# All C++ Functions

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bitset Constructors</strong> (C++ Bitsets)</td>
<td>create new bitsets</td>
</tr>
<tr>
<td><strong>Bitset Operators</strong> (C++ Bitsets)</td>
<td>compare and assign bitsets</td>
</tr>
<tr>
<td><strong>Vector constructors</strong></td>
<td>create vectors and initialize them with some data</td>
</tr>
<tr>
<td><strong>Container constructors</strong> (C++ Double-ended Queues)</td>
<td>create containers and initialize them with some data</td>
</tr>
<tr>
<td><strong>Container constructors</strong> (C++ Lists)</td>
<td>create containers and initialize them with some data</td>
</tr>
<tr>
<td><strong>Container constructors &amp; destructors</strong> (C++ Sets)</td>
<td>default methods to allocate, copy, and deallocate containers</td>
</tr>
<tr>
<td><strong>Container constructors &amp; destructors</strong> (C++ Multisets)</td>
<td>default methods to allocate, copy, and deallocate multisets</td>
</tr>
<tr>
<td><strong>Map constructors &amp; destructors</strong> (C++ Maps)</td>
<td>default methods to allocate, copy, and deallocate maps</td>
</tr>
<tr>
<td><strong>Multimap constructors &amp; destructors</strong> (C++ Multimaps)</td>
<td>default methods to allocate, copy, and deallocate containers</td>
</tr>
<tr>
<td><strong>Container operators</strong> (C++ Lists)</td>
<td>assign and compare containers</td>
</tr>
<tr>
<td><strong>Container operators</strong> (C++ Sets)</td>
<td>assign and compare containers</td>
</tr>
<tr>
<td><strong>Container operators</strong> (C++ Multisets)</td>
<td>assign and compare containers</td>
</tr>
<tr>
<td><strong>Multimap operators</strong> (C++ Multimaps)</td>
<td>assign and compare containers</td>
</tr>
<tr>
<td><strong>Vector operators</strong></td>
<td>compare, assign, and access elements of a vector</td>
</tr>
<tr>
<td><strong>Container operators</strong> (C++ Double-ended Queues)</td>
<td>compare, assign, and access elements of a container</td>
</tr>
<tr>
<td><strong>I/O Constructors</strong> (C++ I/O)</td>
<td>constructors</td>
</tr>
<tr>
<td><strong>Map operators</strong> (C++ Maps)</td>
<td>assign, compare, and access elements of a map</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Priority queue constructors</strong></td>
<td>construct a new priority queue</td>
</tr>
<tr>
<td><strong>Queue constructor</strong></td>
<td>construct a new queue</td>
</tr>
<tr>
<td><strong>Stack constructors</strong></td>
<td>construct a new stack</td>
</tr>
<tr>
<td><strong>String constructors</strong></td>
<td>create strings from arrays of characters and other strings</td>
</tr>
<tr>
<td><strong>String operators</strong></td>
<td>concatenate strings, assign strings, use strings for I/O, compare strings</td>
</tr>
<tr>
<td><strong>accumulate</strong></td>
<td>sum up a range of elements</td>
</tr>
<tr>
<td><strong>adjacent_difference</strong></td>
<td>compute the differences between adjacent elements in a range</td>
</tr>
<tr>
<td><strong>adjacent_find</strong></td>
<td>finds two items that are adjacent to eachother</td>
</tr>
<tr>
<td><strong>any</strong></td>
<td>true if any bits are set</td>
</tr>
<tr>
<td><strong>append</strong></td>
<td>append characters and strings onto a string</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>append</strong></td>
<td>append characters and strings onto a string</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>assign elements to a container</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>returns an element at a specific location</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>returns an element at a specific location</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>returns an element at a specific location</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>returns an element at a specific location</td>
</tr>
<tr>
<td><strong>auto_ptr</strong></td>
<td>create pointers that automatically destroy objects</td>
</tr>
<tr>
<td><strong>back</strong></td>
<td>returns a reference to last element of a container</td>
</tr>
<tr>
<td><strong>back</strong></td>
<td>returns a reference to last element of a container</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>back (C++ Lists)</td>
<td>returns a reference to last element of a container</td>
</tr>
<tr>
<td>back (C++ Queues)</td>
<td>returns a reference to last element of a container</td>
</tr>
<tr>
<td>bad (C++ I/O)</td>
<td>true if an error occurred</td>
</tr>
<tr>
<td>begin (C++ Strings)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Vectors)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Double-ended Queues)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Lists)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Sets)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Multisets)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Multimaps)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>begin (C++ Maps)</td>
<td>returns an iterator to the beginning of the container</td>
</tr>
<tr>
<td>binary_search (C++ Algorithms)</td>
<td>determine if an element exists in a certain range</td>
</tr>
<tr>
<td>c_str (C++ Strings)</td>
<td>returns a standard C character array version of the string</td>
</tr>
<tr>
<td>capacity (C++ Vectors)</td>
<td>returns the number of elements that the container can hold</td>
</tr>
<tr>
<td>capacity (C++ Strings)</td>
<td>returns the number of elements that the container can hold</td>
</tr>
<tr>
<td>clear (C++ I/O)</td>
<td>clear and set status flags</td>
</tr>
<tr>
<td>clear (C++ Strings)</td>
<td>removes all elements from the container</td>
</tr>
<tr>
<td>clear (C++ Vectors)</td>
<td>removes all elements from the container</td>
</tr>
<tr>
<td>clear (C++ Double-ended Queues)</td>
<td>removes all elements from the container</td>
</tr>
<tr>
<td>Function</td>
<td>(C++ Type)</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>(C++ Lists)</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>(C++ Sets)</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>(C++ Multisets)</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>(C++ Maps)</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>(C++ Multimaps)</td>
</tr>
<tr>
<td><strong>close</strong></td>
<td>(C++ I/O)</td>
</tr>
<tr>
<td><strong>compare</strong></td>
<td>(C++ Strings)</td>
</tr>
<tr>
<td><strong>copy</strong></td>
<td>(C++ Strings)</td>
</tr>
<tr>
<td><strong>copy</strong></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><strong>copy_backward</strong></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><strong>copy_n</strong></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><strong>count</strong></td>
<td>(C++ Sets)</td>
</tr>
<tr>
<td><strong>count</strong></td>
<td>(C++ Multisets)</td>
</tr>
<tr>
<td><strong>count</strong></td>
<td>(C++ Maps)</td>
</tr>
<tr>
<td><strong>count</strong></td>
<td>(C++ Multimaps)</td>
</tr>
<tr>
<td><strong>count</strong></td>
<td>(C++ Bitsets)</td>
</tr>
<tr>
<td><strong>count</strong></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><strong>count_if</strong></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><strong>data</strong></td>
<td>(C++ Strings)</td>
</tr>
<tr>
<td><strong>empty</strong></td>
<td>(C++ Strings)</td>
</tr>
<tr>
<td><strong>empty</strong></td>
<td>(C++ Vectors)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Double-ended Queues)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Lists)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Sets)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Multisets)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Multimaps)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Stacks)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Queues)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>empty</strong> (C++ Priority Queues)</td>
<td>true if the container has no elements</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Strings)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Vectors)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Double-ended Queues)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Lists)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Sets)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Multisets)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Multimaps)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Maps)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>end</strong> (C++ Multimaps)</td>
<td>returns an iterator just past the last element of a container</td>
</tr>
<tr>
<td><strong>eof</strong> (C++ I/O)</td>
<td>true if at the end-of-file</td>
</tr>
<tr>
<td><strong>equal</strong> (C++ Algorithms)</td>
<td>determine if two sets of elements are the same</td>
</tr>
<tr>
<td><strong>equal_range</strong> (C++ Sets)</td>
<td>returns iterators to the first and just past the last elements matching a specific key</td>
</tr>
<tr>
<td><strong>equal_range</strong> (C++ Multisets)</td>
<td>returns iterators to the first and just past the last elements matching a specific key</td>
</tr>
<tr>
<td>Function</td>
<td>(C++ Libraries)</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td><code>equal_range</code></td>
<td>Maps</td>
</tr>
<tr>
<td><code>equal_range</code></td>
<td>Multimaps</td>
</tr>
<tr>
<td><code>equal_range</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Strings</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Vectors</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Double-ended Queues</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Lists</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Sets</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Multisets</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Maps</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Multimaps</td>
</tr>
<tr>
<td><code>fail</code></td>
<td>I/O</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>I/O</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td><code>fill_n</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Sets</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Multisets</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Maps</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Multimaps</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Strings</td>
</tr>
<tr>
<td><code>find_end</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td><code>find_first_not_of</code></td>
<td>Strings</td>
</tr>
<tr>
<td><code>find_first_of</code></td>
<td>Strings</td>
</tr>
<tr>
<td><code>find_first_of</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td><code>find_if</code></td>
<td>Algorithms</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>find_last_not_of</code> (C++ Strings)</td>
<td>find last absence of characters</td>
</tr>
<tr>
<td><code>find_last_of</code> (C++ Strings)</td>
<td>find last occurrence of characters</td>
</tr>
<tr>
<td><code>flags</code> (C++ I/O)</td>
<td>access or manipulate io stream format flags</td>
</tr>
<tr>
<td><code>flip</code> (C++ Bitsets)</td>
<td>reverses the bitset</td>
</tr>
<tr>
<td><code>flush</code> (C++ I/O)</td>
<td>empty the buffer</td>
</tr>
<tr>
<td><code>for_each</code> (C++ Algorithms)</td>
<td>apply a function to a range of elements</td>
</tr>
<tr>
<td><code>front</code> (C++ Vectors)</td>
<td>returns a reference to the first element of a container</td>
</tr>
<tr>
<td><code>front</code> (C++ Double-ended Queues)</td>
<td>returns a reference to the first element of a container</td>
</tr>
<tr>
<td><code>front</code> (C++ Lists)</td>
<td>returns a reference to the first element of a container</td>
</tr>
<tr>
<td><code>front</code> (C++ Queues)</td>
<td>returns a reference to the first element of a container</td>
</tr>
<tr>
<td><code>gcount</code> (C++ I/O)</td>
<td>number of characters read during last input</td>
</tr>
<tr>
<td><code>generate</code> (C++ Algorithms)</td>
<td>saves the result of a function in a range</td>
</tr>
<tr>
<td><code>generate_n</code> (C++ Algorithms)</td>
<td>saves the result of N applications of a function</td>
</tr>
<tr>
<td><code>get</code> (C++ I/O)</td>
<td>read characters</td>
</tr>
<tr>
<td><code>getline</code> (C++ I/O)</td>
<td>read a line of characters</td>
</tr>
<tr>
<td><code>getline</code> (C++ Strings)</td>
<td>read data from an I/O stream into a string</td>
</tr>
<tr>
<td><code>good</code> (C++ I/O)</td>
<td>true if no errors have occurred</td>
</tr>
<tr>
<td><code>ignore</code> (C++ I/O)</td>
<td>read and discard characters</td>
</tr>
<tr>
<td><code>includes</code> (C++ Algorithms)</td>
<td>returns true if one set is a subset of another</td>
</tr>
<tr>
<td><code>inner_product</code> (C++ Algorithms)</td>
<td>compute the inner product of two ranges of elements</td>
</tr>
<tr>
<td><code>inplace_merge</code> (C++ Algorithms)</td>
<td>merge two ordered ranges in-place</td>
</tr>
<tr>
<td><code>insert</code> (C++ Strings)</td>
<td>insert characters into a string</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>insert</code> (C++ Vectors)</td>
<td>inserts elements into the container</td>
</tr>
<tr>
<td><code>insert</code> (C++ Double-ended Queues)</td>
<td>inserts elements into the container</td>
</tr>
<tr>
<td><code>insert</code> (C++ Lists)</td>
<td>inserts elements into the container</td>
</tr>
<tr>
<td><code>insert</code> (C++ Sets)</td>
<td>insert items into a container</td>
</tr>
<tr>
<td><code>insert</code> (C++ Multisets)</td>
<td>inserts items into a container</td>
</tr>
<tr>
<td><code>insert</code> (C++ Multimaps)</td>
<td>inserts items into a container</td>
</tr>
<tr>
<td><code>insert</code> (C++ Maps)</td>
<td>insert items into a container</td>
</tr>
<tr>
<td><code>is_heap</code> (C++ Algorithms)</td>
<td>returns true if a given range is a heap</td>
</tr>
<tr>
<td><code>is_sorted</code> (C++ Algorithms)</td>
<td>returns true if a range is sorted in ascending order</td>
</tr>
<tr>
<td><code>iter_swap</code> (C++ Algorithms)</td>
<td>swaps the elements pointed to by two iterators</td>
</tr>
<tr>
<td><code>key_comp</code> (C++ Sets)</td>
<td>returns the function that compares keys</td>
</tr>
<tr>
<td><code>key_comp</code> (C++ Multisets)</td>
<td>returns the function that compares keys</td>
</tr>
<tr>
<td><code>key_comp</code> (C++ Maps)</td>
<td>returns the function that compares keys</td>
</tr>
<tr>
<td><code>key_comp</code> (C++ Multimaps)</td>
<td>returns the function that compares keys</td>
</tr>
<tr>
<td><code>length</code> (C++ Strings)</td>
<td>returns the length of the string</td>
</tr>
<tr>
<td><code>lexicographical_compare</code> (C++ Algorithms)</td>
<td>returns true if one range is lexicographically less than another</td>
</tr>
<tr>
<td><code>lexicographical_compare_3way</code> (C++ Algorithms)</td>
<td>determines if one range is lexicographically less than or greater than another</td>
</tr>
<tr>
<td><code>lower_bound</code> (C++ Sets)</td>
<td>returns an iterator to the first element greater than or equal to a certain value</td>
</tr>
<tr>
<td><code>lower_bound</code> (C++ Multisets)</td>
<td>returns an iterator to the first element greater than or equal to a certain value</td>
</tr>
<tr>
<td><code>lower_bound</code> (C++ Maps)</td>
<td>returns an iterator to the first element greater than or equal to a certain value</td>
</tr>
<tr>
<td><code>lower_bound</code> (C++ Multimaps)</td>
<td>returns an iterator to the first element greater than or equal to a certain value</td>
</tr>
<tr>
<td></td>
<td>search for the first place that a value</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>lower_bound</code></td>
<td>(C++ Algorithms) can be inserted while preserving order</td>
</tr>
<tr>
<td><code>make_heap</code></td>
<td>(C++ Algorithms) creates a heap out of a range of elements</td>
</tr>
<tr>
<td><code>max</code></td>
<td>(C++ Algorithms) returns the larger of two elements</td>
</tr>
<tr>
<td><code>max_element</code></td>
<td>(C++ Algorithms) returns the largest element in a range</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Strings) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Vectors) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Double-ended Queues) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Lists) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Sets) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Multisets) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Maps) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>(C++ Multimaps) returns the maximum number of elements that the container can hold</td>
</tr>
<tr>
<td><code>merge</code></td>
<td>(C++ Lists) merge two lists</td>
</tr>
<tr>
<td><code>merge</code></td>
<td>(C++ Algorithms) merge two sorted ranges</td>
</tr>
<tr>
<td><code>min</code></td>
<td>(C++ Algorithms) returns the smaller of two elements</td>
</tr>
<tr>
<td><code>min_element</code></td>
<td>(C++ Algorithms) returns the smallest element in a range</td>
</tr>
<tr>
<td><code>mismatch</code></td>
<td>(C++ Algorithms) finds the first position where two ranges differ</td>
</tr>
<tr>
<td><code>next_permutation</code></td>
<td>(C++ Algorithms) generates the next greater lexicographic permutation of a range of elements</td>
</tr>
<tr>
<td><code>none</code></td>
<td>(C++ Bitsets) true if no bits are set</td>
</tr>
<tr>
<td><code>nth_element</code></td>
<td>(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</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>open (C++ I/O)</td>
<td>create an input stream</td>
</tr>
<tr>
<td>partial_sort (C++ Algorithms)</td>
<td>sort the first N elements of a range</td>
</tr>
<tr>
<td>partial_sort_copy (C++</td>
<td>copy and partially sort a range of elements</td>
</tr>
<tr>
<td>Algorithms)</td>
<td></td>
</tr>
<tr>
<td>partial_sum (C++ Algorithms)</td>
<td>compute the partial sum of a range of elements</td>
</tr>
<tr>
<td>partition (C++ Algorithms)</td>
<td>divide a range of elements into two groups</td>
</tr>
<tr>
<td>peek (C++ I/O)</td>
<td>check the next input character</td>
</tr>
<tr>
<td>pop (C++ Stacks)</td>
<td>removes the top element of a container</td>
</tr>
<tr>
<td>pop (C++ Queues)</td>
<td>removes the top element of a container</td>
</tr>
<tr>
<td>pop (C++ Priority Queues)</td>
<td>removes the top element of a container</td>
</tr>
<tr>
<td>pop_back (C++ Vectors)</td>
<td>removes the last element of a container</td>
</tr>
<tr>
<td>pop_back (C++ Double-ended</td>
<td>removes the last element of a container</td>
</tr>
<tr>
<td>Queues)</td>
<td></td>
</tr>
<tr>
<td>pop_back (C++ Lists)</td>
<td>removes the last element of a container</td>
</tr>
<tr>
<td>pop_front (C++ Double-ended</td>
<td>removes the first element of the container</td>
</tr>
<tr>
<td>Queues)</td>
<td></td>
</tr>
<tr>
<td>pop_front (C++ Lists)</td>
<td>removes the first element of the container</td>
</tr>
<tr>
<td>pop_heap (C++ Algorithms)</td>
<td>remove the largest element from a heap</td>
</tr>
<tr>
<td>precision (C++ I/O)</td>
<td>manipulate the precision of a stream</td>
</tr>
<tr>
<td>prev_permutation (C++</td>
<td>generates the next smaller lexicographic permutation of a</td>
</tr>
<tr>
<td>Algorithms)</td>
<td>range of elements</td>
</tr>
<tr>
<td>push (C++ Stacks)</td>
<td>adds an element to the top of the container</td>
</tr>
<tr>
<td>push (C++ Queues)</td>
<td>adds an element to the end of the container</td>
</tr>
<tr>
<td>push (C++ Priority Queues)</td>
<td>adds an element to the end of the container</td>
</tr>
<tr>
<td></td>
<td>add an element to the end of the container</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>push_back</strong> (C++ Vectors)</td>
<td>container</td>
</tr>
<tr>
<td><strong>push_back</strong> (C++ Double-ended Queues)</td>
<td>add an element to the end of the container</td>
</tr>
<tr>
<td><strong>push_back</strong> (C++ Lists)</td>
<td>add an element to the end of the container</td>
</tr>
<tr>
<td><strong>push_back</strong> (C++ Strings)</td>
<td>add an element to the end of the container</td>
</tr>
<tr>
<td><strong>push_front</strong> (C++ Double-ended Queues)</td>
<td>add an element to the front of the container</td>
</tr>
<tr>
<td><strong>push_front</strong> (C++ Lists)</td>
<td>add an element to the front of the container</td>
</tr>
<tr>
<td><strong>push_heap</strong> (C++ Algorithms)</td>
<td>add an element to a heap</td>
</tr>
<tr>
<td><strong>put</strong> (C++ I/O)</td>
<td>write characters</td>
</tr>
<tr>
<td><strong>putback</strong> (C++ I/O)</td>
<td>return characters to a stream</td>
</tr>
<tr>
<td><strong>random_sample</strong> (C++ Algorithms)</td>
<td>randomly copy elements from one range to another</td>
</tr>
<tr>
<td><strong>random_sample_n</strong> (C++ Algorithms)</td>
<td>sample N random elements from a range</td>
</tr>
<tr>
<td><strong>random_shuffle</strong> (C++ Algorithms)</td>
<td>randomly re-order elements in some range</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Vectors)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Strings)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Double-ended Queues)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Lists)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Sets)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Multisets)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td><strong>rbegin</strong> (C++ Maps)</td>
<td>returns a <strong>reverse_iterator</strong> to the end of the container</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><code>rbegin</code></td>
<td>(C++ Multimaps)</td>
</tr>
<tr>
<td><code>rdstate</code></td>
<td>(C++ I/O)</td>
</tr>
<tr>
<td><code>read</code></td>
<td>(C++ I/O)</td>
</tr>
<tr>
<td><code>remove</code></td>
<td>(C++ Lists)</td>
</tr>
<tr>
<td><code>remove</code></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><code>remove_copy</code></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><code>remove_copy_if</code></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><code>remove_if</code></td>
<td>(C++ Lists)</td>
</tr>
<tr>
<td><code>remove_if</code></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Vectors)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Strings)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Double-ended Queues)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Lists)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Sets)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Multisets)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Maps)</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>(C++ Multimaps)</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>(C++ Strings)</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td><code>replace_copy</code></td>
<td>(C++ Algorithms)</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>replace_copy_if</code></td>
<td>(C++ Algorithms) copy a range of elements, replacing those for which a predicate is true</td>
</tr>
<tr>
<td><code>replace_if</code></td>
<td>(C++ Algorithms) change the values of elements for which a predicate is true</td>
</tr>
<tr>
<td><code>reserve</code></td>
<td>(C++ Vectors) sets the minimum capacity of the container</td>
</tr>
<tr>
<td><code>reserve</code></td>
<td>(C++ Strings) sets the minimum capacity of the container</td>
</tr>
<tr>
<td><code>reset</code></td>
<td>(C++ Bitsets) sets bits to zero</td>
</tr>
<tr>
<td><code>resize</code></td>
<td>(C++ Vectors) change the size of the container</td>
</tr>
<tr>
<td><code>resize</code></td>
<td>(C++ Double-ended Queues) change the size of the container</td>
</tr>
<tr>
<td><code>resize</code></td>
<td>(C++ Lists) change the size of the container</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>(C++ Lists) reverse the list</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>(C++ Algorithms) reverse elements in some range</td>
</tr>
<tr>
<td><code>reverse_copy</code></td>
<td>(C++ Algorithms) create a copy of a range that is reversed</td>
</tr>
<tr>
<td><code>rfind</code></td>
<td>(C++ Strings) find the last occurrence of a substring</td>
</tr>
<tr>
<td><code>rotate</code></td>
<td>(C++ Algorithms) move the elements in some range to the left by some amount</td>
</tr>
<tr>
<td><code>rotate_copy</code></td>
<td>(C++ Algorithms) copy and rotate a range of elements</td>
</tr>
<tr>
<td><code>search</code></td>
<td>(C++ Algorithms) search for a range of elements</td>
</tr>
<tr>
<td><code>search_n</code></td>
<td>(C++ Algorithms) search for N consecutive copies of an element in some range</td>
</tr>
<tr>
<td><code>seekg</code></td>
<td>(C++ I/O) perform random access on an input stream</td>
</tr>
<tr>
<td><code>seekp</code></td>
<td>(C++ I/O) perform random access on output streams</td>
</tr>
<tr>
<td><code>set</code></td>
<td>(C++ Bitsets) sets bits</td>
</tr>
<tr>
<td><code>set_difference</code></td>
<td>(C++ Algorithms) computes the difference between two sets</td>
</tr>
<tr>
<td><code>set_intersection</code></td>
<td>(C++ Algorithms) computes the intersection of two sets</td>
</tr>
<tr>
<td><code>set_symmetric_difference</code></td>
<td>(C++ Algorithms) computes the symmetric difference</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>set_union</strong> (C++ Algorithms)</td>
<td>computes the union of two sets</td>
</tr>
<tr>
<td><strong>set</strong> (C++ I/O)</td>
<td>set format flags</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Strings)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Vectors)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Double-ended Queues)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Lists)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Sets)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Multisets)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Maps)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Multimaps)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Stacks)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Queues)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Priority Queues)</td>
<td>returns the number of items in the container</td>
</tr>
<tr>
<td><strong>size</strong> (C++ Bitsets)</td>
<td>number of bits that the bitset can hold</td>
</tr>
<tr>
<td><strong>sort</strong> (C++ Lists)</td>
<td>sorts a list into ascending order</td>
</tr>
<tr>
<td><strong>sort</strong> (C++ Algorithms)</td>
<td>sort a range into ascending order</td>
</tr>
<tr>
<td><strong>sort_heap</strong> (C++ Algorithms)</td>
<td>turns a heap into a sorted range of elements</td>
</tr>
<tr>
<td><strong>splice</strong> (C++ Lists)</td>
<td>merge two lists in <strong>constant time</strong></td>
</tr>
<tr>
<td><strong>stable_partition</strong> (C++ Algorithms)</td>
<td>divide elements into two groups while preserving their relative order</td>
</tr>
<tr>
<td><strong>stable_sort</strong> (C++ Algorithms)</td>
<td>sort a range of elements while preserving order between equal</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>substr</code> (C++ Strings)</td>
<td>returns a certain substring</td>
</tr>
<tr>
<td><code>swap</code> (C++ Strings)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Vectors)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Double-ended Queues)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Lists)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Sets)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Multisets)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Multimaps)</td>
<td>swap the contents of this container with another</td>
</tr>
<tr>
<td><code>swap</code> (C++ Algorithms)</td>
<td>swap the values of two objects</td>
</tr>
<tr>
<td><code>swap_ranges</code> (C++ Algorithms)</td>
<td>swaps two ranges of elements</td>
</tr>
<tr>
<td><code>sync_with_stdio</code> (C++ I/O)</td>
<td>synchronize with standard I/O</td>
</tr>
<tr>
<td><code>tellg</code> (C++ I/O)</td>
<td>read input stream pointers</td>
</tr>
<tr>
<td><code>tellp</code> (C++ I/O)</td>
<td>read output stream pointers</td>
</tr>
<tr>
<td><code>test</code> (C++ Bitsets)</td>
<td>returns the value of a given bit</td>
</tr>
<tr>
<td><code>to_string</code> (C++ Bitsets)</td>
<td>string representation of the bitset</td>
</tr>
<tr>
<td><code>to_ulong</code> (C++ Bitsets)</td>
<td>returns an integer representation of the bitset</td>
</tr>
<tr>
<td><code>top</code> (C++ Stacks)</td>
<td>returns the top element of the container</td>
</tr>
<tr>
<td><code>top</code> (C++ Priority Queues)</td>
<td>returns the top element of the container</td>
</tr>
<tr>
<td><code>transform</code> (C++ Algorithms)</td>
<td>applies a function to a range of elements</td>
</tr>
<tr>
<td><code>unique</code> (C++ Lists)</td>
<td>removes consecutive duplicate elements</td>
</tr>
<tr>
<td></td>
<td>remove consecutive duplicate</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>unique</strong> (C++ Algorithms)</td>
<td>elements in a range</td>
</tr>
<tr>
<td><strong>unique_copy</strong> (C++ Algorithms)</td>
<td>create a copy of some range of elements that contains no consecutive duplicates</td>
</tr>
<tr>
<td><strong>unsetf</strong> (C++ I/O)</td>
<td>clear io stream format flags</td>
</tr>
<tr>
<td><strong>upper_bound</strong> (C++ Sets)</td>
<td>returns an iterator to the first element greater than a certain value</td>
</tr>
<tr>
<td><strong>upper_bound</strong> (C++ Multisets)</td>
<td>returns an iterator to the first element greater than a certain value</td>
</tr>
<tr>
<td><strong>upper_bound</strong> (C++ Maps)</td>
<td>returns an iterator to the first element greater than a certain value</td>
</tr>
<tr>
<td><strong>upper_bound</strong> (C++ Multimaps)</td>
<td>returns an iterator to the first element greater than a certain value</td>
</tr>
<tr>
<td><strong>upper_bound</strong> (C++ Algorithms)</td>
<td>searches for the last possible location to insert an element into an ordered range</td>
</tr>
<tr>
<td><strong>value_comp</strong> (C++ Sets)</td>
<td>returns the function that compares values</td>
</tr>
<tr>
<td><strong>value_comp</strong> (C++ Multisets)</td>
<td>returns the function that compares values</td>
</tr>
<tr>
<td><strong>value_comp</strong> (C++ Maps)</td>
<td>returns the function that compares values</td>
</tr>
<tr>
<td><strong>value_comp</strong> (C++ Multimaps)</td>
<td>returns the function that compares values</td>
</tr>
<tr>
<td><strong>width</strong> (C++ I/O)</td>
<td>access and manipulate the minimum field width</td>
</tr>
<tr>
<td><strong>write</strong> (C++ I/O)</td>
<td>write characters</td>
</tr>
</tbody>
</table>
Bitset Constructors

