used to undefine variables
Predefined preprocessor variables	miscellaneous preprocessor variables

## **Other Standard C Functions**

<u>Display all entries</u> for Other Standard C Functions on one page, or view entries individually:

<u>abort</u>	stops the program
<u>assert</u>	stops the program if an expression isn't true
<u>atexit</u>	sets a function to be called when the program exits
<u>bsearch</u>	perform a binary search
<u>exit</u>	stop the program
getenv	get enviornment information about a variable
<u>longjmp</u>	start execution at a certain point in the program
<u>qsort</u>	perform a quicksort
<u>raise</u>	send a signal to the program
<u>rand</u>	returns a pseudorandom number
<u>setjmp</u>	set execution to start at a certain point
<u>signal</u>	register a function as a signal handler
<u>srand</u>	initialize the random number generator
<u>system</u>	perform a system call
<u>va</u> arg	use variable length parameter lists

## **Standard C Math**

<u>Display all entries</u> for Standard C Math on one page, or view entries individually:

<u>abs</u>	absolute value
<u>acos</u>	arc cosine
<u>asin</u>	arc sine
<u>atan</u>	arc tangent
atan2	arc tangent, using signs to determine quadrants
<u>ceil</u>	the smallest integer not less than a certain value
COS	cosine
<u>cosh</u>	hyperbolic cosine
<u>div</u>	returns the quotient and remainder of a division
<u>exp</u>	returns "e" raised to a given power
<u>fabs</u>	absolute value for floating-point numbers
<u>floor</u>	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
<u>frexp</u>	decomposes a number into scientific notation
<u>labs</u>	absolute value for long integers
<u>ldexp</u>	computes a number in scientific notation
ldiv	returns the quotient and remainder of a division, in long integer form
log	natural logarithm (to base e)
<u>log10</u>	common logarithm (to base 10)
modf	decomposes a number into integer and fractional parts
pow	returns a given number raised to another number
sin	sine
<u>sinh</u>	hyperbolic sine
<u>sqrt</u>	square root
<u>tan</u>	tangent

## **Standard C Date & Time**

<u>Display all entries</u> for Standard C Date & Time on one page, or view entries individually:

<u>asctime</u>	a textual version of the time
<u>clock</u>	returns the amount of time that the program has been running
<u>ctime</u>	returns a specifically formatted version of the time
<u>difftime</u>	the difference between two times
<u>gmtime</u>	returns a pointer to the current Greenwich Mean Time
<u>localtime</u>	returns a pointer to the current time
<u>mktime</u>	returns the calendar version of a given time
<u>setlocale</u>	sets the current locale
<u>strftime</u>	returns individual elements of the date and time
<u>time</u>	returns the current calendar time of the system

# **Standard C String and Character**

<u>Display all entries</u> for Standard C String and Character on one page, or view entries individually:

<u>atof</u>	converts a string to a double
<u>atoi</u>	converts a string to an integer
<u>atol</u>	converts a string to a long
<u>isalnum</u>	true if a character is alphanumeric
<u>isalpha</u>	true if a character is alphabetic
<u>iscntrl</u>	true if a character is a control character
<u>isdigit</u>	true if a character is a digit
<u>isgraph</u>	true if a character is a graphical character
<u>islower</u>	true if a character is lowercase
<u>isprint</u>	true if a character is a printing character
<u>ispunct</u>	true if a character is punctuation
<u>isspace</u>	true if a character is a space character
<u>isupper</u>	true if a character is an uppercase character
<u>isxdigit</u>	true if a character is a hexidecimal character
<u>memchr</u>	searches an array for the first occurance of a character
<u>memcmp</u>	compares two buffers
<u>memcpy</u>	copies one buffer to another
<u>memmove</u>	moves one buffer to another
<u>memset</u>	fills a buffer with a character
<u>strcat</u>	concatenates two strings
<u>strchr</u>	finds the first occurance of a character in a string
<u>strcmp</u>	compares two strings
<u>strcoll</u>	compares two strings in accordance to the current locale
strcpy	copies one string to another
<u>strcspn</u>	searches one string for any characters in another
strerror	returns a text version of a given error code

<u>strlen</u>	returns the length of a given string
strncat	concatenates a certain amount of characters of two strings
strncmp	compares a certain amount of characters of two strings
strncpy	copies a certain amount of characters from one string to another
<u>strpbrk</u>	finds the first location of any character in one string, in another string
<u>strrchr</u>	finds the last occurance of a character in a string
<u>strspn</u>	returns the length of a substring of characters of a string
<u>strstr</u>	finds the first occurance of a substring of characters
strtod	converts a string to a double
<u>strtok</u>	finds the next token in a string
<u>strtol</u>	converts a string to a long
strtoul	converts a string to an unsigned long
<u>strxfrm</u>	converts a substring so that it can be used by string comparison functions
tolower	converts a character to lowercase
toupper	converts a character to uppercase

# **Standard C Memory**

<u>Display all entries</u> for Standard C Memory on one page, or view entries individually:

<u>calloc</u>	allocates and clears a two-dimensional chunk of memory
<u>free</u>	returns previously allocated memory to the operating system
<u>malloc</u>	allocates memory
<u>realloc</u>	changes the size of previously allocated memory

## Standard C I/O

Display all entries for Standard C I/O on one page, or view entries individually:

<u>clearerr</u>	clears errors
<u>fclose</u>	close a file
<u>feof</u>	true if at the end-of-file
<u>ferror</u>	checks for a file error
<u>fflush</u>	writes the contents of the output buffer
<u>fgetc</u>	get a character from a stream
<u>fgetpos</u>	get the file position indicator
<u>fgets</u>	get a string of characters from a stream
<u>fopen</u>	open a file
<u>fprintf</u>	print formatted output to a file
<u>fputc</u>	write a character to a file
<u>fputs</u>	write a string to a file
<u>fread</u>	read from a file
<u>freopen</u>	open an existing stream with a different name
<u>fscanf</u>	read formatted input from a file
<u>fseek</u>	move to a specific location in a file
<u>fsetpos</u>	move to a specific location in a file
<u>ftell</u>	returns the current file position indicator
<u>fwrite</u>	write to a file
getc	read a character from a file
getchar	read a character from <b>stdin</b>
<u>gets</u>	read a string from <b>stdin</b>
porror	displays a string version of the current error to
<u>perror</u>	stderr
<u>printf</u>	write formatted output to <b>stdout</b>
<u>putc</u>	write a character to a stream
<u>putchar</u>	write a character to <b>stdout</b>

puts	write a string to <b>stdout</b>
<u>remove</u>	erase a file
<u>rename</u>	rename a file
rewind	move the file position indicator to the
rewind	beginning of a file
<u>scanf</u>	read formatted input from <b>stdin</b>
<u>setbuf</u>	set the buffer for a specific stream
<u>setvbuf</u>	set the buffer and size for a specific stream
<u>sprintf</u>	write formatted output to a buffer
<u>sscanf</u>	read formatted input from a buffer
<u>tmpfile</u>	return a pointer to a temporary file
<u>tmpnam</u>	return a unique filename
ungetc	puts a character back into a stream
vprintf, vfprintf, and	write formatted output with variable argument
<u>vsprintf</u>	lists

#### cppreference.com > Static Return

#### Watch out.

This function returns a variable that is statically located, and therefore overwritten each time this function is called. If you want to save the return value of this function, you should manually save it elsewhere.

Of course, when you save it elsewhere, you should make sure to actually copy the value(s) of this variable to another location. If the return value is a struct, you should make a new struct, then copy over the members of the struct.

cppreference.com > C/C++ Data Types

# C/C++ Data Types

There are five data types for C: **void**, **int**, **float**, **double**, and **char**.

Type	Description
void	associated with no data type
int	integer
float	floating-point number
double	double precision floating-point number
char	character

C++ defines two more: **bool** and **wchar\_t**.

Type	Description
bool	Boolean value, true or false
wchar_	<sup>t</sup> wide character

## **Type Modifiers**

Several of these types can be modified using **signed**, **unsigned**, **short**, and **long**. When one of these type modifiers is used by itself, a data type of **int** is assumed. A complete list of possible data types follows:

bool
char
unsigned char
signed char
int
unsigned int
signed int
short int
unsigned short int
signed short int
long int
signed long int
unsigned long int
float
double
long double
wchar_t

### **Type Sizes and Ranges**

The size and range of any data type is compiler and architecture dependent. The "cfloat" (or "float.h") header file often defines minimum and maximum values for the various data types. You can use the <u>sizeof</u> operator to determine the size of any data type, in bytes. However, many architectures implement data types of a standard size. **ints** and **floats** are often 32-bit, **chars** 8-bit, and **doubles** are usually 64-bit. **bools** are often implemented as 8-bit data types.

#### cppreference.com > Escape Sequences

#### **Constant Escape Sequences**

The following escape sequences can be used to define certain special characters within strings:

#### **Escape Sequence Description**

\'	Single quote
\	Single quote
\''	Double quote
\\	Backslash
\nnn	Octal number (nnn)
\0	Null character (really just the octal number zero)
\a	Audible bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xnnn	Hexadecimal number (nnn)

An example of this is contained in the following code (which assumes that the newline character generates complete newlines, i.e. on Unix systems):

```
printf( "This\nis\na\ntest\n\nShe said, \"How are you?\"\n" );
```

#### which would display

```
This
is
a
test
She said, "How are you?"
```

## C++ Operator Precedence

The operators at the top of this list are evaluated first.

Grouping operator  Grouping operator  Array access Member access from a pointer Member access from an object Post-increment Post-decrement  Logical negation Bitwise complement Pre-decrement  Pre-increment Pre-decrement  Unary plus Dereference Address of Cast to a given type Return size in bytes  Member pointer  Grouping operator  (a + b) / 4; array[4] = 2; ptr->age = 34; obj.age = 34; obj.age = 34; for (i = 0; i < 10; i++) for (i = 10; i > 0; i)  if (!done) flags = ~flags; for (i = 0; i < 10; ++i) for (i = 10; i > 0; -i)  for (i = 10; i > 0; -i) int i = -1; int i = +1; data = *ptr; address = &obj int i = (int) floatNum; int size = sizeof(floatNum);  Member pointer  Scoping operator  (a + b) / 4; array[4] = 2; ptr->age = 34; obj.age = 3	Precedence	Operator	Description	Example	Associativity
Operator   Array access   Array access   Member access   from a pointer   Member access   from an object   Post-increment   Post-decrement   Pre-increment   Pre-increment   Pre-decrement   Operator   Array access   Member access   from an object   Post-increment   Pre-increment   Pre-increment   Onary minus   Onary plus   Dereference   Address of   Cast to a given   type   Return size in   bytes   Operator   Array access   Array access   from a pointer   Array access   for (i = 0; i < 10; i++)   for (i = 10; i > 0; i)   for (i = 10; i > 0; ii)   for	1	::	Scoping operator	Class::age = 2;	none
Itogreat negation Bitwise complement Pre-increment Pre-decrement Unary minus Unary plus Unary plus Dereference Address of Cast to a given type Return size in bytes  Logred negation flags = ~flags; for( i = 0; i < 10; ++i ) for( i = 10; i > 0;i ) int i = -1; int i = +1; data = *ptr; address = &obj int i = (int) floatNum; int size = sizeof(floatNum);  Member pointer  selector  Member pointer	2	[] ->	operator Array access Member access from a pointer Member access from an object Post-increment	array[4] = 2; ptr->age = 34; obj.age = 34; for( i = 0; i < 10; i++) for( i = 10; i > 0;	left to right
-> *	3	++  - + * & (type)	Bitwise complement Pre-increment Pre-decrement Unary minus Unary plus Dereference Address of Cast to a given type Return size in	flags = ~flags; for( i = 0; i < 10; ++i ) for( i = 10; i > 0; - -i ) int i = -1; int i = +1; data = *ptr; address = &obj int i = (int) floatNum; int size =	right to left
.* Member object obj.*var = 24; left to right selector	4		selector Member object	ptr->*var = 24; obj.*var = 24;	left to right

5	*  /  %	Multiplication Division Modulus	int i = 2 * 4; float f = 10 / 3; int rem = 4 % 3;	left to right
6	+	Addition Subtraction	int i = 2 + 3; int i = 5 - 1;	left to right
7	<< >>	Bitwise shift left Bitwise shift right	int flags = 33 << 1; int flags = 33 >> 1;	left to right
8	<	Comparison less-than Comparison less-than-or-equal-to Comparison greater-than Comparison geater-than-or-equal-to	if( i < 42 ) if( i <= 42 ) if( i > 42 ) if( i >= 42 )	left to right
9	== !=	Comparison equal-to Comparison not-equal-to	if( i == 42 ) if( i != 42 )	left to right
10	&	Bitwise AND	flags = flags & 42;	left to right
11	٨	Bitwise exclusive OR	flags = flags ^ 42;	left to right
12		Bitwise inclusive (normal) OR	flags = flags   42;	left to right
13	&&	Logical AND	if( conditionA && conditionB )	left to right
14	II	Logical OR	if( conditionA    conditionB )	left to right
		Ternary	int i = (a > b) ? a :	

15	?:	conditional (if- then-else)	b;	right to left
16	= += -= *= /= %= &= \<=  = <<= >>=	Assignment operator Increment and assign Decrement and assign Multiply and assign Divide and assign Modulo and assign Bitwise AND and assign Bitwise exclusive OR and assign Bitwise inclusive (normal) OR and assign Bitwise shift left and assign Bitwise shift right and assign	int a = b; a += 3; b -= 4; a *= 5; a /= 2; a %= 3; flags &= new_flags; flags \^= new_flags; flags <<= 2; flags >>= 2;	right to left
17	,	Sequential evaluation operator	for( i = 0, j = 0; i < 10; i++, j++ )	left to right

One important aspect of C++ that is related to operator precedence is the **order of evaluation** and the **order of side effects** in expressions. In some circumstances, the order in which things happen is not defined. For example, consider the following code:

```
float x = 1;
x = x / ++x;
```

The value of x is not guaranteed to be consistent across different compilers, because it is not clear whether the computer should evaluate the left or the right side of the division first. Depending on which side is evaluated first, x could take a different value.

Furthermore, while ++x evaluates to x+1, the side effect of actually storing that new value in x could happen at different times, resulting in different values for x.

The bottom line is that expressions like the one above are horribly ambiguous and should be avoided at all costs. When in doubt, break a single ambiguous expression into multiple expressions to ensure that the order of evaluation is correct.

### **ASCII Chart**

The following chart contains ASCII decimal, octal, hexadecimal and character codes for values from 0 to 127.

Decimal Octal Hex Character Description					
0	0	00	NUL		
1	1	01	SOH	start of header	
2	2	02	STX	start of text	
3	3	03	ETX	end of text	
4	4	04	EOT	end of transmission	
5	5	05	ENQ	enquiry	
6	6	06	ACK	acknowledge	
7	7	07	BEL	bell	
8	10	80	BS	backspace	
9	11	09	HT	horizontal tab	
10	12	0A	LF	line feed	
11	13	0B	VT	vertical tab	
12	14	0C	FF	form feed	
13	15	0D	CR	carriage return	
14	16	0E	SO	shift out	
15	17	0F	SI	shift in	
16	20	10	DLE	data link escape	
17	21	11	DC1	no assignment, but usually XON	
18	22	12	DC2		
19	23	13	DC3	no assignment, but usually XOFF	
20	24	14	DC4		
21	25	15	NAK	negative acknowledge	
22	26	16	SYN	synchronous idle	
23	27	17	ETB	end of transmission block	
24	30	18	CAN	cancel	
25	31	19	EM	end of medium	

26	32	1A	SUB	substitute
27	33	1B	ESC	escape
28	34	1C	FS	file seperator
29	35	1D	GS	group seperator
30	36	1E	RS	record seperator
31	37	1F	US	unit seperator
32	40	20	SPC	space
33	41	21	!	
34	42	22	11	
35	43	23	#	
36	44	24	\$	
37	45	25	%	
38	46	26	&	
39	47	27	1	
40	50	28	(	
41	51	29	)	
42	52	2A	*	
43	53	2B	+	
44	54	2C	,	
45	55	2D	-	
46	56	2E	•	
47	57	2F	/	
48	60	30	0	
49	61	31	1	
50	62	32	2	
51	63	33	3	
52	64	34	4	
53	65	35	5	
54	66	36	6	
55	67	37	7	
56	70	38	8	
57	71	39	9	
58	72	3A	:	
59	73	3B	;	

60	74	3C	<
61	75	3D	=
62	76	3E	>
63	77	3F	?
64	100	40	@
65	101	41	A
66	102	42	В
67	103	43	С
68	104	44	D
69	105	45	Е
70	106	46	F
71	107	47	G
72	110	48	Н
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	О
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]

94	136	5E	$\wedge$		
95	137	5F			
96	140	60	`		
97	141	61	a		
98	142	62	b		
99	143	63	C		
100	144	64	d		
101	145	65	e		
102	146	66	f		
103	147	67	g		
104	150	68	h		
105	151	69	i		
106	152	6A	j		
107	153	6B	k		
108	154	6C	l		
109	155	6D	m		
110	156	6E	n		
111	157	6F	0		
112	160	70	p		
113	161	71	q		
114	162	72	r		
115	163	73	S		
116	164	74	t		
117	165	75	u		
118	166	76	V		
119	167	77	W		
120	170	78	X		
121	171	79	y		
122	172	7A	Z		
123	173	7B	{		
124	174	7C			
125	175	7D	}		
126	176	7E	~		
127	177	7F	DEL	delete	

# C/C++ Keywords

<u>Display all entries</u> for C/C++ Keywords on one page, or view entries individually:

<u>asm</u>	insert an assembly instruction
<u>auto</u>	declare a local variable
<u>bool</u>	declare a boolean variable
<u>break</u>	break out of a loop
<u>case</u>	a block of code in a <u>switch</u> statement
<u>catch</u>	handles exceptions from <u>throw</u>
<u>char</u>	declare a character variable
<u>class</u>	declare a class
const	declare immutable data or functions that do not change data
const_cast	cast from const variables
<u>continue</u>	bypass iterations of a loop
<u>default</u>	default handler in a <u>case</u> statement
<u>delete</u>	make memory available
<u>do</u>	looping construct
<u>double</u>	declare a double precision floating-point variable
dynamic_cast	perform runtime casts
<u>else</u>	alternate case for an <u>if</u> statement
<u>enum</u>	create enumeration types
<u>explicit</u>	only use constructors when they exactly match
export	allows template definitions to be separated from their declarations
<u>extern</u>	tell the compiler about variables defined elsewhere
<u>false</u>	the boolean value of false
<u>float</u>	declare a floating-point variable
<u>for</u>	looping construct

<u>friend</u>	grant non-member function access to private data
<u>goto</u>	jump to a different part of the program
<u>if</u>	execute code based off of the result of a test
<u>inline</u>	optimize calls to short functions
<u>int</u>	declare a integer variable
<u>long</u>	declare a long integer variable
<u>mutable</u>	override a const variable
<u>namespace</u>	partition the global namespace by defining a scope
<u>new</u>	allocate dynamic memory for a new variable
<u>operator</u>	create overloaded operator functions
<u>private</u>	declare private members of a class
protected	declare protected members of a class
<u>public</u>	declare public members of a class
<u>register</u>	request that a variable be optimized for speed
<u>reinterpret_cast</u>	change the type of a variable
<u>return</u>	return from a function
<u>short</u>	declare a short integer variable
<u>signed</u>	modify variable type declarations
<u>sizeof</u>	return the size of a variable or type
<u>static</u>	create permanent storage for a variable
static_cast	perform a nonpolymorphic cast
struct	define a new structure
switch	execute code based off of different possible values for a variable
<u>template</u>	create generic functions
<u>this</u>	a pointer to the current object
<u>throw</u>	throws an exception
<u>true</u>	the boolean value of true
try	execute code that can throw an exception
<u>typedef</u>	create a new type name from an existing type
<u>typeid</u>	describes an object
<u>typename</u>	declare a class or undefined type
union	a structure that assigns multiple variables to the same memory location

unsigned	declare an unsigned integer variable
using	import complete or partial <u>namespace</u> s into the current
	scope
virtual	create a function that can be overridden by a derived
	class
<u>void</u>	declare functions or data with no associated data type
volatile	warn the compiler about variables that can be modified
	unexpectedly
<u>wchar_t</u>	declare a wide-character variable
<u>while</u>	looping construct

#### **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 <u>queues</u>, <u>lists</u>, and <u>stacks</u>.

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 <u>push()</u> and <u>pop()</u> 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 <u>a bunch of useful algorithms</u> -- like <u>searching</u>, <u>sorting</u>, and <u>general-purpose iterating</u> algorithms -- that can be used on a variety

of data structures.

<u>cppreference.com</u> > FAQ

## **Frequently Asked Questions**

#### Can I get a copy of this site?

We do provide a <u>downloadable archived version of cppreference.com</u>. If you're interested in getting archived versions of websites in general, you might want to check out utilities like <u>GNU's wget</u> (Windows version <u>here</u>).

In addition, James Heany has compiled a PDF version of the site (as of October 2007) that is available for download. There is a <u>C++ Reference PDF</u> and a <u>STL Reference PDF</u> available for download.

#### Can I [mirror/translate/put up my own version of/etc.] this site?

Sure, that would be great! All that we would ask is that you include a link back to this site so that people know where to get the most up-to-date content.

#### Who is this site meant for?

There are no "Introduction to Programming" tutorials here. This site is meant to be used by more-or-less experienced C++ programmers, who have a good idea of what they want to do and simply need to look up the syntax. If you're interested in learning C/C++, try one of these sites:

- How C Programming Works
- <u>C Programming</u>
- <u>C++ Language Tutorial</u>

# Does this site contain a complete and definitive list of everything I can do with C/C++?

Few things in life are absolute. Many C/C++ compilers have added or missing functionality. If you don't find what you are looking for here, don't assume that it doesn't exist. Do a search on Google for it.

#### Some of the examples on this site don't work on my system.

#### What's going on?

Most of the code on this site was compiled under <u>Linux</u> (<u>Red Hat</u>, <u>Debian</u>, or <u>Ubuntu</u>) with the <u>GNU Compiler Collection</u>. Since this site is merely a reference for the <u>Standard C and C++ specification</u>, not every compiler will support every function listed here. For example,

Header files change like mad. To include the necessary support for <u>C++</u>
 <u>Vectors</u>, you might have to use any of these:

```
#include <vector>
#include <Vector>
#include <vector.h>
```

(according to the spec, the first of those should work, and the compiler should know enough to use it to reference the real vector header file.)