Syntax:

```cpp
#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:

```cpp
// 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;
```
**Bitset Operators**

**Syntax:**

```cpp
#include <bitset>
!=, ==, &=, ^=, |=, ~, <<=, >>=, [x]
```

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:

```cpp
// create a bitset out of a number
bitset<8> bs2( (long) 131 );
cout << "bs2 is " << bs2 << endl;
// shift the bitset to the left by 4 digits
bs2 <<= 4;
cout << "now bs2 is " << bs2 << endl;
```

When the above code is run, it displays:

```plaintext
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++;
```
When run, this code displays the following output:

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

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

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

Syntax:

```cpp
#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:

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

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

```cpp
// 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++;
```
When run, this code displays the following output:

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

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

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

**Syntax:**

```cpp
#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:

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

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

```cpp
// create a vector of random integers
cout << "original vector: ";
vector<int> v;
for( int i = 0; i < 10; i++ ) {
    int num = (int) rand() % 10;
    cout << num << " ";
    v.push_back( num );
}
cout << endl;

// find the first element of v that is even
vector<int>::iterator iter1 = v.begin();
while( iter1 != v.end() && *iter1 % 2 != 0 ) {
    iter1++;
```
// find the last element of v that is even
vector<int>::iterator iter2 = v.end();
do {
    iter2--;
} while( iter2 != v.begin() && *iter2 % 2 != 0 );

cout << "first even number: " << *iter1 << ", last even number: " << *iter2 << endl;

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

When run, this code displays the following output:

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

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

The default destructor is called when the list should be destroyed.
Set constructors & destructors

Syntax:

```cpp
#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 \textit{constant time}. The default copy constructor runs in \textit{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:

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

Related topics:
(C++ Strings) resize
Container constructors & destructors

Syntax:

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

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

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

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

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

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

Related topics:
(C++ Strings) resize
Map Constructors & Destructors

Syntax:

```cpp
#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 \( \text{start} \) and \( \text{end} \), or specify a comparison function \( \text{cmp} \).

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:

```cpp
struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
        return strcmp( s1, s2 ) < 0;
    }
};

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

cout << "Bart is " << ages["Bart"] << " years old" << endl;
```
Related topics:

Map Operators
Multimap constructors & destructors

Syntax:

```cpp
#include <map>
multimap();
multimap( const multimap& c );
multimap( iterator begin, iterator end,
        const key_compare& cmp = Compare(), const allocator& alloc = Allocator() )
~multimap();
```

Multimaps have several constructors:

- The default constructor takes no arguments, creates a new instance of that multimap, and runs in **constant time**.
- The default copy constructor runs in **linear time** and can be used to create a new multimap that is a copy of the given multimap c.
- Multimaps can also be created from a range of elements defined by `begin` and `end`. When using this constructor, an optional comparison function `cmp` and allocator `alloc` can also be provided.

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

The template definition of multimaps requires that both a key type and value type be supplied. For example, you can instantiate a multimap that maps strings to integers with this statement:

```cpp
multimap<string,int> m;
```

You can also supply a comparison function and an allocator in the template:

```cpp
multimap<string,int,myComp,myAlloc> m;
```

For example, the following code uses a multimap to associate a series of employee names with numerical IDs:
```cpp
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 Smitey", employeeID++) );
m.insert( pair<string, int>("Bob Smith", employeeID++) );

    cout << "Number of employees named 'Bob Smith': " << m.count("Bob Smith") << endl;
    cout << "Number of employees named 'Bob Thompson': " << m.count("Bob Thompson") << endl;
    cout << "Number of employees named 'Bob Smitey': " << m.count("Bob Smitey") << endl;
    cout << "Employee list:
    for( multimap<string, int>::iterator iter = m.begin(); iter != m.end(); ++iter )
        cout << " Name: " << iter->first << ", ID #" << iter->second << endl;
```

When run, the above code produces the following output. Note that the employee list is displayed in alphabetical order, because multimaps are sorted associative containers:

```
Number of employees named 'Bob Smith': 2
Number of employees named 'Bob Thompson': 1
Number of employees named 'Bob Smitey': 1
Employee list:
    Name: Bob Smith, ID #0
    Name: Bob Smith, ID #3
    Name: Bob Smitey, ID #2
    Name: Bob Thompson, ID #1
```

**Related topics:**
- `count`
- `insert`
List operators

Syntax:

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

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

Two lists are equal if:

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

Comparisons among lists are done lexicographically.

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

Syntax:

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

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

Two sets are equal if:

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

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

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

Two multisets are equal if:

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

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

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

Two multimaps are equal if:

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

Comparisons among multimaps are done lexicographically.

Related topics:

**Multimap Constructors**
Vector operators

Syntax:

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

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

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

Two vectors are equal if:

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

Comparisons among vectors are done lexicographically.

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

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

Related topics:
at
Container operators

Syntax:

```cpp
#include <deque>

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

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

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

Two `containers` are equal if:

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

Comparisons among dequeues are done lexicographically.

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

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

Related topics:
at
I/O Constructors

Syntax:

```cpp
#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 io stream mode flags. 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.

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

Related topics:

close open
Map operators

Syntax:

```cpp
#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);
```

Maps can be compared and assigned with the standard comparison operators: `==, !=, <=, >=, <, >,` and `=`. Individual elements of a map can be examined with the `[]` operator.

Performing a comparison or assigning one map to another takes linear time.

Two maps are equal if:

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

Comparisons among maps are done lexicographically.

For example, the following code defines a map between strings and integers and loads values into the map using the `[]` operator:

```cpp
struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
        return strcmp( s1, s2 ) < 0;
    }
};

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
```
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

cout << "Bart is " << ages["Bart"] << " years old" << endl;

cout << "In alphabetical order: " << endl;
for( map<const char*, int, strCmp>::iterator iter = ages.begin(); iter != ages.end(); iter++)
    cout << (*iter).first << " is " << (*iter).second << " years old" << endl;

When run, the above code displays this output:

Bart is 11 years old
In alphabetical order:
Bart is 11 years old
Homer is 38 years old
Lisa is 8 years old
Maggie is 1 years old
Marge is 37 years old

Related topics:
insert Map Constructors & Destructors
Priority queue constructors

Syntax:

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

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

**Syntax:**

```cpp
#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:

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

When run, the above code might produce this output:

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

Syntax:

```cpp
#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:**

```cpp
#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,

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

displays

```
cccccc
Now is the time...
time
```

The string constructors usually run in **linear time**, except the empty constructor,
which runs in \textit{constant time}. 
String operators

Syntax:

```cpp
#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 linear time.

Two strings are equal if:

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

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:
Furthermore, 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 constant time.

*Related topics:*
c_str  
compare 
data
accumulate

Syntax:

```cpp
#include <numeric>

TYPE accumulate( iterator start, iterator end, TYPE val );
TYPE accumulate( iterator start, iterator end, TYPE val, BinaryFunction f );
```

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:

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

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

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

`adjacent_difference` runs in linear time.

Related topics:
- `accumulate`
- `count`
- `inner_product`
- `partial_sum`
adjacent_find

Syntax:

```cpp
#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:

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

Related topics:
find find_end
find_first_of
find_if
unique
unique_copy
any

Syntax:

```cpp
#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
append

Syntax:

```cpp
#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` repetitions 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:

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

That code displays:

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

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

```cpp
string str1 = "Eventually I stopped caring..."
string str2 = "but that was the '80s so nobody noticed."
```
When run, the above code displays:

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

Syntax:

```cpp
#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:

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

The above code displays the following output:

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

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

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
}
```
cout << endl;

When run, the above code displays the following output:

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

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

Syntax:

```cpp
#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:

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

The above code displays the following output:

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

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

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

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

When run, the above code displays the following output:

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

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

**Syntax:**

```cpp
#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:

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

The above code displays the following output:

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

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

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

When run, the above code displays the following output:

0 1 2 3 4 5 6 7 8 9

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

Syntax:

```cpp
#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:

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

displays

```
War and Peace
```

This function will destroy the previous contents of the string.

Related topics:
(C++ Lists) assign
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:

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

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

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

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

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

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

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

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

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

Syntax:

```cpp
#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:

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

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

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

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

Related topics:
(C++ Multimaps) Multimap operators
(C++ Double-ended Queues) Container operators
auto_ptr

Syntax:

```cpp
#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:

```cpp
#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;
}
back

Syntax:

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

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

For example:

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

This code produces the following output:

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

The `back()` function runs in \textit{constant time}.

\textit{Related topics}: 

\texttt{front} \hspace{1em} \texttt{pop\_back}
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:

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

This code produces the following output:

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

The back() function runs in constant time.

Related topics:
front pop_back
back

Syntax:

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

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

For example:

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

This code produces the following output:

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

The `back()` function runs in constant time.

Related topics:

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

For example:

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

This code produces the following output:

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

The back() function runs in constant time.

*Related topics:* [front](C++ Lists) [pop_back](C++ Lists)
bad

Syntax:

```cpp
#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
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;
}
```

Related topics:
end rbegin rend
Syntax:

```cpp
#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:

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

while( cin >> str ) words.push_back(str);

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

When given this input:

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

...the above code produces the following output:

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

Related topics:

- `[]` operator
- `at`
begin

Syntax:

```cpp
#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:

```cpp
// 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;
}
```

Related topics:
- end
- rbegin
- rend
begin

Syntax:

```cpp
#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:

```cpp
// 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;
}
```

Related topics:
- **end rbegin**
- **rend**
begin

Syntax:

```cpp
#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:

```cpp
// 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;
}
```

Related topics:
end rbegin
rend
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:

```cpp
// 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;
}
```

Related topics:
- `end`
- `rbegin`
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:

```cpp
map<string,int> stringCounts;
string str;

while( cin >> str ) stringCounts[str]++;

map<string,int>::iterator iter;
for( iter = stringCounts.begin(); iter != stringCounts.end(); iter++ )
    cout << "word: " << iter->first << " count: " << iter->second << endl;
```

When given this input:

```
here are some words and here are some more words
```

...the above code generates this output:

```
word: and, count: 1
word: are, count: 2
word: here, count: 2
word: more, count: 1
word: some, count: 2
word: words, count: 2
```

Related topics:

end
rbegin
rend


**begin**

**Syntax:**

```cpp
#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:

```cpp
// 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;
}
```

**Related topics:**

`end`  
`rbegin`

`rend`
**binary_search**

**Syntax:**

```cpp
#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:

```cpp
int nums[] = { -242, -1, 0, 5, 8, 9, 11 };
int start = 0;
int end = 7;

for( int i = 0; i < 10; i++ ) {
    if( binary_search( nums+start, nums+end, i ) ) {
        cout << "nums[] contains " << i << endl;
    } else {
        cout << "nums[] DOES NOT contain " << i << endl;
    }
}
```

When run, this code displays the following output:

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

*Related topics:*

- [equal_range](#)
- [is_sorted](#)
- [lower_bound](#)
- [partial_sort](#)
- [partial_sort_copy](#)
- [sort](#)
- [stable_sort](#)
- [upper_bound](#)
c_str

Syntax:

```cpp
#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 `const`, the character data that c_str() returns **cannot be modified**. Furthermore, you do not need to call `free()` or `delete` on this pointer.

*Related topics:*

**String operators data**
capacity

Syntax:

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

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

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

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

When run, the above code produces the following output:

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

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

The `capacity()` function runs in **constant time**.

Related topics:
reserve resize
size
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;
```

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:

```cpp
#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:

```cpp
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;
```

Related topics:
- eof
- fail
- good
clear

Syntax:

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

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

Related topics:
(C++ Lists) erase
clear

Syntax:

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

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

clear() runs in linear time.

Related topics:
erase
clear

Syntax:

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

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

Related topics:
erase
clear

Syntax:

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

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

Related topics:
erase
**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](#)
clear

Syntax:

```cpp
#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
clear

Syntax:

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

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

Related topics:
erase
clear

Syntax:

```cpp
#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
close

Syntax:

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

The `close()` function closes the associated file stream.

*Related topics:*

- I/O Constructors open
compare

Syntax:

```cpp
#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,
            size_type length2 );
```

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than zero</td>
<td>this &lt; str</td>
</tr>
<tr>
<td>zero</td>
<td>this == str</td>
</tr>
<tr>
<td>greater than zero</td>
<td>this &gt; str</td>
</tr>
</tbody>
</table>

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 each other:

```cpp
string names[] = {"Homer", "Marge", "3-eyed fish", "inanimate carbon"};
```
```c++
for( int i = 0; i < 4; i++ ) {
    for( int j = 0; j < 4; j++ ) {
        cout << names[i].compare( names[j] ) << " ";
    }
    cout << endl;
}
```

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

<table>
<thead>
<tr>
<th></th>
<th>Homer</th>
<th>Marge</th>
<th>3-eyed fish</th>
<th>inanimate carbon rod</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Homer&quot;.compare(x)</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>&quot;Marge&quot;.compare(x)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>&quot;3-eyed fish&quot;.compare(x)</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>&quot;inanimate carbon rod&quot;.compare(x)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Related topics:

String operators
copy

Syntax:

```cpp
#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:

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

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`
copy

Syntax:

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

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

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

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

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

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

Related topics:
- copy_backward
- copy_n
- generate
**copy_backward**

**Syntax:**

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

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

```plaintext
*(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:

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

vector<int> to_vector(15);

copy_backward( from_vector.begin(), from_vector.end(), to_vector.end() );

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

The above code produces the following output:

```plaintext
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:**

```cpp
#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:

```cpp
#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.
**count**

*Syntax:*

```cpp
#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*. 
count

Syntax:

```cpp
#include <map>

size_type count( const key_type& key );
```

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

Count() should run in logarithmic time.
count

Syntax:

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

The function `count()` returns the number of occurrences of `key` in the multimap. `count()` should run in **logarithmic time**.
count

Syntax:

```cpp
#include <bitset>
size_type count();
```

The function `count()` returns the number of bits that are set to 1 in the bitset.

Related topics:

any
count

Syntax:

```cpp
#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:

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

int target_value = 3;
int num_items = count( v.begin(), v.end(), target_value );

cout << "v contains " << num_items << " items matching " << target_value << endl;
```

The above code displays the following output:

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

Related topics:
- accumulate
- adjacent_difference
- count_if
- inner_product
- partial_sum
count_if

Syntax:

```cpp
#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:

```cpp
int nums[] = { 0, 1, 2, 3, 4, 5, 9, 3, 13 };
int start = 0;
int end = 9;

int target_value = 3;
int num_items = count_if( nums+start, nums+end, bind2nd(equal_to<int>(), target_value) );

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

When run, the above code displays the following output:

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

Related topics:

`count`
data

Syntax:

```cpp
#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(""潟");
string s3("This is a string");
cout.setf(ios::boolalpha);
cout << s1.empty() << endl;
cout << s2.empty() << endl;
cout << s3.empty() << endl;
```

When run, this code produces the following output:

```
true
true
false
```

Related topics:

size
empty

Syntax:

```cpp
#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:

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

Related topics:

size
empty

Syntax:

```cpp
#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:

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

Related topics:

size
empty

Syntax:

```cpp
#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:

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

Related topics:

`size`
empty

Syntax:

```cpp
#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:

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

Related topics: size
empty

Syntax:

```cpp
#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:

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

Related topics:

`size`
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;
age["Marge"] = 37;
age["Lisa"] = 8;
age["Maggie"] = 1;
age["Bart"] = 11;
while( !ages.empty() ) {
    cout << "Erasing: " << (*ages.begin()).first << ", " << (*ages.begin()).second << endl;
    ages.erase( ages.begin() );
}
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
Erasing: Marge, 37
```

Related topics:
begin erase
size
empty

Syntax:

```cpp
#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:

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

Related topics:

`size`
empty

Syntax:

```cpp
#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:

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

Related topics:

- size
empty

Syntax:

```cpp
#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:

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

Related topics:

- size
empty

Syntax:

```cpp
#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:

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

Related topics:
size
Syntax:

```cpp
#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 `begin()` and `end()` to iterate through all of the members of a vector:

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

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

`end()` runs in constant time.

Related topics:

`begin` `rbegin` `rend`
end

Syntax:

```cpp
#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:

```cpp
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );

int bad_val = *(v1.end());
cout << "bad_val is " << bad_val << endl;

int good_val = *(v1.end() - 1);
cout << "good_val is " << good_val << endl;
```

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

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

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

end() runs in constant time.

Related topics:
begin rbegin rend
end

Syntax:

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

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

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

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

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

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

end() runs in constant time.

Related topics:

begin rbegin
rend
The `end()` function returns an iterator just past the end of the list.

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

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

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

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

`end()` runs in **constant time**.

**Related topics:**
- `begin`
- `rbegin`
- `rend`
**end**

*Syntax:*

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

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

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

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

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

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

`end()` runs in *constant time*.

*Related topics:*

- `begin`
- `rbegin`
- `rend`
The end() function returns an iterator just past the end of the multiset.

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

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

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

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

end() runs in constant time.

Related topics:
begin rbegin
rend
end

Syntax:

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

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

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

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

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

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

`end()` runs in **constant time**.

*Related topics:*

- `begin`
- `rbegin`
- `rend`
end

Syntax:

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

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

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

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

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

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

end() runs in **constant time**.

*Related topics:*

begin rbeginn
rend
**eof**

**Syntax:**

```cpp
#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:

```cpp
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:

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

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

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

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

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

Related topics:
- `find_if`
- `lexicographical_compare`
- `mismatch`
- `search`
**equal_range**

*Syntax:*

```cpp
#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`. 
equal_range

Syntax:

```cpp
#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`. 
equal_range

Syntax:

```cpp
#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:

```cpp
#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()`:

```cpp
multimap<string, pair<int, int>> input_config;

// read configuration from file "input.conf" to input_config
readConfigFile( input_config, "input.conf" );

pair<multimap<string, pair<int, int>>::iterator, multimap<string, pair<int, int>>::iterator> ii;
multimap<string, pair<int, int>>::iterator i;

ii = input_config.equal_range("key"); // keyboard key-bindings
// we can iterate over a range just like with begin() and end()
for( i = ii.first; i != ii.second; ++i ) {
    // add a key binding with this key and output
    bindkey(i->second.first, i->second.second);
}

ii = input_config.equal_range("joyb"); // joystick button key-bindings
for( i = ii.first; i != ii.second; ++i ) {
    // add a key binding with this joystick button and output
    bindjoyb(i->second.first, i->second.second);
}
```
equal_range

Syntax:

```cpp
#include <algorithm>
pair<iterator, iterator> equal_range(iterator first, iterator last, const pair<iterator, iterator>& equal_range(iterator first, iterator last, const pair<iterator, iterator>&

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

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

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

```cpp
vector<int> nums;
nums.push_back(-242);
nums.push_back(-1);
nums.push_back(0);
nums.push_back(5);
nums.push_back(8);
nums.push_back(8);
nums.push_back(11);

pair<vector<int>::iterator, vector<int>::iterator> result;
int new_val = 8;

result = equal_range(nums.begin(), nums.end(), new_val);

cout << "The first place that " << new_val << " could be inserted is before " << *result.first << ", and the last place that it could be inserted is before " << *result.second << endl;
```

The above code produces the following output:
The first place that 8 could be inserted is before 8, and the last place that it could be inserted is before 11

*Related topics:*

- [binary_search](#)
- [lower_bound](#)
- [upper_bound](#)
# erase

## Syntax:

```cpp
#include <string>

iterator erase( iterator loc );
named_iterator erase( named_iterator start, named_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:

```cpp
string s("So, you like donuts, eh? Well, have all the donuts in the world!");

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;
```

will display

```
The original string is 'So, you like donuts, eh? Well, have all the donuts in the world!'
Now the string is 'So, you like donuts, eh? Well, have all the donuts in the world!'
Now the string is 'So, you like donuts, eh? Well, have all the donuts in the world!'
Now the string is 'So, you like donuts, eh? Well, have all the donuts in the world!'
```
erase() runs in linear time.

Related topics:
insert
erase

Syntax:

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

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

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

For example:

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

That code would display the following output:

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

```cpp
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
    cout << alphaVector[i];
}
cout << endl;

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

When run, the above code displays:

```
ABCDEF
ABH
```

*Related topics:*
- `clear`
- `insert`
- `pop_back`
- (C++ Lists) `pop_front`
- (C++ Lists) `remove`
- (C++ Lists) `remove_if`
erase

Syntax:

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

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

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

For example:

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

That code would display the following output:

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

```cpp
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
    cout << alphaVector[i];
}
cout << endl;

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

When run, the above code displays:

```
ABCD
AB
```

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

Syntax:

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

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

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

For example:

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

That code would display the following output:

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

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

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

Related topics:
- clear
- insert
- pop_back
- pop_front
- remove
- remove_if
**erase**

*Syntax:*

```cpp
#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*. 
**erase**

Syntax:

```cpp
#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`. 
erase

Syntax:

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

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

For example, the following code uses `erase()` in a `while` loop to incrementally clear a map and display its contents in order:

```c++
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;
age["Marge"] = 37;
age["Lisa"] = 8;
age["Maggie"] = 1;
age["Bart"] = 11;

while( !ages.empty() ) {
    cout << "Erasing: " << (*ages.begin()).first << ", " << (*ages.begin()).second << endl;
    ages.erase( ages.begin() );
}
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
```
Erasing: Marge, 37

Related topics:
begin clear
empty
size
erase

Syntax:

```cpp
#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`. 
fail

Syntax:

```cpp
#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:

```cpp
#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**
fill

Syntax:

```cpp
#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:

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

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

cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
    cout << v1[i] << " ";
}
cout << endl;
```

When run, the above code displays:

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

Related topics:
- `fill_n`
- `generate`
- `transform`
fill\_n

Syntax:

```
#include <algorithm>
#include <algorithm>
iterator fill\_n( iterator start, size\_t n, const TYPE& val );
```

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

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

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

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

When run, this code displays:

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

Related topics:
find

Syntax:

```cpp
#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:

```cpp
int num_to_find = 3;
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back(i);
}
vector<int>::iterator result;
result = find( v1.begin(), v1.end(), num_to_find );

if( result == v1.end() ) {
    cout << "Did not find any element matching " << num_to_find << endl;
} else {
    cout << "Found a matching element: " << *result << endl;
}
```

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

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

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

Related topics:
adjacent_find find_end
find_first_of
find_if
mismatch
search
**find**

*Syntax:*

```cpp
#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**.
find

Syntax:

```cpp
#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**.
find

Syntax:

```cpp
#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:

```cpp
map<string,int> stringCounts;
string str;

while( cin >> str ) stringCounts[str]++;

map<string,int>::iterator iter = stringCounts.find("spoon");
if( iter != stringCounts.end() ) {
    cout << "You typed '" << iter->first << "' " << iter->second << " time(s)" << endl;
}
```

When run with this input:

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

...the above code produces this output:

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

Syntax:

```cpp
#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**.
find

Syntax:

```cpp
#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 length );
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:

```cpp
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;
}
```

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_start, iterator seq_end);
iterator find_end(iterator start, iterator end, iterator seq_start, iterator seq_end, BinPred bp);
```

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

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

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

```cpp
#include <algorithm>

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 + 3 );
if( *result == nums[end] ){
    cout << "Did not find any subsequence matching { 1, 2, 3 }" " endl;
} else {
    cout << "The last matching subsequence is at: " << *result " endl;
}

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

`adjacent_find` `find`
`find_first_of`
`find_if`
`search_n`
```cpp
#include <string>

size_type find_first_not_of( const string& str, size_type index = 0 );
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, size_type size_type
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 hyphen:

```cpp
string lower_case = "abcdefghijklmnopqrstuvwxyz,\-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-CASE part";
cout << "first non-lower-case letter in str at: " << str.find_first_not_of(lower_case);
```

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:

```cpp
#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_type num);
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:

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

The `find_first_of()` function searches for the first occurrence 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:

```cpp
int nums[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int* result;
int start = 0;
int end = 10;

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

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

**Syntax:**

```cpp
#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:

```cpp
int nums[] = { 0, -1, -2, -3, -4, 342, -5 };
int* result;
int start = 0;
int end = 7;

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

**Related topics:**

- adjacent_find
- equal
- find
- find_end
- find_first_of
- search_n
find_last_not_of

Syntax:

```cpp
#include <string>
size_type find_last_not_of( const string& str, size_type index = npos );
size_type find_last_not_of( const char* str, size_type index = npos );
size_type find_last_not_of( const char* str, size_type index, size_type num );
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:

```cpp
string lower_case = "abcdefghijklmnopqrstuvwxyz";
string str = "abcdefgABCDEFGhijklmnop";
cout << "last non-lower-case letter in str at: " << str.find_last_not_of(lower_case) << endl;
```

This code displays the following output:

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

Related topics:
- `find_first_not_of`
- `find_first_of`
- `find_last_of`
rfind
find_last_of

Syntax:

```cpp
#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_type size );
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
flags

Syntax:

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

The flags() function either returns the io stream format flags for the current stream, or sets the flags for the current stream to be \( f \).

Related topics:

setf unsetf
flip

Syntax:

```cpp
#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.
flush

Syntax:

```cpp
#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:

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

The for_each() algorithm applies the function \( f \) to each of the elements between \( \text{start} \) and \( \text{end} \). 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:

```cpp
template<class TYPE>
struct increment : public unary_function<TYPE, void>
{
    void operator()( TYPE& x ) {
        x++;
    }
};
...
int nums[] = {3, 4, 2, 9, 15, 267};
const int N = 6;

cout << "Before, nums[] is: ";
for( int i = 0; i < N; i++ ) {
    cout << nums[i] << " ";
}
cout << endl;
for_each( nums, nums+N, increment<int>() );

cout << "After, nums[] is: ";
for( int i = 0; i < N; i++ ) {
    cout << nums[i] << " ";
}
cout << endl;
```

The above code displays the following output:

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

---

CPPReference page: [C++ Algorithms > for_each](http://cppreference.com)
After, nums[] is: 4 5 3 10 16 268
front

Syntax:

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

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

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

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

while( cin >> str ) words.push_back(str);

sort( words.begin(), words.end() );

cout << "In alphabetical order, the first word is '" << words.front() << "]" << endl;
```

When provided with this input:

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

...the above code displays:

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

\textit{Related topics:}
back (C++ Lists) \texttt{pop_front}
(C++ Lists) \texttt{push_front}
front

Syntax:

```cpp
#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
front

Syntax:

```cpp
#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
front

Syntax:

```cpp
#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
gcount

Syntax:

```cpp
#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
generate

Syntax:

```cpp
#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:

```cpp
#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`
get

Syntax:

```cpp
#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:

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

Related topics:
- `gcount` getline
- (C++ Strings) getline
- ignore
- peek
put
read
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:

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

ifstream fin("tmp.dat");
int MAX_LENGTH = 100;
char line[MAX_LENGTH];

while( fin.getline(line, MAX_LENGTH) ) {
  cout << "read line: " << line << endl;
}
```

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

Those using a Microsoft compiler may find that `getline()` reads an extra character, and should consult the documentation on the [Microsoft getline bug](https://support.microsoft.com/en-us/kb/292343).