 Another header file issue is that newer compilers can use a more platformindependent commands to include standard C libraries. For example, you should be able to use

```
#include <cstdio>
```

instead of

```
#include <stdio.h>
```

 All of the code on this site assumes that the correct namespace has been designated. If your compiler is a little old, then you might be able to get away with using simple statements like:

```
cout << "hello world!";</pre>
```

However, newer compilers require that you either use

```
std::cout << "hello world!";
```

or declare what namespace to use with the "using namespace" command.

• Certain popular compilers (like the one shipped with Microsoft's Visual C++) have added alternative or additional functionality to the C++ Standard Template Library. For example, the MFC in Visual C++ provides you with the string type "CString", which has string functionality but is not part of the C++ STL.

...The list goes on and on. In other words, individual results may vary.

### You've got an error in this site.

If you find any errors in this reference, please feel free to <u>contact us</u> -- feedback and code examples are always welcome.

### What's up with this site?

Think of it as a community service, for geeks.

cppreference.com > Links

## Links

Here are some links to other language references:

- <u>C++ (Dinkumware)</u>
- C++ Language and Library
- Java 6 SE (Sun)
- MySQL
- <u>Perl</u>
- Python
- Ruby
- <u>Tcl</u>
- Visual C++ STL (Microsoft)

#### <u>cppreference.com</u> > Credits

Huge thanks to all these people for sending in bug fixes and suggestions on how to improve the site:

Alex Vinokur - Ted Felix

A.J.M. van den Berg - Adrian Pfisterer - Alex Wilson - Alexandre Kostine -Andre - Andre Gillibert - Andrew L Roth - Annamalai Gurusami - Art Stamness - Arvid Norberg - Benjamin Lee Hansen - Brian Higgins - Brian T Stadler - Carl - Cedric Blaser - Chip Lemon - Chris Frey - Chris H - Chris Rimmer - Chris Yate - Christian Foerg - Christoffer Nyborg - Christoph Otto - Christoph Vogelbusch - Claudio Alberto Andreoni - Colin Hirsch - Damian (doublenegative) - Dan Mergens - Dan Stronger - Daniel Fish - Daniel Goering -Daniel Lorch - Darsh Ranjan - Dave Schuyler - Dave T - David E Freitas -Davoud Taghawi-Nejad - Deepak Goyal - Devin Pratt - Diggory Hardy - Dirk Jagdmann - Drew Dormann - Dzu Nguyen - E.Guadalupe - Edgardo Rossetto -Eirik Stangeland - Emmanuel Viaud - Enrique Pineda - Eric Kinser - Erik Aas -Erik Wikstrom - Fabian Foerg - Florian Schaper - Florian B - Fred Ma - Frederik Hertzum - Gerhard Grossauer - guiliano - Guillaume Bouchez - Hasan Amjad -Henning Diedrich - Henrik Huttunen - Henrik Mattsson - Iain Staffell - Iheanyi Umez-Eronini - Imre Pentek - JP (Pete) Donnell - James Bliese - James Dennett - James Heany - James Jones - Jan - Jann Poppinga - Jari Karppinen - Jeff Bowden - Jeff Dwork - Jeroen Missinne - Jodi Giordano - Joe Crobak - John Feltz - Jonathan Dent - Jonathan Kleid - Joseph Bruni - Joshua Haberman -Joshua R. Warr - Justin M. Lee - Katherine Haines - Keith Knapp - Ken Sedgwick - Kien Nguyen - Kiyoshi Aman - Kuang-che Wu - Kwan Ting Chan -Kurt McKee - Leor Zolman - Lindley French - Lucas Fisher - Mael Herz -Magnus Kulke - Manish Malik - Martin - Martin Milata - Martin Richardt -Martijn van de Giessen - Matthias Britsch - Matthias Hofmann - Matthias Neeracher - Mike Angstadt - Mike Clarke - Mike Ekoka - Mike Jennings - Milan Mimica - Moonrie - Nadia De Bode - Nate Silva - Neelesh Bodas - Neil - Nick Gianakas - Nicolas Boichat - Olivier Ricou - Onur Tugcu - Osku Salerma -Patrick Spendrin - Paul Fee - Paul L. Tomlinson - Philip Dunstan - Phillip Lee -Piers Daniell - Ralf Denzer - Randall Rathbun - Rasmus Hansen - Rex Kerr -Rob Larkins - Rodrigo Cesar Dias - Roger D Pack - Romans Kasperovics -Ronald Cotton - Salman Mahbub - Selim T. Erdogan - Sergio Martinez -Shibukawa Yoshiki - Simon Perkins - snlee - Stefan Suffa - Stefan Voegel -

Steve Davison - Steve Ward - Supermonkey - TT - Tarjei Knapstad - Tetra - Thomas Volk - Tiaan van Aardt - Tom (prkchp) - Tor Husab - Tyler Cole - Vegard Nossum - Victor Rachels - Vijay S. - William Charles Deich IV - William Dye - William K. Austad - William K. Foster - Wouter Lievens - XenteX

Thank you!

```
<u>cppreference.com</u> > <u>C++ Bitsets</u>
```

### 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

## **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</li>
- >>= 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;</pre>
```

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;</pre>
```

### 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:

<u>any</u>

### 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** 

```
<u>cppreference.com</u> > <u>C++ Vectors</u>
```

## 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;</pre>
```

The above code displays the following output:

```
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] << " ";
}</pre>
```

```
cout << endl;</pre>
```

When run, the above code displays the following output:

```
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;
}</pre>
```

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;
}</pre>
```

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>
    <u>TYPE</u>& back();
    const <u>TYPE</u>& 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;</pre>
```

This code produces the following output:

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

The back() function runs in <u>constant time</u>.

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;
}</pre>
```

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;</pre>
```

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 <u>linear time</u>.

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();
}</pre>
```

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;</pre>
```

The next example shows how <u>begin()</u> 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 <u>begin()</u>. 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 <u>constant time</u> for lists and <u>linear time</u> for vectors, dequeues, and strings. The multiple-element version of erase always takes <u>linear time</u>.

For example:

```
// Create a vector, load it with the first ten characters of the a
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 </pre>
```

```
cout << endl;
}</pre>
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
HIJ
HIJ
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 a
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 elemen
// 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;</pre>
```

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 <u>sort() algorithm</u> 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.from</pre>
```

When provided with this input:

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

...the above code displays:

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

Related topics:

```
back
```

(C++ Lists) pop\_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 e
```

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 <u>linear time</u>. 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 <u>List</u> or a <u>Deque</u>).

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

This code would display:

```
CCCCABCDEFGHIJ
```

Here 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: ";</pre>
for( int i = 0; i < v2.size(); i++ ) {</pre>
  cout << v2[i] << " ";
cout << endl;</pre>
v2.insert( v2.end(), v1.begin(), v1.end() );
cout << "After, v2 is: ";</pre>
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 number of elements that the vector will be able to hold before more memory will have to be allocated, respectively.

Related topics:

<u>size</u>

## 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 );
}</pre>
```

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 <u>reverse\_iterator</u> 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 <u>reverse\_iterator</u> to the beginning of the current vector.

rend() runs in constant time.

Related topics:

begin end

<u>rbegin</u>

#### 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 <u>linear time</u>.

## 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 <u>linear time</u>.

*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 <u>constant time</u>.

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;</pre>
```

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
```

### **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: ";</pre>
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:
```

```
cout << "new vector: ";
  vector<int> v2( iter1, iter2 );
  for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
  }
  cout << endl;
}</pre>
```

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 <u>linear time</u> except the first, which runs in <u>constant time</u>.

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 <u>linear time</u>. The [] operator runs in <u>constant time</u>.

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 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;
}</pre>
```

Related topics:

<u>at</u>

#### <u>cppreference.com</u> > <u>C++ Double-ended Queues</u>

## 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;</pre>
```

The above code displays the following output:

```
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] << " ";
}</pre>
```

```
cout << endl;</pre>
```

When run, the above code displays the following output:

```
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;
}</pre>
```

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;
}</pre>
```

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;</pre>
```

This code produces the following output:

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

The back() function runs in <u>constant time</u>.

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
   cout << *theIterator;
}</pre>
```

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

# **Container** [] **operator**

Syntax:

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

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</pre>
```

**Container** [] operator

```
Syntax:
```

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

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</pre>
```

#### Container constructors & destru

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 <u>constant time</u>. The default copy constructor runs in <u>linear time</u> 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
```

#### 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: " <

cout << "new vector: ";
vector<int> v2( iter1, iter2 );
for( int i = 0; i < v2.size(); i++ ) {
   cout << v2[i] << " ";
}
cout << endl;</pre>
```

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 <u>linear time</u> 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 <u>linear time</u>. The [] operator runs in <u>constant time</u>.

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 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;
}</pre>
```

Related topics:

<u>at</u>

## 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();
}</pre>
```

Related topics:

<u>size</u>

## 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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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 <u>constant time</u> for lists and <u>linear time</u> for vectors, dequeues, and strings. The multiple-element version of erase always takes <u>linear time</u>.

For example:

```
// Create a vector, load it with the first ten characters of the al
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 != alphaVec
        cout << *tempIterator;
    }
    cout << endl;
}</pre>
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
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 a
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++ ) {</pre>
 cout << alphaVector[i];</pre>
cout << endl;
// use erase to remove all but the first two and last three element
// of the vector
alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
// display the modified vector
for( int i = 0; i < alphaVector.size(); i++ ) {</pre>
 cout << alphaVector[i];</pre>
}
cout << endl;
```

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

#### Related topics:

```
clear
insert
pop_back
pop_front
```

```
(C++ Lists) <u>remove</u>
(C++ Lists) <u>remove_if</u>
```

### 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, i
```

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 al
```

```
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
   cout << *theIterator;
}</pre>
```

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 number of elements that the dequeue will be able to hold before more memory will have to be allocated, respectively.

<u>size</u>

# 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 erase pop\_front 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 front pop\_back 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 );</pre>
```

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 <u>reverse\_iterator</u> 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 <u>reverse\_iterator</u> 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) <u>Multimap constructors</u> & <u>destructors</u> (C++ Strings) <u>capacity</u> 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) <u>length</u>

<u>max_size</u>

<u>resize</u>
```

## 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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

(C++ Lists) splice

```
<u>cppreference.com</u> > <u>C++ Lists</u>
```

## 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;</pre>
```

The above code displays the following output:

```
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] << " ";
}</pre>
```

```
cout << endl;</pre>
```

When run, the above code displays the following output:

```
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;</pre>
```

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.endo
   cout << *theIterator;
}</pre>
```

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

erase

# 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 <u>constant time</u>. The default copy constructor runs in <u>linear time</u> 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
```

#### 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: "</pre>
cout << "new vector: ";</pre>
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 <u>linear time</u> except the first, which runs in <u>constant time</u>.

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 <u>linear time</u>.

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 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
```

## 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();
}</pre>
```

Related topics:

<u>size</u>

#### 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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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 <u>constant time</u> for lists and <u>linear time</u> for vectors, dequeues, and strings. The multiple-element version of erase always takes <u>linear time</u>.

For example:

```
// Create a vector, load it with the first ten characters of the al
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 != alphaVec
        cout << *tempIterator;
    }
    cout << endl;
}</pre>
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
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 a
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++ ) {</pre>
 cout << alphaVector[i];</pre>
cout << endl;
// use erase to remove all but the first two and last three element
// of the vector
alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
// display the modified vector
for( int i = 0; i < alphaVector.size(); i++ ) {</pre>
 cout << alphaVector[i];</pre>
}
cout << endl;
```

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

#### Related topics:

```
clear
insert
pop_back
pop_front
```

#### 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, i
```

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 al
```

```
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
   cout << *theIterator;
}</pre>
```

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 <u>size()</u> or (C++ Strings) <u>capacity()</u> functions, which return the number of elements currently in the list and the number of elements that the list will be able to hold before more memory will have to be allocated, respectively.

Related topics:

#### 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** 

<u>insert</u>

**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** 

erase

pop front

# 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 );</pre>
```

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' );</pre>
```

Remove runs in <u>linear time</u>.

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 <a href="linear time">linear time</a>.

Related topics:

<u>erase</u>

remove

<u>unique</u>

#### rend

Syntax:

```
#include <list>
  reverse iterator rend();
const_reverse iterator rend() const;
```

The function rend() returns a <a href="reverse\_iterator">reverse\_iterator</a> to the beginning of the current list.

rend() runs in constant time.

Related topics:

**begin** 

<u>end</u>

<u>rbegin</u>

### 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) <u>Multimap constructors</u> & <u>destructors</u> (C++ Strings) <u>capacity</u> <u>size</u>
```

#### reverse

Syntax:

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

The function reverse() reverses the list, and takes <u>linear time</u>.

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
(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 en
```

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

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 <u>constant time</u>.

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;</pre>
```

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

```
<u>cppreference.com</u> > <u>C++ Sets</u>
```

# 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
   cout << *theIterator;
}</pre>
```

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
```

## **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 <u>linear time</u>.

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 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
```

#### 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();
}</pre>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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* before 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 <u>key type</u>& 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 <u>logarithmic time</u>.

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 <u>reverse</u> <u>iterator</u> 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 <u>reverse\_iterator</u> 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 <u>constant time</u>.

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;</pre>
```

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 <u>key type</u>& key );
```

The function upper\_bound() returns an iterator to the first element in the set with a value 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

```
<u>cppreference.com</u> > <u>C++ Multisets</u>
```

## 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 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
   cout << *theIterator;
}</pre>
```

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
```

# 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 <u>constant time</u>. The default copy constructor runs in <u>linear time</u> 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 <u>linear time</u>.

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
```

#### 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();
}</pre>
```

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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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 <u>key type</u>& 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 <u>logarithmic time</u>.

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 number of elements that the multiset will be able to hold before more memory will have to be allocated, respectively.

Related topics:

<u>size</u>

# rbegin

Syntax:

```
#include <set>
   reverse iterator rbegin();
   const_reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse\_iterator</u> 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 <u>reverse\_iterator</u> 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 <u>constant time</u>.

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;</pre>
```

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 <u>key type</u>& 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

```
<u>cppreference.com</u> > <u>C++ Maps</u>
```

# 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 
}
```

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 <u>linear</u> 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 <u>logarithmic time</u>.

## 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.keging).first << (*ages.keging).first << ", " << (*ages.keging).first << (*ages.keging).first << (*ages.keging).first << (*ages.keging).first << (*ages.keging)
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
Erasing: Marge, 37
```

#### Related topics:

<u>begin</u> <u>erase</u>

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 <u>begin()</u> 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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 <u>while</u> 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());
}</pre>
```

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 << }
}</pre>
```

When run with this input:

```
my spoon is too big. my spoon is TOO big! my SPOON is TOO big! I
```

...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
}
```

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 <u>string</u>, the map is sorted alphabetically by key.

## 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:

<u>value\_comp</u>

## lower\_bound

Syntax:

```
#include <map>
iterator lower_bound( const <u>key type</u>& 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 <u>logarithmic time</u>.

Related topics:

upper\_bound

## **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;</pre>
```

#### 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 <u>linear time</u>.

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 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();
   cout << (*iter).first << " is " << (*iter).second << " years old
}</pre>
```

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** 

## 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 <u>size()</u> or (C++ Strings) <u>capacity()</u> functions, which return the number of elements currently in the map and 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 <u>reverse\_iterator</u> 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 <u>reverse\_iterator</u> to the beginning of the current map.

rend() runs in constant time.

Related topics:

<u>begin</u>

<u>end</u>

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 <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

# upper\_bound

Syntax:

```
#include <map>
iterator upper_bound( const <u>key type</u>& 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:

<u>key\_comp</u>

```
<u>cppreference.com</u> > <u>C++ Multimaps</u>
```

# 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
   cout << *theIterator;
}</pre>
```

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 <u>logarithmic time</u>.

## 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();</pre>
```

}

Related topics:

<u>size</u>

#### 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 <a href="begin">begin</a>() 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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

## 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, pai</pre>
multimap<string,pair<int,int> >::iterator i;
ii = input_config.equal_range("key");
                                               // keyboard key-bind
// 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
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*.

## 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.

#### 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 cout << "Number of employees named 'Bob Thompson': " << m.count("Ecout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Employee list: " << endl;
for( multimap<string, int>::iterator iter = m.begin(); iter != m.cout << "Name: " << iter->first << ", ID #" << iter->second << "}</pre>
```

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.

## 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 <u>logarithmic time</u>.

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 number of elements that the multimap will be able to hold before more memory will have to be allocated, respectively.

Related topics:

<u>size</u>

# Multimap constructors & destructors

Syntax:

Multimaps have several constructors:

- The default constructor takes no arguments, creates a new instance of that multimap, and runs in <u>constant time</u>.
- The default copy constructor runs in <u>linear time</u> 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 cout << "Number of employees named 'Bob Thompson': " << m.count("E cout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Number of employees named 'Bob Smithey': " << m.count("Bob cout << "Employee list: " << endl;
for( multimap<string, int>::iterator iter = m.begin(); iter != m.e cout << " Name: " << iter->first << ", ID #" << iter->second << }</pre>
```

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);</pre>
```

```
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 <u>linear time</u>.

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 member in location i in the other multimap.

Comparisons among multimaps are done lexicographically.

Related topics:

**Multimap Constructors** 

# rbegin

Syntax:

```
#include <map>
   reverse iterator rbegin();
   const_reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse</u> <u>iterator</u> 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 <u>reverse\_iterator</u> 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 <u>constant time</u>.

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;</pre>
```

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

 $\underline{\text{cppreference.com}} > \underline{\text{C++ I/O}} > \underline{\text{Examples}}$ 

# C++ I/O Examples

#### **Reading From Files**

Assume that we have a file named *data.txt* that contains this text:

```
Fry: One Jillion dollars.
[Everyone gasps.]
Auctioneer: Sir, that's not a number.
[Everyone gasps.]
```

We could use this code to read data from the file, word by word:

```
ifstream fin("data.txt");
string s;
while( fin >> s ) {
   cout << "Read from file: " << s << endl;
}</pre>
```

When used in this manner, we'll get space-delimited bits of text from the file:

```
Read from file: Fry:
Read from file: One
Read from file: Jillion
Read from file: dollars.
Read from file: [Everyone
Read from file: gasps.]
Read from file: Auctioneer:
Read from file: Sir,
Read from file: that's
Read from file: not
Read from file: a
Read from file: number.
Read from file: [Everyone
Read from file: gasps.]
```

Note that in the previous example, all of the whitespace that separated words (including newlines) was lost. If we were interested in preserving whitespace, we could read the file in line-by-line using the <a href="I/O getline">I/O getline</a>() function.

```
ifstream fin("data.txt");
const int LINE_LENGTH = 100;
char str[LINE_LENGTH];
while( fin.getline(str,LINE_LENGTH) ) {
  cout << "Read from file: " << str << endl;</pre>
```

}

Reading line-by-line produces the following output:

```
Read from file: Fry: One Jillion dollars.
Read from file: [Everyone gasps.]
Read from file: Auctioneer: Sir, that's not a number.
Read from file: [Everyone gasps.]
```

If you want to avoid reading into character arrays, you can use the <u>C++ string</u> getline() function to read lines into <u>strings</u>:

```
ifstream fin("data.txt");
string s;
while( getline(fin,s) ) {
  cout << "Read from file: " << s << endl;
}</pre>
```

## **Checking For Errors**

Simply evaluating an I/O object in a boolean context will return false if any errors have occurred:

```
string filename = "data.txt";
ifstream fin( filename.c_str() );
if( !fin ) {
  cout << "Error opening " << filename << " for input" << endl;
  exit(-1);
}</pre>
```

```
<u>cppreference.com</u> > <u>C++ I/O</u>
```

### bad

Syntax:

```
#include <fstream>
bool bad();
```

The bad() function returns true if a fatal error with the current stream has occurred, false otherwise.

Related topics:

<u>eof fail</u><u>good</u>rdstate

## clear

Syntax:

```
#include <fstream>
void clear( iostate flags = ios::goodbit );
```

The function clear() does two things:

- it clears all io stream state flags associated with the current stream,
- and sets the flags denoted by *flags*

The *flags* argument defaults to ios::goodbit, which means that by default, all flags will be cleared and ios::goodbit will be set.

Example code:

For example, the following code uses the clear() function to reset the flags of an output file stream, after an attempt is made to read from that output stream:

```
fstream outputFile( "output.txt", fstream::out );

// try to read from the output stream; this shouldn't work
int val;
outputFile >> val;
if( outputFile.fail() ) {
  cout << "Error reading from the output stream" << endl;
  // reset the flags associated with the stream
  outputFile.clear();
}

for( int i = 0; i < 10; i++ ) {
  outputFile << i << " ";
}
outputFile << endl;</pre>
```

Related topics:

eof fail good rdstate

### close

Syntax:

```
#include <fstream>
void close();
```

The close() function closes the associated file stream.

Related topics:

**I/O Constructors** 

<u>open</u>

## **I/O Constructors**

Syntax:

```
#include <fstream>
fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

The fstream, ifstream, and ofstream objects are used to do file I/O. The optional *mode* defines how the file is to be opened, according to the <u>io stream mode flag</u>s. The optional *filename* specifies the file to be opened and associated with the stream.

Input and output file streams can be used in a similar manner to C++ predefined I/O streams, cin and cout.

#### Example code:

The following code reads input data and appends the result to an output file.

```
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
  fout << temp + 2 << endl;
fin.close();
fout.close();</pre>
```

Related topics:

<u>close</u> <u>open</u>

### eof

Syntax:

```
#include <fstream>
bool eof();
```

The function eof() returns true if the end of the associated input file has been reached, false otherwise.

For example, the following code reads data from an input stream *in* and writes it to an output stream *out*, using eof() at the end to check if an error occurred:

```
char buf[BUFSIZE];
do {
   in.read( buf, BUFSIZE );
   std::streamsize n = in.gcount();
   out.write( buf, n );
} while( in.good() );
if( in.bad() || !in.eof() ) {
   // fatal error occurred
}
in.close();
```

#### Related topics:

bad clear fail good

**rdstate** 

# C++ I/O Examples

#### **Reading From Files**

Assume that we have a file named *data.txt* that contains this text:

```
Fry: One Jillion dollars.
[Everyone gasps.]
Auctioneer: Sir, that's not a number.
[Everyone gasps.]
```

We could use this code to read data from the file, word by word:

```
ifstream fin("data.txt");
string s;
while( fin >> s ) {
   cout << "Read from file: " << s << endl;
}</pre>
```

When used in this manner, we'll get space-delimited bits of text from the file:

```
Read from file: Fry:
Read from file: One
Read from file: Jillion
Read from file: dollars.
Read from file: [Everyone
Read from file: gasps.]
Read from file: Auctioneer:
Read from file: Sir,
Read from file: that's
Read from file: not
Read from file: a
Read from file: number.
Read from file: [Everyone
Read from file: gasps.]
```

Note that in the previous example, all of the whitespace that separated words (including newlines) was lost. If we were interested in preserving whitespace, we could read the file in line-by-line using the <a href="I/O getline">I/O getline</a>() function.

```
ifstream fin("data.txt");
const int LINE_LENGTH = 100;
char str[LINE_LENGTH];
while( fin.getline(str,LINE_LENGTH) ) {
  cout << "Read from file: " << str << endl;</pre>
```

}

Reading line-by-line produces the following output:

```
Read from file: Fry: One Jillion dollars.
Read from file: [Everyone gasps.]
Read from file: Auctioneer: Sir, that's not a number.
Read from file: [Everyone gasps.]
```

If you want to avoid reading into character arrays, you can use the <u>C++ string</u> getline() function to read lines into <u>strings</u>:

```
ifstream fin("data.txt");
string s;
while( getline(fin,s) ) {
  cout << "Read from file: " << s << endl;
}</pre>
```

#### **Checking For Errors**

Simply evaluating an I/O object in a boolean context will return false if any errors have occurred:

```
string filename = "data.txt";
ifstream fin( filename.c_str() );
if( !fin ) {
  cout << "Error opening " << filename << " for input" << endl;
  exit(-1);
}</pre>
```

## fail

Syntax:

```
#include <fstream>
bool fail();
```

The fail() function returns true if an error has occurred with the current stream, false otherwise.

Related topics:

bad clear eof good rdstate

## fill

Syntax:

```
#include <fstream>
char fill();
char fill( char ch );
```

The function fill() either returns the current fill character, or sets the current fill character to *ch*.

The fill character is defined as the character that is used for padding when a number is smaller than the specified width(). The default fill character is the space character.

Related topics:

precision
width

# flags

Syntax:

```
#include <fstream>
fmtflags flags();
fmtflags flags( fmtflags f );
```

The flags() function either returns the <u>io stream format flags</u> for the current stream, or sets the flags for the current stream to be *f*.

Related topics:

setf unsetf

### flush

Syntax:

```
#include <fstream>
ostream& flush();
```

The flush() function causes the buffer for the current output stream to be actually written out to the attached device.

This function is useful for printing out debugging information, because sometimes programs abort before they have a chance to write their output buffers to the screen. Judicious use of flush() can ensure that all of your debugging statements actually get printed.

Related topics:

put
write

## gcount

Syntax:

```
#include <fstream>
streamsize gcount();
```

The function gcount() is used with input streams, and returns the number of characters read by the last input operation.

Related topics:

getline read

## get

Syntax:

```
#include <fstream>
int get();
istream& get( char& ch );
istream& get( char* buffer, streamsize num );
istream& get( char* buffer, streamsize num, char delim );
istream& get( streambuf& buffer );
istream& get( streambuf& buffer, char delim );
```

The get() function is used with input streams, and either:

- reads a character and returns that value,
- reads a character and stores it as *ch*,
- reads characters into *buffer* until *num* 1 characters have been read, or **EOF** or newline encountered,
- reads characters into *buffer* until *num* 1 characters have been read, or **EOF** or the *delim* character encountered (*delim* is not read until next time),
- reads characters into buffer until a newline or **EOF** is encountered,
- or reads characters into buffer until a newline, **EOF**, or *delim* character is encountered (again, *delim* isn't read until the next get()).

For example, the following code displays the contents of a file called temp.txt, character by character:

```
char ch;
ifstream fin( "temp.txt" );
while( fin.get(ch) )
  cout << ch;
fin.close();</pre>
```

#### Related topics:

```
gcount
getline
(C++ Strings) getline
ignore
peek
put
read
```

# getline

#### Syntax:

```
#include <fstream>
istream& getline( char* buffer, streamsize num );
istream& getline( char* buffer, streamsize num, char delim );
```

The getline() function is used with input streams, and reads characters into *buffer* until either:

- *num* 1 characters have been read,
- a newline is encountered,
- an **EOF** is encountered,
- or, optionally, until the character *delim* is read. The *delim* character is not put into buffer.

For example, the following code uses the getline function to display the first 100 characters from each line of a text file:

```
ifstream fin("tmp.dat");
int MAX_LENGTH = 100;
char line[MAX_LENGTH];
while( fin.getline(line, MAX_LENGTH) ) {
  cout << "read line: " << line << endl;
}</pre>
```

If you'd like to read lines from a file into <u>strings</u> instead of character arrays, consider using the <u>string getline</u> function.

Those using a Microsoft compiler may find that getline() reads an extra character, and should consult the documentation on the Microsoft getline bug.

Related topics:

```
gcount
get
(C++ Strings) getline
ignore
read
```

# good

Syntax:

```
#include <fstream>
bool good();
```

The function good() returns true if no errors have occurred with the current

stream, false otherwise.

Related topics:

bad

<u>clear</u>

eof

**fail** 

<u>rdstate</u>

# ignore

Syntax:

```
#include <fstream>
istream& ignore( streamsize num=1, int delim=EOF );
```

The ignore() function is used with input streams. It reads and throws away characters until *num* characters have been read (where *num* defaults to 1) or until the character *delim* is read (where *delim* defaults to **EOF**).

The ignore() function can sometimes be useful when using the getline() function together with the >> operator. For example, if you read some input that is followed by a newline using the >> operator, the newline will remain in the input as the next thing to be read. Since getline() will by default stop reading input when it reaches a newline, a subsequent call to getline() will return an empty string. In this case, the ignore() function could be called before getline() to "throw away" the newline.

Related topics:

get

<u>getline</u>

### open

Syntax:

```
#include <fstream>
void open( const char *filename );
void open( const char *filename, openmode mode = default_mode );
```

The function open() is used with file streams. It opens *filename* and associates it with the current stream. The optional <u>io stream mode flag</u> *mode* defaults to ios::in for ifstream, ios::out for ofstream, and ios::in|ios::out for fstream.

If open() fails, the resulting stream will evaluate to false when used in a Boolean expression. For example:

```
ifstream inputStream;
inputStream.open("file.txt");
if( !inputStream ) {
  cerr << "Error opening input stream" << endl;
  return;
}</pre>
```

Related topics:

**I/O Constructors** 

**close** 

# peek

Syntax:

```
#include <fstream>
int peek();
```

The function peek() is used with input streams, and returns the next character in the stream or **EOF** if the end of file is read. peek() does not remove the character from the stream.

Related topics:

get putback

# precision

Syntax:

```
#include <fstream>
streamsize precision();
streamsize precision( streamsize p );
```

The precision() function either sets or returns the current number of digits that is displayed for floating-point variables.

For example, the following code sets the precision of the cout stream to 5:

```
float num = 314.15926535;
cout.precision( 5 );
cout << num;</pre>
```

This code displays the following output:

```
314.16
```

Related topics:

<u>fill</u>

width

### put

Syntax:

```
#include <fstream>
ostream& put( char ch );
```

The function put() is used with output streams, and writes the character *ch* to the stream.

Related topics:

<u>flush</u>

# putback

Syntax:

```
#include <fstream>
istream& putback( char ch );
```

The putback() function is used with input streams, and returns the previously-read character *ch* to the input stream.

Related topics:

**peek** 

(Standard C I/O) ungetc

# rdstate

Syntax:

```
#include <fstream>
iostate rdstate();
```

The rdstate() function returns the io stream state flags of the current stream.

Related topics:

bad

clear

eof

fail

good

## read

#### Syntax:

```
#include <fstream>
istream& read( char* buffer, streamsize num );
```

The function read() is used with input streams, and reads *num* bytes from the stream before placing them in *buffer*. If **EOF** is encountered, read() stops, leaving however many bytes it put into *buffer* as they are.

For example:

```
struct {
   int height;
   int width;
} rectangle;

input_file.read( (char *)(&rectangle), sizeof(rectangle) );
if( input_file.bad() ) {
   cerr << "Error reading data" << endl;
   exit( 0 );
}</pre>
```

#### Related topics:

gcount get getline write

# seekg

#### Syntax:

```
#include <fstream>
istream& seekg( off_type offset, ios::seekdir origin );
istream& seekg( pos_type position );
```

The function seekg() is used with input streams, and it repositions the "get" pointer for the current stream to *offset* bytes away from *origin*, or places the "get" pointer at *position*.

Related topics:

seekp tellg tellp

# seekp

Syntax:

```
#include <fstream>
ostream& seekp( off_type offset, ios::seekdir origin );
ostream& seekp( pos_type position );
```

The seekp() function is used with output streams, but is otherwise very similar to seekg().

Related topics:

seekg tellg tellp

### setf

Syntax:

```
#include <fstream>
fmtflags setf( fmtflags flags );
fmtflags setf( fmtflags flags, fmtflags needed );
```

The function setf() sets the <u>io stream format flags</u> of the current stream to *flags*. The optional *needed* argument specifies that only the flags that are in both *flags* and *needed* should be set. The return value is the previous configuration of <u>io stream format flags</u>.

For example:

```
int number = 0x3FF;
```

```
cout.setf( ios::dec );
cout << "Decimal: " << number << endl;
cout.unsetf( ios::dec );
cout.setf( ios::hex );
cout << "Hexadecimal: " << number << endl;</pre>
```

Note that the preceding code is functionally identical to:

```
int number = 0x3FF;
cout << "Decimal: " << number << endl << hex << "Hexadecimal: " <</pre>
```

thanks to io stream manipulators.

Related topics:

flags unsetf

# sync\_with\_stdio

Syntax:

```
#include <fstream>
static bool sync_with_stdio( bool sync=true );
```

The sync\_with\_stdio() function allows you to turn on and off the ability for the C++ I/O system to work with the C I/O system.

# tellg

Syntax:

```
#include <fstream>
pos_type tellg();
```

The tellg() function is used with input streams, and returns the current "get" position of the pointer in the stream.

Related topics:

seekg seekp tellp

# tellp

Syntax:

```
#include <fstream>
pos_type tellp();
```

The tellp() function is used with output streams, and returns the current "put" position of the pointer in the stream.

For example, the following code displays the file pointer as it writes to a stream:

```
string s("In Xanadu did Kubla Khan...");
ofstream fout("output.txt");
for( int i=0; i < s.length(); i++ ) {
  cout << "File pointer: " << fout.tellp();
  fout.put( s[i] );
  cout << " " << s[i] << endl;
}
fout.close();</pre>
```

Related topics:

seekg seekp tellg

### unsetf

Syntax:

```
#include <fstream>
void unsetf( fmtflags flags );
```

The function unsetf() uses *flags* to clear the <u>io stream format flag</u>s associated with the current stream.

Related topics:

flags setf

## width

Syntax:

```
#include <fstream>
int width();
int width( int w );
```

The function width() returns the current width, which is defined as the minimum number of characters to display with each output. The optional argument *w* can be used to set the width.

For example:

```
cout.width( 5 );
cout << "2";
```

displays

2

(that's four spaces followed by a '2')

Related topics:

<u>fill</u>

precision

## write

Syntax:

```
#include <fstream>
ostream& write( const char* buffer, streamsize num );
```

The write() function is used with output streams, and writes *num* bytes from *buffer* to the current output stream.

Related topics:

**flush** 

<u>put</u>

<u>read</u>

```
<u>cppreference.com</u> > <u>C++ Priority Queues</u>
```

# 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();
}</pre>
```

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:

# **Priority queue constructors**

Syntax:

```
#include <queue>
priority_queue( const Compare& cmp = Compare(), const Container&
priority_queue( input iterator start, input iterator end, const C
```

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*.

# 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);</pre>
```

### 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();
}</pre>
```

Related topics:

<u>pop</u>

```
<u>cppreference.com</u> > <u>C++ Queues</u>
```

### 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;</pre>
```

This code produces the following output:

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

The back() function runs in <u>constant time</u>.

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();
}</pre>
```

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);
}</pre>
```

Related topics:

<u>pop</u>

# **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();
}</pre>
```

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
```

#### 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

```
<u>cppreference.com</u> > <u>C++ Stacks</u>
```

# 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();
}</pre>
```

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);</pre>
```

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
```

## **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.

## 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();
}</pre>
```

Related topics:

<u>pop</u>

```
<u>cppreference.com</u> > <u>C++ Strings</u>
```

# append

#### Syntax:

```
#include <string>
  string& append( const string& str );
  string& append( const char* str );
  string& append( const string& str, size_type index, size_type len
  string& append( const char* str, size_type num );
  string& append( size_type num, char ch );
  string& append( input iterator start, input iterator end );
```

The append() function either:

- appends *str* on to the end of the current string,
- appends a substring of *str* starting at *index* that is *len* characters long on to the end of the current string,
- appends *num* characters of *str* on to the end of the current string,
- appends *num* repititions of *ch* on to the end of the current string,
- or appends the sequence denoted by *start* and *end* on to the end of the current string.

For example, the following code uses append() to add 10 copies of the '!' character to a string:

```
string str = "Hello World";
str.append( 10, '!' );
cout << str << endl;</pre>
```

That code displays:

```
Hello World!!!!!!!!
```

In the next example, append() is used to concatenate a substring of one string onto another string:

```
string str1 = "Eventually I stopped caring...";
string str2 = "but that was the '80s so nobody noticed.";
```

```
str1.append( str2, 25, 15 );
cout << "str1 is " << str1 << endl;
```

When run, the above code displays:

```
str1 is Eventually I stopped caring...nobody noticed.
```

# assign

Syntax:

```
#include <string>
void assign( size_type num, const char& val );
void assign( input iterator start, input iterator end );
string& assign( const string& str );
string& assign( const char* str );
string& assign( const char* str, size_type num );
string& assign( const string& str, size_type index, size_type len string& assign( size_type num, const char& ch );
```

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

In addition to the normal assign functionality that all C++ containers have, strings possess an assign() function that also allows them to:

- assign *str* to the current string,
- assign the first *num* characters of *str* to the current string,
- assign a substring of *str* starting at *index* that is *len* characters long to the current string,

For example, the following code:

```
string str1, str2 = "War and Peace";
str1.assign( str2, 4, 3 );
cout << str1 << endl;</pre>
```

displays

and

This function will destroy the previous contents of the string.

```
Related topics:
```

```
(C++ Lists) assign
```

### at

Syntax:

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

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

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;
}</pre>
```

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;
}</pre>
```

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) <u>Multimap operators</u>
(C++ Double-ended Queues) <u>Container operators</u>
```

# begin

Syntax:

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

The function begin() returns an iterator to the first element of the string. 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
   cout << *theIterator;
}</pre>
```

Related topics:

end rbegin rend

## c\_str

Syntax:

```
#include <string>
const char* c_str();
```

The function c\_str() returns a const pointer to a regular C string, identical to the

current string. The returned string is null-terminated.

Note that since the returned pointer is of type <u>const</u>, the character data that c\_str() returns **cannot be modified**. Furthermore, you do not need to call <u>free()</u> or <u>delete</u> on this pointer.

Related topics:

String operators data

# capacity

Syntax:

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

The capacity() function returns the number of elements that the string 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;</pre>
```

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 <a href="reserve">reserve</a>() 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 <string>
void clear();
```

The function clear() deletes all of the elements in the string. clear() runs in <u>linear</u> time.

Related topics:

(C++ Lists) erase

### compare

Syntax:

```
#include <string>
int compare( const string& str );
int compare( const char* str );
int compare( size_type index, size_type length, const string& str
int compare( size_type index, size_type length, const string& str
    size_type length2 );
int compare( size_type index, size_type length, const char* str,
```

The compare() function either compares *str* to the current string in a variety of ways, returning

Return Value	Case		
less than zero	this < str		
zero	this == str		
greater than zero	this > str		

The various functions either:

- compare *str* to the current string,
- compare *str* to a substring of the current string, starting at *index* for *length* characters,
- compare a substring of *str* to a substring of the current string, where *index2* and *length2* refer to *str* and *index* and *length* refer to the current string,
- or compare a substring of *str* to a substring of the current string, where the substring of *str* begins at zero and is *length2* characters long, and the substring of the current string begins at *index* and is *length* characters long.

For example, the following code uses compare() to compare four strings with eachother:

```
string names[] = {"Homer", "Marge", "3-eyed fish", "inanimate carbo

for( int i = 0; i < 4; i++ ) {
   for( int j = 0; j < 4; j++ ) {
     cout << names[i].compare( names[j] ) << " ";
   }
   cout << endl;
}</pre>
```

Data from the above code was used to generate this table, which shows how the various strings compare to eachother:

	Homer	Marge	3-eyed fish	inanimate carbon rod
"Homer".compare(x)	0	-1	1	-1

"Marge".compare( x )	1	0	1	-1
"3-eyed fish".compare( x )	-1	-1	0	-1
"inanimate carbon rod".compare( x )	1	1	1	0

Related topics:

**String operators** 

## copy

Syntax:

```
#include <string>
size_type copy( char* str, size_type num, size_type index = 0 );
```

The copy() function copies *num* characters of the current string (starting at *index* if it's specified, 0 otherwise) into *str*.

The return value of copy() is the number of characters copied.

For example, the following code uses copy() to extract a substring of a string into an array of characters:

```
char buf[30];
memset( buf, '\0', 30 );
string str = "Trying is the first step towards failure.";
str.copy( buf, 24 );
cout << buf << endl;</pre>
```

When run, this code displays:

```
Trying is the first step
```

Note that before calling copy(), we first call (Standard C String and Character) memset() to fill the destination array with copies of the **NULL** character. This step is included to make sure that the resulting array of characters is **NULL**-terminated.

Related topics:

## data

Syntax:

```
#include <string>
const char *data();
```

The function data() returns a pointer to the first character in the current string.

Related topics:

**String operators** 

c str

## empty

Syntax:

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

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

For example:

```
string s1;
string s2("");
string s3("This is a string");
cout.setf(ios::boolalpha);
cout << s1.empty() << endl;
cout << s2.empty() << endl;
cout << s3.empty() << endl;</pre>
```

When run, this code produces the following output:

```
true
true
```

false

Related topics:

**size** 

### end

Syntax:

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

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

Note that before you can access the last element of the string 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 <u>begin()</u> 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;
}</pre>
```

The iterator is initialized with a call to <a href="begin">begin</a>(). 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:

<u>begin</u> rbegin rend

### erase

#### *Syntax:*

```
#include <string>
  iterator erase( iterator loc );
  iterator erase( iterator start, iterator end );
  string& erase( size_type index = 0, size_type num = npos );
```

The erase() function either:

- removes the character pointed to by *loc*, returning an iterator to the next character,
- removes the characters between *start* and *end* (including the one at *start* but not the one at *end*), returning an iterator to the character after the last character removed,
- or removes *num* characters from the current string, starting at *index*, and returns \*this.

The parameters *index* and *num* have default values, which means that erase() can be called with just *index* to erase all characters after *index* or with no arguments to erase all characters.

### For example:

```
string s("So, you like donuts, eh? Well, have all the donuts in cout << "The original string is '" << s << "'" << endl;
s.erase( 50, 14 );
cout << "Now the string is '" << s << "'" << endl;
s.erase( 24 );
cout << "Now the string is '" << s << "'" << endl;
s.erase();
cout << "Now the string is '" << s << "'" << endl;</pre>
```

### will display

```
The original string is 'So, you like donuts, eh? Well, have all to Now the string is 'So, you like donuts, eh? Well, have all the donut the string is 'So, you like donuts, eh?'

Now the string is ''
```

erase() runs in <u>linear time</u>.

Related topics:

insert

## find

Syntax:

```
#include <string>
size_type find( const string& str, size_type index );
size_type find( const char* str, size_type index );
size_type find( const char* str, size_type index, size_type lengt
size_type find( char ch, size_type index );
```

The function find() either:

- returns the first occurrence of *str* within the current string, starting at *index*, string::npos if nothing is found,
- if the *length* parameter is given, then find() returns the first occurrence of the first *length* characters of *str* within the current string, starting at *index*, string::npos if nothing is found,
- or returns the index of the first occurrence *ch* within the current string, starting at *index*, string::npos if nothing is found.

For example:

```
string str1( "Alpha Beta Gamma Delta" );
string::size_type loc = str1.find( "Omega", 0 );
if( loc != string::npos ) {
  cout << "Found Omega at " << loc << endl;
} else {
  cout << "Didn't find Omega" << endl;
}</pre>
```

### Related topics:

```
find_first_not_of
find_first_of
find_last_not_of
find_last_of
```

## find\_first\_not\_of

Syntax:

```
#include <string>
size_type find_first_not_of( const string& str, size_type index =
size_type find_first_not_of( const char* str, size_type index = 0
size_type find_first_not_of( const char* str, size_type index, si
size_type find_first_not_of( char ch, size_type index = 0 );
```

The find\_first\_not\_of() function either:

- returns the index of the first character within the current string that does not match any character in *str*, beginning the search at *index*, string::npos if nothing is found,
- searches the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning string::npos,
- or returns the index of the first occurrence of a character that does not match *ch* in the current string, starting the search at *index*, string::npos if nothing is found.

For example, the following code searches a string of text for the first character that is not a lower-case character, space, comma, or hypen:

```
string lower_case = "abcdefghijklmnopqrstuvwxyz ,-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-Cout << "first non-lower-case letter in str at: " << str.find_first
```

When run, find\_first\_not\_of() finds the first upper-case letter in *str* at index 29 and displays this output:

```
first non-lower-case letter in str at: 29
```

Related topics:

**find** 

find\_first\_not\_of find\_first\_of find\_last\_not\_of find\_last\_of rfind

## find\_first\_of

#### Syntax:

```
#include <string>
size_type find_first_of( const string &str, size_type index = 0 )
size_type find_first_of( const char* str, size_type index = 0 );
size_type find_first_of( const char* str, size_type index, size_t
size_type find_first_of( char ch, size_type index = 0 );
```

The find\_first\_of() function either:

- returns the index of the first character within the current string that matches any character in *str*, beginning the search at *index*, string::npos if nothing is found,
- searches the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or string::npos if no characters match,
- or returns the index of the first occurrence of *ch* in the current string, starting the search at *index*, string::npos if nothing is found.