**Related topics:**
- `gcount get` (C++ Strings)
- `getline`
ignore
read
getline

Syntax:

```cpp
#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:

```cpp
string s;
getline( cin, s );
cout << "You entered " << s << endl;
```

After getting a line of data in a string, you may find that string streams 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:

```cpp
// expects either space-delimited numbers or lines that start with double 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;
        }
    }
}
```
When run with a user supplying input, the above code might produce this output:

```plaintext
// 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) get
(C++ I/O) getline
string streams
good

Syntax:

```cpp
#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
ignore

Syntax:

```cpp
#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`
includes

Syntax:

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

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

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

`includes()` runs in linear time.

Related topics:

- `set_difference`
- `set_intersection`
- `set_symmetric_difference`
- `set_union`
inner_product

Syntax:

```cpp
#include <numeric>

TYPE inner_product( iterator start1, iterator end1, iterator start2,
                    TYPE inner_product( iterator start1, iterator end1, iterator start2,
```}

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

`inner_product()` runs in **linear time**.

**Related topics:**

- `accumulate`
- `adjacent_difference`
- `count`
- `partial_sum`
inplace_merge

Syntax:

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

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

Related topics:
merge
insert

Syntax:

```cpp
#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 index2,
                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`
insert

Syntax:

```
#include <vector>

iterator insert( iterator loc, const TYPE& val );
void insert( iterator loc, size_type num, const TYPE& val );
void insert( iterator loc, input_iterator start, input_iterator end );
```

The `insert()` function either:

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

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

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

```
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end(); cout << *theIterator;
}
This code would display:

```
CCCCABCDEFGHJ
```

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:

```cpp
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );

vector<int> v2;
v2.push_back( 5 );
v2.push_back( 6 );
v2.push_back( 7 );
v2.push_back( 8 );

cout << "Before, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
}
cout << endl;

v2.insert( v2.end(), v1.begin(), v1.end() );

cout << "After, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
}
cout << endl;
```

When run, this code displays:

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

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

Syntax:

```cpp
#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:

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

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end();
    cout << *theIterator;
}
```

This code would display:

```
CCCCABCDDEFGHIJ
```

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

```cpp
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end();
    cout << *theIterator;
}
```

This code would display:

```
CCCABCDEFGHIJ
```

Related topics:
- assign
- erase
- merge
- push_back
push_front
splice
insert

Syntax:

```cpp
#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
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`. 
insert

Syntax:

```cpp
#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:

```cpp
multimap<string,int> m;
int employeeID = 0;
m.insert( pair<string,int>("Bob Smith",employeeID++ ) );
m.insert( pair<string,int>("Bob Thompson",employeeID++ ) );
m.insert( pair<string,int>("Bob Smithey",employeeID++ ) );
m.insert( pair<string,int>("Bob Smith",employeeID++ ) );

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

When run, the above code produces the following output:

```
Number of employees named 'Bob Smith': 2
```

Number of employees named 'Bob Thompson': 1
Number of employees named 'Bob Smithey': 1
Employee list:
  Name: Bob Smith, ID #0
  Name: Bob Smith, ID #3
  Name: Bob Smithey, ID #2
  Name: Bob Thompson, ID #1
insert

Syntax:

```
#include <map>

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

The function insert() either:

- inserts pair after the element at pos (where pos is really just a suggestion as to where pair should go, since sets and maps are ordered), and returns an iterator to that element.
- inserts a range of elements from start to end.
- inserts pair<key,val>, but only if no element with key key already exists. The return value is an iterator to the element inserted (or an existing pair with key key), and a boolean which is true if an insertion took place.

For example, the following code uses the insert() function (along with the make_pair() function) to insert some data into a map and then displays that data:

```
map<string,int> theMap;
theMap.insert( make_pair( "Key 1", -1 ) );
theMap.insert( make_pair( "Another key!", 32 ) );
theMap.insert( make_pair( "Key the Three", 66667 ));

map<string,int>::iterator iter;
for( iter = theMap.begin(); iter != theMap.end(); ++iter ) {
    cout << "Key: " << iter->first << " Value: " << iter->second << endl;
}
```

When run, the above code displays this output:

```
Key: 'Another key!', Value: 32
Key: 'Key 1', Value: -1
Key: 'Key the Three', Value: 66667
```

Note that because maps are sorted containers, the output is sorted by the key
value. In this case, since the map key data type is `string`, the map is sorted alphabetically by key.

*Related topics:*

**Map operators**
is_heap

Syntax:

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

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

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

`is_heap()` runs in linear time.

Related topics:
- `make_heap`
- `pop_heap`
- `push_heap`
- `sort_heap`
**is_sorted**

**Syntax:**

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

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

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

`is_sorted()` runs in linear time.

**Related topics:**
- `binary_search`
- `partial_sort`
- `partial_sort_copy`
- `sort`
- `stable_sort`
iter_swap

Syntax:

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

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

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

would.

Related topics:

`swap swap_ranges`
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
key_comp

Syntax:

```cpp
#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
key_comp

Syntax:

```cpp
#include <map>
key_compare key_comp() const;
```

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

key_comp() runs in constant time.

Related topics:

value_comp
key_comp

Syntax:

```cpp
#include <map>
key_compare key_comp() const;
```

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

`key_comp()` runs in **constant time**.

*Related topics:*

value_comp
length

Syntax:

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

The `length()` function returns the number of elements in the current string, performing the same role as the `size()` function.

*Related topics:*  
`size`
lexicographical_compare

Syntax:

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

The lexicographical_compare() function returns true if the range of elements 
`[start1,end1)` is lexicographically less than the range of elements `[start2,end2)`. If you're confused about what lexicographic means, it might help to know that dictionaries are ordered lexicographically.

`lexicographical_compare()` runs in linear time.

Related topics:
- `equal`
- `lexicographical_compare_3way`
- `mismatch`
- `search`
lexicographical_compare_3way

Syntax:

```cpp
#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:

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

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

`lower_bound()` runs in logarithmic time.

Related topics:
upper_bound
lower_bound

Syntax:

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

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

lower_bound() runs in logarithmic time.

Related topics:
upper_bound
lower_bound

Syntax:

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

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

`lower_bound()` runs in logarithmic time.

Related topics:

upper_bound
lower_bound

Syntax:

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

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

`lower_bound()` runs in logarithmic time.

Related topics:

`upper_bound`
lower_bound

Syntax:

```cpp
#include <algorithm>

iterator lower_bound( iterator first, iterator last, const TYPE& val );
iterator lower_bound( iterator first, iterator last, const TYPE& val, CompFn f );
```

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

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

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

```cpp
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );

cout << "Before nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
    cout << nums[i] << " ";
}
cout << endl;

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

The above code produces the following output:

<table>
<thead>
<tr>
<th>Before nums is:</th>
<th>-242 -1 0 5 8 8 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>After, nums is:</td>
<td>-242 -1 0 5 7 8 8 11</td>
</tr>
</tbody>
</table>

Related topics:

*binary_search*  *equal_range*
make_heap

Syntax:

```cpp
#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 **linear time**.

Related topics:
- `is_heap`
- `pop_heap`
- `push_heap`
- `sort_heap`
**max**

**Syntax:**

```cpp
#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:

```cpp
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:

```cpp
#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:

```cpp
int array[] = { 3, 1, 4, 1, 5, 9 };
unsigned int array_size = 6;
cout << "Max element in array is " << *max_element( array, array+array_size ) << endl;

vector<char> v;
v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back('d');
cout << "Max element in the vector v is " << *max_element( v.begin(), v.end() ) << endl;
```

When run, the above code displays this output:

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

**Related topics:**

- max
- min
- min_element
max_size

Syntax:

```cpp
#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:

- `size`
max_size

Syntax:

```cpp
#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:
- size
max_size

Syntax:

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

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

Related topics:

- `size`
max_size

Syntax:

```cpp
#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:

size
max_size

Syntax:

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

Related topics:
- `size`
max_size

Syntax:

```cpp
#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:

`size`
max_size

Syntax:

```cpp
#include <map>
size_type max_size() const;
```

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

Related topics:

size
max_size

Syntax:

```cpp
#include <map>
size_type max_size() const;
```

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

Related topics:

`size`
merge

Syntax:

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

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

`merge()` runs in linear time.

Related topics:
- Container operators
- insert
- splice
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:

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

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

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

Related topics:  
- [max](#)  
- [max_element](#)  
- [min_element](#)
min_element

Syntax:

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

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

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

Related topics:
- max max_element
- min
mismatch

Syntax:

```cpp
#include <algorithm>

pair<iterator1,iterator2> mismatch(iterator start1, iterator end1);

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

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

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

The `mismatch()` algorithm runs in **linear time**.

**Related topics:**
- [equal](#)
- [find](#)
- [lexicographical_compare](#)
- [search](#)
next_permutation

Syntax:

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

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

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

*Related topics:*

- `prev_permutation`
- `random_sample`
- `random_sample_n`
- `random_shuffle`
none

Syntax:

```cpp
#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, StrictWeakOrdering cmp );
```

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

nth_element() runs in linear time on average.

Related topics:
- partial_sort
open

Syntax:

```cpp
#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 `io stream mode flag 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:

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

Related topics:
[I/O Constructors close](#)
### partial_sort

**Syntax:**

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

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

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

**Related topics:**

- `binary_search`
- `is_sorted`
- `nth_element`
- `partial_sort_copy`
- `sort`
- `stable_sort`
partial_sort_copy

Syntax:

```cpp
#include <algorithm>

iterator partial_sort_copy( iterator start, iterator end, iterator result_start, iterator result_end );

iterator partial_sort_copy( iterator start, iterator end, iterator result_start, iterator result_end, StrictWeakOrdering cmp );
```

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

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

Related topics:
- `binary_search`
- `is_sorted`
- `partial_sort`
- `sort`
- `stable_sort`
partial_sum

Syntax:

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

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

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

`partial_sum()` runs in linear time.

Related topics: accumulate adjacent_difference count inner_product
partition

Syntax:

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

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

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

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

`partition()` runs in linear time.

Related topics:

`stable_partition`
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:

gaet putback
pop

Syntax:

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

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

*Related topics:*

push top
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
pop

Syntax:

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

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

Related topics:
push top
pop_back

Syntax:

```cpp
#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
**pop_back**

*Syntax:*

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

The `pop_back()` function removes the last element of the deque.

`pop_back()` runs in **constant time**.

*Related topics:*

- back
- erase
- pop_front
- push_back
pop_back

Syntax:

```cpp
#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
pop_front

Syntax:

```cpp
#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`
pop_front

Syntax:

```cpp
#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
pop_heap

Syntax:

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

The `pop_heap()` function removes the largest 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`
precision

Syntax:

```cpp
#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:

```cpp
float num = 314.15926535;
cout.precision( 5 );
cout << num;
```

This code displays the following output:

```
314.16
```

Related topics:

- **fill width**
prev_permutation

Syntax:

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

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

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

Related topics:
- next_permutation
- random_sample
- random_sample_n
- random_shuffle
push

Syntax:

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

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

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

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

Related topics:

pop
push

Syntax:

```cpp
#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:

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

Related topics:

pop
push

Syntax:

```cpp
#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:

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

Syntax:

```cpp
#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:

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

When displayed, the resulting vector would look like this:

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

`push_back()` runs in **constant time**.

*Related topics:*  
assign insert  
**pop_back**  
(C++ Lists) **push_front**
The `push_back()` function appends `val` to the end of the deque.

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

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

When displayed, the resulting list would look like this:

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

`push_back()` runs in **constant time**.

**Related topics:**
- assign
- insert
- pop_back
- push_front
push_back

Syntax:

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

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

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

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

When displayed, the resulting list would look like this:

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

`push_back()` runs in **constant time**.

Related topics:
- `assign`
- `insert`
- `pop_back`
- `push_front`
push_back

Syntax:

```cpp
#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:

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

When displayed, the resulting string would look like this:

```
abcdefghij
```

`push_back()` runs in `constant time`.

`Related topics:`
- assign
- insert
push_front

Syntax:

```cpp
#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
push_front

Syntax:

```cpp
#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
**push_heap**

**Syntax:**

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

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

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

`push_heap()` runs in **logarithmic time**.

*Related topics:*

- is_heap
- make_heap
- pop_heap
- sort_heap
put

**Syntax:**

```cpp
#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](#)
putback

Syntax:

```cpp
#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:

```cpp
#include <algorithm>

iterator random_sample( iterator start1, iterator end1, iterator start2, iterator end2 );

iterator random_sample( iterator start1, iterator end1, iterator start2, iterator end2, RandomNumberGenerator& rnd );
```

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

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

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

`random_sample()` runs in linear time.

Related topics:
- `next_permutation`
- `prev_permutation`
- `random_sample_n`
- `random_shuffle`
random_sample_n

Syntax:

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

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

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

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

random_sample_n() runs in linear time.

**Related topics:**
- next_permutation
- prev_permutation
- random_sample
- random_shuffle
random_shuffle

Syntax:

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

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

Related topics:
- next_permutation
- prev_permutation
- random_sample
- random_sample_n
rbegin

Syntax:

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

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

rbegin() runs in `constant time`.

Related topics:
- `begin`
- `end`
- `rend`
**rbegin**

**Syntax:**

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

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

`rbegin()` runs in **constant time**.

**Related topics:**

- `begin`
- `end`
- `rend`
rbegin

Syntax:

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

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

rbegin() runs in constant time.

Related topics:
begin end rend
rbegin

Syntax:

```cpp
#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
### rbegin

#### Syntax:

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

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

`rbegin()` runs in **constant time**.

**Related topics:**
- `begin`
- `end`
- `rend`
rbegin

Syntax:

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

The rbegin() function returns a `reverse_iterator` to the end of the current multiset. rbegin() runs in `constant time`.

Related topics:
begin end rend
rbegin

Syntax:

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

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

rbegin() runs in `constant time`.

Related topics:
- `begin`
- `end`
- `rend`
rbegin

Syntax:

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

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

`rbegin()` runs in constant time.

Related topics:
- `begin`
- `end`
- `rend`
rdstate

Syntax:

```cpp
#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:

```cpp
#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:

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

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

Related topics:
- `gcount`
- `get`
- `getline`
- `write`
remove

Syntax:

```cpp
#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:

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

Remove runs in linear time.

Related topics:
- `erase`
- `remove_if`
- `unique`
remove

Syntax:

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

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

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

The remove() function runs in linear time.

Related topics:

- remove_copy
- remove_copy_if
- remove_if
- unique
- unique_copy
The remove_copy() algorithm copies the range \([start, end)\) to result but omits any elements that are equal to \(val\).

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

Related topics:
- copy
- remove
- remove_copy_if
- remove_if
remove_copy_if

Syntax:

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

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

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

`remove_copy_if()` runs in linear time.

Related topics:
- remove
- remove_copy
- remove_if
remove_if

Syntax:

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

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

`remove_if()` runs in linear time.

Related topics:
- `erase`
- `remove`
- `unique`
**remove_if**

*Syntax:*

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

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

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

`remove_if()` runs in *linear time*.

*Related topics:*

- [remove](#)
- [remove_copy](#)
- [remove_copy_if](#)
rend

Syntax:

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

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

rend() runs in `constant time`.

Related topics:
`begin` `end` `rbegin`
rend

Syntax:

```c++
#include <string>
 reverse_iterator rend();
 const_reverse_iterator rend() const;
```

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

rend() runs in constant time.

Related topics:

begin end
rbegin
**rend**

*Syntax:*

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

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

`rend()` runs in *constant time*.

*Related topics:*

*begin* | *end* | *rbegin*
**rend**

**Syntax:**

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

The function `rend()` returns a `reverse_iterator` to the beginning of the current list. `rend()` runs in **constant time**.

**Related topics:**
- `begin`
- `end`
- `rbegin`
rend

Syntax:

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

The function `rend()` returns a `reverse_iterator` to the beginning of the current set. `rend()` runs in `constant time`.

Related topics:

- `begin`
- `end`
- `rbegin`
rend

Syntax:

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

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

rend() runs in constant time.

Related topics:
- begin
- end
- rbegin
rend

Syntax:

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

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

`rend()` runs in constant time.

Related topics: 
`begin` `end` `rbegin`
**rend**

*Syntax:*

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

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

`rend()` runs in **constant time**.

*Related topics:*

- `begin`
- `end`
- `rbegin`
replace

Syntax:

```cpp
#include <string>
string& replace( size_type index, size_type num, const string& str );
string& replace( size_type index1, size_type num1, 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, char ch );
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, size_type num, char ch );
```

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."

```cpp
string s = "They say he carved it himself...from a BIGGER spoon";
string s2 = "find your soul-mate, Homer.";
```
```cpp
s.replace( 32, s2.length(), s2 );
cout << s << endl;
```

Related topics:

* insert
replace

Syntax:

```cpp
#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:

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

The replace_copy() function copies the elements in the range \([\text{start}, \text{end})\) to the destination \(\text{result}\). Any elements in the range that are equal to \(\text{old\_value}\) are replaced with \(\text{new\_value}\).

Related topics:
replace

replace
replace_copy_if

Syntax:

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

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

Related topics:
replace
replace_if

Syntax:

```cpp
#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 \textit{linear time}.

Related topics:

\texttt{replace}
reservereserve

Syntax:

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

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

reserve() runs in linear time.

Related topics:
capacity
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 linear time.

*Related topics:* capacity
reset

Syntax:

```cpp
#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.
resize

Syntax:

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

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

This function runs in linear time.

Related topics:
- Vector constructors & destructors
- capacity
- size
**resize**

**Syntax:**

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

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

This function runs in **linear time**.

**Related topics:**

(C++ Multimaps) [Multimap constructors & destructors](https://cppreference.com)  
(C++ Strings) [capacity](https://cppreference.com)  
[size](https://cppreference.com)
resize

Syntax:

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

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

This function runs in linear time.

Related topics:
(C++ Multimaps) Multimap constructors & destructors
(C++ Strings) capacity
size
```cpp
#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) [Multimap constructors & destructors](capacity size)
reverse

Syntax:

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

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

Related topics:
sort
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:

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

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

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

Related topics: reverse
rfind

Syntax:

```cpp
#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.

```cpp
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;
```

Related topics:
- find
- find_first_not_of
- find_first_of
- find_last_not_of
- find_last_of
rotate

Syntax:

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

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

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

rotate() runs in linear time.

Related topics:
rotate_copy
rotate_copy

Syntax:

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

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

Related topics:

rotate
search

Syntax:

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

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

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

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

Related topics:
- `equal`
- `find`
- `lexicographical_compare`
- `mismatch`
- `search_n`
search_n

Syntax:

```
#include <algorithm>

iterator search_n( iterator start, iterator end, size_t num, const

iterator search_n( iterator start, iterator end, size_t num, const
```

The `search_n()` function looks for `num` occurrences 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`
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:

```cpp
#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
**set**

**Syntax:**

```cpp
#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:

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

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

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

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

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

Related topics:
- `includes set_intersection`
- `set_symmetric_difference`
- `set_union`
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:

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

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

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

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

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

Related topics:
- `includes set_difference`
- `set_intersection`
- `set_union`
set_union

Syntax:

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

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

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

`set_union()` runs in linear time.

Related topics:
- includes
  - merge
- set_difference
- set_intersection
- set_symmetric_difference
**setf**

**Syntax:**

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

The function `setf()` sets the io stream format flags 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 io stream format flags.

For example:

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

Note that the preceding code is functionally identical to:

```cpp
int number = 0x3FF;
cout << "Decimal: " << number << endl << hex; 
cout << "Hexadecimal: " << number << endl;
```

thanks to io stream manipulators.

**Related topics:**

flags unset
**size**

*Syntax:*

```cpp
#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](#)
size

Syntax:

```cpp
#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
size

Syntax:

```cpp
#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
size

Syntax:

```cpp
#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
size

Syntax:

```cpp
#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
**size**

Syntax:

```cpp
#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`
size

Syntax:

```cpp
#include <map>
size_type size() const;
```

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

*Related topics:*

- empty
- max_size
size

Syntax:

```cpp
#include <map>
size_type size() const;
```

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

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

Syntax:

```cpp
#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 (C++ Strings)
size

Syntax:

```cpp
#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++ Strings)
- `max_size` (C++ Multimaps)
- `resize` (C++ Strings)
size

Syntax:

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

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

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

Syntax:

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

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

Syntax:

```cpp
#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:

```cpp
#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:

```cpp
vector<int> v;
v.push_back( 23 );
v.push_back( -1 );
v.push_back( 9999 );
v.push_back( 0 );
v.push_back( 4 );

cout << "Before sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
    cout << v[i] << " ";
}
cout << endl;

sort( v.begin(), v.end() );

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

When run, the above code displays this output:

<table>
<thead>
<tr>
<th>Before sorting: 23 -1 9999 0 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>After sorting: -1 0 4 23 9999</td>
</tr>
</tbody>
</table>

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

```cpp
int array[] = { 23, -1, 9999, 0, 4 };
unsigned int array_size = 5;

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

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

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

```cpp
bool cmp( int a, int b ) {
    return a > b;
}
...

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

cout << "Before: ";
for( int i = 0; i < 10; i++ ) {
```
cout << v[i] << " ";
}
cout << endl;

sort( v.begin(), v.end(), cmp );

cout << "After: ";
for( int i = 0; i < 10; i++ ) {
    cout << v[i] << " ";
}
cout << endl;

Related topics:

**binary_search**  **is_sorted**  
**merge**  
**partial_sort**  
**partial_sort_copy**  
**stable_sort**  
(Other Standard C Functions) **qsort**
sort_heap

Syntax:

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

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

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

Related topics:
- `is_heap`
- `make_heap`
- `pop_heap`
- `push_heap`
splice

Syntax:

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

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

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

Related topics:
- `insert`
- `merge`
- `swap`
stable_partition

Syntax:

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

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

Related topics:

`partition`
stable_sort

Syntax:

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

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

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

Related topics:
- `binary_search`
- `is_sorted`
- `partial_sort`
- `partial_sort_copy`
- `sort`
substr

Syntax:

```cpp
#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:

```cpp
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;
```

displays

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

Related topics:

copy
swap

Syntax:

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

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

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

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

The above code displays:

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

**Related topics:**
(C++ Lists) **splice**
swap

Syntax:

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

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

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

```cpp
vector v1;
v1.push_back("I'm in v1!");

vector v2;
v2.push_back("And I'm in v2!");

v1.swap(v2);

cout << "The first element in v1 is " << v1.front() << endl;
cout << "The first element in v2 is " << v2.front() << endl;
```

The above code displays:

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

Related topics:

= operator (C++ Lists) splice
swap

Syntax:

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

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

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

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

The above code displays:

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

Related topics:
(C++ Lists) `splice`
### swap

**Syntax:**

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

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

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

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

The above code displays:

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

**Related topics:**

splice
swap

Syntax:

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

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

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

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

The above code displays:

And this is second
This comes first

Related topics:
(C++ Lists) splice
swap

Syntax:

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

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

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

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

The above code displays:

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

Related topics:
(C++ Lists) splice
The swap() function exchanges the elements of the current map with those of `from`. This function operates in constant time.

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

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

The above code displays:

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

**Related topics:**

(C++ Lists) [splice](https://cppreference.com)
**swap**

*Syntax:*

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

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

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

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

The above code displays:

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

*Related topics:*

*(C++ Lists)* [splice]
swap

Syntax:

```cpp
#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:

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

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

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

Related topics:
- `iter_swap`
- `swap`
sync_with_stdio

Syntax:

```cpp
#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:

```cpp
#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:

```cpp
#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:

```cpp
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();
```

Related topics:

- seekg
- seekp
- tellg
test

Syntax:

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

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

Syntax:

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

The to_string() function returns a string representation of the bitset.

Related topics:

to_ulong
to_ulong

Syntax:

```cpp
#include <bitset>
unsigned long to_ulong();
```

The function `to_ulong()` returns the bitset, converted into an unsigned long integer.

*Related topics:*

[to_string]
**top**

*Syntax:*

```cpp
#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:

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

*Related topics:*

- [pop](#)
**top**

*Syntax:*

```cpp
#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:

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

*Related topics:*

`pop`
transform

Syntax:

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

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

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

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

Related topics:
- copy
- fill
- generate
unique

Syntax:

```cpp
#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
unique

Syntax:

```cpp
#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 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:

```cpp
#include <algorithm>

iterator unique_copy( iterator start, iterator end, iterator result);
iterator unique_copy( iterator start, iterator end, iterator result, BinPred p);
```

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

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

`unique_copy()` runs in linear time.

Related topics:
- adjacent_find
- remove
- unique
**unsetf**

*Syntax:*

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

The function `unsetf()` uses `flags` to clear the **io stream format flags** associated with the current stream.

*Related topics:*

`flags setf`
upper_bound

Syntax:

```cpp
#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:
lower_bound
upper_bound

Syntax:

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

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

Related topics:
lower_bound
upper_bound

Syntax:

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

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

Related topics:
lower_bound
upper_bound

**Syntax:**

```cpp
#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*
**upper_bound**

Syntax:

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

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

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

`upper_bound()` runs in **logarithmic time**.

*Related topics:*

- [binary_search](#)
- [equal_range](#)
value_comp

Syntax:

```cpp
#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`
value_comp

Syntax:

```cpp
#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](#)
value_comp

Syntax:

```cpp
#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](#)
value_comp

Syntax:

```cpp
#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](#)
width

Syntax:

```cpp
#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:

```cpp
cout.width( 5 );
cout << "2";
```

displays

```
  2
```

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

*Related topics:*

*fill precision*
write

Syntax:

```cpp
#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
<table>
<thead>
<tr>
<th>Predefined preprocessor variables</th>
<th>Miscellaneous preprocessor variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>stops the program</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>acos</td>
<td>arc cosine</td>
</tr>
<tr>
<td>asctime</td>
<td>a textual version of the time</td>
</tr>
<tr>
<td>asin</td>
<td>arc sine</td>
</tr>
<tr>
<td>assert</td>
<td>stops the program if an expression isn't true</td>
</tr>
<tr>
<td>atan</td>
<td>arc tangent</td>
</tr>
<tr>
<td>atan2</td>
<td>arc tangent, using signs to determine quadrants</td>
</tr>
<tr>
<td>atexit</td>
<td>sets a function to be called when the program exits</td>
</tr>
<tr>
<td>atof</td>
<td>converts a string to a double</td>
</tr>
<tr>
<td>atoi</td>
<td>converts a string to an integer</td>
</tr>
<tr>
<td>atol</td>
<td>converts a string to a long</td>
</tr>
<tr>
<td>bsearch</td>
<td>perform a binary search</td>
</tr>
<tr>
<td>calloc</td>
<td>allocates and clears a two-dimensional chunk of memory</td>
</tr>
<tr>
<td>ceil</td>
<td>the smallest integer not less than a certain value</td>
</tr>
<tr>
<td>clearerr</td>
<td>clears errors</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>clock</td>
<td>returns the amount of time that the program has been running</td>
</tr>
<tr>
<td>cos</td>
<td>cosine</td>
</tr>
<tr>
<td>cosh</td>
<td>hyperbolic cosine</td>
</tr>
<tr>
<td>ctime</td>
<td>returns a specifically formatted version of the time</td>
</tr>
<tr>
<td>difftime</td>
<td>the difference between two times</td>
</tr>
<tr>
<td>div</td>
<td>returns the quotient and remainder of a division</td>
</tr>
<tr>
<td>exit</td>
<td>stop the program</td>
</tr>
<tr>
<td>exp</td>
<td>returns &quot;e&quot; raised to a given power</td>
</tr>
<tr>
<td>fabs</td>
<td>absolute value for floating-point numbers</td>
</tr>
<tr>
<td>fclose</td>
<td>close a file</td>
</tr>
<tr>
<td>feof</td>
<td>true if at the end-of-file</td>
</tr>
<tr>
<td>ferror</td>
<td>checks for a file error</td>
</tr>
<tr>
<td>fflush</td>
<td>writes the contents of the output buffer</td>
</tr>
<tr>
<td>fgetc</td>
<td>get a character from a stream</td>
</tr>
<tr>
<td>fgetpos</td>
<td>get the file position indicator</td>
</tr>
<tr>
<td>fgets</td>
<td>get a string of characters from a stream</td>
</tr>
<tr>
<td>floor</td>
<td>returns the largest integer not greater than a given value</td>
</tr>
<tr>
<td>fmod</td>
<td>returns the remainder of a division</td>
</tr>
<tr>
<td>fopen</td>
<td>open a file</td>
</tr>
<tr>
<td>fprintf</td>
<td>print formatted output to a file</td>
</tr>
<tr>
<td>fputc</td>
<td>write a character to a file</td>
</tr>
<tr>
<td>fputs</td>
<td>write a string to a file</td>
</tr>
<tr>
<td>fread</td>
<td>read from a file</td>
</tr>
<tr>
<td>free</td>
<td>returns previously allocated memory to the operating system</td>
</tr>
<tr>
<td>freopen</td>
<td>open an existing stream with a different name</td>
</tr>
<tr>
<td>frexp</td>
<td>decomposes a number into scientific notation</td>
</tr>
<tr>
<td>fscanf</td>
<td>read formatted input from a file</td>
</tr>
<tr>
<td>fseek</td>
<td>move to a specific location in a file</td>
</tr>
<tr>
<td>fsetpos</td>
<td>move to a specific location in a file</td>
</tr>
<tr>
<td>ftell</td>
<td>returns the current file position indicator</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>fwrite</td>
<td>write to a file</td>
</tr>
<tr>
<td>getc</td>
<td>read a character from a file</td>
</tr>
<tr>
<td>getchar</td>
<td>read a character from STDIN</td>
</tr>
<tr>
<td>getenv</td>
<td>get environment information about a variable</td>
</tr>
<tr>
<td>gets</td>
<td>read a string from STDIN</td>
</tr>
<tr>
<td>gmttime</td>
<td>returns a pointer to the current Greenwich Mean Time</td>
</tr>
<tr>
<td>isalnum</td>
<td>true if a character is alphanumeric</td>
</tr>
<tr>
<td>isalpha</td>
<td>true if a character is alphabetic</td>
</tr>
<tr>
<td>iscntrl</td>
<td>true if a character is a control character</td>
</tr>
<tr>
<td>isdigit</td>
<td>true if a character is a digit</td>
</tr>
<tr>
<td>isgraph</td>
<td>true if a character is a graphical character</td>
</tr>
<tr>
<td>islower</td>
<td>true if a character is lowercase</td>
</tr>
<tr>
<td>isprint</td>
<td>true if a character is a printing character</td>
</tr>
<tr>
<td>ispunct</td>
<td>true if a character is punctuation</td>
</tr>
<tr>
<td>isspace</td>
<td>true if a character is a space character</td>
</tr>
<tr>
<td>isupper</td>
<td>true if a character is an uppercase character</td>
</tr>
<tr>
<td>isxdigit</td>
<td>true if a character is a hexadecimal character</td>
</tr>
<tr>
<td>labs</td>
<td>absolute value for long integers</td>
</tr>
<tr>
<td>ldexp</td>
<td>computes a number in scientific notation</td>
</tr>
<tr>
<td>ldiv</td>
<td>returns the quotient and remainder of a division, in long integer form</td>
</tr>
<tr>
<td>localtime</td>
<td>returns a pointer to the current time</td>
</tr>
<tr>
<td>log</td>
<td>natural logarithm</td>
</tr>
<tr>
<td>log10</td>
<td>natural logarithm, in base 10</td>
</tr>
<tr>
<td>longjmp</td>
<td>start execution at a certain point in the program</td>
</tr>
<tr>
<td>malloc</td>
<td>allocates memory</td>
</tr>
<tr>
<td>memchr</td>
<td>searches an array for the first occurrence of a character</td>
</tr>
<tr>
<td>memcmp</td>
<td>compares two buffers</td>
</tr>
<tr>
<td>memcpy</td>
<td>copies one buffer to another</td>
</tr>
<tr>
<td>memmove</td>
<td>moves one buffer to another</td>
</tr>
<tr>
<td>memset</td>
<td>fills a buffer with a character</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>mktime</td>
<td>returns the calendar version of a given time</td>
</tr>
<tr>
<td>modf</td>
<td>decomposes a number into integer and fractional parts</td>
</tr>
<tr>
<td>perror</td>
<td>displays a string version of the current error to <strong>STDERR</strong></td>
</tr>
<tr>
<td>pow</td>
<td>returns a given number raised to another number</td>
</tr>
<tr>
<td>printf</td>
<td>write formatted output to <strong>STDOUT</strong></td>
</tr>
<tr>
<td>putc</td>
<td>write a character to a stream</td>
</tr>
<tr>
<td>putchar</td>
<td>write a character to <strong>STDOUT</strong></td>
</tr>
<tr>
<td>puts</td>
<td>write a string to <strong>STDOUT</strong></td>
</tr>
<tr>
<td>qsort</td>
<td>perform a quicksort</td>
</tr>
<tr>
<td>raise</td>
<td>send a signal to the program</td>
</tr>
<tr>
<td>rand</td>
<td>returns a pseudorandom number</td>
</tr>
<tr>
<td>realloc</td>
<td>changes the size of previously allocated memory</td>
</tr>
<tr>
<td>remove</td>
<td>erase a file</td>
</tr>
<tr>
<td>rename</td>
<td>rename a file</td>
</tr>
<tr>
<td>rewind</td>
<td>move the file position indicator to the beginning of a file</td>
</tr>
<tr>
<td>scanf</td>
<td>read formatted input from <strong>STDIN</strong></td>
</tr>
<tr>
<td>setbuf</td>
<td>set the buffer for a specific stream</td>
</tr>
<tr>
<td>setjmp</td>
<td>set execution to start at a certain point</td>
</tr>
<tr>
<td>setlocale</td>
<td>sets the current locale</td>
</tr>
<tr>
<td>setvbuf</td>
<td>set the buffer and size for a specific stream</td>
</tr>
<tr>
<td>signal</td>
<td>register a function as a signal handler</td>
</tr>
<tr>
<td>sin</td>
<td>sine</td>
</tr>
<tr>
<td>sinh</td>
<td>hyperbolic sine</td>
</tr>
<tr>
<td>sprintf</td>
<td>write formatted output to a buffer</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
<tr>
<td>srand</td>
<td>initialize the random number generator</td>
</tr>
<tr>
<td>sscanf</td>
<td>read formatted input from a buffer</td>
</tr>
<tr>
<td>strcat</td>
<td>concatenates two strings</td>
</tr>
<tr>
<td>strchr</td>
<td>finds the first occurrence of a character in a string</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strcmp</td>
<td>compares two strings</td>
</tr>
<tr>
<td>strcoll</td>
<td>compares two strings in accordance to the current locale</td>
</tr>
<tr>
<td>strcpy</td>
<td>copies one string to another</td>
</tr>
<tr>
<td>strcspn</td>
<td>searches one string for any characters in another</td>
</tr>
<tr>
<td>strerror</td>
<td>returns a text version of a given error code</td>
</tr>
<tr>
<td>strftime</td>
<td>returns individual elements of the date and time</td>
</tr>
<tr>
<td>strlen</td>
<td>returns the length of a given string</td>
</tr>
<tr>
<td>strncat</td>
<td>concatenates a certain amount of characters of two strings</td>
</tr>
<tr>
<td>strncmp</td>
<td>compares a certain amount of characters of two strings</td>
</tr>
<tr>
<td>strncpy</td>
<td>copies a certain amount of characters from one string to another</td>
</tr>
<tr>
<td>strpbrk</td>
<td>finds the first location of any character in one string, in another string</td>
</tr>
<tr>
<td>strrchr</td>
<td>finds the last occurrence of a character in a string</td>
</tr>
<tr>
<td>strspn</td>
<td>returns the length of a substring of characters of a string</td>
</tr>
<tr>
<td>strstr</td>
<td>finds the first occurrence of a substring of characters</td>
</tr>
<tr>
<td>strtod</td>
<td>converts a string to a double</td>
</tr>
<tr>
<td>strtok</td>
<td>finds the next token in a string</td>
</tr>
<tr>
<td>strtol</td>
<td>converts a string to a long</td>
</tr>
<tr>
<td>strtoul</td>
<td>converts a string to an unsigned long</td>
</tr>
<tr>
<td>strxfrm</td>
<td>converts a substring so that it can be used by string comparison functions</td>
</tr>
<tr>
<td>system</td>
<td>perform a system call</td>
</tr>
<tr>
<td>tan</td>
<td>tangent</td>
</tr>
<tr>
<td>tanh</td>
<td>hyperbolic tangent</td>
</tr>
<tr>
<td>time</td>
<td>returns the current calendar time of the system</td>
</tr>
<tr>
<td>tmpfile</td>
<td>return a pointer to a temporary file</td>
</tr>
<tr>
<td>tmpnam</td>
<td>return a unique filename</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>tolower</td>
<td>converts a character to lowercase</td>
</tr>
<tr>
<td>toupper</td>
<td>converts a character to uppercase</td>
</tr>
<tr>
<td>ungetc</td>
<td>puts a character back into a stream</td>
</tr>
<tr>
<td>va_arg</td>
<td>use variable length parameter lists</td>
</tr>
<tr>
<td>vprintf, vfprintf, and vsprintf</td>
<td>write formatted output with variable argument lists</td>
</tr>
</tbody>
</table>
The # and ## operators are used with the #define 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

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

will make the compiler turn this command

```c
cout << to_string( Hello World! ) << endl;
```

into

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

Here is an example of the ## command:

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

This code will make the compiler turn

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

into

```c
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 ) ) {
    ...
}
```

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 its 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:

```c
#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 );
}
```

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.
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 `#define` statement, then the code immediately following the command will be compiled.

`#ifndef` macro

If the `macro` has not been defined by a `#define` 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:

```cpp
#ifdef DEBUG
    cout << "This is the test version, i=" << i << endl;
#else
    cout << "This is the production version!" << endl;
```
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
#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".
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.
#undef

The #undef command undefines a previously defined macro variable, such as a variable defined by a #define.

Related topics:
#define
Predefined preprocessor variables

Syntax:

```c
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__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++.
**abort**

*Syntax:*

```c
#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`
abs

Syntax:

```c
#include <cstdlib>
int abs( int num );
```

The `abs()` function returns the absolute value of `num`. For example:

```c
int magic_number = 10;
cout << "Enter a guess: ";
cin >> x;
cout << "Your guess was " << abs( magic_number - x ) << " away from the magic number."
```

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:
asin atan atan2 cos cosh sin sinh tan tanh
asctime

Syntax:

```c
#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
# asin

**Syntax:**

```cpp
#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
- cos
- cosh
- sin
- sinh
- tan
- tanh
assert

Syntax:

```c
#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`
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:

```c
#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
atan
cos
cosh
sin
sinh
tan
tanh
## atexit

### Syntax:

```c
#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`
atof

Syntax:

```c
#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,

```c
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:**

```c
#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:

```c
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:

```c
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:

- atof
- atoi
- (Standard C I/O) `sprintf`
- `strtol`
### bsearch

**Syntax:**

```c
#include <cstdlib>
void *bsearch( const void *key, const void *buf, size_t num, size_t size,
               int (*compare)(const void *, const void *) );
```

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`
malloc

Syntax:

```c
#include <stdlib>
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
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
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 perror() to figure out which error actually occurred.

Related topics:
feof ferror perror
clock

Syntax:

```c
#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`
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 coth  sinh  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  
tan  
tanh
ctime

Syntax:

```c
#include <ctime>
char *ctime( const time_t *time );
```

The `ctime()` function converts the calendar time `time` to local time of the format:

```
day month date hours:minutes:seconds year
```

using `ctime()` is equivalent to

```c
asctime( localtime( tp ) );
```

Related topics:
- `asctime`
- `clock`
- `gmtime`
- `localtime`
- `mktime`
- `time`
The function difftime() returns \textit{time2 - time1}, in seconds.

\textit{Related topics:}
\texttt{asctime gmtime localtime time}
div

Syntax:

```c
#include <cstdlib>

struct div_t {
    int quot;  // The quotient
    int rem;   // The remainder
};
```

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:

```c
int quot;  // The quotient
int rem;   // The remainder
```

For example, the following code displays the quotient and remainder of `x/y`:

```c
    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:

- `ldiv`
exit

Syntax:

```c
#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`
exp

Syntax:

```cpp
#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:

```cpp
#include <cmath>
double fabs( double arg );
```

The function fabs() returns the absolute value of arg.

Related topics:
- abs
- fmod
- labs
fclose

Syntax:

```c
#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:

```c
#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`
ferror

Syntax:

```c
#include <cstdio>
int ferror( FILE *stream );
```

The `ferror()` function looks for errors with `stream`, returning zero if no errors have occurred, and non-zero if there is an error. In case of an error, use `perror()` to determine which error has occurred.

Related topics:
- `clearerr`
- `feof`
- `perror`
**fflush**

*Syntax:*

```c
#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.

```c
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:

```c
#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:

```c
#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`
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
floor

Syntax:

```cpp
#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:

```cpp
#include <cmath>
double fmod( double x, double y );
```

The fmod() function returns the remainder of $x/y$.

Related topics:
ceil fabs floor
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)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Open a text file for reading</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Create a text file for writing</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Append to a text file</td>
</tr>
<tr>
<td>&quot;rb&quot;</td>
<td>Open a binary file for reading</td>
</tr>
<tr>
<td>&quot;wb&quot;</td>
<td>Create a binary file for writing</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>Append to a binary file</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>Open a text file for read/write</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>Create a text file for read/write</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>Open a text file for read/write</td>
</tr>
<tr>
<td>&quot;rb+&quot;</td>
<td>Open a binary file for read/write</td>
</tr>
<tr>
<td>&quot;wb+&quot;</td>
<td>Create a binary file for read/write</td>
</tr>
<tr>
<td>&quot;ab+&quot;</td>
<td>Open a binary file for read/write</td>
</tr>
</tbody>
</table>

An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```
Related topics:
fclose fflush fputc fread freopen fseek fwrite getc getchar setbuf
fprintf

Syntax:

```c
#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 `printf()` 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:

```c
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:

```c
#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
fputs

Syntax:

```c
#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:

```c
#include <cstdio>
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

The function fread() reads \textit{num} number of objects (where each object is \textit{size} bytes) and places them into the array pointed to by \textit{buffer}. The data comes from the given input \textit{stream}. The return value of the function is the number of things read. You can use \texttt{feof}() or \texttt{ferror}() to figure out if an error occurs.

\textit{Related topics:}
\texttt{fflush} \texttt{fgetc}  
\texttt{fopen}  
\texttt{fputc}  
\texttt{fscanf}  
\texttt{fwrite}  
\texttt{getc}
free

Syntax:

```c
#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 `malloc()`, `calloc()`, or `realloc()`. An example:

```c
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`
- `new`
- `realloc`
freopen

Syntax:

```c
#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`
frexp

Syntax:

```cpp
#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`
fscanf

Syntax:

```c
#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
fseek

Syntax:

```c
#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):

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_SET</td>
<td>Seek from the start of the file</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>Seek from the current location</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>Seek from the end of the file</td>
</tr>
</tbody>
</table>

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:**
- fseek
- fopen
- fgetpos
- ftell
- fsetpos
- rewind
fsetpos

Syntax:

```c
#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:*

```c
#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:

```c
#include <cstdio>

int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

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
- fscanf
- fopen
- fputc
- fread
- fwrite
- fscanf
- getc
getc

Syntax:

```c
#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 `fgetc()`. For example:

```c
int ch;
FILE *input = fopen( "stuff", "r" );

ch = getc( input );
while( ch != EOF ){
    printf( "%c", ch );
    ch = getc( input );
}
```

Related topics:

- `feof`
- `fflush`
- `fgetc`
- `fopen`
- `fputc`
- `fread`
- `fwrite`
- `putc`
- `ungetc`
getchar

Syntax:

```c
#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
getenv

Syntax:

```c
#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`
gets

Syntax:

```c
#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 `fgets()`.

Related topics:
- `fgetc`
- `fgets`
- `fputs`
- `puts`
gmtime

Syntax:

```c
#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`
**isalnum**

**Syntax:**

```cpp
#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.

```cpp
char c;
scanf( "%c", &c );
if( isalnum(c) )
    printf( "You entered the alphanumerical character %c\n", c );
```

**Related topics:**

- isalpha
- iscntrl
- isdigit
- isgraph
- isprint
- ispunct
- isspace
- isxdigit
isalpha

**Syntax:**

```c
#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.

```c
char c;
scanf( "%c", &c );
if( isalpha(c) )
    printf( "You entered a letter of the alphabet\n" );
```

**Related topics:**
- isalnum
- iscntrl
- isdigit
- isgraph
- isprint
- ispunct
- isspace
- isxdigit
iscntrl

Syntax:

```cpp
#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:
isalnum isalpha
isdigit
isgraph
isprint
ispunct
isspace
isxdigit
**isdigit**

**Syntax:**

```c
#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.

```c
char c;
scanf( "%c", &c );
if( isdigit(c) )
    printf( "You entered the digit %c\n", c );
```

**Related topics:**
- `isalnum`
- `isalpha`
- `iscntrl`
- `isgraph`
- `isprint`
- `ispunct`
- `isspace`
- `isxdigit`
isgraph

Syntax:

```c
#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:

- isalnum
- isalpha
- 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:
isupper
isprint

Syntax:

```c
#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  isalpha  
iscntrl  
isdigit  
isgraph  
ispunct  
isspace
ispunct

Syntax:

```c
#include <cctype>
int ispunct( int ch );
```

The ispunct() function returns non-zero if its argument is a printing character but neither alphanumerical nor a space. Otherwise, zero is returned.

Related topics:

- isalnum
- isalpha
- iscntrl
- isdigit
- isgraph
- isprint
- isspace
- isxdigit
**isspace**

*Syntax:*

```cpp
#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`
- `isalpha`
- `iscntrl`
- `isdigit`
- `isgraph`
- `isprint`
- `ispunct`
- `isxdigit`
isupper

Syntax:

```c
#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:

```c
#include <ctype>
int isxdigit( int ch );
```

The function isxdigit() returns non-zero if its argument is a hexadecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics:
isalnum  isalpha
iscntrl
isdigit
isgraph
ispunct
isspace
labs

Syntax:

```
#include <cstdlib>
long labs( long num );
```

The function labs() returns the absolute value of `num`.

Related topics:

`abs fabs`
ldexp

Syntax:

```c
#include <cmath>
double ldexp( double num, int exp );
```

The ldexp() function returns \( num \times (2^\exp) \). And get this: if an overflow occurs, HUGE_VAL is returned.

Related topics:
frexp modf
# ldiv

**Syntax:**

```c
#include <cstdlib>
ldiv_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:

```c
long quot;    // the quotient
long rem;     // the remainder
```

*Related topics:*

*div*


**localtime**

*Syntax:*

```c
#include <ctime>
struct tm *localtime( const time_t *time );
```

The function `localtime()` converts calendar time `time` into local time. Watch out for the `static` return.

*Related topics:*
- `asctime`  
- `ctime`  
- `difftime`  
- `gmtime`  
- `strftime`  
- `time`
log

Syntax:

```cpp
#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:

```cpp
double answer = log(x) / log(b);
```

Related topics:
- `exp`
- `log10`
- `pow`
- `sqrt`
**log10**

*Syntax:*

```cpp
#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](#)
longjmp

Syntax:

```c
#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 `setjmp()`. `envbuf` is usually set through a call to `setjmp()`. `status` becomes the return value of `setjmp()` and can be used to figure out where `longjmp()` came from. `status` should not be set to zero.

Related topics:

`setjmp`
**malloc**

*Syntax:*

```c
#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:

```c
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` (C/C++ Keywords)
- `free`
- `new` (C/C++ Keywords)
- `realloc`
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:

```c
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 count );
```

The function memcmp() compares the first `count` characters of `buffer1` and `buffer2`. The return values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than 0</td>
<td>buffer1 is less than buffer2</td>
</tr>
<tr>
<td>equal to 0</td>
<td>buffer1 is equal to buffer2</td>
</tr>
<tr>
<td>greater than 0</td>
<td>buffer1 is greater than buffer2</td>
</tr>
</tbody>
</table>

Related topics:
- memchr
- memcpy
- memset
- 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:**

```c
#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:**

```c
#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 initializing a section of memory to some value. For example, this command:

```c
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:

<table>
<thead>
<tr>
<th>Input size</th>
<th>Initialized with a for-loop</th>
<th>Initialized with <code>memset()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.016</td>
<td>0.017</td>
</tr>
<tr>
<td>10000</td>
<td>0.055</td>
<td>0.013</td>
</tr>
<tr>
<td>100000</td>
<td>0.443</td>
<td>0.029</td>
</tr>
<tr>
<td>1000000</td>
<td>4.337</td>
<td>0.291</td>
</tr>
</tbody>
</table>

**Related topics:**

- `memcmp`
- `memcpy`
- `memmove`
**mktime**

**Syntax:**

```c
#include <ctime>

#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`
**modf**

*Syntax:*

```cpp
#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
**perror**

**Syntax:**

```c
#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:

```c
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 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:**
- clearerr
- feof
- ferror
**pow**

Syntax:

```cpp
#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`
printf

Syntax:

```c
#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

```c
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>character</td>
</tr>
<tr>
<td>%d</td>
<td>signed integers</td>
</tr>
<tr>
<td>%i</td>
<td>signed integers</td>
</tr>
<tr>
<td>%e</td>
<td>scientific notation, with a lowercase &quot;e&quot;</td>
</tr>
<tr>
<td>%E</td>
<td>scientific notation, with an uppercase &quot;E&quot;</td>
</tr>
<tr>
<td>%f</td>
<td>floating point</td>
</tr>
<tr>
<td>----</td>
<td>---------------</td>
</tr>
<tr>
<td>%g</td>
<td>use %e or %f, whichever is shorter</td>
</tr>
<tr>
<td>%G</td>
<td>use %E or %f, whichever is shorter</td>
</tr>
<tr>
<td>%o</td>
<td>octal</td>
</tr>
<tr>
<td>%s</td>
<td>a string of characters</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>%x</td>
<td>unsigned hexadecimal, with lowercase letters</td>
</tr>
<tr>
<td>%X</td>
<td>unsigned hexadecimal, with uppercase letters</td>
</tr>
<tr>
<td>%p</td>
<td>a pointer</td>
</tr>
<tr>
<td>%n</td>
<td>the argument shall be a pointer to an integer into which is placed the number of characters written so far</td>
</tr>
<tr>
<td>%%</td>
<td>a ‘%’ sign</td>
</tr>
</tbody>
</table>

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 modifier simply acts as a maximum field 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 hexadecimal 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
# Standard C I/O

## putc

### Syntax:

```c
#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:

```c
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:

```c
#include <stdio>
int putchar( int ch );
```

The `putchar()` function writes `ch` to `stdout`. The code

```c
putchar( ch );
```

is the same as

```c
putc( ch, stdout );
```

The return value of `putchar()` is the written character, or `EOF` if there is an error.

*Related topics:*

`putc`
puts

Syntax:

```c
#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:

```c
#include <cstdlib>
void qsort( void *buf, size_t num, size_t size, int (*compare)(const void*, const void*) );
```

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:

```c
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] );
}
```

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:

```c
#include <csignal>
int raise( int signal );
```

The `raise()` function sends the specified `signal` to the program. Some signals:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Termination error</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Floating pointer error</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Bad instruction</td>
</tr>
<tr>
<td>SIGINT</td>
<td>User pressed CTRL-C</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Illegal memory access</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Terminate program</td>
</tr>
</tbody>
</table>

The return value is zero upon success, nonzero on failure.