#### Related topics:

find find first not of find last not of find last of rfind

# find\_last\_not\_of

#### Syntax:

```
#include <string>
size_type find_last_not_of( const string& str, size_type index =
size_type find_last_not_of( const char* str, size_type index = np
size_type find_last_not_of( const char* str, size_type index, siz
size_type find_last_not_of( char ch, size_type index = npos );
```

The find\_last\_not\_of() function either:

- returns the index of the last character within the current string that does not match any character in *str*, doing a reverse search from *index*, string::npos if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning string::npos,
- or returns the index of the last occurrence of a character that does not match *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, the following code searches for the last non-lower-case character in a mixed string of characters:

```
string lower_case = "abcdefghijklmnopqrstuvwxyz";
string str = "abcdefgABCDEFGhijklmnop";
cout << "last non-lower-case letter in str at: " << str.find_last_
```

This code displays the following output:

```
last non-lower-case letter in str at: 13
```

### Related topics:

find find\_first\_not\_of find\_first\_of find\_last\_of rfind

## find\_last\_of

#### Syntax:

```
#include <string>
    size_type find_last_of( const string& str, size_type index = npos
    size_type find_last_of( const char* str, size_type index = npos )
    size_type find_last_of( const char* str, size_type index, size_ty
    size_type find_last_of( char ch, size_type index = npos );
```

The find\_last\_of() function either:

- does a reverse search from *index*, returning the index of the first character within the current string that matches any character in *str*, or string::npos if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or string::npos if no characters match,
- or does a reverse search from *index*, returning the index of the first occurrence of *ch* in the current string, string::npos if nothing is found.

### Related topics:

find first not of find first of find last not of rfind

# getline

### Syntax:

```
#include <string>
istream& getline( istream& is, string& s, char delimiter = '\n' )
```

The C++ string class defines the global function getline() to read strings from an I/O stream. The getline() function, which is not part of the string class, reads a

line from *is* and stores it into *s*. If a character *delimiter* is specified, then getline() will use *delimiter* to decide when to stop reading data.

For example, the following code reads a line of text from **stdin** and displays it to **stdout**:

```
string s;
getline( cin, s );
cout << "You entered " << s << endl;</pre>
```

After getting a line of data in a string, you may find that <u>string streams</u> are useful in extracting data from that string. For example, the following code reads numbers from standard input, ignoring any "commented" lines that begin with double slashes:

```
// expects either space-delimited numbers or lines that start with
// two forward slashes (//)
string s;
while( getline(cin,s) ) {
  if( s.size() >= 2 && s[0] == '/' && s[1] == '/' ) {
    cout << " ignoring comment: " << s << endl;
  } else {
    istringstream ss(s);
    double d;
    while( ss >> d ) {
      cout << " got a number: " << d << endl;
    }
  }
}</pre>
```

When run with a user supplying input, the above code might produce this output:

```
// test
  ignoring comment: // test
23.3 -1 3.14159
  got a number: 23.3
  got a number: -1
  got a number: 3.14159
// next batch
  ignoring comment: // next batch
1 2 3 4 5
  got a number: 1
  got a number: 2
  got a number: 3
  got a number: 4
  got a number: 5
```

```
50
got a number: 50
```

Related topics:

(C++ I/O) <u>get</u> (C++ I/O) <u>getline</u> <u>string streams</u>

### insert

#### Syntax:

```
#include <string>
iterator insert( iterator i, const char& ch );
string& insert( size_type index, const string& str );
string& insert( size_type index, const char* str );
string& insert( size_type index1, const string& str, size_type in
string& insert( size_type index, const char* str, size_type num )
string& insert( size_type index, size_type num, char ch );
void insert( iterator i, size_type num, const char& ch );
void insert( iterator i, iterator start, iterator end );
```

The very multi-purpose insert() function either:

- inserts *ch* before the character denoted by *i*,
- inserts *str* into the current string, at location *index*,
- inserts a substring of *str* (starting at *index2* and *num* characters long) into the current string, at location *index1*,
- inserts *num* characters of *str* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, before the character denoted by *i*,
- or inserts the characters denoted by *start* and *end* into the current string, before the character specified by *i*.

#### Related topics:

erase replace

# length

Syntax:

```
#include <string>
size_type length() const;
```

The length() function returns the number of elements in the current string, performing the same role as the <u>size()</u> function.

Related topics:

<u>size</u>

## max\_size

Syntax:

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

The max\_size() function returns the maximum number of elements that the string 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 string and the the number of elements that the string will be able to hold before more memory will have to be allocated, respectively.

Related topics:

size

# push\_back

Syntax:

```
#include <string>
void push_back( char c );
```

The push\_back() function appends *c* to the end of the string.

For example, the following code adds 10 characters to a string:

```
string the_string;
for( int i = 0; i < 10; i++ )
  the_string.push_back( i+'a' );</pre>
```

When displayed, the resulting string would look like this:

```
abcdefghij
```

push\_back() runs in constant time.

Related topics:

assign insert

# rbegin

Syntax:

```
#include <string>
  reverse iterator rbegin();
const_reverse iterator rbegin() const;
```

The rbegin() function returns a <u>reverse\_iterator</u> to the end of the current string.

rbegin() runs in constant time.

Related topics:

begin end rend

### rend

#### Syntax:

```
#include <string>
  reverse iterator rend();
const_reverse iterator rend() const;
```

The function rend() returns a <u>reverse\_iterator</u> to the beginning of the current string.

rend() runs in <u>constant time</u>.

Related topics:

begin end rbegin

## replace

#### Syntax:

```
#include <string>
  string& replace( size_type index, size_type num, const string& st
  string& replace( size_type index1, size_type num1, const string&
  string& replace( size_type index, size_type num, const char* str
  string& replace( size_type index, size_type num1, const char* str
  string& replace( size_type index, size_type num1, size_type num2,
  string& replace( iterator start, iterator end, const string& str
  string& replace( iterator start, iterator end, const char* str, s
  string& replace( iterator start, iterator end, const char* str, s
  string& replace( iterator start, iterator end, size_type num, cha
```

The function replace() either:

- replaces characters of the current string with up to *num* characters from *str*, beginning at *index*,
- replaces up to *num1* characters of the current string (starting at *index1*) with up to *num2* characters from *str* beginning at *index2*,
- replaces up to *num* characters of the current string with characters from *str*, beginning at *index* in *str*,

- replaces up to *num1* characters in the current string (beginning at *index1*) with *num2* characters from *str* beginning at *index2*,
- replaces up to *num1* characters in the current string (beginning at *index*) with *num2* copies of *ch*,
- replaces the characters in the current string from *start* to *end* with *str*,
- replaces characters in the current string from *start* to *end* with *num* characters from *str*,
- or replaces the characters in the current string from *start* to *end* with *num* copies of *ch*.

For example, the following code displays the string "They say he carved it himself...find your soul-mate, Homer."

```
string s = "They say he carved it himself...from a BIGGER spoon"
string s2 = "find your soul-mate, Homer.";
s.replace( 32, s2.length(), s2 );
cout << s << endl;</pre>
```

Related topics:

<u>insert</u>

### reserve

Syntax:

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

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

reserve() runs in <u>linear time</u>.

Related topics:

**capacity** 

## resize

#### Syntax:

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

The function resize() changes the size of the string 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) <u>Multimap constructors & destructors</u>
<a href="mailto:capacity">capacity</a>
size

## rfind

#### Syntax:

```
#include <string>
size_type rfind( const string& str, size_type index );
size_type rfind( const char* str, size_type index );
size_type rfind( const char* str, size_type index, size_type num
size_type rfind( char ch, size_type index );
```

The rfind() function either:

- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, string::npos if nothing is found,
- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, searching at most *num* characters, string::npos if nothing is found,
- or returns the location of the first occurrence of *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, in the following code, the first call to rfind() returns string::npos, because the target word is not within the first 8 characters of the string. However, the second call returns 9, because the target word is within 20 characters of the beginning of the string.

```
int loc;
string s = "My cat's breath smells like cat food.";
loc = s.rfind( "breath", 8 );
cout << "The word breath is at index " << loc << endl;
loc = s.rfind( "breath", 20 );
cout << "The word breath is at index " << loc << endl;</pre>
```

#### Related topics:

**find** 

find\_first\_not\_of

find first of

find\_last\_not\_of

find last of

## size

Syntax:

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

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

### Related topics:

**capacity** 

**empty** 

length

max\_size

resize

# **String constructors**

Syntax:

```
#include <string>
string();
string( const string& s );
string( size_type length, const char& ch );
```

```
string( const char* str );
string( const char* str, size_type length );
string( const string& str, size_type index, size_type length );
string( input iterator start, input iterator end );
~string();
```

The string constructors create a new string containing:

- nothing; an empty string,
- a copy of the given string *s*,
- *length* copies of *ch*,
- a duplicate of *str* (optionally up to *length* characters long),
- a substring of *str* starting at *index* and *length* characters long
- a string of characters denoted by the start and end iterators

For example,

```
string str1( 5, 'c' );
string str2( "Now is the time..." );
string str3( str2, 11, 4 );
cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;</pre>
```

displays

```
ccccc
Now is the time...
time
```

The string constructors usually run in <u>linear time</u>, except the empty constructor, which runs in <u>constant time</u>.

# **String operators**

Syntax:

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

```
bool operator<(const string& c1, const string& c2);
bool operator>(const string& c1, const string& c2);
bool operator>=(const string& c1, const string& c2);
bool operator>=(const string& c1, const string& c2);
string operator+(const string& s1, const string& s2 );
string operator+(const char* s, const string& s2 );
string operator+( char c, const string& s2 );
string operator+( const string& s1, const char* s );
string operator+( const string& s1, char c );
ostream& operator<<( ostream& os, const string& s );
istream& operator>( istream& is, string& s );
string& operator=( const string& s );
string& operator=( const char* s );
string& operator=( char ch );
char& operator[]( size_type index );
```

C++ strings can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one string to another takes <u>linear time</u>.

Two strings are equal if:

```
    Their size is the same, and
    Each member in location i in one string is equal to the the mer
```

Comparisons among strings are done lexicographically.

In addition to the normal container operators, strings can also be concatenated with the + operator and fed to the C++ I/O stream classes with the << and >> operators.

For example, the following code concatenates two strings and displays the result:

```
string s1 = "Now is the time...";
string s2 = "for all good men...";
string s3 = s1 + s2;
cout << "s3 is " << s3 << endl;</pre>
```

Futhermore, strings can be assigned values that are other strings, character arrays, or even single characters. The following code is perfectly valid:

```
char ch = 'N';
string s;
s = ch;
```

Individual characters of a string can be examined with the [] operator, which runs in <u>constant time</u>.

Related topics:

<u>c\_str</u> <u>compare</u> data

### substr

#### Syntax:

```
#include <string>
string substr( size_type index, size_type length = npos );
```

The substr() function returns a substring of the current string, starting at *index*, and *length* characters long. If *length* is omitted, it will default to string::npos, and the substr() function will simply return the remainder of the string starting at *index*.

For example:

```
string s("What we have here is a failure to communicate");
string sub = s.substr(21);
cout << "The original string is " << s << endl;
cout << "The substring is " << sub << endl;</pre>
```

displays

```
The original string is What we have here is a failure to communicate
```

Related topics:

**copy** 

## swap

Syntax:

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

The swap() function exchanges the elements of the current string with those of *from*. This function operates in <u>constant time</u>.

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;</pre>
```

The above code displays:

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

Related topics:

(C++ Lists) splice

```
<u>cppreference.com</u> > <u>C++ Algorithms</u>
```

## accumulate

#### Syntax:

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

The accumulate() 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, itera
iterator adjacent_difference( iterator start, iterator end, itera
```

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 " << *resul
}</pre>
```

Related topics:

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
```

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 ) ) {</pre>
```

```
cout << "nums[] contains " << i << endl;
} else {
  cout << "nums[] DOES NOT contain " << i << endl;
}
</pre>
```

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
is_sorted
lower_bound
partial_sort
partial_sort_copy
sort
stable_sort
upper_bound
```

## 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 <u>linear</u> 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;</pre>
```

Related topics:

copy\_backward copy\_n generate remove\_copy swap transform

# copy\_backward

Syntax:

```
#include <algorithm>
iterator copy_backward( iterator start, iterator end, iterator de
```

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.er

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

The above code produces the following output:

```
to_vector contains: 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_</pre>
```

The above code displays the following output:

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

Related topics:
accumulate
adjacent\_difference
count\_if
inner\_product
partial sum

## 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:

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, BinP
```

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;
}</pre>
```

Related topics:

find\_if lexicographical\_compare mismatch search

## equal\_range

Syntax:

```
#include <algorithm>
pair<iterator,iterator> equal_range( iterator first, iterator las
pair<iterator,iterator> equal_range( iterator first, iterator las
```

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 <a href="lower\_bound">lower\_bound</a>() and `upper\_bound1`() functions, since the first of the pair of iterators that it returns is what <a href="lower\_bound">lower\_bound</a>() 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:

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;</pre>
```

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
```

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;</pre>
```

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 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 << er
}

else {
   cout << "Found a matching element: " << *result << endl;
}</pre>
```

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 << end } else {
   cout << "Found a matching number: " << *result << endl;
}</pre>
```

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_sta
iterator find_end( iterator start, iterator end, iterator seq_sta
```

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
if( *result == nums[end] ) {
 cout << "Did not find any subsequence matching { 1, 2, 3 }" << er</pre>
} else {
  cout << "The last matching subsequence is at: " << *result << en</pre>
int target2[] = \{ 3, 2, 3 \};
result = find_end( nums + start, nums + end, target2 + 0, target2
if( *result == nums[end] ) {
 cout << "Did not find any subsequence matching { 3, 2, 3 }" << er</pre>
} else {
  cout << "The last matching subsequence is at: " << *result << end
```

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 fi
iterator find_first_of( iterator start, iterator end, iterator fi
```

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 + 0, targets + 0 target
```

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>(),
if( *result == nums[end] ) {
  cout << "Did not find any number greater than zero" << endl;
} else {
  cout << "Found a positive non-zero number: " << *result << endl;
}</pre>
```

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, UnaryFuncti
```

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 operator() (TYPE& x) {
          x++;
     }
};
...
int nums[] = {3, 4, 2, 9, 15, 267};</pre>
```

```
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;</pre>
```

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 fill generate\_n 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, i
bool includes( iterator start1, iterator end1, iterator start2, i
```

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 <u>linear time</u>.

Related topics:

set difference
set intersection
set symmetric difference
set union

# inner\_product

Syntax:

```
#include <numeric>
    TYPE inner_product( iterator start1, iterator end1, iterator star
    TYPE inner_product( iterator start1, iterator end1, iterator star
```

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 <u>linear time</u>.

Related topics:

accumulate
adjacent\_difference
count
partial\_sum

# inplace\_merge

Syntax:

```
#include <algorithm>
inline void inplace_merge( iterator start, iterator middle, itera
inline void inplace_merge( iterator start, iterator middle, itera
```

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:

<u>merge</u>

# is\_heap

Syntax:

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

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 <u>linear time</u>.

```
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
```

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 <a href="linear-time">linear time</a>.

### Related topics:

binary search

partial\_sort

partial\_sort\_copy

<u>sort</u>

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

```
would.

Related topics:
```

<u>swap</u>

swap\_ranges

# lexicographical\_compare

Syntax:

```
#include <algorithm>
bool lexicographical_compare( iterator start1, iterator end1, ite
bool lexicographical_compare( iterator start1, iterator end1, ite
```

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 <a href="mailto:linear time">linear time</a>.

Related topics:

equal lexicographical\_compare\_3way mismatch

# lexicographical\_compare\_3way

Syntax:

```
#include <algorithm>
int lexicographical_compare_3way( iterator start1, iterator end1,
```

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&
iterator lower_bound( iterator first, iterator last, const TYPE&
```

The lower\_bound() function is a type of <u>binary\_search(</u>). 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: ";</pre>
for( unsigned int i = 0; i < nums.size(); i++ ) {</pre>
  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: ";</pre>
for( unsigned int i = 0; i < nums.size(); i++ ) {</pre>
  cout << nums[i] << " ";</pre>
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
```

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 <u>linear time</u>.

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)
```

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+ar)

vector<char> v;
v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back(
cout << "Max element in the vector v is " << *max_element( v.begin)</pre>
```

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 merge( iterator start1, iterator end1, iterator start2,
```

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:

<u>max</u>

max\_element

min

## mismatch

Syntax:

```
#include <algorithm>
```

```
pair <iterator1,iterator2> mismatch( iterator start1, iterator en
pair <iterator1,iterator2> mismatch( iterator start1, iterator en
```

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 <u>linear time</u>.

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, StrictWeakOr
```

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,
```

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 <u>linear time</u> 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,
```

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

#### Syntax:

```
#include <algorithm>
iterator partial_sort_copy( iterator start, iterator end, iterato
iterator partial_sort_copy( iterator start, iterator end, iterato
```

The partial\_sort\_copy() algorithm behaves like <u>partial\_sort(</u>), 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 resu
  iterator partial_sum( iterator start, iterator end, iterator resu
```

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 <u>linear time</u>.

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 <u>linear time</u>.

Related topics: stable\_partition

# pop\_heap

#### Syntax:

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

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 <u>logarithmic time</u>.

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, StrictWeakOr
```

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

# push\_heap

#### Syntax:

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

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 make\_heap pop\_heap sort\_heap

# random\_sample

### Syntax:

```
#include <algorithm>
iterator random_sample( iterator start1, iterator end1, iterator
iterator random_sample( iterator start1, iterator end1, iterator
```

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 prev\_permutation random\_sample\_n random\_shuffle

## random\_sample\_n

#### Syntax:

```
#include <algorithm>
iterator random_sample_n( iterator start, iterator end, iterator
iterator random_sample_n( iterator start, iterator end, iterator
```

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

#### Syntax:

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

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 <u>linear time</u>.

### Related topics:

```
remove_copy
remove_if
remove_if
unique
unique_copy
```

## remove\_copy

#### Syntax:

```
#include <algorithm>
iterator remove_copy( iterator start, iterator end, iterator resu
```

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 <u>linear</u> 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 r
```

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 <u>linear time</u>.

Related topics:

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 <u>linear time</u>.

Related topics:

remove\_copy remove\_copy\_if

# replace

Syntax:

```
#include <algorithm>
void replace( iterator start, iterator end, const TYPE& old_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 res
```

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:

<u>replace</u>

# replace\_copy\_if

Syntax:

```
#include <algorithm>
iterator replace_copy_if( iterator start, iterator end, iterator
```

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:

<u>replace</u>

## replace\_if

Syntax:

```
#include <algorithm>
void replace_if( iterator start, iterator end, Predicate p, const
```

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 res
```

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
```

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 <u>linear time</u>.

Related topics:

rotate\_copy

## rotate\_copy

Syntax:

```
#include <algorithm>
iterator rotate_copy( iterator start, iterator middle, iterator e
```

The rotate\_copy() algorithm is similar to the <u>rotate()</u> algorithm, except that the range of elements is copied to *result* before being rotated.

## search

Syntax:

```
#include <algorithm>
iterator search( iterator start1, iterator end1, iterator start2,
iterator search( iterator start1, iterator end1, iterator start2,
```

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 <u>linear time</u>.

Related topics:

equal
find
lexicographical\_compare
mismatch
search\_n

## search\_n

Syntax:

```
#include <algorithm>
iterator search_n( iterator start, iterator end, size_t num, cons
iterator search_n( iterator start, iterator end, size_t num, cons
```

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 find\_if search

## set\_difference

#### Syntax:

```
#include <algorithm>
iterator set_difference( iterator start1, iterator end1, iterator
iterator set_difference( iterator start1, iterator end1, iterator
```

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\_intersection
set\_symmetric\_difference
set\_union

## set\_intersection

#### Syntax:

```
#include <algorithm>
iterator set_intersection( iterator start1, iterator end1, iterat
iterator set_intersection( iterator start1, iterator end1, iterat
```

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\_difference set\_symmetric\_difference set\_union

# set\_symmetric\_difference

### Syntax:

```
#include <algorithm>
iterator set_symmetric_difference( iterator start1, iterator end1
iterator set_symmetric_difference( iterator start1, iterator end1
```

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\_difference
set\_intersection
set\_union

## set\_union

#### Syntax:

```
#include <algorithm>
iterator set_union( iterator start1, iterator end1, iterator star
iterator set_union( iterator start1, iterator end1, iterator star
```

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
merge
set\_difference
set\_intersection
set\_symmetric\_difference

### 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 <u>linear time</u>.

#### 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 << out << v[i] << " ";
}
cout << endl;</pre>
```

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;</pre>
```

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;</pre>
```

```
sort( v.begin(), v.end(), cmp );

cout << "After: ";
for( int i = 0; i < 10; i++ ) {
   cout << v[i] << " ";
}
cout << endl;</pre>
```

#### 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
```

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, Predicat
```

The stable\_partition() function behaves similarly to <u>partition()</u>. 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, StrictWeakOrderin
```

The stable\_sort() algorithm is like the <u>sort()</u> algorithm, in that it sorts a range of elements into ascending order. Unlike <u>sort()</u>, 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 st
```

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:

```
<u>iter_swap</u>
<u>swap</u>
```

## transform

#### *Syntax:*

```
#include <algorithm>
iterator transform( iterator start, iterator end, iterator result
iterator transform( iterator start1, iterator end1, iterator star
```

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 resu
iterator unique_copy( iterator start, iterator end, iterator resu
```

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 <a href="mailto:lineartime">lineartime</a>.

Related topics: adjacent find

remove

unique

# upper\_bound

#### Syntax:

```
#include <algorithm>
iterator upper_bound( iterator start, iterator end, const <u>TYPE</u>& v
iterator upper_bound( iterator start, iterator end, const <u>TYPE</u>& v
```

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

## auto\_ptr

Syntax:

```
#include <memory>
auto_ptr<class TYPE> name
```

The auto\_ptr class allows the programmer to create pointers that point to other objects. When auto\_ptr pointers are destroyed, the objects to which they point are also destroyed.

The auto\_ptr class supports normal pointer operations like =, \*, and ->, as well as two functions TYPE\* get() and TYPE\* release(). The get() function returns a pointer to the object that the auto\_ptr points to. The release() function acts similarly to the get() function, but also relieves the auto\_ptr of its memory destruction duties. When an auto\_ptr that has been released goes out of scope, it will not call the destructor of the object that it points to.

**Warning:** It is generally a **bad idea** to put auto\_ptr objects inside C++ STL containers. C++ containers can do funny things with the data inside them, including frequent reallocation (when being copied, for instance). Since calling the destructor of an auto\_ptr object will free up the memory associated with that object, any C++ container reallocation will cause any auto\_ptr objects to become invalid.

#### Example code:

```
#include <memory>
using namespace std;

class MyClass {
public:
   MyClass() {} // nothing
   ~MyClass() {} // nothing
   void myFunc() {} // nothing
};

int main() {
   auto_ptr<MyClass> ptr1(new MyClass), ptr2;
```

```
ptr2 = ptr1;
ptr2->myFunc();

MyClass* ptr = ptr2.get();

ptr->myFunc();

return 0;
}
```

```
<u>cppreference.com</u> > <u>C++ String Streams</u>
```

# **String Stream Constructors**

Syntax:

```
#include <sstream>
  stringstream()
  stringstream( openmode mode )
  stringstream( string s, openmode mode )
  ostringstream()
  ostringstream( openmode mode )
  ostringstream( string s, openmode mode )
  istringstream()
  istringstream( openmode mode )
  istringstream( string s, openmode mode )
```

The stringstream, ostringstream, and istringstream objects are used for input and output to a string. They behave in a manner similar to fstream, ofstream and ifstream objects.

The optional *mode* parameter defines how the file is to be opened, according to the <u>io stream mode flags</u>.

An ostringstream object can be used to write to a string. This is similar to the C sprintf() function. For example:

```
ostringstream s1;
int i = 22;
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;</pre>
```

An istringstream object can be used to read from a string. This is similar to the C sscanf() function. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25</pre>
```

You can also specify the input string in the istringstream constructor as in this example:

```
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25</pre>
```

A stringstream object can be used for both input and output to a string like an fstream object.

Related topics:

C++ I/O Streams

# **String Stream Operators**

Syntax:

```
#include <sstream>
operator<<
operator>>
```

Like <u>C++ I/O Streams</u>, the simplest way to use string streams is to take advantage of the overloaded << and >> operators.

The << operator inserts data into the stream. For example:

```
stream1 << "hello" << i;
```

This example inserts the string "hello" and the variable i into stream1. In contrast, the >> operator extracts data out of a string stream:

```
stream1 >> i;
```

This code reads a value from *stream1* and assigns the variable *i* that value.

Related topics:

C++ I/O Streams

## rdbuf

Syntax:

```
#include <sstream>
stringbuf* rdbuf();
```

The *rdbuf()* function returns a pointer to the string buffer for the current string stream.

Related topics:

str()

C++ I/O Streams

#### str

Syntax:

```
#include <sstream>
void str( string s );
string str();
```

The function *str()* can be used in two ways. First, it can be used to get a copy of the string that is being manipulated by the current stream string. This is most useful with output strings. For example:

```
ostringstream stream1;
stream1 << "Testing!" << endl;
cout << stream1.str();</pre>
```

Second, *str()* can be used to copy a string into the stream. This is most useful with input strings. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
```

*str()*, along with *clear()*, is also handy when you need to clear the stream so that it can be reused:

```
istringstream stream1;
float num;

// use it once
string string1 = "25 1 3.235\n11111111\n222222";
stream1.str(string1);
while( stream1 >> num ) cout << "num: " << num << endl; // displa

// use the same string stream again with clear() and str()
string string2 = "1 2 3 4 5 6 7 8 9 10";
stream1.clear();
stream1.str(string2);

while( stream1 >> num ) cout << "num: " << num << endl; // displa</pre>
```

*Related topics:* 

rdbuf()

C++ I/O Streams

#### <u>cppreference.com</u> > <u>C/C++ Pre-processor Commands</u>

## #define

Syntax:

```
#define macro-name replacement-string
```

The #define command is used to make substitutions throughout the file in which it is located. In other words, #define causes the compiler to go through the file, replacing every occurrence of *macro-name* with *replacement-string*. The replacement string stops at the end of the line.

Example code:

Here's a typical use for a #define (at least in C):

```
#define TRUE 1
#define FALSE 0
...
int done = 0;
while( done != TRUE ) {
    ...
}
```

Another feature of the #define command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define absolute_value( x ) ( ((x) < 0) ? -(x) : (x) )
...
int num = -1;
while( absolute_value( num ) ) {
...
}</pre>
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within it's own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run

the risk of having the compiler misinterpret your code.

Here is an example of how to use the #define command to create a general purpose incrementing for loop that prints out the integers 1 through 20:

```
#define count_up( v, low, high ) \
  for( (v) = (low); (v) <= (high); (v)++ )
  ...
int i;
count_up( i, 1, 20 ) {
  printf( "i is %d\n", i );
}</pre>
```

#### Related topics:

#, ## #if, #ifdef, #ifndef, #else, #elif, #endif #undef

## #error

Syntax:

```
#error message
```

The #error command simply causes the compiler to stop when it is encountered. When an #error is encountered, the compiler spits out the line number and whatever *message* is. This command is mostly used for debugging.

## #include

Syntax:

```
#include <filename>
#include "filename"
```

This command slurps in a file and inserts it at the current location. The main

difference between the syntax of the two items is that if *filename* is enclosed in angled brackets, then the compiler searches for it somehow. If it is enclosed in quotes, then the compiler doesn't search very hard for the file.

While the behavior of these two searches is up to the compiler, usually the angled brackets means to search through the standard library directories, while the quotes indicate a search in the current directory. The spiffy new C++ #include commands don't need to map directly to filenames, at least not for the standard libraries. That's why you can get away with

```
#include <iostream>
```

and not have the compiler choke on you.

## #line

Syntax:

```
#line line_number "filename"
```

The #line command is simply used to change the value of the \_\_LINE\_\_ and \_\_FILE\_\_ variables. The filename is optional. The \_\_LINE\_\_ and \_\_FILE\_\_ variables represent the current file and which line is being read. The command

```
#line 10 "main.cpp"
```

changes the current line number to 10, and the current file to "main.cpp".

# #pragma

The #pragma command gives the programmer the ability to tell the compiler to do certain things. Since the #pragma command is implementation specific, uses vary from compiler to compiler. One option might be to trace program execution.

# #if, #ifdef, #ifndef, #else, #elif, #endif

These commands give simple logic control to the compiler. As a file is being compiled, you can use these commands to cause certain lines of code to be included or not included.

```
#if expression
```

If the value of expression is true, then the code that immediately follows the command will be compiled.

```
#ifdef macro
```

If the *macro* has been defined by a <u>#define</u> statement, then the code immediately following the command will be compiled.

```
#ifndef macro
```

If the *macro* has not been defined by a <u>#define</u> statement, then the code immediately following the command will be compiled.

A few side notes: The command #elif is simply a horribly truncated way to say "elseif" and works like you think it would. You can also throw in a "defined" or "!defined" after an #if to get added functionality.

Example code:

Here's an example of all these:

```
#ifdef DEBUG
  cout << "This is the test version, i=" << i << endl;
#else
  cout << "This is the production version!" << endl;
#endif</pre>
```

You might notice how that second example could make debugging a lot easier than inserting and removing a million "cout"s in your code.

Related topics:

#define

# **Predefined preprocessor variables**

Syntax:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

The following variables can vary by compiler, but generally work:

- The \_\_LINE\_\_ and \_\_FILE\_\_ variables represent the current line and current file being processed.
- The \_\_DATE\_\_ variable contains the current date, in the form month/day/year. This is the date that the file was compiled, not necessarily the current date.
- The \_\_TIME\_\_ variable represents the current time, in the form hour:minute:second. This is the time that the file was compiled, not necessarily the current time.
- The \_\_cplusplus variable is only defined when compiling a C++ program. In some older compilers, this is also called c\_plusplus.
- The \_\_STDC\_\_ variable is defined when compiling a C program, and may also be defined when compiling C++.

# **#,** ##

The # and ## operators are used with the <u>#define</u> macro. Using # causes the first

argument after the # to be returned as a string in quotes. Using ## concatenates what's before the ## with what's after it.

Example code:

For example, the command

```
#define to_string( s ) # s
```

will make the compiler turn this command

```
cout << to_string( Hello World! ) << endl;</pre>
```

into

```
cout << "Hello World!" << endl;
```

Here is an example of the ## command:

```
#define concatenate( x, y ) x ## y
...
int xy = 10;
...
```

This code will make the compiler turn

```
cout << concatenate( x, y ) << endl;</pre>
```

into

```
cout << xy << endl;
```

which will, of course, display '10' to standard output.

Related topics:

#define

# #undef

The #undef command undefines a previously defined macro variable, such as a variable defined by a <u>#define</u>.

Related topics:

#define

#### cppreference.com > Other Standard C Functions

## abort

*Syntax:* 

```
#include <cstdlib>
void abort( void );
```

The function abort() terminates the current program. Depending on the implementation, the return value can indicate failure.

*Related topics:* 

assert atexit

exit

#### assert

Syntax:

```
#include <cassert>
assert( exp );
```

The assert() macro is used to test for errors. If *exp* evaluates to zero, assert() writes information to **stderr** and exits the program. If the macro NDEBUG is defined, the assert() macros will be ignored.

Related topics:

abort

# atexit

*Syntax:* 

#include <cstdlib>

```
int atexit( void (*func)(void) );
```

The function atexit() causes the function pointed to by *func* to be called when the program terminates. You can make multiple calls to atexit() (at least 32, depending on your compiler) and those functions will be called in reverse order of their establishment. The return value of atexit() is zero upon success, and non-zero on failure.

Related topics:

<u>abort</u> exit

## bsearch

Syntax:

```
#include <cstdlib>
void *bsearch( const void *key, const void *buf, size_t num, size
```

The bsearch() function searches *buf*[0] to *buf*[num-1] for an item that matches *key*, using a binary search. The function *compare* should return negative if its first argument is less than its second, zero if equal, and positive if greater. The items in the array *buf* should be in ascending order. The return value of bsearch() is a pointer to the matching item, or **NULL** if none is found.

Related topics:

**qsort** 

#### exit

Syntax:

```
#include <cstdlib>
void exit( int exit_code );
```

The exit() function stops the program. *exit\_code* is passed on to be the return

value of the program, where usually zero indicates success and non-zero indicates an error.

Related topics:

abort
atexit
system

## getenv

Syntax:

```
#include <cstdlib>
char *getenv( const char *name );
```

The function getenv() returns environmental information associated with *name*, and is very implementation dependent. **NULL** is returned if no information about *name* is available.

Related topics:

**system** 

# longjmp

Syntax:

```
#include <csetjmp>
void longjmp( jmp_buf envbuf, int status );
```

The function longjmp() causes the program to start executing code at the point of the last call to <a href="setjmp">setjmp</a>(). <a href="setjmp">envbuf</a> is usually set through a call to <a href="setjmp">setjmp</a>(). <a href="setjmp">status</a> becomes the return value of <a href="setjmp">setjmp</a>() and can be used to figure out where longjmp() came from. <a href="setjmp">status</a> should not be set to zero.

Related topics:

<u>setjmp</u>

## qsort

Syntax:

```
#include <cstdlib>
void qsort( void *buf, size_t num, size_t size, int (*compare)(co
```

The qsort() function sorts *buf* (which contains *num* items, each of size *size*) using Quicksort. The *compare* function is used to compare the items in *buf. compare* should return negative if the first argument is less than the second, zero if they are equal, and positive if the first argument is greater than the second. qsort() sorts *buf* in ascending order.

#### Example code:

For example, the following bit of code uses qsort() to sort an array of integers:

```
int compare_ints( const void* a, const void* b ) {
 int* arg1 = (int*) a;
  int* arg2 = (int*) b;
 if( *arg1 < *arg2 ) return -1;
  else if( *arg1 == *arg2 ) return 0;
  else return 1;
}
int array[] = \{-2, 99, 0, -743, 2, 3, 4\};
int array_size = 7;
. . .
printf( "Before sorting: " );
for( int i = 0; i < array_size; i++ ) {
 printf( "%d ", array[i] );
printf( "\n" );
qsort( array, array_size, sizeof(int), compare_ints );
printf( "After sorting: " );
for( int i = 0; i < array_size; i++ ) {
 printf( "%d ", array[i] );
printf( "\n" );
```

When run, this code displays the following output:

```
Before sorting: -2 99 0 -743 2 3 4
After sorting: -743 -2 0 2 3 4 99
```

Related topics:

**bsearch** 

(C++ Algorithms) sort

## raise

Syntax:

```
#include <csignal>
int raise( int signal );
```

The raise() function sends the specified *signal* to the program. Some signals:

Signal	Meaning
SIGABRT	Termination error
SIGFPE	Floating pointer error
SIGILL	Bad instruction
SIGINT	User presed CTRL-C
SIGSEGV	Illegal memory access
SIGTERM	Terminate program

The return value is zero upon success, nonzero on failure.

Related topics:

<u>signal</u>

# rand

Syntax:

```
#include <cstdlib>
int rand( void );
```

The function rand() returns a pseudorandom integer between zero and RAND\_MAX. An example:

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );</pre>
```

Related topics:

**srand** 

# setjmp

Syntax:

```
#include <csetjmp>
int setjmp( jmp_buf envbuf );
```

The setjmp() function saves the system stack in *envbuf* for use by a later call to <u>longjmp</u>(). When you first call setjmp(), its return value is zero. Later, when you call <u>longjmp</u>(), the second argument of <u>longjmp</u>() is what the return value of setjmp() will be. Confused? Read about <u>longjmp</u>().

Related topics:

**longjmp** 

# signal

Syntax:

```
#include <csignal>
void ( *signal( int signal, void (* func) (int)) ) (int);
```

The signal() function sets *func* to be called when *signal* is recieved by your program. *func* can be a custom signal handler, or one of these macros (defined in the csignal header file):

Macro	Explanation
SIG_DFL	default signal handling
SIG_IGN	ignore the signal

Some basic signals that you can attach a signal handler to are:

Signal	Description
SIGTERM	Generic stop signal that can be caught.
SIGINT	Interrupt program, normally ctrl-c.
SIGQUIT	Interrupt program, similar to SIGINT.
SIGKILL	Stops the program. Cannot be caught.
SIGHUP	Reports a disconnected terminal.

The return value of signal() is the address of the previously defined function for this signal, or SIG\_ERR is there is an error.

#### Example code:

The following example uses the signal() function to call an arbitrary number of functions when the user aborts the program. The functions are stored in a vector, and a single "clean-up" function calls each function in that vector of functions when the program is aborted:

```
void f1() {
  cout << "calling f1()..." << endl;
}

void f2() {
  cout << "calling f2()..." << endl;
}

typedef void(*endFunc)(void);
vector<endFunc> endFuncs;
```

```
void cleanUp( int dummy ) {
  for( unsigned int i = 0; i < endFuncs.size(); i++ ) {</pre>
    endFunc f = endFuncs.at(i);
    (*f)();
 exit(-1);
int main() {
  // connect various signals to our clean-up function
  signal( SIGTERM, cleanUp );
  signal( SIGINT, cleanUp );
  signal( SIGQUIT, cleanUp');
  signal( SIGHUP, cleanUp );
  // add two specific clean-up functions to a list of functions
  endFuncs.push_back( f1 );
  endFuncs.push_back( f2 );
  // loop until the user breaks
 while( 1 );
  return 0;
```

Related topics:

<u>raise</u>

### srand

Syntax:

```
#include <cstdlib>
void srand( unsigned seed );
```

The function srand() is used to seed the random sequence generated by <u>rand()</u>. For any given *seed*, rand() will generate a specific "random" sequence over and over again.

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );</pre>
```

Related topics:

rand

(Standard C Date & Time) time

## system

Syntax:

```
#include <cstdlib>
int system( const char *command );
```

The system() function runs the given *command* by passing it to the default command interpreter.

The return value is usually zero if the command executed without errors. If *command* is **NULL**, system() will test to see if there is a command interpreter available. Non-zero will be returned if there is a command interpreter available, zero if not.

Related topics:

exit

<u>getenv</u>

## va\_arg

Syntax:

```
#include <cstdarg>
type va_arg( va_list argptr, type );
void va_end( va_list argptr );
void va_start( va_list argptr, last_parm );
```

The va\_arg() macros are used to pass a variable number of arguments to a function.

1. First, you must have a call to va\_start() passing a valid va\_list and the

- mandatory first argument of the function. This first argument can be anything; one way to use it is to have it be an integer describing the number of parameters being passed.
- 2. Next, you call va\_arg() passing the **va\_list** and the type of the argument to be returned. The return value of va\_arg() is the current parameter.
- 3. Repeat calls to va\_arg() for however many arguments you have.
- 4. Finally, a call to va\_end() passing the **va\_list** is necessary for proper cleanup.

#### For example:

```
int sum( int num, ... ) {
   int answer = 0;
   va_list argptr;

  va_start( argptr, num );

  for( ; num > 0; num-- ) {
      answer += va_arg( argptr, int );
   }

  va_end( argptr );

  return( answer );
}

int main( void ) {
  int answer = sum( 4, 4, 3, 2, 1 );
  printf( "The answer is %d\n", answer );
  return( 0 );
}
```

This code displays 10, which is 4+3+2+1.

Here is another example of variable argument function, which is a simple printing function:

```
void my_printf( char *format, ... ) {
  va_list argptr;

va_start( argptr, format );
```

```
while( *format != '\0' ) {
    // string
    if( *format == 's' ) {
      char* s = va_arg( argptr, char * );
      printf( "Printing a string: %s\n", s );
    // character
    else if( *format == 'c' ) {
      char c = (char) va_arg( argptr, int );
      printf( "Printing a character: %c\n", c );
      break;
    }
    // integer
    else if( *format == 'd' ) {
      int d = va_arg( argptr, int );
      printf( "Printing an integer: %d\n", d );
    *format++;
 va_end( argptr );
int main( void ) {
 my_printf( "sdc", "This is a string", 29, 'X' );
  return( 0 );
}
```

This code displays the following output when run:

```
Printing a string: This is a string
Printing an integer: 29
Printing a character: X
```

#### cppreference.com > Standard C Math

## abs

Syntax:

```
#include <cstdlib>
int abs( int num );
```

The abs() function returns the absolute value of *num*. For example:

```
int magic_number = 10;
cout << "Enter a guess: ";
cin >> x;
cout << "Your guess was " << abs( magic_number - x ) << " away fi</pre>
```

Related topics:

fabs labs

#### acos

Syntax:

```
#include <cmath>
double acos( double arg );
```

The acos() function returns the arc cosine of *arg*, which will be in the range [0, pi]. *arg* should be between -1 and 1. If *arg* is outside this range, acos() returns NAN and raises a floating-point exception.

Related topics:

<u>asin</u>

<u>atan</u>

atan2

<u>**COS**</u>

<u>cosh</u>

<u>sin</u>

sinh tan tanh

## asin

Syntax:

```
#include <cmath>
double asin( double arg );
```

The asin() function returns the arc sine of *arg*, which will be in the range [-pi/2, +pi/2]. *arg* should be between -1 and 1. If *arg* is outside this range, asin() returns NAN and raises a floating-point exception.

Related topics:

acos

<u>atan</u>

atan2

COS

cosh

sin

sinh

<u>tan</u>

tanh

#### atan

Syntax:

```
#include <cmath>
double atan( double arg );
```

The function atan() returns the arc tangent of *arg*, which will be in the range [-pi/2, +pi/2].

```
Related topics:
acos
asin
atan2
cos
cosh
sin
sinh
tan
tanh
```

## atan2

Syntax:

```
#include <cmath>
double atan2( double y, double x );
```

The atan2() function computes the arc tangent of y/x, using the signs of the arguments to compute the quadrant of the return value.

Note the order of the arguments passed to this function.

Related topics: <a href="https://acos">acos</a>

<u>asin</u>

<u>atan</u>

<u>COS</u>

cosh

sin sinh

<u>tan</u>

<u>tanh</u>

## ceil

#### Syntax:

```
#include <cmath>
double ceil( double num );
```

The ceil() function returns the smallest integer no less than *num*. For example,

```
y = 6.04;
x = ceil( y );
```

would set x to 7.0.

Related topics:

floor fmod

#### COS

Syntax:

```
#include <cmath>
double cos( double arg );
```

The cos() function returns the cosine of *arg*, where *arg* is expressed in radians. The return value of cos() is in the range [-1,1]. If *arg* is infinite, cos() will return NAN and raise a floating-point exception.

Related topics:

acos

asin

atan

atan2

cosh

sin

sinh

tan

tanh

## cosh

Syntax:

```
#include <cmath>
double cosh( double arg );
```

The function cosh() returns the hyperbolic cosine of *arg*.

Related topics:

acos

asin

atan

atan2

**COS** 

sin

sinh

<u>tan</u>

tanh

## div

Syntax:

```
#include <cstdlib>
div_t div( int numerator, int denominator );
```

The function div() returns the quotient and remainder of the operation *numerator* / *denominator*. The **div\_t** structure is defined in cstdlib, and has at least:

```
int quot; // The quotient
int rem; // The remainder
```

For example, the following code displays the quotient and remainder of x/y:

```
div_t temp;
temp = div( x, y );
printf( "%d divided by %d yields %d with a remainder of %d\n",
```

```
x, y, temp.quot, temp.rem );
```

### Related topics:

<u>ldiv</u>

### exp

Syntax:

```
#include <cmath>
double exp( double arg );
```

The exp() function returns e (2.7182818) raised to the *arg*th power.

Related topics:

log

**pow** 

sqrt

## fabs

Syntax:

```
#include <cmath>
double fabs( double arg );
```

The function fabs() returns the absolute value of *arg*.

Related topics:

<u>abs</u>

**fmod** 

labs

## floor

Syntax:

```
#include <cmath>
double floor( double arg );
```

The function floor() returns the largest integer not greater than *arg*. For example,

```
y = 6.04;
x = floor( y );
```

would result in x being set to 6.0.

Related topics:

ceil

fmod

## fmod

Syntax:

```
#include <cmath>
double fmod( double x, double y );
```

The fmod() function returns the remainder of x/y.

Related topics:

ceil

fabs

floor

# frexp

Syntax:

```
#include <cmath>
double frexp( double num, int* exp );
```

The function frexp() is used to decompose *num* into two parts: a mantissa between 0.5 and 1 (returned by the function) and an exponent returned as *exp*. Scientific notation works like this:

```
num = mantissa * (2 ^ exp)
```

Related topics:

ldexp modf

## labs

Syntax:

```
#include <cstdlib>
long labs( long num );
```

The function labs() returns the absolute value of *num*.

Related topics:

<u>abs</u> fabs

# ldexp

Syntax:

```
#include <cmath>
double ldexp( double num, int exp );
```

The ldexp() function returns  $num * (2 \land exp)$ . And get this: if an overflow occurs, **HUGE\_VAL** is returned.

Related topics:

frexp modf

### ldiv

Syntax:

```
#include <cstdlib>
Idiv_t ldiv( long numerator, long denominator );
```

Testing: adiv\_t, div\_t, ldiv\_t.

The ldiv() function returns the quotient and remainder of the operation *numerator* / *denominator*. The ldiv\_t structure is defined in cstdlib and has at least:

```
long quot; // the quotient long rem; // the remainder
```

Related topics:

<u>div</u>

# log

Syntax:

```
#include <cmath>
double log( double num );
```

The function log() returns the natural (base e) logarithm of *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

In order to calculate the logarithm of *x* to an arbitrary base *b*, you can use:

```
double answer = log(x) / log(b);
```

Related topics:

```
exp
log10
pow
```

# log10

Syntax:

```
#include <cmath>
double log10( double num );
```

The log10() function returns the base 10 (or common) logarithm for *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

Related topics:

log

## modf

Syntax:

```
#include <cmath>
double modf( double num, double *i );
```

The function modf() splits *num* into its integer and fraction parts. It returns the fractional part and loads the integer part into *i*.

Related topics:

frexp ldexp

### pow

Syntax:

```
#include <cmath>
```

#### double pow( double base, double exp );

The pow() function returns *base* raised to the *exp*th power. There's a domain error if *base* is zero and *exp* is less than or equal to zero. There's also a domain error if *base* is negative and *exp* is not an integer. There's a range error if an overflow occurs.

#### Related topics:

<u>exp</u>

log

sart

### sin

#### Syntax:

```
#include <cmath>
double sin( double arg );
```

The function sin() returns the sine of *arg*, where *arg* is given in radians. The return value of sin() will be in the range [-1,1]. If *arg* is infinite, sin() will return NAN and raise a floating-point exception.

#### Related topics:

acos

asin

atan

atan2

**COS** 

cosh

sinh

tan

tanh

### sinh

### Syntax:

```
#include <cmath>
double sinh( double arg );
```

The function sinh() returns the hyperbolic sine of *arg*.

Related topics:

<u>acos</u>

asin

atan

atan2

**COS** 

cosh

sin

<u>tan</u>

<u>tanh</u>

## sqrt

#### Syntax:

```
#include <cmath>
double sqrt( double num );
```

The sqrt() function returns the square root of *num*. If *num* is negative, a domain error occurs.

Related topics:

<u>exp</u>

log

**pow** 

### tan

Syntax:

```
#include <cmath>
double tan( double arg );
```

The tan() function returns the tangent of *arg*, where *arg* is given in radians. If *arg* is infinite, tan() will return NAN and raise a floating-point exception.

#### Related topics:

acos

asin

<u>atan</u>

atan2

**COS** 

cosh

<u>sin</u>

<u>sinh</u>

tanh

## tanh

Syntax:

```
#include <cmath>
double tanh( double arg );
```

The function tanh() returns the hyperbolic tangent of *arg*.

### Related topics:

acos

asin

atan

atan2

COS

cosh

sin

<u>sinh</u>

tan

#### <u>cppreference.com</u> > <u>Standard C Date & Time</u>

### asctime

Syntax:

```
#include <ctime>
char *asctime( const struct tm *ptr );
```

The function asctime() converts the time in the struct 'ptr' to a character string of the following format:

```
day month date hours:minutes:seconds year
```

An example:

```
Mon Jun 26 12:03:53 2000
```

Related topics:

clock ctime

<u>difftime</u>

gmtime

**localtime** 

mktime

**time** 

## clock

Syntax:

```
#include <ctime>
clock_t clock( void );
```

The clock() function returns the processor time since the program started, or -1 if that information is unavailable. To convert the return value to seconds, divide it by CLOCKS\_PER\_SEC. (Note: if your compiler is POSIX compliant, then CLOCKS\_PER\_SEC is always defined as 1000000.)

Related topics: asctime

ctime time

## ctime

Syntax:

```
#include <ctime>
  char *ctime( const time_t *time );
```

The ctime() function converts the calendar time to local time of the format:

```
day month date hours:minutes:seconds year
```

using ctime() is equivalent to

```
asctime( localtime( tp ) );
```

Related topics:

asctime

clock

<u>gmtime</u>

**localtime** 

**mktime** 

time

## difftime

Syntax:

```
#include <ctime>
double difftime( time_t time2, time_t time1 );
```

The function difftime() returns *time2 - time1*, in seconds.

Related topics:

asctime gmtime localtime time

# gmtime

Syntax:

```
#include <ctime>
struct tm *gmtime( const time_t *time );
```

The gmtime() function returns the given *time* in Coordinated Universal Time (usually Greenwich mean time), unless it's not supported by the system, in which case **NULL** is returned. Watch out for static return.

Related topics:

**asctime** 

ctime

difftime

localtime

mktime

**strftime** 

time

## localtime

Syntax:

```
#include <ctime>
struct tm *localtime( const time_t *time );
```

The function localtime() converts calendar time into local time. Watch out for the <u>static return</u>.

# Related topics: asctime

<u>ctime</u>

<u>difftime</u>

gmtime strftime

time

## mktime

Syntax:

```
#include <ctime>
  time_t mktime( struct tm *time );
```

The mktime() function converts the local time in *time* to calendar time, and returns it. If there is an error, -1 is returned.

Related topics:

<u>asctime</u>

ctime

**gmtime** 

time

## setlocale

*Syntax:* 

```
#include <clocale>
char *setlocale( int category, const char * locale );
```

The setlocale() function is used to set and retrieve the current locale. If *locale* is **NULL**, the current locale is returned. Otherwise, *locale* is used to set the locale for the given *category*.

category can have the following values:

Value	Description	
LC_ALL	All of the locale	
LC_TIME	Date and time formatting	
LC_NUMERIC	Number formatting	
LC_COLLATE	String collation and regular expression matching	
LC_CTYPE	Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.	
LC_MONETARY	For monetary formatting	
LC_MESSAGES	For natural language messages	

Related topics:

(Standard C String and Character) strcoll

## strftime

### Syntax:

```
#include <ctime>
size_t strftime( char *str, size_t maxsize, const char *fmt, stru
```

The function strftime() formats date and time information from *time* to a format specified by *fmt*, then stores the result in *str* (up to *maxsize* characters). Certain codes may be used in *fmt* to specify different types of time:

Code	Meaning
%a	abbreviated weekday name (e.g. Fri)
%A	full weekday name (e.g. Friday)
%b	abbreviated month name (e.g. Oct)
%B	full month name (e.g. October)
%с	the standard date and time string

%d	day of the month, as a number (1-31)
%Н	hour, 24 hour format (0-23)
%I	hour, 12 hour format (1-12)
%j	day of the year, as a number (1-366)
%m	month as a number (1-12). Note: some versions of Microsoft Visual C++ may use values that range from 0-11.
%M	minute as a number (0-59)
%p	locale's equivalent of AM or PM
%S	second as a number (0-59)
%U	week of the year, (0-53), where week 1 has the first Sunday
%w	weekday as a decimal (0-6), where Sunday is 0
%W	week of the year, (0-53), where week 1 has the first Monday
%x	standard date string
%X	standard time string
%y	year in decimal, without the century (0-99)
%Y	year in decimal, with the century
%Z	time zone name
%%	a percent sign

The strftime() function returns the number of characters put into *str*, or zero if an error occurs.

### Related topics:

gmtime localtime time

## time

### Syntax:

```
#include <ctime>
time_t time( time_t *time );
```

The function time() returns the current time, or -1 if there is an error. If the argument 'time' is given, then the current time is stored in 'time'.

### Related topics:

**asctime** 

clock

**ctime** 

**difftime** 

**gmtime** 

**localtime** 

mktime

(Other Standard C Functions) srand

**strftime** 

#### <u>cppreference.com</u> > <u>Standard C String and Character</u>

### atof

Syntax:

```
#include <cstdlib>
double atof( const char *str );
```

The function atof() converts *str* into a double, then returns that value. *str* must start with a valid number, but can be terminated with any non-numerical character, other than "E" or "e". For example,

```
x = atof( "42.0is_the_answer" );
```

results in x being set to 42.0.

Related topics:

atoi atol

(Standard C I/O) sprintf

**strtod** 

### atoi

Syntax:

```
#include <cstdlib>
int atoi( const char *str );
```

The atoi() function converts *str* into an integer, and returns that integer. *str* should start with whitespace or some sort of number, and atoi() will stop reading from *str* as soon as a non-numerical character has been read. For example:

```
int i;
i = atoi( "512" );
i = atoi( "512.035" );
```

```
i = atoi( " 512.035");
i = atoi( " 512+34");
i = atoi( " 512 bottles of beer on the wall");
```

All five of the above assignments to the variable *i* would result in it being set to 512.

If the conversion cannot be performed, then atoi() will return zero:

```
int i = atoi( " does not work: 512" ); // results in i == 0
```

You can use sprintf() to convert a number into a string.

Related topics:

atof atol

(Standard C I/O) sprintf

### atol

Syntax:

```
#include <cstdlib>
long atol( const char *str );
```

The function atol() converts *str* into a long, then returns that value. atol() will read from *str* until it finds any character that should not be in a long. The resulting truncated value is then converted and returned. For example,

```
x = atol( "1024.0001" );
```

results in x being set to 1024L.

Related topics:

<u>atof</u> atoi

(Standard C I/O) sprintf

strtol

## isalnum

Syntax:

```
#include <cctype>
int isalnum( int ch );
```

The function isalnum() returns non-zero if its argument is a numeric digit or a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalnum(c) )
  printf( "You entered the alphanumeric character %c\n", c );
```

Related topics:

<u>isalpha</u>

**iscntrl** 

**isdigit** 

<u>isgraph</u>

**isprint** 

**ispunct** 

**isspace** 

isxdigit

# isalpha

Syntax:

```
#include <cctype>
int isalpha( int ch );
```

The function isalpha() returns non-zero if its argument is a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
```

```
if( isalpha(c) )
  printf( "You entered a letter of the alphabet\n" );
```

#### Related topics:

<u>isalnum</u>

iscntrl

**isdigit** 

**isgraph** 

**isprint** 

**ispunct** 

**isspace** 

<u>isxdigit</u>

## iscntrl

Syntax:

```
#include <cctype>
int iscntrl( int ch );
```

The iscntrl() function returns non-zero if its argument is a control character (between 0 and 0x1F or equal to 0x7F). Otherwise, zero is returned.

#### Related topics:

<u>isalnum</u>

<u>isalpha</u>

**isdigit** 

**isgraph** 

**isprint** 

**ispunct** 

**isspace** 

**isxdigit** 

# isdigit

*Syntax:* 

```
#include <cctype>
int isdigit( int ch );
```

The function isdigit() returns non-zero if its argument is a digit between 0 and 9. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isdigit(c) )
  printf( "You entered the digit %c\n", c );
```

#### Related topics:

**isalnum** 

<u>isalpha</u>

iscntrl

**isgraph** 

**isprint** 

**ispunct** 

**isspace** 

**isxdigit** 

# isgraph

Syntax:

```
#include <cctype>
int isgraph( int ch );
```

The function isgraph() returns non-zero if its argument is any printable character other than a space (if you can see the character, then isgraph() will return a non-zero value). Otherwise, zero is returned.

### Related topics:

<u>isalnum</u>

<u>isalpha</u>

iscntrl

**isdigit** 

**isprint** 

```
ispunct
isspace
isxdigit
```

## islower

Syntax:

```
#include <cctype>
int islower( int ch );
```

The islower() function returns non-zero if its argument is a lowercase letter. Otherwise, zero is returned.

Related topics:

<u>isupper</u>

# isprint

Syntax:

```
#include <cctype>
int isprint( int ch );
```

The function isprint() returns non-zero if its argument is a printable character (including a space). Otherwise, zero is returned.

Related topics:

isalnum

<u>isalpha</u>

**iscntrl** 

<u>isdigit</u>

**isgraph** 

**ispunct** 

**isspace** 

# ispunct

Syntax:

```
#include <cctype>
int ispunct( int ch );
```

The ispunct() function returns non-zero if its argument is a printing character but neither alphanumeric nor a space. Otherwise, zero is returned.

#### Related topics:

isalnum

<u>isalpha</u>

**iscntrl** 

**isdigit** 

<u>isgraph</u>

**isprint** 

**isspace** 

**isxdigit** 

# isspace

Syntax:

```
#include <cctype>
int isspace( int ch );
```

The isspace() function returns non-zero if its argument is some sort of space (i.e. single space, tab, vertical tab, form feed, carriage return, or newline). Otherwise, zero is returned.

### Related topics:

**isalnum** 

<u>isalpha</u>

iscntrl

**isdigit** 

isgraph isprint ispunct isxdigit

# isupper

Syntax:

```
#include <cctype>
int isupper( int ch );
```

The isupper() function returns non-zero if its argument is an uppercase letter. Otherwise, zero is returned.

Related topics:

**islower** 

**tolower** 

# isxdigit

Syntax:

```
#include <cctype>
int isxdigit( int ch );
```

The function isxdigit() returns non-zero if its argument is a hexidecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics:

**isalnum** 

**isalpha** 

**iscntrl** 

**isdigit** 

**isgraph** 

**ispunct** 

### memchr

Syntax:

```
#include <cstring>
void *memchr( const void *buffer, int ch, size_t count );
```

The memchr() function looks for the first occurrence of *ch* within *count* characters in the array pointed to by *buffer*. The return value points to the location of the first occurrence of *ch*, or **NULL** if *ch* isn't found. For example:

```
char names[] = "Alan Bob Chris X Dave";
if( memchr(names,'X',strlen(names)) == NULL )
  printf( "Didn't find an X\n" );
else
  printf( "Found an X\n" );
```

Related topics:

memcmp memcpy

**strstr** 

### memcmp

Syntax:

```
#include <cstring>
int memcmp( const void *buffer1, const void *buffer2, size_t coun
```

The function memcmp() compares the first *count* characters of *buffer1* and *buffer2*. The return values are as follows:

Value	Explanation
less than 0	buffer1 is less than buffer2

equal to 0	buffer1 is equal to buffer2			
greater than 0	buffer1 is greater than buffer2			
Related topics:				
<u>memchr</u>				
<u>memcpy</u>				
<u>memset</u>				
strcmp				

## memcpy

#### Syntax:

```
#include <cstring>
void *memcpy( void *to, const void *from, size_t count );
```

The function memcpy() copies *count* characters from the array *from* to the array *to*. The return value of memcpy() is *to*. The behavior of memcpy() is undefined if *to* and *from* overlap.

Related topics:

memchr

**memcmp** 

memmove

memset

strcpy

**strlen** 

**strncpy** 

### memmove

#### Syntax:

```
#include <cstring>
void *memmove( void *to, const void *from, size_t count );
```

The memmove() function is identical to memcpy(), except that it works even if *to* and *from* overlap.

Related topics:

memcpy memset

### memset

Syntax:

```
#include <cstring>
void* memset( void* buffer, int ch, size_t count );
```

The function memset() copies *ch* into the first *count* characters of *buffer*, and returns *buffer*. memset() is useful for intializing a section of memory to some value. For example, this command:

```
const int ARRAY_LENGTH;
char the_array[ARRAY_LENGTH];
...
// zero out the contents of the_array
memset( the_array, '\0', ARRAY_LENGTH );
```

...is a very efficient way to set all values of the\_array to zero.

The table below compares two different methods for initializing an array of characters: a for-loop versus memset(). As the size of the data being initialized increases, memset() clearly gets the job done much more quickly:

Input size	e Initializ	ed with a for-loop Initialized with memset()
1000	0.016	0.017
10000	0.055	0.013
100000	0.443	0.029
1000000	4.337	0.291

**Related topics:** 

memcmp memcpy

### strcat

Syntax:

```
#include <cstring>
char *strcat( char *str1, const char *str2 );
```

The strcat() function concatenates str2 onto the end of str1, and returns str1. For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
title = strcat( name, " the Great" );
printf( "Hello, %s\n", title );
```

Note that strcat() does not perform bounds checking, and thus risks overrunning str1 or str2. For a similar (and safer) function that includes bounds checking, see strncat().

Related topics:

strchr strcmp strcpy strncat

Another set of related (but non-standard) functions are <u>strlcpy and strlcat</u>.

### strchr

Syntax:

```
#include <cstring>
char *strchr( const char *str, int ch );
```

The function strchr() returns a pointer to the first occurence of *ch* in *str*, or **NULL** if *ch* is not found.

### Related topics:

**strcat** 

**strcmp** 

**strcpy** 

strlen

**strncat** 

**strncmp** 

**strncpy** 

**strpbrk** 

**strspn** 

**strstr** 

**strtok** 

## strcmp

#### Syntax:

```
#include <cstring>
int strcmp( const char *str1, const char *str2 );
```

The function strcmp() compares *str1* and *str2*, then returns:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than "str2"

### For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 ) {
  printf( "Hello, Dr. Mary!\n" );
```

}

Note that if str1 or str2 are missing a null-termination character, then strcmp() may not produce valid results. For a similar (and safer) function that includes explicit bounds checking, see strncmp().

#### Related topics:

**memcmp** 

strcat

strchr

strcoll

strcpy

<u>strlen</u>

<u>strncmp</u>

**strxfrm** 

## strcoll

### Syntax:

```
#include <cstring>
int strcoll( const char *str1, const char *str2 );
```

The strcoll() function compares *str1* and *str2*, much like <u>strcmp</u>(). However, strcoll() performs the comparison using the locale specified by the (Standard C Date & Time) <u>setlocale()</u> function.

#### Related topics:

(Standard C Date & Time) setlocale

strcmp

<u>strxfrm</u>

## strcpy

Syntax:

```
#include <cstring>
char *strcpy( char *to, const char *from );
```

The strcpy() function copies characters in the string *from* to the string *to*, including the null termination. The return value is *to*.

Note that strcpy() does not perform bounds checking, and thus risks overrunning *from* or *to*. For a similar (and safer) function that includes bounds checking, see <a href="strncpy">strncpy()</a>.

Related topics:

memcpy strcat strchr strcmp strncmp strncpy

Another set of related (but non-standard) functions are strlcpy and strlcat.

## strcspn

Syntax:

```
#include <cstring>
size_t strcspn( const char *str1, const char *str2 );
```

The function strcspn() returns the index of the first character in *str1* that matches any of the characters in *str2*.

Related topics:

strpbrk strrchr strstr strtok

### strerror

*Syntax:* 

```
#include <cstring>
char *strerror( int num );
```

The function strerror() returns an implementation defined string corresponding to *num*.

### strlen

Syntax:

```
#include <cstring>
size_t strlen( char *str );
```

The strlen() function returns the length of *str* (determined by the number of characters before null termination).

Related topics:

memcpy strchr strcmp strncmp

### strncat

Syntax:

```
#include <cstring>
char *strncat( char *str1, const char *str2, size_t count );
```

The function strncat() concatenates at most *count* characters of *str2* onto *str1*,

adding a null termination. The resulting string is returned.

Related topics:

strcat strchr

**strncmp** 

**strncpy** 

Another set of related (but non-standard) functions are strlcpy and strlcat.

## strncmp

#### Syntax:

```
#include <cstring>
int strncmp( const char *str1, const char *str2, size_t count );
```

The strncmp() function compares at most *count* characters of *str1* and *str2*. The return value is as follows:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than str2"

If there are less than *count* characters in either string, then the comparison will stop after the first null termination is encountered.

### Related topics:

**strchr** 

**strcmp** 

**strcpy** 

<u>strlen</u>

**strncat** 

**strncpy** 

## strncpy

Syntax:

```
#include <cstring>
char *strncpy( char *to, const char *from, size_t count );
```

The strncpy() function copies at most *count* characters of *from* to the string *to*. If from has less than *count* characters, the remainder is padded with '\0' characters. The return value is the resulting string.

#### Related topics:

memcpy strchr strcpy strncat

**strncmp** 

Another set of related (but non-standard) functions are strlcpy and strlcat.

# strpbrk

Syntax:

```
#include <cstring>
char* strpbrk( const char* str1, const char* str2 );
```

The function strpbrk() returns a pointer to the first ocurrence in *str1* of any character in *str2*, or **NULL** if no such characters are present.

#### Related topics:

```
(C++ Algorithms) find_first_of

strchr

strcspn

strrchr

strspn
```

## strrchr

Syntax:

```
#include <cstring>
char *strrchr( const char *str, int ch );
```

The function strrchr() returns a pointer to the last occurrence of *ch* in *str*, or **NULL** if no match is found.

Related topics:

strcspn

**strpbrk** 

strspn

strstr

strtok

# strspn

Syntax:

```
#include <cstring>
size_t strspn( const char *str1, const char *str2 );
```

The strspn() function returns the index of the first character in *str1* that doesn't match any character in *str2*.

Related topics:

**strchr** 

**strpbrk** 

strrchr

strstr

strtok

### strstr

Syntax:

```
#include <cstring>
char *strstr( const char *str1, const char *str2 );
```

The function strstr() returns a pointer to the first occurrence of *str2* in *str1*, or **NULL** if no match is found. If the length of *str2* is zero, then strstr() will simply return *str1*.

For example, the following code checks for the existence of one string within another string:

```
char* str1 = "this is a string of characters";
char* str2 = "a string";
char* result = strstr( str1, str2 );
if( result == NULL ) printf( "Could not find '%s' in '%s'\n", str2
else printf( "Found a substring: '%s'\n", result );
```

When run, the above code displays this output:

```
Found a substring: 'a string of characters'

Related topics:

memchr

strchr

strcspn

strpbrk
```

## strtod

Syntax:

strrchr strspn strtok

```
#include <cstdlib>
double strtod( const char *start, char **end );
```

The function strtod() returns whatever it encounters first in *start* as a double. *end* is set to point at whatever is left in *start* after that double. If overflow occurs, strtod() returns either **HUGE\_VAL** or **-HUGE\_VAL**.

Related topics:

<u>atof</u>

### strtok

Syntax:

```
#include <cstring>
char *strtok( char *str1, const char *str2 );
```

The strtok() function returns a pointer to the next "token" in str1, where str2 contains the delimiters that determine the token. strtok() returns **NULL** if no token is found. In order to convert a string to tokens, the first call to strtok() should have str1 point to the string to be tokenized. All calls after this should have str1 be **NULL**.

For example:

```
char str[] = "now # is the time for all # good men to come to the
char delims[] = "#";
char *result = NULL;
result = strtok( str, delims );
while( result != NULL ) {
    printf( "result is \"%s\"\n", result );
    result = strtok( NULL, delims );
}
```

The above code will display the following output:

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

#### Related topics:

**strchr** 

**strcspn** 

<u>strpbrk</u>

<u>strrchr</u>

**strspn** 

**strstr** 

### strtol

Syntax:

```
#include <cstdlib>
long strtol( const char *start, char **end, int base );
```

The strtol() function returns whatever it encounters first in *start* as a long, doing the conversion to *base* if necessary. *end* is set to point to whatever is left in *start* after the long. If the result can not be represented by a long, then strtol() returns either **LONG\_MAX** or **LONG\_MIN**. Zero is returned upon error.

Related topics:

<u>atol</u>

strtoul

### strtoul

Syntax:

```
#include <cstdlib>
unsigned long strtoul( const char *start, char **end, int base );
```

The function strtoul() behaves exactly like <u>strtol</u>(), except that it returns an unsigned long rather than a mere long.

Related topics:

**strtol** 

### strxfrm

Syntax:

```
#include <cstring>
size_t strxfrm( char *str1, const char *str2, size_t num );
```

The strxfrm() function manipulates the first *num* characters of str2 and stores them in str1. The result is such that if a strcoll() is performed on str1 and the old str2, you will get the same result as with a strcmp().