Related topics:

[signal]
rand

Syntax:

```cpp
#include <cstdlib>
int rand( void );
```

The function `rand()` returns a pseudorandom integer between zero and `RAND_MAX`. An example:

```c
srand( time(NULL) );
for( i = 0; i < 10; i++ )
    printf( "Random number #d: %d\n", i, rand() );
```

Related topics:

`srand`
realloc

Syntax:

```c
#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
remove

Syntax:

```c
#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:

```c
#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:*

```c
#include <stdio>
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 `printf()`. 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:

<table>
<thead>
<tr>
<th>Control Character</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%c</code></td>
<td>a single character</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>a decimal integer</td>
</tr>
<tr>
<td><code>%i</code></td>
<td>an integer</td>
</tr>
<tr>
<td><code>%e</code>, <code>%f</code>, <code>%g</code></td>
<td>a floating-point number</td>
</tr>
<tr>
<td><code>%lf</code></td>
<td>a double</td>
</tr>
<tr>
<td><code>%o</code></td>
<td>an octal number</td>
</tr>
<tr>
<td><code>%s</code></td>
<td>a string</td>
</tr>
<tr>
<td><code>%x</code></td>
<td>a hexadecimal number</td>
</tr>
<tr>
<td><code>%p</code></td>
<td>a pointer</td>
</tr>
<tr>
<td><code>%n</code></td>
<td>an integer equal to the number of characters read so far</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>an unsigned integer</td>
</tr>
<tr>
<td><code>[]</code></td>
<td>a set of characters</td>
</tr>
<tr>
<td><code>%%</code></td>
<td>a percent sign</td>
</tr>
</tbody>
</table>

`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:

```c
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:

```c
#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`
setjmp

Syntax:

```c
#include <csetjmp>
int setjmp( jmp_buf envbuf );
```

The `setjmp()` function saves the system stack in `envbuf` for use by a later call to `longjmp()`. When you first call `setjmp()`, its return value is zero. Later, when you call `longjmp()`, the second argument of `longjmp()` is what the return value of `setjmp()` will be. Confused? Read about `longjmp()`.

Related topics:
`longjmp`
setlocale

Syntax:

```c
#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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_ALL</td>
<td>All of the locale</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>Date and time formatting</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>Number formatting</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>String collation and regular expression matching</td>
</tr>
<tr>
<td>LCCTYPE</td>
<td>Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>For monetary formatting</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>For natural language messages</td>
</tr>
</tbody>
</table>

Related topics:
(Standard C String and Character) `strcoll`
setvbuf

Syntax:

```c
#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`
signal

Syntax:

```c
#include <csignal>
void (*signal( int signal, void (*func)(int)))(int);
```

The `signal()` function sets `func` to be called when `signal` is received by your program. `func` can be a custom signal handler, or one of these macros (defined in the csignal header file):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIG_DFL</td>
<td>default signal handling</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>ignore the signal</td>
</tr>
</tbody>
</table>

Some basic signals that you can attach a signal handler to are:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGTERM</td>
<td>Generic stop signal that can be caught.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Interrupt program, normally ctrl-c.</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Interrupt program, similar to SIGINT.</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Stops the program. Cannot be caught.</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>Reports a disconnected terminal.</td>
</tr>
</tbody>
</table>

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:

```cpp
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++ ) {
        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:
raise
**sin**

**Syntax:**

```c
#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:**
acos  asin
atan
atan2
cos
cosh
sin
tan
tanh
**sprintf**

*Syntax:*

```c
#include <cstdio>
int sprintf( char *buffer, const char *format, ... );
```

The `sprintf()` function is just like `printf()`, except that the output is sent to `buffer`. The return value is the number of characters written. For example:

```c
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 `atoi()` -- where `atoi()` 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:

```c
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:

```c
char result[100];
float fnum = 3.14159;
sprintf( result, "%f", fnum );
```

*Related topics:*

(Standard C String and Character) `atof`
(Standard C String and Character) `atoi`
(Standard C String and Character) \texttt{atol}\n\texttt{fprintf}\n\texttt{printf}
sqrt

Syntax:

```c
#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
**srand**

*Syntax:*

```c
#include <cstdlib>
void srand( unsigned seed );
```

The function `srand()` is used to seed the random sequence generated by `rand()`. For any given `seed`, `rand()` will generate a specific "random" sequence over and over again.

```c
srand( time(NULL) );
for( i = 0; i < 10; i++ )
    printf( "Random number #%d: %d\n", i, rand() );
```

*Related topics:*

- [rand](Standard C Date & Time) [time](Standard C Date & Time)
sscanf

Syntax:

```
#include <cstdio>
int sscanf( const char *buffer, const char *format, ... );
```

The function `sscanf()` is just like `scanf()`, except that the input is read from `buffer`.

Related topics:
`fscanf` `scanf`
**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
", 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`
- `strncpy`
- `strlcpy`
- `strlcat`

Another set of related (but non-standard) functions are `strlcpy` and `strlcat`. 
strchr

Syntax:

```c
#include <cstring>
char *strchr( const char *str, int ch );
```

The function strchr() returns a pointer to the first occurrence of `ch` in `str`, or `NULL` if `ch` is not found.

Related topics:

- `strcat`
- `strncpy`
- `strcmp`
- `strcpy`
- `strlen`
- `strncat`
- `strncmp`
- `strncpy`
- `strpbrk`
- `strspn`
- `strstr`
- `strtok`
**strcmp**

*Syntax:*

```c
#include <cstring>
int strcmp( const char *str1, const char *str2 );
```

The function `strcmp()` compares `str1` and `str2`, then returns:

<table>
<thead>
<tr>
<th>Return value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than 0</td>
<td>&quot;str1&quot; is less than &quot;str2&quot;</td>
</tr>
<tr>
<td>equal to 0</td>
<td>&quot;str1&quot; is equal to &quot;str2&quot;</td>
</tr>
<tr>
<td>greater than 0</td>
<td>&quot;str1&quot; is greater than &quot;str2&quot;</td>
</tr>
</tbody>
</table>

For example:

```c
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`
- `strcpyp`
- `strlen`
- `strncmp`
- `strxfrm`
**strcoll**

**Syntax:**

```c
#include <cstring>
int strcoll( const char *str1, const char *str2 );
```

The `strcoll()` function compares `str1` and `str2`, much like `strcmp()`. However, `strcoll()` performs the comparison using the locale specified by the (Standard C Date & Time) `setlocale()` function.

**Related topics:**
(Standard C Date & Time) `setlocale`
`strcmp`
`strxfrm`


**strncpy**

*Syntax:*

```c
#include <cstring>
char *strncpy( char *to, const char *from );
```

The `strncpy()` function copies characters in the string `from` to the string `to`, including the null termination. The return value is `to`.

Note that `strncpy()` does not perform bounds checking, and thus risks overrunning `from` or `to`. For a similar (and safer) function that includes bounds checking, see `strncpy()`.

*Related topics:*

- `memcpy`  
- `strcat`  
- `strchr`  
- `strcmp`  
- `strncmp`  
- `memcpy`  
- `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  
strchr  
strstr  
strtok
strerror

Syntax:

```
#include <cstring>
char *strerror( int num );
```

The function `strerror()` returns an implementation defined string corresponding to `num`. 
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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>abbreviated weekday name (e.g. Fri)</td>
</tr>
<tr>
<td>%A</td>
<td>full weekday name (e.g. Friday)</td>
</tr>
<tr>
<td>%b</td>
<td>abbreviated month name (e.g. Oct)</td>
</tr>
<tr>
<td>%B</td>
<td>full month name (e.g. October)</td>
</tr>
<tr>
<td>%c</td>
<td>the standard date and time string</td>
</tr>
<tr>
<td>%d</td>
<td>day of the month, as a number (1-31)</td>
</tr>
<tr>
<td>%H</td>
<td>hour, 24 hour format (0-23)</td>
</tr>
<tr>
<td>%I</td>
<td>hour, 12 hour format (1-12)</td>
</tr>
<tr>
<td>%j</td>
<td>day of the year, as a number (1-366)</td>
</tr>
<tr>
<td>%m</td>
<td>month as a number (1-12). Note: some versions of Microsoft Visual C++ may use values that range from 0-11.</td>
</tr>
<tr>
<td>%M</td>
<td>minute as a number (0-59)</td>
</tr>
<tr>
<td>%p</td>
<td>locale's equivalent of AM or PM</td>
</tr>
<tr>
<td>%S</td>
<td>second as a number (0-59)</td>
</tr>
<tr>
<td>%U</td>
<td>week of the year, (0-53), where week 1 has the first Sunday</td>
</tr>
<tr>
<td>%w</td>
<td>weekday as a decimal (0-6), where Sunday is 0</td>
</tr>
<tr>
<td>%W</td>
<td>week of the year, (0-53), where week 1 has the first Monday</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>%x</td>
<td>standard date string</td>
</tr>
<tr>
<td>%X</td>
<td>standard time string</td>
</tr>
<tr>
<td>%y</td>
<td>year in decimal, without the century (0-99)</td>
</tr>
<tr>
<td>%Y</td>
<td>year in decimal, with the century</td>
</tr>
<tr>
<td>%Z</td>
<td>time zone name</td>
</tr>
<tr>
<td>%%</td>
<td>a percent sign</td>
</tr>
</tbody>
</table>

The strftime() function returns the number of characters put into str, or zero if an error occurs.

*Related topics:*

- gmtime
- localtime
- time
**strlen**

*Syntax:*

```c
#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`  
- `strncpy`
strncat

Syntax:

```c
#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`  `strlcpy`  `strlcat`

Another set of related (but non-standard) functions are `strlcpy` and `strlcat`. 
strncmp

Syntax:

```c
#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:

<table>
<thead>
<tr>
<th>Return value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than 0</td>
<td>&quot;str1&quot; is less than &quot;str2&quot;</td>
</tr>
<tr>
<td>equal to 0</td>
<td>&quot;str1&quot; is equal to &quot;str2&quot;</td>
</tr>
<tr>
<td>greater than 0</td>
<td>&quot;str1&quot; is greater than str2</td>
</tr>
</tbody>
</table>

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`
- `strlen`
- `strncat`
- `strncpy`
**strncpy**

*Syntax:*

```c
#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`
- `strncpy` and `strlcpy` and `strlcat`

Another set of related (but non-standard) functions are `strlcpy` and `strlcat`. 
strpbrk

Syntax:

```c
#include <cstring>
char* strpbrk( const char* str1, const char* str2 );
```

The function strpbrk() returns a pointer to the first occurrence in `str1` of any character in `str2`, or `NULL` if no such characters are present.

Related topics:
(C++ Algorithms) find_first_of
strchr
strcspn
strchr
strrchr
strspn
strstr
strtok
**strrchr**

*Syntax:*

```c
#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:

```c
#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`
- `strstr`
- `strrchr`
- `strtok`
**strstr**

**Syntax:**

```c
#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:

```c
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, str1);
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](#)
**strtod**

**Syntax:**

```c
#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:

```c
#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:

```c
char str[] = "now # is the time for all # good men to come to the # aid of their country";
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**
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`
strtol

Syntax:

```c
#include <cstdlib>
unsigned long strtol( const char *start, char **end, int base );
```

The function strtol() behaves exactly like strtol(), except that it returns an unsigned long rather than a mere long.

*Related topics:* 
strtol
**strxfrm**

Syntax:

```c
#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`
system

Syntax:

```c
#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
- getenv
tan

Syntax:

```cpp
#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
- sin
- sinh
- tanh
tanh

Syntax:

```cpp
#include <cmath>
double tanh( double arg );
```

The function `tanh()` returns the hyperbolic tangent of `arg`.

Related topics:
- acos
- asin
- atan
- atan2
- cos
- cosh
- sin
- sinh
- tan
time

Syntax:

```c
#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
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
tmpnam

Syntax:

```c
#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
tolower

Syntax:

```cpp
#include <cctype>
int tolower( int ch );
```

The function tolower() returns the lowercase version of the character `ch`.

Related topics:

* isupper toupper
toupper

Syntax:

```cpp
#include <cctype>
int toupper( int ch );
```

The toupper() function returns the uppercase version of the character `ch`.

Related topics:

tolower
ungetc

Syntax:

```c
#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:

```c
#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:

```c
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 printf(), fprintf(), and sprintf(). 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) va_arg(). 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);
}
```
General C/C++

- Pre-processor commands
- 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
C++

- C++ I/O
- C++ Strings
- C++ String Streams
- Miscellaneous C++
C++ Standard Template Library

- About the Standard Template Library
- Iterators
- C++ Algorithms
- C++ Vectors
- C++ Double-Ended Queues
- C++ Lists
- C++ Priority Queues
- C++ Queues
- C++ Stacks
- C++ Sets
- C++ Multisets
- C++ Maps
- C++ Multimaps
- C++ Bitsets

Questions? Check out the FAQ, look at these other language references, or contact us.

Last modified on 3/18/2008 by Nate Kohl, with help from a lot of people.
Windows compiled help (CHM) file created by James Brown.
String Stream Constructors

Syntax:

```cpp
#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 )
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 [iostream mode flags](https://en.cppreference.com/w/cpp/io/stream/ios_base/flags).

An ostringstream object can be used to write to a string. This is similar to the C `sprintf()` function. For example:

```cpp
ostringstream s1;
int i = 22;
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;
```

An istringstream object can be used to read from a string. This is similar to the C `sscanf()` function. For example:

```cpp
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25
```
You can also specify the input string in the istringstream constructor as in this example:

```cpp
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25
```

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:

```cpp
#include <sstream>
operator<<
operator>>
```

Like C++ I/O Streams, 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:

```cpp
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:

```cpp
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:**

```cpp
#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:

```cpp
ostringstream stream1;
stream1 << "Testing!" << endl;
cout << stream1.str();
```

Second, `str()` can be used to copy a string into the stream. This is most useful with input strings. For example:

```cpp
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:

```cpp
istringstream stream1;
float num;

// use it once
string string1 = "25 1 3.235
111111
222222"
stream1.str(string1);
while( stream1 >> num ) cout << "num: " << num << endl; // displays

// 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; // displays
```
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 );
The keyword auto is used to declare local variables, and is purely optional.

Related topics:
register
**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.

```cpp
bool done = false;
while( !done ) {
    ...
}
```

Also see the [data types](http://cppreference.com) 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 do, for, or while loop. It is also used to finish each clause of a switch statement, keeping the program from "falling through" to the next case in the code. An example:

```cpp
while( x < 100 ) {
    if( x < 0 )
        break;
    cout << x << endl;
    x++;
}
```

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 goto statement to break out of the loop.

Related topics:
- continue
- do
- for
- goto
- switch
- while
The `case` keyword is used to test a variable against a certain value in a `switch` statement.

*Related topics:*
- `default` `switch`
catch

The catch statement handles exceptions generated by the throw statement.

*Related topics:*

- throw
- try
The char keyword is used to declare character variables. For more information about variable types, see the data types page.

Related topics:
bool
double
float
int
long
short
signed
unsigned
void
wchar_t
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:

```cpp
class Date {
    int Day;
    int Month;
    int Year;
    public:
        void display();
};
```
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:

```cpp
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 `const`.

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:

```cpp
for( int i = 0; i < 21; i++) {
    if( i == 10 ) {
        continue;
    }
    cout << i << " ";
}
```

Related topics:
- **break**
- **do**
- **for**
- **while**
default

A default case in the switch statement.

Related topics:
case switch
**delete**

**Syntax:**

```cpp
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 `new`. The second form of delete should be used to delete an array.

**Related topics:**
(Standard C Memory) [free](#)
(Standard C Memory) [malloc](#)
[new](#)
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 data types 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`
The else keyword is used as an alternative case for the if statement.

*Related topics:*

*if*
### enum

**Syntax:**

```cpp
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:

```cpp
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
...
ColorT c1 = indigo;
if( c1 == indigo ) {
    cout << "c1 is indigo" << endl;
}
```

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:

```cpp
enum ColorT { red = 10, blue = 15, green };  
...
ColorT c = green;        
cout << "c is " << c << endl;
```

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:

```cpp
enum ColorT { red = 10, blue = 15, green };  
...
enum ColorT c = green;  // note the additional 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:

```cpp
class foo {
public:
    explicit foo(int a)
        : a_(a)
    {
    }
    int a_;  // explicit call to explicit constructor.
};

int bar(const foo &f) {
    return f.a_;  // with automatic conversion from float to int.
}
```

bar(1);  // fails because an implicit conversion from int to foo
          // is forbidden by explicit.

bar(foo(1));  // works -- explicit call to explicit constructor.

bar(foo(1.0));  // works -- explicit call to explicit constructor
```
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++.
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
The float keyword is used to declare floating-point variables. Also see the data types page.

Related topics:
bool char
double
int
long
short
signed
unsigned
void
wchar_t
for

Syntax:

```cpp
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:

```cpp
for( int i = 0; i < 10; i++ ) {
    cout << "i is " << i << endl;
}
int j, k;
for( j = 0, k = 10;
    j < k;
    j++, k-- ) {
    cout << "j is " << j << " and k is " << k << endl;
}
for( ; ; ) {
    // loop forever!
}
```

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:*

```plaintext
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 considered harmful, it can occasionally be useful. For example, it may be cleaner to use a `goto` to break out of a deeply-nested `for` loop, compared to the space and time that extra `break` logic would consume.

*Related topics:*

- `break`
if

Syntax:

```cpp
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 else statement is present, then the statement list within the else block will be executed. All of the else blocks are optional.

Related topics:
- else
- for
- switch
- while
inline

Syntax:

```cpp
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.
The `int` keyword is used to declare integer variables. Also see the data types page.

*Related topics:* `bool char double float long short signed unsigned void wchar_t`
The long keyword is a data type modifier that is used to declare long integer variables. For more information on long, see the data types page.

Related topics:
- bool
- char
- double
- float
- int
- short
- signed
- void
mutable

The mutable keyword overrides any enclosing `const` statement. A mutable member of a `const` 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 `using` keyword.

Example code:

```
namespace CartoonNameSpace {
    int HomersAge;
    void incrementHomersAge() {
        HomersAge++;
    }
}

int main() {
    CartoonNameSpace::HomersAge = 39;
    CartoonNameSpace::incrementHomersAge();
    cout << CartoonNameSpace::HomersAge << endl;
}
```

Related topics:

using
new

Syntax:

```c
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:

```c
class Base {
    public:
        Base() { }

        void *operator new( unsigned int size, string str ) {
            cout << "Logging an allocation of " << size << " bytes for new object '" << str << "'
            return malloc( size );
        }

        int var;
        double var2;
    }

    ...}

Base* b = new ("Base instance 1") Base;
```

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) free
(Standard C Memory) malloc
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:*  
[this](#)
Private data of a class can only be accessed by members of that class, except when friend is used. The private keyword can also be used to inherit a base class privately, which causes all public and protected 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 public and protected members of the base class to become protected members of the derived class.

Related topics:
class private
class 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 protected members of the base class to become public and protected members of the derived class.

Related topics:
class private
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:

```cpp
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`
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 data types page.

Related topics:

bool char
double
float
int
long
signed
unsigned
void
wchar_t
The signed keyword is a data type modifier that is usually used to declare signed char variables. See the data types page.

Related topics:
- bool
- char
- double
- float
- int
- long
- short
- unsigned
- void
- wchar_t
**sizeof**

The sizeof 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 `<climits>` header file.

For example, the following code uses sizeof to display the sizes of a number of variables:

```cpp
struct EmployeeRecord {
    int ID;
    int age;
    double salary;
    EmployeeRecord* boss;
};
...
int i;
float f;
double d;
char c;
EmployeeRecord er;

cout << "sizeof(int): " << sizeof(int) << endl
     << "sizeof(float): " << sizeof(float) << endl
     << "sizeof(double): " << sizeof(double) << endl
     << "sizeof(char): " << sizeof(char) << endl
     << "sizeof(EmployeeRecord): " << sizeof(EmployeeRecord) << 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 sizeof 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 sizeof with a variable type (e.g. sizeof(int)).

Related topics:
C/C++ Data Types
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 class, all instantiations of that class share one copy of the variable.
static_cast

**Syntax:**

```cpp
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`
struct

Syntax:

```c
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 public rather than private. In C, structs can only contain data and are not permitted to have inheritance lists. For example:

```c
struct Date {
  int Day;
  int Month;
  int Year;
};
```

Related topics:

class union
switch

Syntax:

```cpp
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 `if()...else if()...else if()...` statements. `break` statements are required between each `case` statement, otherwise execution will "fall-through" to the next `case` statement. The `default` case is optional. If provided, it will match any case not explicitly covered by the preceding cases in the switch statement. For example:

```cpp
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:**

```cpp
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:

```cpp
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 << endl;
    genericSwap( num1, num2 );
    cout << "After, num1 is " << num1 << " and num2 is " << num2 << endl;
    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
The `this` keyword is a pointer to the current object. All member functions of a `class` have a `this` pointer.

*Related topics:* `class operator`
**throw**

**Syntax:**

```cpp
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 `try` and `catch` statements, the C++ exception handling system gives programmers an elegant mechanism for error recovery.

You will generally use a `try` 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 `try` block and into one of the `catch` blocks. For example:

```cpp
try {
    cout << "Before throwing exception" << endl;
    throw 42;
    cout << "Shouldn't ever see this" << endl;
}
catch( int error ) {
    cout << "Error: caught exception " << error << endl;
}
```

**Related topics:**

- `catch`
- `try`
The Boolean value of "true".

*Related topics:*

`bool false`
try

The try statement attempts to execute exception-generating code. See the throw statement for more details.

Related topics:
catch 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
pui_t data1;
unsigned int* data2;
```
**typeid**

*Syntax:*

```cpp
typeid( object );
```

The typeid operator returns a reference to a type_info object that describes `object`. 
The typename keyword can be used to describe an undefined type or in place of the class keyword in a template declaration.

*Related topics:*

class template
union

Syntax:

```cpp
union union-name {
    public-members-list;
    private:
    private-members-list;
} object-list;
```

A union is like a class, except that all members of a union share the same memory location and are by default public rather than private. For example:

```cpp
union Data {
    int i;
    char c;
};
```

Related topics:
class struct
The unsigned keyword is a data type modifier that is usually used to declare unsigned `int` variables. See the data types page.

Related topics:
- bool
- char
- double
- float
- int
- short
- signed
- void
- wchar_t
The using keyword is used to import a namespace (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 preceeding "std::".

```cpp
using namespace std;
```

Alternatively, the next code snippet just imports a single element of the std namespace into the current namespace:

```cpp
using std::cout;
```

Related topics:

namespace
virtual

Syntax:

```cpp
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:

```cpp
class Base {
public:
  void nonVirtualFunc() {
    cout << "Base: non-virtual function" << endl;
  }
  virtual void virtualFunc() {
    cout << "Base: virtual function" << endl;
  }
};
```
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
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 data types 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.
The keyword wchar_t is used to declare wide character variables. Also see the data types 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
do continue
for continue
if continue
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 `flags()`, `setf()`, and `unsetf()` functions. For example,

```
cout.setf(ios::left);
```

turns on left justification for all output directed to `cout`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolalpha</td>
<td>Boolean values can be input/output using the words &quot;true&quot; and &quot;false&quot;.</td>
</tr>
<tr>
<td>dec</td>
<td>Numeric values are displayed in decimal.</td>
</tr>
<tr>
<td>fixed</td>
<td>Display floating point values using normal notation (as opposed to scientific).</td>
</tr>
<tr>
<td>hex</td>
<td>Numeric values are displayed in hexadecimal.</td>
</tr>
<tr>
<td>internal</td>
<td>If a numeric value is padded to fill a field, spaces are inserted between the sign and base character.</td>
</tr>
<tr>
<td>left</td>
<td>Output is left justified.</td>
</tr>
<tr>
<td>oct</td>
<td>Numeric values are displayed in octal.</td>
</tr>
<tr>
<td>right</td>
<td>Output is right justified.</td>
</tr>
<tr>
<td>scientific</td>
<td>Display floating point values using scientific notation.</td>
</tr>
<tr>
<td>showbase</td>
<td>Display the base of all numeric values.</td>
</tr>
<tr>
<td>showpoint</td>
<td>Display a decimal and extra zeros, even when not needed.</td>
</tr>
<tr>
<td>showpos</td>
<td>Display a leading plus sign before positive numeric values.</td>
</tr>
<tr>
<td>skipws</td>
<td>Discard whitespace characters (spaces, tabs, newlines) when reading from a stream.</td>
</tr>
<tr>
<td>unitbuf</td>
<td>Flush the buffer after each insertion.</td>
</tr>
<tr>
<td>uppercase</td>
<td>Display the &quot;e&quot; of scientific notation and the &quot;x&quot; of hexadecimal notation as capital letters.</td>
</tr>
</tbody>
</table>
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:

```cpp
cout << dec;
```

<table>
<thead>
<tr>
<th>Manipulator defined in <code>&lt;iostream&gt;</code></th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolalpha</code></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><code>dec</code></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><code>endl</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ends</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fixed</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>flush</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>hex</code></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><code>internal</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>left</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>noboolalpha</code></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><code>noshownum_base</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>noshownum_point</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>noshownum_pos</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>noskipws</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nounitbuf</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nouppercase</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>oct</code></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><code>right</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>scientific</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>showbase</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>showpoint</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>showpos</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>skipws</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: `X` indicates that the flag is turned on or off by the manipulator.
<table>
<thead>
<tr>
<th><strong>Manipulator</strong></th>
<th><strong>Description</strong></th>
<th><strong>Input</strong></th>
<th><strong>Output</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>unitbuf</td>
<td>Turns on the unitbuf flag</td>
<td></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>uppercase</td>
<td>Turns on the uppercase flag</td>
<td></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>ws</td>
<td>Skip any leading whitespace</td>
<td></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td><strong>Manipulators defined in <code>&lt;iomanip&gt;</code></strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>resetiosflags( long f )</td>
<td>Turn off the flags specified by f</td>
<td><strong>X</strong></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>setbase( int base )</td>
<td>Sets the number base to <code>base</code></td>
<td></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>setfill( int ch )</td>
<td>Sets the fill character to <code>ch</code></td>
<td></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>setiosflags( long f )</td>
<td>Turn on the flags specified by <code>f</code></td>
<td><strong>X</strong></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>setprecision( int p )</td>
<td>Sets the number of digits of precision</td>
<td></td>
<td><strong>X</strong></td>
</tr>
<tr>
<td>setw( int w )</td>
<td>Sets the field width to <code>w</code></td>
<td></td>
<td><strong>X</strong></td>
</tr>
</tbody>
</table>
**State flags**

The I/O stream state flags tell you the current state of an I/O stream. The flags are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbit</td>
<td>a fatal error has occurred</td>
</tr>
<tr>
<td>eofbit</td>
<td>EOF has been found</td>
</tr>
<tr>
<td>failbit</td>
<td>a nonfatal error has occurred</td>
</tr>
<tr>
<td>goodbit</td>
<td>no errors have occurred</td>
</tr>
</tbody>
</table>
Mode flags

The I/O stream mode flags allow you to access files in different ways. The flags are:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios::app</td>
<td>append output</td>
</tr>
<tr>
<td>ios::ate</td>
<td>seek to EOF when opened</td>
</tr>
<tr>
<td>ios::binary</td>
<td>open the file in binary mode</td>
</tr>
<tr>
<td>ios::in</td>
<td>open the file for reading</td>
</tr>
<tr>
<td>ios::out</td>
<td>open the file for writing</td>
</tr>
<tr>
<td>ios::trunc</td>
<td>overwrite the existing file</td>
</tr>
</tbody>
</table>
C++ Iterators

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

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

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:

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

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

There are different measurements of the speed of any given algorithm. Given an input size of \( N \), they can be described as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Speed</th>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponential time</td>
<td>slow</td>
<td>takes an amount of time proportional to a constant raised to the ( N )th power</td>
<td>( K^N )</td>
</tr>
<tr>
<td>polynomial time</td>
<td>fast</td>
<td>takes an amount of time proportional to ( N ) raised to some constant power</td>
<td>( N^K )</td>
</tr>
<tr>
<td>linear time</td>
<td>faster</td>
<td>takes an amount of time directly proportional to ( N )</td>
<td>( K \times N )</td>
</tr>
<tr>
<td>logarithmic time</td>
<td>much faster</td>
<td>takes an amount of time proportional to the logarithm of ( N )</td>
<td>( K \times \log(N) )</td>
</tr>
<tr>
<td>constant time</td>
<td>fastest</td>
<td>takes a fixed amount of time, no matter how large the input is</td>
<td>( K )</td>
</tr>
</tbody>
</table>
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.

Display all entries for C++ Bitsets on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Bitset Constructors</th>
<th>create new bitsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitset Operators</td>
<td>compare and assign bitsets</td>
</tr>
<tr>
<td>any</td>
<td>true if any bits are set</td>
</tr>
<tr>
<td>count</td>
<td>returns the number of set bits</td>
</tr>
<tr>
<td>flip</td>
<td>reverses the bitset</td>
</tr>
<tr>
<td>none</td>
<td>true if no bits are set</td>
</tr>
<tr>
<td>reset</td>
<td>sets bits to zero</td>
</tr>
<tr>
<td>set</td>
<td>sets bits</td>
</tr>
<tr>
<td>size</td>
<td>number of bits that the bitset can hold</td>
</tr>
<tr>
<td>test</td>
<td>returns the value of a given bit</td>
</tr>
<tr>
<td>to_string</td>
<td>string representation of the bitset</td>
</tr>
<tr>
<td>to_ulong</td>
<td>returns an integer representation of the bitset</td>
</tr>
</tbody>
</table>
C++ Vectors

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

Display all entries for C++ Vectors on one page, or view entries individually:

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

```c++
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.

Display all entries for C++ Double-ended Queues on one page, or view entries individually:

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

Display all entries for C++ Lists on one page, or view entries individually:

<table>
<thead>
<tr>
<th>List constructors</th>
<th>create lists and initialize them with some data</th>
</tr>
</thead>
<tbody>
<tr>
<td>List operators</td>
<td>assign and compare lists</td>
</tr>
<tr>
<td>assign</td>
<td>assign elements to a list</td>
</tr>
<tr>
<td>back</td>
<td>returns a reference to last element of a list</td>
</tr>
<tr>
<td>begin</td>
<td>returns an iterator to the beginning of the list</td>
</tr>
<tr>
<td>clear</td>
<td>removes all elements from the list</td>
</tr>
<tr>
<td>empty</td>
<td>true if the list has no elements</td>
</tr>
<tr>
<td>end</td>
<td>returns an iterator just past the last element of a list</td>
</tr>
<tr>
<td>erase</td>
<td>removes elements from a list</td>
</tr>
<tr>
<td>front</td>
<td>returns a reference to the first element of a list</td>
</tr>
<tr>
<td>insert</td>
<td>inserts elements into the list</td>
</tr>
<tr>
<td>max_size</td>
<td>returns the maximum number of elements that the list can hold</td>
</tr>
<tr>
<td>merge</td>
<td>merge two lists</td>
</tr>
<tr>
<td>pop_back</td>
<td>removes the last element of a list</td>
</tr>
<tr>
<td>pop_front</td>
<td>removes the first element of the list</td>
</tr>
<tr>
<td>push_back</td>
<td>add an element to the end of the list</td>
</tr>
<tr>
<td>push_front</td>
<td>add an element to the front of the list</td>
</tr>
<tr>
<td>rbegin</td>
<td>returns a reverse_iterator to the end of the list</td>
</tr>
<tr>
<td>remove</td>
<td>removes elements from a list</td>
</tr>
<tr>
<td>remove_if</td>
<td>removes elements conditionally</td>
</tr>
<tr>
<td>rend</td>
<td>returns a reverse_iterator to the beginning of the list</td>
</tr>
<tr>
<td>resize</td>
<td>change the size of the list</td>
</tr>
<tr>
<td>reverse</td>
<td>reverse the list</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>returns the number of items in the list</td>
</tr>
<tr>
<td><strong>sort</strong></td>
<td>sorts a list into ascending order</td>
</tr>
<tr>
<td><strong>splice</strong></td>
<td>merge two lists in <em>constant time</em></td>
</tr>
<tr>
<td><strong>swap</strong></td>
<td>swap the contents of this list with another</td>
</tr>
<tr>
<td><strong>unique</strong></td>
<td>removes consecutive duplicate elements</td>
</tr>
</tbody>
</table>
The C++ Set is an associative container that contains a sorted set of unique objects.

**Display all entries** for C++ Sets on one page, or view entries individually:

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

Display all entries for C++ Multisets on one page, or view entries individually:

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

C++ Maps are sorted associative containers that contain unique key/value pairs. For example, you could create a map that associates a `string` with an integer, and then use that map to associate the number of days in each month with the name of each month.

Display all entries for C++ Maps on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Map constructors &amp; destructors</th>
<th>default methods to allocate, copy, and deallocate maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map operators</td>
<td>assign, compare, and access elements of a map</td>
</tr>
<tr>
<td><code>begin</code></td>
<td>returns an iterator to the beginning of the map</td>
</tr>
<tr>
<td><code>clear</code></td>
<td>removes all elements from the map</td>
</tr>
<tr>
<td><code>count</code></td>
<td>returns the number of elements matching a certain key</td>
</tr>
<tr>
<td><code>empty</code></td>
<td>true if the map has no elements</td>
</tr>
<tr>
<td><code>end</code></td>
<td>returns an iterator just past the last element of a map</td>
</tr>
<tr>
<td><code>equal_range</code></td>
<td>returns iterators to the first and just past the last elements matching a specific key</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>removes elements from a map</td>
</tr>
<tr>
<td><code>find</code></td>
<td>returns an iterator to specific elements</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>insert items into a map</td>
</tr>
<tr>
<td><code>key_comp</code></td>
<td>returns the function that compares keys</td>
</tr>
<tr>
<td><code>lower_bound</code></td>
<td>returns an iterator to the first element greater than or equal to a certain value</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>returns the maximum number of elements that the map can hold</td>
</tr>
<tr>
<td><code>rbegin</code></td>
<td>returns a <code>reverse_iterator</code> to the end of the map</td>
</tr>
<tr>
<td><code>rend</code></td>
<td>returns a <code>reverse_iterator</code> to the beginning of the map</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>returns the number of items in the map</td>
</tr>
<tr>
<td><strong>swap</strong></td>
<td>swap the contents of this map with another</td>
</tr>
<tr>
<td><strong>upper_bound</strong></td>
<td>returns an iterator to the first element greater than a certain value</td>
</tr>
<tr>
<td><strong>value_comp</strong></td>
<td>returns the function that compares values</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Multimap constructors &amp; destructors</th>
<th>default methods to allocate, copy, and deallocate multimaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multimap operators</td>
<td>assign and compare multimaps</td>
</tr>
<tr>
<td>begin</td>
<td>returns an iterator to the beginning of the multimap</td>
</tr>
<tr>
<td>clear</td>
<td>removes all elements from the multimap</td>
</tr>
<tr>
<td>count</td>
<td>returns the number of elements matching a certain key</td>
</tr>
<tr>
<td>empty</td>
<td>true if the multimap has no elements</td>
</tr>
<tr>
<td>end</td>
<td>returns an iterator just past the last element of a multimap</td>
</tr>
<tr>
<td>equal_range</td>
<td>returns iterators to the first and just past the last elements matching a specific key</td>
</tr>
<tr>
<td>erase</td>
<td>removes elements from a multimap</td>
</tr>
<tr>
<td>find</td>
<td>returns an iterator to specific elements</td>
</tr>
<tr>
<td>insert</td>
<td>inserts items into a multimap</td>
</tr>
<tr>
<td>key_comp</td>
<td>returns the function that compares keys</td>
</tr>
<tr>
<td>lower_bound</td>
<td>returns an iterator to the first element greater than or equal to a certain value</td>
</tr>
<tr>
<td>max_size</td>
<td>returns the maximum number of elements that the multimap can hold</td>
</tr>
<tr>
<td>rbegin</td>
<td>returns a reverse_iterator to the end of the multimap</td>
</tr>
<tr>
<td>rend</td>
<td>returns a reverse_iterator to the beginning of the multimap</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------</td>
</tr>
<tr>
<td>size</td>
<td>returns the number of items in the multimap</td>
</tr>
<tr>
<td>swap</td>
<td>swap the contents of this multimap with another</td>
</tr>
<tr>
<td>upper_bound</td>
<td>returns an iterator to the first element greater than a certain value</td>
</tr>
<tr>
<td>value_comp</td>
<td>returns the function that compares values</td>
</tr>
</tbody>
</table>
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 `String Stream` class.

Some of the behavior of the C++ I/O streams (precision, justification, etc) may be modified by manipulating various `io stream format flags`.

Here are some examples of what you can do with C++ I/O.

Display all entries for C++ I/O on one page, or view entries individually:

<table>
<thead>
<tr>
<th>I/O Constructors</th>
<th>constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bad</code></td>
<td>true if an error occurred</td>
</tr>
<tr>
<td><code>clear</code></td>
<td>clear and set status flags</td>
</tr>
<tr>
<td><code>close</code></td>
<td>close a stream</td>
</tr>
<tr>
<td><code>eof</code></td>
<td>true if at the end-of-file</td>
</tr>
<tr>
<td><code>fail</code></td>
<td>true if an error occurred</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>manipulate the default fill character</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>access or manipulate <code>io stream format flags</code></td>
</tr>
<tr>
<td><code>flush</code></td>
<td>empty the buffer</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td><code>gcount</code></td>
<td>number of characters read during last input</td>
</tr>
<tr>
<td><code>get</code></td>
<td>read characters</td>
</tr>
<tr>
<td><code>getline</code></td>
<td>read a line of characters</td>
</tr>
<tr>
<td><code>good</code></td>
<td>true if no errors have occurred</td>
</tr>
<tr>
<td><code>ignore</code></td>
<td>read and discard characters</td>
</tr>
<tr>
<td><code>open</code></td>
<td>open a new stream</td>
</tr>
<tr>
<td><code>peek</code></td>
<td>check the next input character</td>
</tr>
<tr>
<td><code>precision</code></td>
<td>manipulate the precision of a stream</td>
</tr>
<tr>
<td><code>put</code></td>
<td>write characters</td>
</tr>
<tr>
<td><code>putback</code></td>
<td>return characters to a stream</td>
</tr>
<tr>
<td><code>rdstate</code></td>
<td>returns the state flags of the stream</td>
</tr>
<tr>
<td><code>read</code></td>
<td>read data into a buffer</td>
</tr>
<tr>
<td><code>seekg</code></td>
<td>perform random access on an input stream</td>
</tr>
<tr>
<td><code>seekp</code></td>
<td>perform random access on output streams</td>
</tr>
<tr>
<td><code>setf</code></td>
<td>set format flags</td>
</tr>
<tr>
<td><code>sync_with_stdio</code></td>
<td>synchronize with standard I/O</td>
</tr>
<tr>
<td><code>tellg</code></td>
<td>read input stream pointers</td>
</tr>
<tr>
<td><code>tellp</code></td>
<td>read output stream pointers</td>
</tr>
<tr>
<td><code>unsetf</code></td>
<td>clear io stream format flags</td>
</tr>
<tr>
<td><code>width</code></td>
<td>access and manipulate the minimum field width</td>
</tr>
<tr>
<td><code>write</code></td>
<td>write characters</td>
</tr>
</tbody>
</table>
C++ Priority Queues

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

Display all entries for C++ Priority Queues on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Priority queue constructors</th>
<th>construct a new priority queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>true if the priority queue has no elements</td>
</tr>
<tr>
<td>pop</td>
<td>removes the top element of a priority queue</td>
</tr>
<tr>
<td>push</td>
<td>adds an element to the end of the priority queue</td>
</tr>
<tr>
<td>size</td>
<td>returns the number of items in the priority queue</td>
</tr>
<tr>
<td>top</td>
<td>returns the top element of the priority queue</td>
</tr>
</tbody>
</table>
C++ Queues

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

Display all entries for C++ Queues on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Queue constructor</strong></td>
<td>construct a new queue</td>
</tr>
<tr>
<td><strong>back</strong></td>
<td>returns a reference to last element of a queue</td>
</tr>
<tr>
<td><strong>empty</strong></td>
<td>true if the queue has no elements</td>
</tr>
<tr>
<td><strong>front</strong></td>
<td>returns a reference to the first element of a queue</td>
</tr>
<tr>
<td><strong>pop</strong></td>
<td>removes the first element of a queue</td>
</tr>
<tr>
<td><strong>push</strong></td>
<td>adds an element to the end of the queue</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>returns the number of items in the queue</td>
</tr>
</tbody>
</table>
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.

Display all entries for C++ Stacks on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Stack constructors</th>
<th>construct a new stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>true if the stack has no elements</td>
</tr>
<tr>
<td>pop</td>
<td>removes the top element of a stack</td>
</tr>
<tr>
<td>push</td>
<td>adds an element to the top of the stack</td>
</tr>
<tr>
<td>size</td>
<td>returns the number of items in the stack</td>
</tr>
<tr>
<td>top</td>
<td>returns the top element of the stack</td>
</tr>
</tbody>
</table>
C++ Strings

**Display all entries** for C++ Strings on one page, or view entries individually:

<table>
<thead>
<tr>
<th><strong>String constructors</strong></th>
<th>create strings from arrays of characters and other strings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String operators</strong></td>
<td>concatenate strings, assign strings, use strings for I/O, compare strings</td>
</tr>
<tr>
<td><strong>append</strong></td>
<td>append characters and strings onto a string</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>give a string values from strings of characters and other C++ strings</td>
</tr>
<tr>
<td><strong>at</strong></td>
<td>returns the character at a specific location</td>
</tr>
<tr>
<td><strong>begin</strong></td>
<td>returns an iterator to the beginning of the string</td>
</tr>
<tr>
<td><strong>c_str</strong></td>
<td>returns a non-modifiable standard C character array version of the string</td>
</tr>
<tr>
<td><strong>capacity</strong></td>
<td>returns the number of characters that the string can hold</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>removes all characters from the string</td>
</tr>
<tr>
<td><strong>compare</strong></td>
<td>compares two strings</td>
</tr>
<tr>
<td><strong>copy</strong></td>
<td>copies characters from a string into an array</td>
</tr>
<tr>
<td><strong>data</strong></td>
<td>returns a pointer to the first character of a string</td>
</tr>
<tr>
<td><strong>empty</strong></td>
<td>true if the string has no characters</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td>returns an iterator just past the last character of a string</td>
</tr>
<tr>
<td><strong>erase</strong></td>
<td>removes characters from a string</td>
</tr>
<tr>
<td><strong>find</strong></td>
<td>find characters in the string</td>
</tr>
<tr>
<td><strong>find_first_not_of</strong></td>
<td>find first absence of characters</td>
</tr>
<tr>
<td><strong>find_first_of</strong></td>
<td>find first occurrence of characters</td>
</tr>
<tr>
<td><strong>find_last_not_of</strong></td>
<td>find last absence of characters</td>
</tr>
<tr>
<td><strong>find_last_of</strong></td>
<td>find last occurrence of characters</td>
</tr>
<tr>
<td><strong>getline</strong></td>
<td>read data from an I/O stream into a string</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>insert characters into a string</td>
</tr>
<tr>
<td><strong>length</strong></td>
<td>returns the length of the string</td>
</tr>
<tr>
<td><strong>max_size</strong></td>
<td>returns the maximum number of characters that the string can hold</td>
</tr>
<tr>
<td><strong>push_back</strong></td>
<td>add a character to the end of the string</td>
</tr>
<tr>
<td><strong>rbegin</strong></td>
<td>returns a reverse_iterator to the end of the string</td>
</tr>
<tr>
<td><strong>rend</strong></td>
<td>returns a reverse_iterator to the beginning of the string</td>
</tr>
<tr>
<td><strong>replace</strong></td>
<td>replace characters in the string</td>
</tr>
<tr>
<td><strong>reserve</strong></td>
<td>sets the minimum capacity of the string</td>
</tr>
<tr>
<td><strong>resize</strong></td>
<td>change the size of the string</td>
</tr>
<tr>
<td><strong>rfind</strong></td>
<td>find the last occurrence of a substring</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>returns the number of items in the string</td>
</tr>
<tr>
<td><strong>substr</strong></td>
<td>returns a certain substring</td>
</tr>
<tr>
<td><strong>swap</strong></td>
<td>swap the contents of this string with another</td>
</tr>
</tbody>
</table>
### C++ Algorithms

Display all entries for C++ Algorithms on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate</td>
<td>sum up a range of elements</td>
</tr>
<tr>
<td>adjacent_difference</td>
<td>compute the differences between adjacent elements in a range</td>
</tr>
<tr>
<td>adjacent_find</td>
<td>finds two items that are adjacent to each other</td>
</tr>
<tr>
<td>binary_search</td>
<td>determine if an element exists in a certain range</td>
</tr>
<tr>
<td>copy</td>
<td>copy some range of elements to a new location</td>
</tr>
<tr>
<td>copy_backward</td>
<td>copy a range of elements in backwards order</td>
</tr>
<tr>
<td>copy_n</td>
<td>copy N elements</td>
</tr>
<tr>
<td>count</td>
<td>return the number of elements matching a given value</td>
</tr>
<tr>
<td>count_if</td>
<td>return the number of elements for which a predicate is true</td>
</tr>
<tr>
<td>equal</td>
<td>determine if two sets of elements are the same</td>
</tr>
<tr>
<td>equal_range</td>
<td>search for a range of elements that are all equal to a certain element</td>
</tr>
<tr>
<td>fill</td>
<td>assign a range of elements a certain value</td>
</tr>
<tr>
<td>fill_n</td>
<td>assign a value to some number of elements</td>
</tr>
<tr>
<td>find</td>
<td>find a value in a given range</td>
</tr>
<tr>
<td>find_end</td>
<td>find the last sequence of elements in a certain range</td>
</tr>
<tr>
<td>find_first_of</td>
<td>search for any one of a set of elements</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>find_if</td>
<td>find the first element for which a certain predicate is true</td>
</tr>
<tr>
<td>for_each</td>
<td>apply a function to a range of elements</td>
</tr>
<tr>
<td>generate</td>
<td>saves the result of a function in a range</td>
</tr>
<tr>
<td>generate_n</td>
<td>saves the result of N applications of a function</td>
</tr>
<tr>
<td>includes</td>
<td>returns true if one set is a subset of another</td>
</tr>
<tr>
<td>inner_product</td>
<td>compute the inner product of two ranges of elements</td>
</tr>
<tr>
<td>inplace_merge</td>
<td>merge two ordered ranges in-place</td>
</tr>
<tr>
<td>is_heap</td>
<td>returns true if a given range is a heap</td>
</tr>
<tr>
<td>is_sorted</td>
<td>returns true if a range is sorted in ascending order</td>
</tr>
<tr>
<td>iter_swap</td>
<td>swaps the elements pointed to by two iterators</td>
</tr>
<tr>
<td>lexicographical_compare</td>
<td>returns true if one range is lexicographically less than another</td>
</tr>
<tr>
<td>lexicographical_compare_3way</td>
<td>determines if one range is lexicographically less than or greater than another</td>
</tr>
<tr>
<td>lower_bound</td>
<td>search for the first place that a value can be inserted while preserving order</td>
</tr>
<tr>
<td>make_heap</td>
<td>creates a heap out of a range of elements</td>
</tr>
<tr>
<td>max</td>
<td>returns the larger of two elements</td>
</tr>
<tr>
<td>max_element</td>
<td>returns the largest element in a range</td>
</tr>
<tr>
<td>merge</td>
<td>merge two sorted ranges</td>
</tr>
<tr>
<td>min</td>
<td>returns the smaller of two elements</td>
</tr>
<tr>
<td>min_element</td>
<td>returns the smallest element in a range</td>
</tr>
<tr>
<td>mismatch</td>
<td>finds the first position where two ranges differ</td>
</tr>
<tr>
<td>next_permutation</td>
<td>generates the next greater lexicographic permutation of a range of elements</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>nth_element</td>
<td>Put one element in its sorted location and make sure that no elements to its left are greater than any elements to its right</td>
</tr>
<tr>
<td>partial_sort</td>
<td>Sort the first N elements of a range</td>
</tr>
<tr>
<td>partial_sort_copy</td>
<td>Copy and partially sort a range of elements</td>
</tr>
<tr>
<td>partial_sum</td>
<td>Compute the partial sum of a range of elements</td>
</tr>
<tr>
<td>partition</td>
<td>Divide a range of elements into two groups</td>
</tr>
<tr>
<td>pop_heap</td>
<td>Remove the largest element from a heap</td>
</tr>
<tr>
<td>prev_permutation</td>
<td>Generates the next smaller lexicographic permutation of a range of elements</td>
</tr>
<tr>
<td>push_heap</td>
<td>Add an element to a heap</td>
</tr>
<tr>
<td>random_sample</td>
<td>Randomly copy elements from one range to another</td>
</tr>
<tr>
<td>random_sample_n</td>
<td>Sample N random elements from a range</td>
</tr>
<tr>
<td>random_shuffle</td>
<td>Randomly re-order elements in some range</td>
</tr>
<tr>
<td>remove</td>
<td>Remove elements equal to certain value</td>
</tr>
<tr>
<td>remove_copy</td>
<td>Copy a range of elements omitting those that match a certain value</td>
</tr>
<tr>
<td>remove_copy_if</td>
<td>Create a copy of a range of elements, omitting any for which a predicate is true</td>
</tr>
<tr>
<td>remove_if</td>
<td>Remove all elements for which a predicate is true</td>
</tr>
<tr>
<td>replace</td>
<td>Replace every occurrence of some value in a range with another value</td>
</tr>
<tr>
<td>replace_copy</td>
<td>Copy a range, replacing certain elements with new ones</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>replace_copy_if</code></td>
<td>copy a range of elements, replacing those for which a predicate is true</td>
</tr>
<tr>
<td><code>replace_if</code></td>
<td>change the values of elements for which a predicate is true</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>reverse elements in some range</td>
</tr>
<tr>
<td><code>reverse_copy</code></td>
<td>create a copy of a range that is reversed</td>
</tr>
<tr>
<td><code>rotate</code></td>
<td>move the elements in some range to the left by some amount</td>
</tr>
<tr>
<td><code>rotate_copy</code></td>
<td>copy and rotate a range of elements</td>
</tr>
<tr>
<td><code>search</code></td>
<td>search for a range of elements</td>
</tr>
<tr>
<td><code>search_n</code></td>
<td>search for N consecutive copies of an element in some range</td>
</tr>
<tr>
<td><code>set_difference</code></td>
<td>computes the difference between two sets</td>
</tr>
<tr>
<td><code>set_intersection</code></td>
<td>computes the intersection of two sets</td>
</tr>
<tr>
<td><code>set_symmetric_difference</code></td>
<td>computes the symmetric difference between two sets</td>
</tr>
<tr>
<td><code>set_union</code></td>
<td>computes the union of two sets</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>sort a range into ascending order</td>
</tr>
<tr>
<td><code>sort_heap</code></td>
<td>turns a heap into a sorted range of elements</td>
</tr>
<tr>
<td><code>stable_partition</code></td>
<td>divide elements into two groups while preserving their relative order</td>
</tr>
<tr>
<td><code>stable_sort</code></td>
<td>sort a range of elements while preserving order between equal elements</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>swap the values of two objects</td>
</tr>
<tr>
<td><code>swap_ranges</code></td>
<td>swaps two ranges of elements</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>applies a function to a range of elements</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>remove consecutive duplicate elements in a range</td>
</tr>
<tr>
<td><code>unique_copy</code></td>
<td>create a copy of some range of elements that contains no consecutive duplicates</td>
</tr>
<tr>
<td><code>upper_bound</code></td>
<td>searches for the last possible location to insert an element into an ordered range</td>
</tr>
</tbody>
</table>
Miscellaneous C++

Display all entries for Miscellaneous C++ on one page, or view entries individually:

| auto_ptr | create pointers that automatically destroy objects |
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 C++ I/O functions for more information.

Display all entries for C++ String Streams on one page, or view entries individually:

<table>
<thead>
<tr>
<th><strong>Constructors</strong></th>
<th>create new string streams</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operators</strong></td>
<td>read from and write to string streams</td>
</tr>
<tr>
<td><strong>rdbuf</strong></td>
<td>get the buffer for a string stream</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>get or set the stream's string</td>
</tr>
</tbody>
</table>
C/C++ Pre-processor Commands

Display all entries for C/C++ Pre-processor Commands on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#, ##</td>
<td>manipulate strings</td>
</tr>
<tr>
<td>#define</td>
<td>define variables</td>
</tr>
<tr>
<td>#error</td>
<td>display an error message</td>
</tr>
<tr>
<td>#if, #ifdef, #ifndef, #else, #elif, #endif</td>
<td>conditional operators</td>
</tr>
<tr>
<td>#include</td>
<td>insert the contents of another file</td>
</tr>
<tr>
<td>#line</td>
<td>set line and file information</td>
</tr>
<tr>
<td>#pragma</td>
<td>implementation specific command</td>
</tr>
<tr>
<td>#undef</td>
<td>used to undefine variables</td>
</tr>
<tr>
<td><strong>Predefined preprocessor variables</strong></td>
<td>miscellaneous preprocessor variables</td>
</tr>
</tbody>
</table>
**Other Standard C Functions**

*Display all entries* for Other Standard C Functions on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>stops the program</td>
</tr>
<tr>
<td>assert</td>
<td>stops the program if an expression isn't true</td>
</tr>
<tr>
<td>atexit</td>
<td>sets a function to be called when the program exits</td>
</tr>
<tr>
<td>bsearch</td>
<td>perform a binary search</td>
</tr>
<tr>
<td>exit</td>
<td>stop the program</td>
</tr>
<tr>
<td>getenv</td>
<td>get enviornment information about a variable</td>
</tr>
<tr>
<td>longjmp</td>
<td>start execution at a certain point in the program</td>
</tr>
<tr>
<td>qsort</td>
<td>perform a quicksort</td>
</tr>
<tr>
<td>raise</td>
<td>send a signal to the program</td>
</tr>
<tr>
<td>rand</td>
<td>returns a pseudorandom number</td>
</tr>
<tr>
<td>setjmp</td>
<td>set execution to start at a certain point</td>
</tr>
<tr>
<td>signal</td>
<td>register a function as a signal handler</td>
</tr>
<tr>
<td>srand</td>
<td>initialize the random number generator</td>
</tr>
<tr>
<td>system</td>
<td>perform a system call</td>
</tr>
<tr>
<td>va_arg</td>
<td>use variable length parameter lists</td>
</tr>
</tbody>
</table>
### Standard C Math

Display all entries for Standard C Math on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>acos</td>
<td>arc cosine</td>
</tr>
<tr>
<td>asin</td>
<td>arc sine</td>
</tr>
<tr>
<td>atan</td>
<td>arc tangent</td>
</tr>
<tr>
<td>atan2</td>
<td>arc tangent, using signs to determine quadrants</td>
</tr>
<tr>
<td>ceil</td>
<td>the smallest integer not less than a certain value</td>
</tr>
<tr>
<td>cos</td>
<td>cosine</td>
</tr>
<tr>
<td>cosh</td>
<td>hyperbolic cosine</td>
</tr>
<tr>
<td>div</td>
<td>returns the quotient and remainder of a division</td>
</tr>
<tr>
<td>exp</td>
<td>returns &quot;e&quot; raised to a given power</td>
</tr>
<tr>
<td>fabs</td>
<td>absolute value for floating-point numbers</td>
</tr>
<tr>
<td>floor</td>
<td>returns the largest integer not greater than a given value</td>
</tr>
<tr>
<td>fmod</td>
<td>returns the remainder of a division</td>
</tr>
<tr>
<td>frexp</td>
<td>decomposes a number into scientific notation</td>
</tr>
<tr>
<td>labs</td>
<td>absolute value for long integers</td>
</tr>
<tr>
<td>ldexp</td>
<td>computes a number in scientific notation</td>
</tr>
<tr>
<td>ldiv</td>
<td>returns the quotient and remainder of a division, in long integer form</td>
</tr>
<tr>
<td>log</td>
<td>natural logarithm (to base e)</td>
</tr>
<tr>
<td>log10</td>
<td>common logarithm (to base 10)</td>
</tr>
<tr>
<td>modf</td>
<td>decomposes a number into integer and fractional parts</td>
</tr>
<tr>
<td>pow</td>
<td>returns a given number raised to another number</td>
</tr>
<tr>
<td>sin</td>
<td>sine</td>
</tr>
<tr>
<td>sinh</td>
<td>hyperbolic sine</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
<tr>
<td>tan</td>
<td>tangent</td>
</tr>
<tr>
<td>tanh</td>
<td>hyperbolic tangent</td>
</tr>
</tbody>
</table>
Display all entries for Standard C Date & Time on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asctime</code></td>
<td>a textual version of the time</td>
</tr>
<tr>
<td><code>clock</code></td>
<td>returns the amount of time that the program has been running</td>
</tr>
<tr>
<td><code>ctime</code></td>
<td>returns a specifically formatted version of the time</td>
</tr>
<tr>
<td><code>difftime</code></td>
<td>the difference between two times</td>
</tr>
<tr>
<td><code>gmtime</code></td>
<td>returns a pointer to the current Greenwich Mean Time</td>
</tr>
<tr>
<td><code>localtime</code></td>
<td>returns a pointer to the current time</td>
</tr>
<tr>
<td><code>mktime</code></td>
<td>returns the calendar version of a given time</td>
</tr>
<tr>
<td><code>setlocale</code></td>
<td>sets the current locale</td>
</tr>
<tr>
<td><code>strftime</code></td>
<td>returns individual elements of the date and time</td>
</tr>
<tr>
<td><code>time</code></td>
<td>returns the current calendar time of the system</td>
</tr>
</tbody>
</table>
Display all entries for Standard C String and Character on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atof</code></td>
<td>converts a string to a double</td>
</tr>
<tr>
<td><code>atoi</code></td>
<td>converts a string to an integer</td>
</tr>
<tr>
<td><code>atol</code></td>
<td>converts a string to a long</td>
</tr>
<tr>
<td><code>isalnum</code></td>
<td>true if a character is alphanumeric</td>
</tr>
<tr>
<td><code>isalpha</code></td>
<td>true if a character is alphabetic</td>
</tr>
<tr>
<td><code>isctlr</code></td>
<td>true if a character is a control character</td>
</tr>
<tr>
<td><code>isdigit</code></td>
<td>true if a character is a digit</td>
</tr>
<tr>
<td><code>isgraph</code></td>
<td>true if a character is a graphical character</td>
</tr>
<tr>
<td><code>islower</code></td>
<td>true if a character is lowercase</td>
</tr>
<tr>
<td><code>isprint</code></td>
<td>true if a character is a printing character</td>
</tr>
<tr>
<td><code>ispunct</code></td>
<td>true if a character is punctuation</td>
</tr>
<tr>
<td><code>isspace</code></td>
<td>true if a character is a space character</td>
</tr>
<tr>
<td><code>isupper</code></td>
<td>true if a character is an uppercase character</td>
</tr>
<tr>
<td><code>isxdigit</code></td>
<td>true if a character is a hexadecimal character</td>
</tr>
<tr>
<td><code>memchr</code></td>
<td>searches an array for the first occurrence of a character</td>
</tr>
<tr>
<td><code>memcmp</code></td>
<td>compares two buffers</td>
</tr>
<tr>
<td><code>memcpy</code></td>
<td>copies one buffer to another</td>
</tr>
<tr>
<td><code>memmove</code></td>
<td>moves one buffer to another</td>
</tr>
<tr>
<td><code>memset</code></td>
<td>fills a buffer with a character</td>
</tr>
<tr>
<td><code>strcat</code></td>
<td>concatenates two strings</td>
</tr>
<tr>
<td><code>strchr</code></td>
<td>finds the first occurrence of a character in a string</td>
</tr>
<tr>
<td><code>strcmp</code></td>
<td>compares two strings</td>
</tr>
<tr>
<td><code>strcoll</code></td>
<td>compares two strings in accordance to the current locale</td>
</tr>
<tr>
<td><code>strcpy</code></td>
<td>copies one string to another</td>
</tr>
<tr>
<td><code>strcspn</code></td>
<td>searches one string for any characters in another</td>
</tr>
<tr>
<td><code>strerror</code></td>
<td>returns a text version of a given error code</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>strlen</code></td>
<td>returns the length of a given string</td>
</tr>
<tr>
<td><code>strncat</code></td>
<td>concatenates a certain amount of characters of two strings</td>
</tr>
<tr>
<td><code>strncmp</code></td>
<td>compares a certain amount of characters of two strings</td>
</tr>
<tr>
<td><code>strncpy</code></td>
<td>copies a certain amount of characters from one string to another</td>
</tr>
<tr>
<td><code>strpbrk</code></td>
<td>finds the first location of any character in one string, in another string</td>
</tr>
<tr>
<td><code>strrchr</code></td>
<td>finds the last occurrence of a character in a string</td>
</tr>
<tr>
<td><code>strspn</code></td>
<td>returns the length of a substring of characters of a string</td>
</tr>
<tr>
<td><code>strstr</code></td>
<td>finds the first occurrence of a substring of characters</td>
</tr>
<tr>
<td><code>strtod</code></td>
<td>converts a string to a double</td>
</tr>
<tr>
<td><code>strtok</code></td>
<td>finds the next token in a string</td>
</tr>
<tr>
<td><code>strtol</code></td>
<td>converts a string to a long</td>
</tr>
<tr>
<td><code>strtoul</code></td>
<td>converts a string to an unsigned long</td>
</tr>
<tr>
<td><code>strxfrm</code></td>
<td>converts a substring so that it can be used by string comparison functions</td>
</tr>
<tr>
<td><code>tolower</code></td>
<td>converts a character to lowercase</td>
</tr>
<tr>
<td><code>toupper</code></td>
<td>converts a character to uppercase</td>
</tr>
</tbody>
</table>
Standard C Memory

Display all entries for Standard C Memory on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calloc</code></td>
<td>allocates and clears a two-dimensional chunk of memory</td>
</tr>
<tr>
<td><code>free</code></td>
<td>returns previously allocated memory to the operating system</td>
</tr>
<tr>
<td><code>malloc</code></td>
<td>allocates memory</td>
</tr>
<tr>
<td><code>realloc</code></td>
<td>changes the size of previously allocated memory</td>
</tr>
</tbody>
</table>
# Standard C I/O

Display all entries for Standard C I/O on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clearerr</code></td>
<td>clears errors</td>
</tr>
<tr>
<td><code>fclose</code></td>
<td>close a file</td>
</tr>
<tr>
<td><code>feof</code></td>
<td>true if at the end-of-file</td>
</tr>
<tr>
<td><code>ferror</code></td>
<td>checks for a file error</td>
</tr>
<tr>
<td><code>fflush</code></td>
<td>writes the contents of the output buffer</td>
</tr>
<tr>
<td><code>fgetc</code></td>
<td>get a character from a stream</td>
</tr>
<tr>
<td><code>fgetpos</code></td>
<td>get the file position indicator</td>
</tr>
<tr>
<td><code>fgets</code></td>
<td>get a string of characters from a stream</td>
</tr>
<tr>
<td><code>fopen</code></td>
<td>open a file</td>
</tr>
<tr>
<td><code>fprintf</code></td>
<td>print formatted output to a file</td>
</tr>
<tr>
<td><code>fputc</code></td>
<td>write a character to a file</td>
</tr>
<tr>
<td><code>fputs</code></td>
<td>write a string to a file</td>
</tr>
<tr>
<td><code>fread</code></td>
<td>read from a file</td>
</tr>
<tr>
<td><code>freopen</code></td>
<td>open an existing stream with a different name</td>
</tr>
<tr>
<td><code>fscanf</code></td>
<td>read formatted input from a file</td>
</tr>
<tr>
<td><code>fseek</code></td>
<td>move to a specific location in a file</td>
</tr>
<tr>
<td><code>fsetpos</code></td>
<td>move to a specific location in a file</td>
</tr>
<tr>
<td><code>ftell</code></td>
<td>returns the current file position indicator</td>
</tr>
<tr>
<td><code>fwrite</code></td>
<td>write to a file</td>
</tr>
<tr>
<td><code>getc</code></td>
<td>read a character from a file</td>
</tr>
<tr>
<td><code>getchar</code></td>
<td>read a character from <code>stdin</code></td>
</tr>
<tr>
<td><code>gets</code></td>
<td>read a string from <code>stdin</code></td>
</tr>
<tr>
<td><code>perror</code></td>
<td>displays a string version of the current error to <code>stderr</code></td>
</tr>
<tr>
<td><code>printf</code></td>
<td>write formatted output to <code>stdout</code></td>
</tr>
<tr>
<td><code>putc</code></td>
<td>write a character to a stream</td>
</tr>
<tr>
<td><code>putchar</code></td>
<td>write a character to <code>stdout</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>puts</code></td>
<td>write a string to <code>stdout</code></td>
</tr>
<tr>
<td><code>remove</code></td>
<td>erase a file</td>
</tr>
<tr>
<td><code>rename</code></td>
<td>rename a file</td>
</tr>
<tr>
<td><code>rewind</code></td>
<td>move the file position indicator to the beginning of a file</td>
</tr>
<tr>
<td><code>scanf</code></td>
<td>read formatted input from <code>stdin</code></td>
</tr>
<tr>
<td><code>setbuf</code></td>
<td>set the buffer for a specific stream</td>
</tr>
<tr>
<td><code>setvbuf</code></td>
<td>set the buffer and size for a specific stream</td>
</tr>
<tr>
<td><code>sprintf</code></td>
<td>write formatted output to a buffer</td>
</tr>
<tr>
<td><code>sscanf</code></td>
<td>read formatted input from a buffer</td>
</tr>
<tr>
<td><code>tmpfile</code></td>
<td>return a pointer to a temporary file</td>
</tr>
<tr>
<td><code>tmpnam</code></td>
<td>return a unique filename</td>
</tr>
<tr>
<td><code>ungetc</code></td>
<td>puts a character back into a stream</td>
</tr>
<tr>
<td><code>vprintf, vfprintf, and vsprintf</code></td>
<td>write formatted output with variable argument lists</td>
</tr>
</tbody>
</table>
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.
C/C++ Data Types

There are five data types for C: **void, int, float, double, and char**.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>associated with no data type</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>float</td>
<td>floating-point number</td>
</tr>
<tr>
<td>double</td>
<td>double precision floating-point number</td>
</tr>
<tr>
<td>char</td>
<td>character</td>
</tr>
</tbody>
</table>

C++ defines two more: **bool and wchar_t**.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Boolean value, true or false</td>
</tr>
<tr>
<td>wchar_t</td>
<td>wide character</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>bool</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsigned char</strong></td>
<td><strong>signed char</strong></td>
</tr>
<tr>
<td>int</td>
<td><strong>unsigned int</strong></td>
</tr>
<tr>
<td><strong>signed int</strong></td>
<td><strong>short int</strong></td>
</tr>
<tr>
<td><strong>unsigned short int</strong></td>
<td><strong>signed short int</strong></td>
</tr>
<tr>
<td><strong>long int</strong></td>
<td><strong>signed long int</strong></td>
</tr>
<tr>
<td><strong>unsigned long int</strong></td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td><strong>long double</strong></td>
</tr>
<tr>
<td>wchar_t</td>
<td></td>
</tr>
</tbody>
</table>
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 `sizeof` 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.
**Constant Escape Sequences**

The following escape sequences can be used to define certain special characters within strings:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'</code></td>
<td>Single quote</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>Double quote</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash</td>
</tr>
<tr>
<td><code>\nnn</code></td>
<td>Octal number (nnn)</td>
</tr>
<tr>
<td><code>\0</code></td>
<td>Null character (really just the octal number zero)</td>
</tr>
<tr>
<td><code>\a</code></td>
<td>Audible bell</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>Backspace</td>
</tr>
<tr>
<td><code>\f</code></td>
<td>Formfeed</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>Newline</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage return</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Horizontal tab</td>
</tr>
<tr>
<td><code>\v</code></td>
<td>Vertical tab</td>
</tr>
<tr>
<td><code>\xnnn</code></td>
<td>Hexadecimal number (nnn)</td>
</tr>
</tbody>
</table>

An example of this is contained in the following code (which assumes that the newline character generates complete newlines, i.e. on Unix systems):