Related topics:

strcmp strcoll

### tolower

Syntax:

```
#include <cctype>
int tolower( int ch );
```

The function tolower() returns the lowercase version of the character *ch*.

Related topics:

<u>isupper</u> <u>toupper</u>

## toupper

Syntax:

```
#include <cctype>
int toupper( int ch );
```

The toupper(	) function returns	the uppercase	version of	the character <i>ch</i> .
--------------	--------------------	---------------	------------	---------------------------

Related topics: tolower

<u>cppreference.com</u> > <u>Standard C Memory</u>

### calloc

Syntax:

```
#include <cstdlib>
void* calloc( size_t num, size_t size );
```

The calloc() function returns a pointer to space for an array of *num* objects, each of size *size*. The newly allocated memory is initialized to zero.

calloc() returns NULL if there is an error.

Related topics:

<u>free malloc</u> realloc

### free

*Syntax:* 

```
#include <cstdlib>
void free( void* ptr );
```

The free() function deallocates the space pointed to by *ptr*, freeing it up for future use. *ptr* must have been used in a previous call to <u>malloc()</u>, <u>calloc()</u>, or <u>realloc()</u>. An example:

```
typedef struct data_type {
   int age;
   char name[20];
} data;

data *willy;
willy = (data*) malloc( sizeof(*willy) );
...
free( willy );
```

```
Related topics:

calloc
(C/C++ Keywords) delete
malloc
(C/C++ Keywords) new
realloc
```

## malloc

Syntax:

```
#include <cstdlib>
void *malloc( size_t size );
```

The function malloc() returns a pointer to a chunk of memory of size *size*, or **NULL** if there is an error. The memory pointed to will be on the heap, not the stack, so make sure to free it when you are done with it. An example:

```
typedef struct data_type {
   int age;
   char name[20];
} data;

data *bob;
bob = (data*) malloc( sizeof(data) );
if( bob != NULL ) {
   bob->age = 22;
   strcpy( bob->name, "Robert" );
   printf( "%s is %d years old\n", bob->name, bob->age );
}
free( bob );
```

```
Related topics:
<a href="mailto:calloc">calloc</a>
(C/C++ Keywords) delete
free
```

(C/C++ Keywords) new

<u>realloc</u>

## realloc

Syntax:

```
#include <cstdlib>
void *realloc( void *ptr, size_t size );
```

The realloc() function changes the size of the object pointed to by ptr to the given size. size can be any size, larger or smaller than the original. The return value is a pointer to the new space, or **NULL** if there is an error.

Related topics:

**calloc** 

**free** 

**malloc** 

<u>cppreference.com</u> > <u>Standard C I/O</u>

#### clearerr

Syntax:

```
#include <cstdio>
void clearerr( FILE *stream );
```

The clearerr function resets the error flags and **EOF** indicator for the given *stream*. When an error occurs, you can use <u>perror</u>() to figure out which error actually occurred.

Related topics:

feof ferror perror

### fclose

*Syntax:* 

```
#include <cstdio>
int fclose( FILE *stream );
```

The function fclose() closes the given file stream, deallocating any buffers associated with that stream. fclose() returns 0 upon success, and **EOF** otherwise.

Related topics:

fflush fopen freopen setbuf

### feof

#### Syntax:

```
#include <cstdio>
int feof( FILE *stream );
```

The function feof() returns a nonzero value if the end of the given file *stream* has been reached.

Related topics:

<u>clearerr</u>

<u>ferror</u>

getc

<u>perror</u>

**putc** 

### ferror

#### Syntax:

```
#include <cstdio>
int ferror( FILE *stream );
```

The ferror() function looks for errors with *stream*, returning zero if no errors have occured, and non-zero if there is an error. In case of an error, use <u>perror()</u> to determine which error has occured.

Related topics:

<u>clearerr</u>

feof

<u>perror</u>

## fflush

#### Syntax:

```
#include <cstdio>
```

```
int fflush( FILE *stream );
```

If the given file *stream* is an output stream, then fflush() causes the output buffer to be written to the file. If the given *stream* is of the input type, then fflush() causes the input buffer to be cleared. fflush() is useful when debugging, if a program segfaults before it has a chance to write output to the screen. Calling fflush( **stdout** ) directly after debugging output will ensure that your output is displayed at the correct time.

```
printf( "Before first call\n" );
fflush( stdout );
shady_function();
printf( "Before second call\n" );
fflush( stdout );
dangerous_dereference();
```

#### Related topics:

fclose

**fopen** 

**fread** 

**fwrite** 

getc

**putc** 

# fgetc

Syntax:

```
#include <cstdio>
int fgetc( FILE *stream );
```

The fgetc() function returns the next character from *stream*, or **EOF** if the end of file is reached or if there is an error.

Related topics:

**fopen** 

**fputc** 

fread

**fwrite** 

```
getc
getchar
gets
putc
```

# fgetpos

Syntax:

```
#include <cstdio>
int fgetpos( FILE *stream, fpos_t *position );
```

The fgetpos() function stores the file position indicator of the given file *stream* in the given *position* variable. The position variable is of type fpos\_t (which is defined in cstdio) and is an object that can hold every possible position in a FILE. fgetpos() returns zero upon success, and a non-zero value upon failure.

Related topics:

<u>fseek</u> <u>fsetpos</u> ftell

# fgets

Syntax:

```
#include <cstdio>
char *fgets( char *str, int num, FILE *stream );
```

The function fgets() reads up to *num* - 1 characters from the given file *stream* and dumps them into *str*. The string that fgets() produces is always **NULL**-terminated. fgets() will stop when it reaches the end of a line, in which case *str* will contain that newline character. Otherwise, fgets() will stop when it reaches *num* - 1 characters or encounters the **EOF** character. fgets() returns *str* on success, and **NULL** on an error.

Related topics:

fputs fscanf gets scanf

# fopen

Syntax:

```
#include <cstdio>
FILE *fopen( const char *fname, const char *mode );
```

The fopen() function opens a file indicated by *fname* and returns a stream associated with that file. If there is an error, fopen() returns **NULL**. *mode* is used to determine how the file will be treated (i.e. for input, output, etc)

Mode	Meaning
"r"	Open a text file for reading
"w"	Create a text file for writing
"a"	Append to a text file
"rb"	Open a binary file for reading
"wb"	Create a binary file for writing
"ab"	Append to a binary file
"r+"	Open a text file for read/write
"w+"	Create a text file for read/write
"a+"	Open a text file for read/write
"rb+"	Open a binary file for read/write
"wb+"	Create a binary file for read/write
"ab+"	Open a binary file for read/write

#### An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

#### *Related topics:*

**fclose** 

fflush

**fgetc** 

**fputc** 

fread

freopen

fseek

**fwrite** 

getc

getchar

<u>setbuf</u>

# fprintf

#### Syntax:

```
#include <cstdio>
int fprintf( FILE *stream, const char *format, ... );
```

The fprintf() function sends information (the arguments) according to the specified *format* to the file indicated by *stream*. fprintf() works just like <u>printf()</u> as far as the format goes. The return value of fprintf() is the number of characters outputted, or a negative number if an error occurs. An example:

```
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
  fprintf( out, "Hello %s\n", name );
```

#### Related topics:

**fputc** 

fputs fscanf printf sprintf

## fputc

Syntax:

```
#include <cstdio>
int fputc( int ch, FILE *stream );
```

The function fputc() writes the given character *ch* to the given output *stream*. The return value is the character, unless there is an error, in which case the return value is **EOF**.

Related topics:

**fgetc** 

**fopen** 

**fprintf** 

fread

**fwrite** 

getc

getchar

<u>putc</u>

# **fputs**

Syntax:

```
#include <cstdio>
int fputs( const char *str, FILE *stream );
```

The fputs() function writes an array of characters pointed to by *str* to the given output *stream*. The return value is non-negative on success, and **EOF** on failure.

Related topics:

**fgets** 

**fprintf** 

<u>fscanf</u>

gets

**puts** 

### fread

#### Syntax:

```
#include <cstdio>
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

The function fread() reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by buffer. The data comes from the given input *stream*. The return value of the function is the number of things read. You can use <u>feof()</u> or <u>ferror()</u> to figure out if an error occurs.

Related topics:

**fflush** 

fgetc

<u>fopen</u>

**fputc** 

**fscanf** 

**fwrite** 

getc

# freopen

#### Syntax:

```
#include <cstdio>
FILE *freopen( const char *fname, const char *mode, FILE *stream
```

The freopen() function is used to reassign an existing *stream* to a different file

and mode. After a call to this function, the given file *stream* will refer to *fname* with access given by *mode*. The return value of freopen() is the new stream, or **NULL** if there is an error.

Related topics:

fclose fopen

### fscanf

Syntax:

```
#include <cstdio>
int fscanf( FILE *stream, const char *format, ... );
```

The function fscanf() reads data from the given file *stream* in a manner exactly like scanf(). The return value of fscanf() is the number of variables that are actually assigned values, or **EOF** if no assignments could be made.

Related topics:

**fgets** 

**fprintf** 

**fputs** 

fread

**fwrite** 

<u>scanf</u>

**sscanf** 

## fseek

Syntax:

```
#include <cstdio>
int fseek( FILE *stream, long offset, int origin );
```

The function fseek() sets the file position data for the given *stream*. The origin

value should have one of the following values (defined in cstdio):

Name	Explanation
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

fseek() returns zero upon success, non-zero on failure. You can use fseek() to move beyond a file, but not before the beginning. Using fseek() clears the **EOF** flag associated with that stream.

Related topics:

**fgetpos** 

**fopen** 

**fsetpos** 

ftell

rewind

# **fsetpos**

Syntax:

```
#include <cstdio>
int fsetpos( FILE *stream, const fpos_t *position );
```

The fsetpos() function moves the file position indicator for the given *stream* to a location specified by the *position* object. fpos\_t is defined in cstdio. The return value for fsetpos() is zero upon success, non-zero on failure.

Related topics:

**fgetpos** 

fseek

**ftell** 

## ftell

#### Syntax:

```
#include <cstdio>
long ftell( FILE *stream );
```

The ftell() function returns the current file position for *stream*, or -1 if an error occurs.

Related topics:

**fgetpos** 

fseek

**fsetpos** 

## **fwrite**

#### Syntax:

```
#include <cstdio>
int fwrite( const void *buffer, size_t size, size_t count, FILE *
```

The fwrite() function writes, from the array *buffer*, *count* objects of size *size* to *stream*. The return value is the number of objects written.

Related topics:

fflush

**fgetc** 

**fopen** 

**fputc** 

**fread** 

**fscanf** 

getc

## getc

Syntax:

```
#include <cstdio>
int getc( FILE *stream );
```

The getc() function returns the next character from *stream*, or **EOF** if the end of file is reached. getc() is identical to <u>fgetc()</u>. For example:

```
int ch;
FILE *input = fopen( "stuff", "r" );

ch = getc( input );
while( ch != EOF ) {
   printf( "%c", ch );
   ch = getc( input );
}
```

Related topics:

**feof** 

**fflush** 

**fgetc** 

<u>fopen</u>

**fputc** 

**fread** 

**fwrite** 

<u>putc</u>

<u>ungetc</u>

# getchar

Syntax:

```
#include <cstdio>
int getchar( void );
```

The getchar() function returns the next character from **stdin**, or **EOF** if the end

of file is reached.

Related topics:

**fgetc** 

**fopen** 

**fputc** 

**putc** 

## gets

Syntax:

```
#include <cstdio>
char *gets( char *str );
```

The gets() function reads characters from **stdin** and loads them into *str*, until a newline or **EOF** is reached. The newline character is translated into a null termination. The return value of gets() is the read-in string, or **NULL** if there is an error.

Note that gets() does not perform bounds checking, and thus risks overrunning *str*. For a similar (and safer) function that includes bounds checking, see <a href="fgets()">fgets()</a>.

Related topics:

**fgetc** 

**fgets** 

**fputs** 

<u>puts</u>

### perror

Syntax:

```
#include <cstdio>
void perror( const char *str );
```

The perror() function prints *str* and an implementation-defined error message corresponding to the global variable *errno*. For example:

```
char* input_filename = "not_found.txt";
FILE* input = fopen( input_filename, "r" );
if( input == NULL ) {
   char error_msg[255];
   sprintf( error_msg, "Error opening file '%s'", input_filename );
   perror( error_msg );
   exit( -1 );
}
```

The the file called *not\_found.txt* is not found, this code will produce the following output:

```
Error opening file 'not_found.txt': No such file or directory

Related topics:

clearer

feof
ferror
```

## printf

Syntax:

```
#include <cstdio>
int printf( const char *format, ... );
```

The printf() function prints output to **stdout**, according to *format* and other arguments passed to printf(). The string *format* consists of two types of items characters that will be printed to the screen, and format commands that define how the other arguments to printf() are displayed. Basically, you specify a format string that has text in it, as well as "special" characters that map to the other arguments of printf(). For example, this code

```
char name[20] = "Bob";
int age = 21;
printf( "Hello %s, you are %d years old\n", name, age );
```

displays the following output:

```
Hello Bob, you are 21 years old
```

The %s means, "insert the first argument, a string, right here." The %d indicates that the second argument (an integer) should be placed there. There are different %-codes for different variable types, as well as options to limit the length of the variables and whatnot.

Code	Format	
%с	character	
%d	signed integers	
%i	signed integers	
%e	scientific notation, with a lowercase "e"	
%E	scientific notation, with a uppercase "E"	
%f	floating point	
%g	use %e or %f, whichever is shorter	
%G	use %E or %f, whichever is shorter	
%o	octal	
%s	a string of characters	
%u	unsigned integer	
%x	unsigned hexadecimal, with lowercase letters	
%X	unsigned hexadecimal, with uppercase letters	
%p	a pointer	
%n	the argument shall be a pointer to an integer into which is placed the number of characters written so far	
%%	a '%' sign	

An integer placed between a % sign and the format command acts as a minimum field width specifier, and pads the output with spaces or zeros to make it long enough. If you want to pad with zeros, place a zero before the minimum field

width specifier:

%012d

You can also include a precision modifier, in the form of a .N where N is some number, before the format command:

%012.4d

The precision modifier has different meanings depending on the format command being used:

- With %e, %E, and %f, the precision modifier lets you specify the number of decimal places desired. For example, %12.6f will display a floating number at least 12 digits wide, with six decimal places.
- With %g and %G, the precision modifier determines the maximum number of significant digits displayed.
- With %s, the precision modifer simply acts as a maximumfield length, to complement the minimum field length that precedes the period.

All of printf()'s output is right-justified, unless you place a minus sign right after the % sign. For example,

%-12.4f

will display a floating point number with a minimum of 12 characters, 4 decimal places, and left justified. You may modify the %d, %i, %o, %u, and %x type specifiers with the letter l and the letter h to specify long and short data types (e.g. %hd means a short integer). The %e, %f, and %g type specifiers can have the letter l before them to indicate that a double follows. The %g, %f, and %e type specifiers can be preceded with the character '#' to ensure that the decimal point will be present, even if there are no decimal digits. The use of the '#' character with the %x type specifier indicates that the hexidecimal number should be printed with the '0x' prefix. The use of the '#' character with the %o type specifier indicates that the octal value should be displayed with a 0 prefix.

Inserting a plus sign '+' into the type specifier will force positive values to be preceded by a '+' sign. Putting a space character ' ' there will force positive values to be preceded by a single space character.

You can also include constant escape sequences in the output string.

The return value of printf() is the number of characters printed, or a negative number if an error occurred.

```
Related topics:
```

fprintf puts scanf sprintf

### putc

Syntax:

```
#include <cstdio>
int putc( int ch, FILE *stream );
```

The putc() function writes the character *ch* to *stream*. The return value is the character written, or **EOF** if there is an error. For example:

```
int ch;
FILE *input, *output;
input = fopen( "tmp.c", "r" );
output = fopen( "tmpCopy.c", "w" );
ch = getc( input );
while( ch != EOF ) {
   putc( ch, output );
   ch = getc( input );
}
fclose( input );
fclose( output );
```

generates a copy of the file tmp.c called tmpCopy.c.

Related topics:

feof fflush fgetc fputc

```
getc
getchar
putchar
puts
```

## putchar

Syntax:

```
#include <cstdio>
int putchar( int ch );
```

The putchar() function writes *ch* to **stdout**. The code

```
putchar( ch );
```

is the same as

```
putc( ch, stdout );
```

The return value of putchar() is the written character, or **EOF** if there is an error.

Related topics:

<u>putc</u>

## puts

Syntax:

```
#include <cstdio>
int puts( char *str );
```

The function puts() writes *str* to **stdout**. puts() returns non-negative on success, or **EOF** on failure.

Related topics:

fputs
gets
printf
putc

#### remove

Syntax:

```
#include <cstdio>
int remove( const char *fname );
```

The remove() function erases the file specified by *fname*. The return value of remove() is zero upon success, and non-zero if there is an error.

Related topics:

**rename** 

#### rename

Syntax:

```
#include <cstdio>
int rename( const char *oldfname, const char *newfname );
```

The function rename() changes the name of the file *oldfname* to *newfname*. The return value of rename() is zero upon success, non-zero on error.

Related topics:

remove

### rewind

Syntax:

```
#include <cstdio>
void rewind( FILE *stream );
```

The function rewind() moves the file position indicator to the beginning of the specified *stream*, also clearing the error and **EOF** flags associated with that stream.

Related topics:

fseek

### scanf

Syntax:

```
#include <cstdio>
int scanf( const char *format, ... );
```

The scanf() function reads input from **stdin**, according to the given *format*, and stores the data in the other arguments. It works a lot like <u>printf()</u>. The *format* string consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are as follows:

Control Character	Explanation
%с	a single character
%d	a decimal integer
%i	an integer
%e, %f, %g	a floating-point number
%lf	a double
%0	an octal number
%s	a string
%x	a hexadecimal number
%p	a pointer

%n	an integer equal to the number of characters read so far
%u	an unsigned integer
%[]	a set of characters
%%	a percent sign

scanf() reads the input, matching the characters from format. When a control character is read, it puts the value in the next variable. Whitespace (tabs, spaces, etc) are skipped. Non-whitespace characters are matched to the input, then discarded. If a number comes between the % sign and the control character, then only that many characters will be converted into the variable. If scanf() encounters a set of characters, denoted by the %[] control character, then any characters found within the brackets are read into the variable. The return value of scanf() is the number of variables that were successfully assigned values, or **EOF** if there is an error.

#### Example code:

This code snippet uses scanf() to read an int, float, and a double from the user. Note that the variable arguments to scanf() are passed in by address, as denoted by the ampersand (&) preceding each variable:

```
int i;
float f;
double d;

printf( "Enter an integer: " );
scanf( "%d", &i );

printf( "Enter a float: " );
scanf( "%f", &f );

printf( "Enter a double: " );
scanf( "%lf", &d );

printf( "You entered %d, %f, and %f\n", i, f, d );
```

#### Related topics:

fgets fscanf printf sscanf

### setbuf

Syntax:

```
#include <cstdio>
void setbuf( FILE *stream, char *buffer );
```

The setbuf() function sets *stream* to use *buffer*, or, if *buffer* is null, turns off buffering. If a non-standard buffer size is used, it should be BUFSIZ characters long.

Related topics:

fclose fopen setvbuf

## setvbuf

Syntax:

```
#include <cstdio>
int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
```

The function setvbuf() sets the buffer for *stream* to be *buffer*, with a size of *size*. *mode* can be:

- \_IOFBF, which indicates full buffering
- \_IOLBF, which means line buffering
- \_IONBF, which means no buffering

Related topics:

<u>setbuf</u>

# sprintf

Syntax:

```
#include <cstdio>
int sprintf( char *buffer, const char *format, ... );
```

The sprintf() function is just like <u>printf(</u>), except that the output is sent to *buffer*. The return value is the number of characters written. For example:

```
char string[50];
int file_number = 0;
sprintf( string, "file.%d", file_number );
file_number++;
output_file = fopen( string, "w" );
```

Note that sprintf() does the opposite of a function like <u>atoi()</u> -- where <u>atoi()</u> converts a string into a number, sprintf() can be used to convert a number into a string.

For example, the following code uses sprintf() to convert an integer into a string of characters:

```
char result[100];
int num = 24;
sprintf( result, "%d", num );
```

This code is similar, except that it converts a floating-point number into an array of characters:

```
char result[100];
float fnum = 3.14159;
sprintf( result, "%f", fnum );
```

#### Related topics:

```
(Standard C String and Character) <u>atof</u> (Standard C String and Character) <u>atoi</u> (Standard C String and Character) <u>atol</u> <u>fprintf</u>
```

### sscanf

Syntax:

```
#include <cstdio>
int sscanf( const char *buffer, const char *format, ... );
```

The function sscanf() is just like <u>scanf()</u>, except that the input is read from *buffer*.

Related topics:

**fscanf** 

**scanf** 

# tmpfile

Syntax:

```
#include <cstdio>
FILE *tmpfile( void );
```

The function tmpfile() opens a temporary file with an unique filename and returns a pointer to that file. If there is an error, null is returned.

Related topics:

<u>tmpnam</u>

## tmpnam

Syntax:

```
#include <cstdio>
```

```
char *tmpnam( char *name );
```

The tmpnam() function creates an unique filename and stores it in *name*. tmpnam() can be called up to **TMP\_MAX** times.

Related topics:

**tmpfile** 

## ungetc

Syntax:

```
#include <cstdio>
int ungetc( int ch, FILE *stream );
```

The function ungetc() puts the character *ch* back in *stream*.

Related topics:

getc

(C++ I/O) putback

# vprintf, vfprintf, and vsprintf

Syntax:

```
#include <cstdarg>
#include <cstdio>
int vprintf( char *format, va_list arg_ptr );
int vfprintf( FILE *stream, const char *format, va_list arg_ptr )
int vsprintf( char *buffer, char *format, va_list arg_ptr );
```

These functions are very much like <u>printf()</u>, <u>fprintf()</u>, and <u>sprintf()</u>. The difference is that the argument list is a pointer to a list of arguments. **va\_list** is defined in cstdarg, and is also used by (Other Standard C Functions) <u>va\_arg()</u>. For example:

```
void error( char *fmt, ... ) {
```

```
va_list args;
va_start( args, fmt );
fprintf( stderr, "Error: " );
vfprintf( stderr, fmt, args );
fprintf( stderr, "\n" );
va_end( args );
exit( 1 );
}
```

```
<u>cppreference.com</u> > <u>C/C++ Keywords</u>
```

#### asm

Syntax:

```
asm( "instruction" );
```

The asm command allows you to insert assembly language commands directly into your code. Various different compilers allow differing forms for this command, such as

```
asm {
  instruction-sequence
}
```

or

```
asm( instruction );
```

#### auto

The keyword auto is used to declare local variables, and is purely optional.

Related topics:

<u>register</u>

## bool

The keyword bool is used to declare Boolean logic variables; that is, variables which can be either true or false.

For example, the following code declares a boolean variable called *done*, initializes it to false, and then loops until that variable is set to true.