```c
printf( "This\nis\na\ntest\n\nShe said, \"How are you?\"\n" );
```

which would display

```text
This
is
a
test

She said, "How are you?"
```
# C++ Operator Precedence

The operators at the top of this list are evaluated first.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>::</td>
<td>Scoping operator</td>
<td>Class::age = 2;</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>() [ ] -&gt; . ++ --</td>
<td>Grouping operator Array access from a pointer Member access from an object Post-increment Post-decrement</td>
<td>(a + b) / 4; array[4] = 2; ptr-&gt;age = 34; obj.age = 34; for( i = 0; i &lt; 10; i++ ) ... for( i = 10; i &gt; 0; i-- ) ...</td>
<td>left to right</td>
</tr>
<tr>
<td>3</td>
<td>! ~ ++ -- - + * &amp; (type) sizeof</td>
<td>Logical negation Bitwise complement Pre-increment Pre-decrement Unary minus Unary plus Dereference Address of Cast to a given type Return size in bytes</td>
<td>if( !done ) ... flags = ~flags; for( i = 0; i &lt; 10; ++i ) ... for( i = 10; i &gt; 0; - -i ) ... int i = -1; int i = +1; data = *ptr; address = &amp;obj; int i = (int) floatNum; int size = sizeof(floatNum);</td>
<td>right to left</td>
</tr>
<tr>
<td>4</td>
<td>- &gt; * . *</td>
<td>Member pointer selector Member object selector</td>
<td>ptr-&gt;*var = 24; obj.*var = 24;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>** operators</td>
<td>Operation</td>
<td>Example</td>
<td>Rule</td>
</tr>
<tr>
<td>---</td>
<td>--------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
<td>---------------</td>
</tr>
</tbody>
</table>
| 5 | */%           | Multiplication  | int \( i = 2 * 4; 
float \( f = 10 / 3; 
int \( rem = 4 \% 3; 
| left to right |
| 6 | + -          | Addition        | int \( i = 2 + 3; 
int \( i = 5 - 1; 
| left to right |
| 7 | << >>        | Bitwise shift   | int \( flags = 33 << 1; 
int \( flags = 33 >> 1; 
| left to right |
| 8 | < <= > >=    | Comparison      | if( \( i < 42 \) .... 
if( \( i <= 42 \) .... 
if( \( i > 42 \) .... 
if( \( i >= 42 \) .... 
| left to right |
| 9 | == !=        | Comparison      | if( \( i == 42 \) .... 
if( \( i != 42 \) .... 
| left to right |
| 10| &            | Bitwise AND     | flags = flags & 42; 
| left to right |
| 11| ^            | 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| ||           | Logical OR      | if( conditionA || conditionB ) .... 
| left to right |
|   |              | Ternary         | int \( i = (a > b) ? a : 
<p>|</p>
<table>
<thead>
<tr>
<th>15</th>
<th>? :</th>
<th>conditional (if-then-else)</th>
<th>b;</th>
<th>right to left</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>=</td>
<td>Assignment operator</td>
<td>int a = b;</td>
<td>right to left</td>
</tr>
<tr>
<td></td>
<td>+=</td>
<td>Increment and assign</td>
<td>a += 3;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-=</td>
<td>Decrement and assign</td>
<td>b -= 4;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*=</td>
<td>Multiply and assign</td>
<td>a *= 5;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/=</td>
<td>Divide and assign</td>
<td>a /= 2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%=</td>
<td>Modulo and assign</td>
<td>a %= 3;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;=</td>
<td>Bitwise AND and assign</td>
<td>flags &amp;=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^=</td>
<td>Bitwise exclusive OR and assign</td>
<td>flags ^=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=</td>
<td>Bitwise inclusive (normal) OR and assign</td>
<td>flags</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;=</td>
<td>Bitwise shift left and assign</td>
<td>flags &lt;&lt;=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;=</td>
<td>Bitwise shift right and assign</td>
<td>flags &gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>,</td>
<td>Sequential evaluation operator</td>
<td>for( i = 0, j = 0; i &lt; 10; i++, j++ ) ...</td>
<td>left to right</td>
</tr>
</tbody>
</table>

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:

```c++
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.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
<th>Hex</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>NUL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>SOH</td>
<td>start of header</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>02</td>
<td>STX</td>
<td>start of text</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>03</td>
<td>ETX</td>
<td>end of text</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>04</td>
<td>EOT</td>
<td>end of transmission</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>05</td>
<td>ENQ</td>
<td>enquiry</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>06</td>
<td>ACK</td>
<td>acknowledge</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>07</td>
<td>BEL</td>
<td>bell</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>08</td>
<td>BS</td>
<td>backspace</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>09</td>
<td>HT</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>0A</td>
<td>LF</td>
<td>line feed</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>0B</td>
<td>VT</td>
<td>vertical tab</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>0C</td>
<td>FF</td>
<td>form feed</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
<td>0D</td>
<td>CR</td>
<td>carriage return</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>0E</td>
<td>SO</td>
<td>shift out</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>0F</td>
<td>SI</td>
<td>shift in</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>10</td>
<td>DLE</td>
<td>data link escape</td>
</tr>
<tr>
<td>17</td>
<td>21</td>
<td>11</td>
<td>DC1</td>
<td>no assignment, but usually XON</td>
</tr>
<tr>
<td>18</td>
<td>22</td>
<td>12</td>
<td>DC2</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>23</td>
<td>13</td>
<td>DC3</td>
<td>no assignment, but usually XOFF</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
<td>14</td>
<td>DC4</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>25</td>
<td>15</td>
<td>NAK</td>
<td>negative acknowledge</td>
</tr>
<tr>
<td>22</td>
<td>26</td>
<td>16</td>
<td>SYN</td>
<td>synchronous idle</td>
</tr>
<tr>
<td>23</td>
<td>27</td>
<td>17</td>
<td>ETB</td>
<td>end of transmission block</td>
</tr>
<tr>
<td>24</td>
<td>30</td>
<td>18</td>
<td>CAN</td>
<td>cancel</td>
</tr>
<tr>
<td>25</td>
<td>31</td>
<td>19</td>
<td>EM</td>
<td>end of medium</td>
</tr>
<tr>
<td>26</td>
<td>32</td>
<td>1A</td>
<td>SUB</td>
<td>substitute</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>27</td>
<td>33</td>
<td>1B</td>
<td>ESC</td>
<td>escape</td>
</tr>
<tr>
<td>28</td>
<td>34</td>
<td>1C</td>
<td>FS</td>
<td>file separator</td>
</tr>
<tr>
<td>29</td>
<td>35</td>
<td>1D</td>
<td>GS</td>
<td>group separator</td>
</tr>
<tr>
<td>30</td>
<td>36</td>
<td>1E</td>
<td>RS</td>
<td>record separator</td>
</tr>
<tr>
<td>31</td>
<td>37</td>
<td>1F</td>
<td>US</td>
<td>unit separator</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>20</td>
<td>SPC</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>41</td>
<td>21</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>42</td>
<td>22</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>43</td>
<td>23</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>44</td>
<td>24</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>45</td>
<td>25</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>46</td>
<td>26</td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>47</td>
<td>27</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>50</td>
<td>28</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>51</td>
<td>29</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>52</td>
<td>2A</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>53</td>
<td>2B</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>54</td>
<td>2C</td>
<td>,</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>55</td>
<td>2D</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>56</td>
<td>2E</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>57</td>
<td>2F</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>60</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>61</td>
<td>31</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>62</td>
<td>32</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>63</td>
<td>33</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>64</td>
<td>34</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>65</td>
<td>35</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>66</td>
<td>36</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>67</td>
<td>37</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>70</td>
<td>38</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>71</td>
<td>39</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>72</td>
<td>3A</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>73</td>
<td>3B</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>74</td>
<td>3C</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>75</td>
<td>3D</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>76</td>
<td>3E</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>77</td>
<td>3F</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>40</td>
<td>@</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>101</td>
<td>41</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>102</td>
<td>42</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>103</td>
<td>43</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>104</td>
<td>44</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>105</td>
<td>45</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>106</td>
<td>46</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>107</td>
<td>47</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>110</td>
<td>48</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>111</td>
<td>49</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>112</td>
<td>4A</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>113</td>
<td>4B</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>114</td>
<td>4C</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>115</td>
<td>4D</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>116</td>
<td>4E</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>117</td>
<td>4F</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>120</td>
<td>50</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>121</td>
<td>51</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>122</td>
<td>52</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>123</td>
<td>53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>124</td>
<td>54</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>125</td>
<td>55</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>126</td>
<td>56</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>127</td>
<td>57</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>130</td>
<td>58</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>131</td>
<td>59</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>132</td>
<td>5A</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>133</td>
<td>5B</td>
<td>[</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>134</td>
<td>5C</td>
<td>\</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>135</td>
<td>5D</td>
<td>]</td>
<td></td>
</tr>
<tr>
<td>ASCII Code</td>
<td>Character</td>
<td>Hex Code</td>
<td>Unicode Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>----------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>94</td>
<td>~</td>
<td>5E</td>
<td>^</td>
<td>escape</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
<td>5F</td>
<td>_</td>
<td>underscore</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>60</td>
<td>`</td>
<td>quote or single quotation mark</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>61</td>
<td>a</td>
<td>letter a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>62</td>
<td>b</td>
<td>letter b</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td>63</td>
<td>c</td>
<td>letter c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td>64</td>
<td>d</td>
<td>letter d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td>65</td>
<td>e</td>
<td>letter e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
<td>66</td>
<td>f</td>
<td>letter f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
<td>67</td>
<td>g</td>
<td>letter g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
<td>68</td>
<td>h</td>
<td>letter h</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
<td>69</td>
<td>i</td>
<td>letter i</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
<td>6A</td>
<td>j</td>
<td>letter j</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
<td>6B</td>
<td>k</td>
<td>letter k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
<td>6C</td>
<td>l</td>
<td>letter l</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
<td>6D</td>
<td>m</td>
<td>letter m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td>6E</td>
<td>n</td>
<td>letter n</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
<td>6F</td>
<td>o</td>
<td>letter o</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
<td>70</td>
<td>p</td>
<td>letter p</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
<td>71</td>
<td>q</td>
<td>letter q</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
<td>72</td>
<td>r</td>
<td>letter r</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
<td>73</td>
<td>s</td>
<td>letter s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
<td>74</td>
<td>t</td>
<td>letter t</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
<td>75</td>
<td>u</td>
<td>letter u</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
<td>76</td>
<td>v</td>
<td>letter v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
<td>77</td>
<td>w</td>
<td>letter w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td>78</td>
<td>x</td>
<td>letter x</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
<td>79</td>
<td>y</td>
<td>letter y</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
<td>7A</td>
<td>z</td>
<td>letter z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
<td>7B</td>
<td>{</td>
<td>opening brace</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
<td>7D</td>
<td>}</td>
<td>closing brace</td>
</tr>
<tr>
<td>126</td>
<td>~</td>
<td>7E</td>
<td>~</td>
<td>tilde</td>
</tr>
<tr>
<td>127</td>
<td>DEL</td>
<td>7F</td>
<td>DEL</td>
<td>delete</td>
</tr>
</tbody>
</table>
C/C++ Keywords