```
bool done = false;
while( !done ) {
   ...
}
```

Also see the <u>data types</u> page.

Related topics:

char double

false

float

int

long

**short** 

<u>signed</u>

true

<u>unsigned</u>

wchar\_t

## break

The break keyword is used to break out of a <u>do</u>, <u>for</u>, or <u>while</u> loop. It is also used to finish each clause of a <u>switch</u> statement, keeping the program from "falling through" to the next case in the code. An example:

```
while( x < 100 ) {
   if( x < 0 )
      break;
   cout << x << endl;
   x++;
}</pre>
```

A given break statement will break out of only the closest loop, no further. If you have a triply-nested for loop, for example, you might want to include extra logic or a <u>goto</u> statement to break out of the loop.

Related topics:
<a href="mailto:continue">continue</a>
<a href="mailto:do">do</a>
<a href="mailto:for">for</a>

goto switch while

#### case

The case keyword is used to test a variable against a certain value in a <u>switch</u> statement.

Related topics:

default switch

### catch

The catch statement handles exceptions generated by the **throw** statement.

Related topics:

throw

<u>try</u>

### char

The char keyword is used to declare character variables. For more information about variable types, see the <u>data types</u> page.

Related topics:

bool
double
float
int
long
short
signed
unsigned
void
wchar t

### class

Syntax:

```
class class-name : inheritance-list {
  private-members-list;
  protected:
  protected-members-list;
  public:
  public-members-list;
} object-list;
```

The class keyword allows you to create new classes. *class-name* is the name of the class that you wish to create, and *inheritance-list* is an optional list of classes inherited by the new class. Members of the class are private by default, unless listed under either the protected or public labels. *object-list* can be used to immediately instantiate one or more instances of the class, and is also optional. For example:

```
class Date {
  int Day;
  int Month;
  int Year;
  public:
    void display();
};
```

Related topics:

**friend** 

private
protected
public
struct
this
typename
union
virtual

### const

The const keyword can be used to tell the compiler that a certain variable should not be modified once it has been initialized.

It can also be used to declare functions of a class that do not alter any class data.

Related topics:

const\_cast
mutable

### const\_cast

Syntax:

```
const_cast<type> (object);
```

The const\_cast keyword can be used to remove the **const** or **volatile** property from some variable. The target data type must be the same as the source type, except (of course) that the target type doesn't have to be <u>const</u>.

### Related topics:

const
dynamic\_cast
reinterpret\_cast
static\_cast

## continue

The continue statement can be used to bypass iterations of a given loop.

For example, the following code will display all of the numbers between 0 and 20 except 10:

```
for( int i = 0; i < 21; i++ ) {
  if( i == 10 ) {
    continue;
  }
  cout << i << " ";
}</pre>
```

Related topics:

**break** 

do

for

**while** 

## default

A default <u>case</u> in the <u>switch</u> statement.

Related topics:

<u>case</u>

**switch** 

## delete

Syntax:

```
delete p;
```

#### delete[] pArray;

The delete operator frees the memory pointed to by p. The argument should have been previously allocated by a call to  $\underline{\text{new}}$ . The second form of delete should be used to delete an array.

#### Related topics:

```
(Standard C Memory) <u>free</u>
(Standard C Memory) <u>malloc</u>
<u>new</u>
```

## do

#### Syntax:

```
do {
  statement-list;
} while( condition );
```

The do construct evaluates the given *statement-list* repeatedly, until *condition* becomes false. Note that every do loop will evaluate its statement list at least once, because the terminating condition is tested at the end of the loop.

Related topics:

break continue for while

## double

The double keyword is used to declare double precision floating-point variables. Also see the <u>data types</u> page.

### Related topics:

bool
char
float
int
long
short
signed
unsigned
void
wchar t

# dynamic\_cast

Syntax:

```
dynamic_cast<type> (object);
```

The dynamic\_cast keyword casts a datum from one type to another, performing a runtime check to ensure the validity of the cast. If you attempt to cast between incompatible types, the result of the cast will be **NULL**.

Related topics:

const\_cast
reinterpret\_cast
static\_cast

## else

The else keyword is used as an alternative case for the <u>if</u> statement.

Related topics:

if

### enum

Syntax:

```
enum name {name-list} var-list;
```

The enum keyword is used to create an enumerated type named name that consists of the elements in *name-list*. The *var-list* argument is optional, and can be used to create instances of the type along with the declaration. For example, the following code creates an enumerated type for colors:

```
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
...
ColorT c1 = indigo;
if( c1 == indigo ) {
  cout << "c1 is indigo" << endl;
}</pre>
```

In the above example, the effect of the enumeration is to introduce several new constants named *red*, *orange*, *yellow*, etc. By default, these constants are assigned consecutive integer values starting at zero. You can change the values of those constants, as shown by the next example:

```
enum ColorT { red = 10, blue = 15, green };
...
ColorT c = green;
cout << "c is " << c << endl;</pre>
```

When executed, the above code will display the following output:

```
c is 16
```

Note that the above examples will only work with C++ compilers. If you're working in regular C, you will need to specify the *enum* keyword whenever you create an instance of an enumerated type:

```
enum ColorT { red = 10, blue = 15, green };
...
enum ColorT c = green; // note the aditional enum keyword
printf( "c is %d\n", c );
```

## explicit

When a constructor is specified as explicit, no automatic conversion will be used with that constructor -- but parameters passed to the constructor may still be converted. For example:

## export

The export keyword is intended to allow definitions of C++ templates to be separated from their declarations. While officially part of the C++ standard, the export keyword is only supported by a few compilers (such as the Comeau C++ compiler) and is not supported by such mainstream compilers as GCC and Visual C++.

### extern

The extern keyword is used to inform the compiler about variables declared outside of the current scope. Variables described by extern statements will not have any space allocated for them, as they should be properly defined elsewhere.

Extern statements are frequently used to allow data to span the scope of multiple files.

## false

The Boolean value of "false".

Related topics:

**bool** 

true

## float

The float keyword is used to declare floating-point variables. Also see the <u>data</u> <u>types</u> page.

Related topics:

**bool** 

char

double

int

long

short

**signed** 

## for

Syntax:

```
for( initialization; test-condition; increment ) {
  statement-list;
}
```

The for construct is a general looping mechanism consisting of 4 parts:

- 1. the initialization, which consists of 0 or more comma-delimited variable initialization statements
- 2. the test-condition, which is evaluated to determine if the execution of the for loop will continue
- 3. the increment, which consists of 0 or more comma-delimited statements that increment variables
- 4. and the statement-list, which consists of 0 or more statements that will be executed each time the loop is executed.

For example:

Related topics:

break

```
continue
do
if
while
```

## friend

The friend keyword allows classes or functions not normally associated with a given class to have access to the private data of that class.

Related topics:

**class** 

## goto

Syntax:

```
goto labelA;
...
labelA:
```

The goto statement causes the current thread of execution to jump to the specified label. While the use of the goto statement is generally <u>considered harmful</u>, it can occasionally be useful. For example, it may be cleaner to use a goto to break out of a deeply-nested <u>for loop</u>, compared to the space and time that extra <u>break logic</u> would consume.

Related topics:

**break** 

if

Syntax:

```
if( conditionA ) {
    statement-listA;
}
else if( conditionB ) {
    statement-listB;
}
...
else {
    statement-listN;
}
```

The if construct is a branching mechanism that allows different code to execute under different conditions. The conditions are evaluated in order, and the statement-list of the first condition to evaluate to true is executed. If no conditions evaluate to true and an <u>else</u> statement is present, then the statement list within the else block will be executed. All of the <u>else</u> blocks are optional.

Related topics:

else for switch while

## inline

Syntax:

```
inline int functionA( int i ) {
   ...
}
```

The inline keyword requests that the compiler expand a given function in place, as opposed to inserting a call to that function. The inline keyword is a request, not a command, and the compiler is free to ignore it for whatever reason.

When a function declaration is included in a class definition, the compiler should try to automatically inline that function. No inline keyword is necessary in this case.

## int

The int keyword is used to declare integer variables. Also see the <u>data types</u> page.

### Related topics:

bool

char

double

**float** 

long

**short** 

**signed** 

unsigned

**void** 

wchar\_t

# long

The long keyword is a data type modifier that is used to declare long integer variables. For more information on long, see the <u>data types</u> page.

### Related topics:

**bool** 

char

**double** 

float

int

**short** 

signed

void

## mutable

The mutable keyword overrides any enclosing <u>const</u> statement. A mutable member of a <u>const</u> object can be modified.

Related topics:

const

### namespace

Syntax:

```
namespace name {
  declaration-list;
}
```

The namespace keyword allows you to create a new scope. The name is optional, and can be omitted to create an unnamed namespace. Once you create a namespace, you'll have to refer to it explicitly or use the <u>using</u> keyword.

Example code:

```
namespace CartoonNameSpace {
   int HomersAge;
   void incrementHomersAge() {
      HomersAge++;
   }
}
int main() {
   ...
   CartoonNameSpace::HomersAge = 39;
   CartoonNameSpace::incrementHomersAge();
   cout << CartoonNameSpace::HomersAge << endl;
   ...
}</pre>
```

Related topics:

using

#### new

Syntax:

```
pointer = new type;
pointer = new type( initializer );
pointer = new type[size];
pointer = new( arg-list ) type...
```

The new operator (valid only in C++) allocates a new chunk of memory to hold a variable of type *type* and returns a pointer to that memory. An optional initializer can be used to initialize the memory. Allocating arrays can be accomplished by providing a *size* parameter in brackets.

The optional *arg-list* parameter can be used with any of the other formats to pass a variable number of arguments to an overloaded version of new(). For example, the following code shows how the new() function can be overloaded for a class and then passed arbitrary arguments:

```
class Base {
public:
    Base() { }

    void *operator new( unsigned int size, string str ) {
        cout << "Logging an allocation of " << size << " bytes for new
        return malloc( size );
    }

    int var;
    double var2;
};

...

Base* b = new ("Base instance 1") Base;</pre>
```

If an int is 4 bytes and a double is 8 bytes, the above code generates the following output when run:

Logging an allocation of 12 bytes for new object 'Base instance 1

Related topics:

**delete** 

(Standard C Memory) <u>free</u> (Standard C Memory) <u>malloc</u>

## operator

Syntax:

```
return-type class-name::operator#(parameter-list) {
    ...
}
return-type operator#(parameter-list) {
    ...
}
```

The operator keyword is used to overload operators. The sharp sign (#) listed above in the syntax description represents the operator which will be overloaded. If part of a class, the *class-name* should be specified. For unary operators, *parameter-list* should be empty, and for binary operators, *parameter-list* should contain the operand on the right side of the operator (the operand on the left side is passed as this).

For the non-member operator overload function, the operand on the left side should be passed as the first parameter and the operand on the right side should be passed as the second parameter.

You cannot overload the #, ##, ., :, .\*, or ? tokens.

Related topics:

<u>this</u>

# private

Private data of a class can only be accessed by members of that class, except when <u>friend</u> is used. The <u>private</u> keyword can also be used to inherit a base class

privately, which causes all <u>public</u> and <u>protected</u> members of the base class to become private members of the derived class.

Related topics:

<u>class</u> <u>protected</u> public

## protected

Protected data are private to their own class but can be inherited by derived classes. The protected keyword can also be used as an inheritance specifier, which causes all <u>public</u> and protected members of the base class to become protected members of the derived class.

Related topics:

<u>class</u> private public

## public

Public data in a class are accessible to everyone. The public keyword can also be used as an inheritance specifier, which causes all public and <u>protected</u> members of the base class to become public and protected members of the derived class.

Related topics:

<u>class</u> <u>private</u> protected

# register

The register keyword requests that a variable be optimized for speed, and fell out of common use when computers became better at most code optimizations than humans.

Related topics:

auto

## reinterpret\_cast

Syntax:

```
reinterpret_cast<type> (object);
```

The reinterpret\_cast operator changes one data type into another. It should be used to cast between incompatible pointer types.

Related topics:

const\_cast
dynamic\_cast
static\_cast

### return

Syntax:

```
return;
return( value );
```

The return statement causes execution to jump from the current function to whatever function called the current function. An optional *value* can be returned. A function may have more than one return statement.

## short

The short keyword is a data type modifier that is used to declare short integer variables. See the <u>data types</u> page.

### Related topics:

**bool** 

**char** 

double

**float** 

int

long

**signed** 

unsigned

void

wchar t

# signed

The signed keyword is a data type modifier that is usually used to declare signed char variables. See the <u>data types</u> page.

### Related topics:

**bool** 

**char** 

double

**float** 

int

long

**short** 

unsigned

**void** 

wchar t

## sizeof

The size of operator is a compile-time operator that returns the size of the argument passed to it. The size is a multiple of the size of a char, which on many personal computers is 1 byte (or 8 bits). The number of bits in a char is stored in the CHAR\_BIT constant defined in the <cli>climits> header file.

For example, the following code uses size of to display the sizes of a number of variables:

```
struct EmployeeRecord {
  int ID;
  int age;
  double salary;
  EmployeeRecord* boss;
};
cout << "sizeof(int): " << sizeof(int) << endl</pre>
     << "sizeof(float): " << sizeof(float) << endl
     << "sizeof(double): " << sizeof(double) << endl</pre>
     << "sizeof(char): " << sizeof(char) << endl
     << "sizeof(EmployeeRecord): " << sizeof(EmployeeRecord) << er</pre>
int i;
float f;
double d;
char c;
EmployeeRecord er;
cout << "sizeof(i): " << sizeof(i) << endl</pre>
     << "sizeof(f): " << sizeof(f) << endl
     << "sizeof(d): " << sizeof(d) << endl
     << "sizeof(c): " << sizeof(c) << endl
     << "sizeof(er): " << sizeof(er) << endl;
```

On some machines, the above code displays this output:

```
sizeof(int): 4
sizeof(float): 4
sizeof(double): 8
sizeof(char): 1
```

```
sizeof(EmployeeRecord): 20
sizeof(i): 4
sizeof(f): 4
sizeof(d): 8
sizeof(c): 1
sizeof(er): 20
```

Note that size of can either take a variable type (such as **int**) or a variable name (such as **i** in the example above).

It is also important to note that the sizes of various types of variables can change depending on what system you're on. Check out a description of the C and C++ data types for more information.

The parentheses around the argument are not required if you are using size of with a variable type (e.g. size of (int)).

Related topics:

C/C++ Data Types

### static

The static data type modifier is used to create permanent storage for variables. Static variables keep their value between function calls. When used in a <u>class</u>, all instantiations of that class share one copy of the variable.

## static\_cast

Syntax:

```
static_cast<type> (object);
```

The static\_cast keyword can be used for any normal conversion between types. No runtime checks are performed.

Related topics:

### struct

Syntax:

```
struct struct-name : inheritance-list {
public-members-list;
protected:
protected-members-list;
private:
private-members-list;
} object-list;
```

Structs are like `classes`, except that by default members of a struct are <u>public</u> rather than <u>private</u>. In C, structs can only contain data and are not permitted to have inheritance lists. For example:

```
struct Date {
  int Day;
  int Month;
  int Year;
};
```

Related topics:

<u>class</u> union

## switch

Syntax:

```
switch( expression ) {
  case A:
  statement list;
  break;
  case B:
```

```
statement list;
break;
...
case N:
statement list;
break;
default:
statement list;
break;
}
```

The switch statement allows you to test an expression for many values, and is commonly used as a replacement for multiple <u>if()...else if()...else if()...</u> statements. <u>break</u> statements are required between each <u>case</u> statement, otherwise execution will "fall-through" to the next <u>case</u> statement. The <u>default</u> case is optional. If provided, it will match any case not explicitly covered by the preceding cases in the switch statement. For example:

```
char keystroke = getch();
switch( keystroke ) {
   case 'a':
   case 'b':
   case 'c':
   case 'd':
        KeyABCDPressed();
        break;
   case 'e':
        KeyEPressed();
        break;
   default:
        UnknownKeyPressed();
        break;
}
```

Related topics:

break
case
default
if

# template

#### *Syntax:*

```
template <class data-type> return-type name( parameter-list ) {
  statement-list;
}
```

Templates are used to create generic functions and can operate on data without knowing the nature of that data. They accomplish this by using a placeholder data-type for which many other data types can be substituted.

#### Example code:

For example, the following code uses a template to define a generic swap function that can swap two variables of any type:

```
template<class X> void genericSwap( X &a, X &b ) {
  X tmp;
  tmp = a;
  a = b;
  b = tmp;
int main(void) {
  int num1 = 5;
  int num2 = 21;
  cout << "Before, num1 is " << num1 << " and num2 is " << num2</pre>
  genericSwap( num1, num2 );
  cout << "After, num1 is " << num1 << " and num2 is " << num2 <
  char c1 = 'a';
  char c2 = 'z';
  cout << "Before, c1 is " << c1 << " and c2 is " << c2 << endl;
  genericSwap( c1, c2 );
  cout << "After, c1 is " << c1 << " and c2 is " << c2 << endl;
  return( 0 );
```

### Related topics:

#### typename

## this

The this keyword is a pointer to the current object. All member functions of a <u>class</u> have a this pointer.

Related topics:

<u>class</u>

**operator** 

### throw

Syntax:

```
try {
  statement list;
  }
  catch( typeA arg ) {
   statement list;
  }
  catch( typeB arg ) {
   statement list;
  }
  ...
  catch( typeN arg ) {
   statement list;
  }
}
```

The throw statement is part of the C++ mechanism for exception handling. This statement, together with the <u>try</u> and <u>catch</u> statements, the C++ exception handling system gives programmers an elegant mechanism for error recovery.

You will generally use a <u>try</u> block to execute potentially error-prone code. Somewhere in this code, a throw statement can be executed, which will cause execution to jump out of the <u>try</u> block and into one of the <u>catch</u> blocks. For example:

```
try {
   cout << "Before throwing exception" << endl;
   throw 42;
   cout << "Shouldn't ever see this" << endl;
}
catch( int error ) {
   cout << "Error: caught exception " << error << endl;
}</pre>
```

Related topics:

catch

<u>try</u>

### true

The Boolean value of "true".

Related topics:

**bool** 

**false** 

## try

The try statement attempts to execute exception-generating code. See the <u>throw</u> statement for more details.

Related topics:

<u>catch</u>

**throw** 

# typedef

Syntax:

typedef existing-type new-type;

The typedef keyword allows you to create a new alias for an existing data type.

This is often useful if you find yourself using a unwieldy data type -- you can use typedef to create a shorter, easier-to-use name for that data type. For

#### example:

```
typedef unsigned int* pui_t;

// data1 and data2 have the same type
piu_t data1;
unsigned int* data2;
```

# typeid

Syntax:

```
typeid( object );
```

The typeid operator returns a reference to a type\_info object that describes `object`.

## typename

The typename keyword can be used to describe an undefined type or in place of the <u>class</u> keyword in a <u>template</u> declaration.

Related topics:

<u>class</u>

**template** 

### union

*Syntax:* 

```
union union-name {
  public-members-list;
  private:
  private-members-list;
```

```
} object-list;
```

A union is like a <u>class</u>, except that all members of a union share the same memory location and are by default <u>public</u> rather than <u>private</u>. For example:

```
union Data {
   int i;
   char c;
};
```

Related topics:

<u>class</u> struct

# unsigned

The unsigned keyword is a data type modifier that is usually used to declare unsigned <u>int</u> variables. See the <u>data types</u> page.

Related topics:

bool

char

double

float

int

**short** 

**signed** 

void

wchar t

# using

The using keyword is used to import a <u>namespace</u> (or parts of a namespace) into the current scope.

#### Example code:

For example, the following code imports the entire *std* namespace into the current scope so that items within that namespace can be used without a preceding "std::".

```
using namespace std;
```

Alternatively, the next code snippet just imports a single element of the *std* namespace into the current namespace:

```
using std::cout;
```

Related topics:

<u>namespace</u>

### virtual

#### Syntax:

```
virtual return-type name( parameter-list );
virtual return-type name( parameter-list ) = 0;
```

The virtual keyword can be used to create virtual functions, which can be overridden by derived classes.

- A virtual function indicates that a function can be overridden in a subclass, and that the overridden function will actually be used.
- When a base object pointer points to a derived object that contains a virtual function, the decision about which version of that function to call is based on the type of object pointed to by the pointer, and this process happens at runtime.
- A base object can point to different derived objects and have different versions of the virtual function run.

If the function is specified as a pure virtual function (denoted by the = 0), it must be overridden by a derived class.

#### Example code:

For example, the following code snippet shows how a child class can override a virtual method of its parent, and how a non-virtual method in the parent cannot be overridden:

```
class Base {
public:
 void nonVirtualFunc() {
   cout << "Base: non-virtual function" << endl;</pre>
 virtual void virtualFunc() {
   cout << "Base: virtual function" << endl;</pre>
}
};
class Child : public Base {
public:
 void nonVirtualFunc() {
   cout << "Child: non-virtual function" << endl;</pre>
 void virtualFunc() {
   cout << "Child: virtual function" << endl;</pre>
};
int main() {
 Base* basePointer = new Child();
 basePointer->nonVirtualFunc();
 basePointer->virtualFunc();
 return 0;
```

When run, the above code displays:

```
Base: non-virtual function
Child: virtual function
```

#### Related topics:

<u>class</u>

## void

The void keyword is used to denote functions that return no value, or generic variables which can point to any type of data. Void can also be used to declare an empty parameter list. Also see the <u>data types</u> page.

#### Related topics:

**char** 

double

float

int

long

**short** 

signed

unsigned

wchar\_t

## volatile

The volatile keyword is an implementation-dependent modifier, used when declaring variables, which prevents the compiler from optimizing those variables. Volatile should be used with variables whose value can change in unexpected ways (i.e. through an interrupt), which could conflict with optimizations that the compiler might perform.

## wchar\_t

The keyword wchar\_t is used to declare wide character variables. Also see the <u>data types</u> page.

Related topics:

bool

char

**double** 

**float** 

int short signed unsigned void

## while

Syntax:

```
while( condition ) {
  statement-list;
}
```

The while keyword is used as a looping construct that will evaluate the *statement-list* as long as *condition* is true. Note that if the *condition* starts off as false, the *statement-list* will never be executed. (You can use a do loop to guarantee that the statement-list will be executed at least once.) For example:

```
bool done = false;
while( !done ) {
   ProcessData();
   if( StopLooping() ) {
      done = true;
   }
}
```

Related topics:

break continue

<u>do</u>

<u>for</u>

<u>if</u>