Display all entries for C/C++ Keywords on one page, or view entries individually:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asm</code></td>
<td>insert an assembly instruction</td>
</tr>
<tr>
<td><code>auto</code></td>
<td>declare a local variable</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>declare a boolean variable</td>
</tr>
<tr>
<td><code>break</code></td>
<td>break out of a loop</td>
</tr>
<tr>
<td><code>case</code></td>
<td>a block of code in a <code>switch</code> statement</td>
</tr>
<tr>
<td><code>catch</code></td>
<td>handles exceptions from <code>throw</code></td>
</tr>
<tr>
<td><code>char</code></td>
<td>declare a character variable</td>
</tr>
<tr>
<td><code>class</code></td>
<td>declare a class</td>
</tr>
<tr>
<td><code>const</code></td>
<td>declare immutable data or functions that do not change data</td>
</tr>
<tr>
<td><code>const_cast</code></td>
<td>cast from const variables</td>
</tr>
<tr>
<td><code>continue</code></td>
<td>bypass iterations of a loop</td>
</tr>
<tr>
<td><code>default</code></td>
<td>default handler in a <code>case</code> statement</td>
</tr>
<tr>
<td><code>delete</code></td>
<td>make memory available</td>
</tr>
<tr>
<td><code>do</code></td>
<td>looping construct</td>
</tr>
<tr>
<td><code>double</code></td>
<td>declare a double precision floating-point variable</td>
</tr>
<tr>
<td><code>dynamic_cast</code></td>
<td>perform runtime casts</td>
</tr>
<tr>
<td><code>else</code></td>
<td>alternate case for an <code>if</code> statement</td>
</tr>
<tr>
<td><code>enum</code></td>
<td>create enumeration types</td>
</tr>
<tr>
<td><code>explicit</code></td>
<td>only use constructors when they exactly match</td>
</tr>
<tr>
<td><code>export</code></td>
<td>allows template definitions to be separated from their declarations</td>
</tr>
<tr>
<td><code>extern</code></td>
<td>tell the compiler about variables defined elsewhere</td>
</tr>
<tr>
<td><code>false</code></td>
<td>the boolean value of false</td>
</tr>
<tr>
<td><code>float</code></td>
<td>declare a floating-point variable</td>
</tr>
<tr>
<td><code>for</code></td>
<td>looping construct</td>
</tr>
<tr>
<td><strong>friend</strong></td>
<td>grant non-member function access to private data</td>
</tr>
<tr>
<td><strong>goto</strong></td>
<td>jump to a different part of the program</td>
</tr>
<tr>
<td><strong>if</strong></td>
<td>execute code based off of the result of a test</td>
</tr>
<tr>
<td><strong>inline</strong></td>
<td>optimize calls to short functions</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td>declare a integer variable</td>
</tr>
<tr>
<td><strong>long</strong></td>
<td>declare a long integer variable</td>
</tr>
<tr>
<td><strong>mutable</strong></td>
<td>override a const variable</td>
</tr>
<tr>
<td><strong>namespace</strong></td>
<td>partition the global namespace by defining a scope</td>
</tr>
<tr>
<td><strong>new</strong></td>
<td>allocate dynamic memory for a new variable</td>
</tr>
<tr>
<td><strong>operator</strong></td>
<td>create overloaded operator functions</td>
</tr>
<tr>
<td><strong>private</strong></td>
<td>declare private members of a class</td>
</tr>
<tr>
<td><strong>protected</strong></td>
<td>declare protected members of a class</td>
</tr>
<tr>
<td><strong>public</strong></td>
<td>declare public members of a class</td>
</tr>
<tr>
<td><strong>register</strong></td>
<td>request that a variable be optimized for speed</td>
</tr>
<tr>
<td><strong>reinterpret_cast</strong></td>
<td>change the type of a variable</td>
</tr>
<tr>
<td><strong>return</strong></td>
<td>return from a function</td>
</tr>
<tr>
<td><strong>short</strong></td>
<td>declare a short integer variable</td>
</tr>
<tr>
<td><strong>signed</strong></td>
<td>modify variable type declarations</td>
</tr>
<tr>
<td><strong>sizeof</strong></td>
<td>return the size of a variable or type</td>
</tr>
<tr>
<td><strong>static</strong></td>
<td>create permanent storage for a variable</td>
</tr>
<tr>
<td><strong>static_cast</strong></td>
<td>perform a nonpolymorphic cast</td>
</tr>
<tr>
<td><strong>struct</strong></td>
<td>define a new structure</td>
</tr>
<tr>
<td><strong>switch</strong></td>
<td>execute code based off of different possible values for a variable</td>
</tr>
<tr>
<td><strong>template</strong></td>
<td>create generic functions</td>
</tr>
<tr>
<td><strong>this</strong></td>
<td>a pointer to the current object</td>
</tr>
<tr>
<td><strong>throw</strong></td>
<td>throws an exception</td>
</tr>
<tr>
<td><strong>true</strong></td>
<td>the boolean value of true</td>
</tr>
<tr>
<td><strong>try</strong></td>
<td>execute code that can throw an exception</td>
</tr>
<tr>
<td><strong>typedef</strong></td>
<td>create a new type name from an existing type</td>
</tr>
<tr>
<td><strong>typeid</strong></td>
<td>describes an object</td>
</tr>
<tr>
<td><strong>typename</strong></td>
<td>declare a class or undefined type</td>
</tr>
<tr>
<td><strong>union</strong></td>
<td>a structure that assigns multiple variables to the same memory location</td>
</tr>
<tr>
<td><strong>unsigned</strong></td>
<td>declare an unsigned integer variable</td>
</tr>
<tr>
<td><strong>using</strong></td>
<td>import complete or partial namespaces into the current scope</td>
</tr>
<tr>
<td><strong>virtual</strong></td>
<td>create a function that can be overridden by a derived class</td>
</tr>
<tr>
<td><strong>void</strong></td>
<td>declare functions or data with no associated data type</td>
</tr>
<tr>
<td><strong>volatile</strong></td>
<td>warn the compiler about variables that can be modified unexpectedly</td>
</tr>
<tr>
<td><strong>wchar_t</strong></td>
<td>declare a wide-character variable</td>
</tr>
<tr>
<td><strong>while</strong></td>
<td>looping construct</td>
</tr>
</tbody>
</table>
The C++ STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists, and stacks.

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

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

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

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

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

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

Can I get a copy of this site?

We do provide a downloadable archived version of cppreference.com. If you're interested in getting archived versions of websites in general, you might want to check out utilities like GNU's wget (Windows version here).

In addition, James Heany has compiled a PDF version of the site (as of October 2007) that is available for download. There is a C++ Reference PDF and a STL Reference PDF 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
- C Programming
- C++ Language Tutorial

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 Linux (Red Hat, Debian, or Ubuntu) with the GNU Compiler Collection. Since this site is merely a reference for the Standard C and C++ specification, not every compiler will support every function listed here. For example,

- Header files change like mad. To include the necessary support for C++ Vectors, you might have to use any of these:

```c
#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 platform-independent commands to include standard C libraries. For example, you should be able to use

```c
#include <cstdio>
```

instead of

```c
#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:

```c
cout << "hello world!";
```

However, newer compilers require that you either use

```c
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 [contact us] -- feedback and code examples are always welcome.

**What's up with this site?**

Think of it as a community service, for geeks.
Links

Here are some links to other language references:

- C++ (Dinkumware)
- C++ Language and Library
- Java 6 SE (Sun)
- MySQL
- Perl
- Python
- Ruby
- Tcl
- Visual C++ STL (Microsoft)
Huge thanks to all these people for sending in bug fixes and suggestions on how to improve the site:

Alex Vinokur - Ted Felix

Thank you!
any

Syntax:

```cpp
code snippet
#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:

```cpp
code snippet
#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:

```cpp
code snippet
// create a bitset out of a number
```cpp
bitset<8> bs2( (long) 131 );
cout << "bs2 is " << bs2 << endl;
// shift the bitset to the left by 4 digits
bs2 <<= 4;
cout << "now bs2 is " << bs2 << endl;
```

When the above code is run, it displays:

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

### Bitset Constructors

**Syntax:**

```cpp
#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:

```cpp
// 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;
```
**count**

*Syntax:*

```cpp
#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:*

```cpp
#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:*

```cpp
#include <bitset>
bool none();
```

The `none()` function only returns true if none of the bits in the bitset are set to 1.

*Related topics:*

any
reset

Syntax:

```cpp
#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:

```cpp
#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:

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

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

---

test
Syntax:

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

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

---

**to_string**

Syntax:

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

The `to_string()` function returns a string representation of the bitset.

**Related topics:**

*to_ulong*

---

**to_ulong**

Syntax:

```cpp
#include <bitset>
unsigned long to_ulong();
```

The function `to_ulong()` returns the bitset, converted into an unsigned long integer.

**Related topics:**

*to_string*
**assign**

**Syntax:**

```cpp
#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:

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

The above code displays the following output:

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

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

```cpp
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] << " ";
}
```
When run, the above code displays the following output:

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

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

---

### at

**Syntax:**

```cpp
#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:

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

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

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

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

*Related topics:*

**Vector operators**

---

### back

*Syntax:*

```cpp
#include <vector>

TYPE& back();
const TYPE& back() const;
```

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

For example:

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

This code produces the following output:

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

The `back()` function runs in *constant time*.

*Related topics:*

**front**

**pop_back**

---

### begin

*Syntax:*

```cpp
```
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:

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

When given this input:

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

...the above code produces the following output:

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

`Related topics:`

- `[]` operator
- `at`
- `end`
- `rbegin`
- `rend`

---

`capacity`
Syntax:

```cpp
#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:

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

When run, the above code produces the following output:

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

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

The `capacity()` function runs in constant time.

**Related topics:**

- `reserve`
- `resize`
- `size`
clear

Syntax:

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

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

clear() runs in linear time.

Related topics:
 erase

empty

Syntax:

```cpp
#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:

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

Related topics:
 size
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:

```cpp
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );

int bad_val = *(v1.end());
cout << "bad_val is " << bad_val << endl;

int good_val = *(v1.end() - 1);
cout << "good_val is " << good_val << endl;
```

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

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

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

end() runs in constant time.

Related topics:
begin
rbegin
rend

---

**erase**

Syntax:

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

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

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

For example:

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

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

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

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
    cout << alphaVector[i];
}
cout << endl;

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

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

**Related topics:**

clear
**front**

**Syntax:**

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

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

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

```cpp
vector<string> words;
string str;
while( cin >> str ) words.push_back(str);
sort( words.begin(), words.end() );
```

When provided with this input:

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

...the above code displays:

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

**Related topics:**

- **back**
- (C++ Lists) **pop_front**
(C++ Lists) **push_front**

# insert

**Syntax:**

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

The `insert()` function either:

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

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

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

```cpp
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end(); 
    cout << *theIterator;
}
```
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:

```cpp
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );

vector<int> v2;
v2.push_back( 5 );
v2.push_back( 6 );
v2.push_back( 7 );
v2.push_back( 8 );

cout << "Before, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
}
cout << endl;

v2.insert( v2.end(), v1.begin(), v1.end() );

cout << "After, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
}
cout << endl;
```

When run, this code displays:

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

Related topics:
- assign
- erase
- `push_back`
- `merge`
- `push_front`
- `splice`
max_size

Syntax:

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

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

Related topics:
size

pop_back

Syntax:

```cpp
#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:**

```cpp
#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:

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

When displayed, the resulting vector would look like this:

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

`push_back()` runs in **constant time**.

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

### rbegin

**Syntax:**

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

The `rbegin()` function returns a `reverse_iterator` to the end of the current vector.
rbegin() runs in constant time.

Related topics:
begin
end
rend

**rend**

Syntax:

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

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

rend() runs in constant time.

Related topics:
begin
end
rbegin

**reserve**

Syntax:

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

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

reserve() runs in linear time.
Related topics:
capacity

---

**resize**

**Syntax:**

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

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

This function runs in **linear time**.

**Related topics:**
Vector constructors & destructors
capacity
size

---

**size**

**Syntax:**

```cpp
#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:

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

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

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

```cpp
vector v1;
v1.push_back("I'm in v1!");

vector v2;
v2.push_back("And I'm in v2!");

v1.swap(v2);

cout << "The first element in v1 is " << v1.front() << endl;
cout << "The first element in v2 is " << v2.front() << endl;
```

The above code displays:

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

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

Vector constructors

Syntax:

```cpp
#include <vector>
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:

```c++
// 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;
}

When run, this code displays the following output:

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

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

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

---

**Vector operators**

**Syntax:**

```cpp
#include <vector>

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

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

Performing a comparison or assigning one vector to another takes **linear time**. The [] operator runs in **constant time**.
Two vectors are equal if:

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

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

Related topics:
at
assign

Syntax:

```cpp
#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:

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

The above code displays the following output:

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

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

```cpp
vector<int> v1;
for (int i = 0; i < 10; i++) {
    v1.push_back(i);
}

vector<int> v2;
v2.assign(v1.begin(), v1.end());

for (int i = 0; i < v2.size(); i++) {
    cout << v2[i] << " ";
}
```
cout << endl;

When run, the above code displays the following output:

0 1 2 3 4 5 6 7 8 9

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

at

Syntax:

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

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

For example, consider the following code:

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

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

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

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

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

---

## back

**Syntax:**

```cpp
#include <deque>

TYPE & back();
const TYPE & back() const;
```

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

For example:

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

This code produces the following output:

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

The back() function runs in **constant time**.

*Related topics:*  
**front**  
**pop_back**

---

## begin
Syntax:

```cpp
#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:

```cpp
// 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;
}
```

**Related topics:**
- end
- rbegin
- rend

---

**clear**

Syntax:

```cpp
#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:

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

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:

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

The [] operator runs in constant time.

Related topics:

at
Syntax:

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

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:

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

The [] operator runs in constant time.

Related topics:

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

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

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

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

```cpp
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: " << *iter2 << endl;

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

When run, this code displays the following output:

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

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

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

---

**Container operators**

**Syntax:**

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

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

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

Two `containers` are equal if:

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

Comparisons among dequeues are done lexicographically.

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

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

**Related topics:**

at

---

**empty**

**Syntax:**

```cpp
#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:

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

Related topics:
- `size`

---

## end

**Syntax:**

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

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

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

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

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

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

`end()` runs in **constant time**.
Related topics:
begin
rbegin
rend

erase

Syntax:

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

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

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

For example:

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

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

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

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for (int i=0; i < 10; i++) {
    alphaVector.push_back( i + 65 );
}
// display the complete vector
for (int i = 0; i < alphaVector.size(); i++) {
    cout << alphaVector[i];
}
cout << endl;

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

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

*Related topics:*
- clear
- insert
- pop_back
- pop_front
front

Syntax:

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

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

Related topics:
- `back`
- `pop_front`
- `push_front`

insert

Syntax:

```cpp
#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:

```cpp
// Create a vector, load it with the first 10 characters of the alphabet
```
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end(); theIterator++ )
    cout << *theIterator;

This code would display:

```
CCCCABCDEF
```

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.
Related topics:
size

---

**pop_back**

Syntax:

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

The `pop_back()` function removes the last element of the deque.

`pop_back()` runs in **constant time**.

Related topics:
back
erase
pop_front
push_back

---

**pop_front**

Syntax:

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

The function `pop_front()` removes the first element of the deque.

The `pop_front()` function runs in **constant time**.

Related topics:
erase
front
pop_back
push_front
push_back

Syntax:

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

The `push_back()` function appends `val` to the end of the deuque.

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

```cpp
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_back`
- `push_front`

---

push_front

Syntax:

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

The `push_front()` function inserts `val` at the beginning of deuque.

`push_front()` runs in **constant time**.
rbegin

Syntax:

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

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

rbegin() runs in constant time.

Related topics:
begin
end
rend

rend

Syntax:

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

The function rend() returns a reverse_iterator to the beginning of the current deque.
rend() runs in constant time.

Related topics:
begin
derbegin
end

**resize**

Syntax:

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

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

This function runs in linear time.

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

**size**

Syntax:

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

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

Related topics:
(C++ Strings) capacity
empty
swap

Syntax:

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

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

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

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

The above code displays:

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

Related topics:
(C++ Lists) splice
assign

Syntax:

```cpp
#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:

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

The above code displays the following output:

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

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

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

When run, the above code displays the following output:

0 1 2 3 4 5 6 7 8 9

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

back

Syntax:

```cpp
#include <list>

TYPE& back();
const TYPE& back() const;
```

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

For example:

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

This code produces the following output:

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

The back() function runs in constant time.

Related topics:
front
pop_back
begin

Syntax:

```cpp
#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:

```cpp
// 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;
```

Related topics:

- end
- rbegin
- rend

clear

Syntax:

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

The function `clear()` deletes all of the elements in the list. `clear()` runs in linear
Container constructors & destructors

Syntax:

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

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

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

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

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

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

Related topics:

Special container constructors, resize
List constructors

Syntax:

```cpp
#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:

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

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

```cpp
// 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;

When run, this code displays the following output:

<table>
<thead>
<tr>
<th>original vector:</th>
<th>1 9 7 9 2 7 2 1 9 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>first even number:</td>
<td>2, last even number:</td>
</tr>
<tr>
<td>new vector:</td>
<td>2 7 2 1 9</td>
</tr>
</tbody>
</table>

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

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

---

**List operators**

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

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

Two lists are equal if:

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

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

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

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a list and display its contents in reverse order:
vector<int> v;
for( int i = 0; i < 5; i++ ) {
    v.push_back(i);
}
while( !v.empty() ) {
    cout << v.back() << endl;
    v.pop_back();
}

Related topics:
size

end

Syntax:

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

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

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

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

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

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

end() runs in constant time.
erase

Syntax:

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

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

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

For example:

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

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

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

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
    cout << alphaVector[i];
}
cout << endl;

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

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

Related topics:
- **clear**
- **insert**
- **pop_back**
- **pop_front**
front

Syntax:

```cpp
#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:

```cpp
#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 alphabet
```
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
    alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end(); theIterator++ )
    cout << *theIterator;

This code would display:

CCCCABCDEFGHIJ

Related topics:
assign
erase
merge
push_back
push_front
splice

max_size

Syntax:

#include <list>
size_type max_size() const;

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

Related topics:
merge

Syntax:

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

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

`merge()` runs in **linear time**.

**Related topics:**
- Container operators
- insert
- splice

pop_back

Syntax:

```cpp
#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

pop_front

Syntax:

```cpp
#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:

```cpp
#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:

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

When displayed, the resulting list would look like this:

```
0 1 2 3 4 5 6 7 8 9
```
push_back() runs in constant time.

Related topics:
assign
insert
pop_back
push_front

---

**push_front**

Syntax:

```cpp
#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_front

---

**rbegin**

Syntax:

```cpp
#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:**

```cpp
#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:

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

Remove runs in linear time.

**Related topics:**
- erase
- remove_if
- unique

---

**remove_if**

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

remove_if() runs in **linear time**.

**Related topics:**
- erase
- remove
- unique

---

### rend

**Syntax:**

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

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

rend() runs in **constant time**.

**Related topics:**
- begin
- end
- rbegin

---

### resize

**Syntax:**

```cpp
#include <list>
void resize( size_type num, const TYPE& val = TYPE() );
```
The function `resize()` changes the size of the list to `size`. If `val` is specified then any newly-created elements will be initialized to have a value of `val`.

This function runs in **linear time**.

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

---

**reverse**

**Syntax:**

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

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

**Related topics:**
**sort**

---

**size**

**Syntax:**

```cpp
#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:

```cpp
#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 $O(N \log N)$ time.

Related topics:
reverse

splice

Syntax:

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

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

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

Related topics:
insert
merge
swap

Syntax:

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

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

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

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

The above code displays:

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

Related topics:
splice

unique

Syntax:

```cpp
#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`
**begin**

**Syntax:**

```cpp
#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:

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

**Related topics:**
- **end**
- **rbegin**
- **rend**

---

**clear**

**Syntax:**

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

The function `clear()` deletes all of the elements in the set. `clear()` runs in **linear**
Set constructors & destructors

Syntax:

```cpp
#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:

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

Set operators

Syntax:

```cpp
#include <set>
set operator=(const set& c2);
bool operator==(const set& c1, const set& c2);
```
All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one set to another takes linear time.

Two sets are equal if:

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

```cpp
#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:

```cpp
#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:

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

Related topics:
size

---

end

Syntax:

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

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

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

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

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

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

`end()` runs in **constant time**.

**Related topics:**
- `begin`
- `rbegin`
- `rend`

---

**equal_range**

**Syntax:**

```cpp
#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:**

```cpp
#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:

```cpp
#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:

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

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

`lower_bound()` runs in **logarithmic time**.

**Related topics:**
- `upper_bound`

max_size

Syntax:

```cpp
#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 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:

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

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

rbegin() runs in constant time.

Related topics:
begin  
end  
rend

rend

Syntax:

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

The function rend() returns a reverse_iterator to the beginning of the current set.
rend() runs in **constant time**.

*Related topics:*  
**begin**  
**end**  
**rbegin**

---

**size**

**Syntax:**

```cpp
#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:**

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

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

For example, the following code uses the swap() function to exchange the values of two strings:
```cpp
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

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

Related topics:
(C++ Lists) splice

---

**upper_bound**

**Syntax:**

```cpp
#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:
lower_bound

---

**value_comp**

**Syntax:**

```cpp
#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**
begin

Syntax:

```cpp
#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:

```cpp
// 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;
}
```

Related topics:
- end
- rbegin
- rend

---

clear

Syntax:

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

The function `clear()` deletes all of the elements in the multiset. `clear()` runs in
Related topics:  
(C++ Lists) erase

---

**Container constructors & destructors**

**Syntax:**

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

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

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

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

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

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

**Related topics:**  
(C++ Strings) resize

---

**Container operators**

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

Two multisets are equal if:

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

```c++
#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:

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

Related topics:

- `size`

---

either

Syntax:

```c++
#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 \texttt{begin()} and \texttt{end()} to iterate through all of the members of a vector:

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

The iterator is initialized with a call to \texttt{begin()}. After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling \texttt{end()}. Since \texttt{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.

\texttt{end()} runs in \textit{constant time}.

\textit{Related topics:}
\texttt{begin}
\texttt{rbegin}
\texttt{rend}

---

**equal\_range**

\textit{Syntax:}

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

The function \texttt{equal\_range()} returns two iterators - one to the first element that contains \textit{key}, another to a point just after the last element that contains \textit{key}.

---

**erase**

\textit{Syntax:}
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:**

```cpp
#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](https://en.wikipedia.org/wiki/Logarithmic_time).

---

**insert**

**Syntax:**

```cpp
#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:*

```cpp
#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:*

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

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

`lower_bound()` runs in **logarithmic time**.

*Related topics:*

- **upper_bound**

**max_size**

*Syntax:*

```cpp
#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:
size

rbegin

Syntax:

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

The rbegin() function returns a reverse_iterator to the end of the current multiset. rbegin() runs in constant time.

Related topics:
begin
end
rend

rend

Syntax:

```cpp
#include <set>
reverse_iterator rend();
const_reverse_iterator rend() const;
```
The function `rend()` returns a `reverse_iterator` to the beginning of the current multiset.

`rend()` runs in \textit{constant time}.

\textit{Related topics:}
\begin{itemize}
  \item \texttt{begin}
  \item \texttt{end}
  \item \texttt{rbegin}
\end{itemize}

\section*{size}

\textbf{Syntax:}

\begin{verbatim}
#include <set>
size_type size() const;
\end{verbatim}

The \textit{size()} function returns the number of elements in the current multiset.

\textit{Related topics:}
\begin{itemize}
  \item (C++ Strings) \texttt{capacity}
  \item \texttt{empty}
  \item (C++ Strings) \texttt{length}
  \item \texttt{max_size}
  \item (C++ Strings) \texttt{resize}
\end{itemize}

\section*{swap}

\textbf{Syntax:}

\begin{verbatim}
#include <set>
void swap( container& from );
\end{verbatim}

The \textit{swap()} function exchanges the elements of the current multiset with those of \texttt{from}. This function operates in \textit{constant time}. 
For example, the following code uses the swap() function to exchange the values of two strings:

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

The above code displays:

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

**Related topics:**
(C++ Lists) splice

---

**upper_bound**

*Syntax:*

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

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

**Related topics:**
lower_bound

---

**value_comp**

*Syntax:*

```cpp
#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
**begin**

Syntax:

```cpp
#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:

```cpp
map<string,int> stringCounts;
string str;

while( cin >> str ) stringCounts[str]++;

map<string,int>::iterator iter;
for( iter = stringCounts.begin(); iter != stringCounts.end(); iter++ )
  cout << "word: " << iter->first << ", count: " << iter->second << endl;
```

When given this input:

```plaintext
here are some words and here are some more words
```

...the above code generates this output:

```plaintext
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:

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

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

time.

Related topics:
erase

count

Syntax:

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

The function count() returns the number of occurrences of key in the map.
count() should run in logarithmic time.

empty

Syntax:

```cpp
#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:

```cpp
struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
        return strcmp( s1, s2 ) < 0;
    }
};
...

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

while( !ages.empty() ) {
    cout << "Erasing: " << (*ages.begin()).first << ", " << (*ages.begin()).second << endl;
    ages.erase( ages.begin() );
}
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
Erasing: Marge, 37
```

**Related topics:**
*begin*
*erase*
*size*

**Syntax:**

```cpp
#include <map>
iterator end();
const_iterator end() const;
```
The end() function returns an iterator just past the end of the map.

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

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

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

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

end() runs in constant time.

Related topics:
begin
rbegin
rend

equal_range

Syntax:

```cpp
text
#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:*

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

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

For example, the following code uses `erase()` in a `while` loop to incrementally clear a map and display its contents in order:

```cpp
struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
        return strcmp( s1, s2 ) < 0;
    }
};
...
map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
ages["Bart"] = 11;

while( !ages.empty() ) {
    cout << "Erasing: " << (*ages.begin()).first << ", " << (*ages.begin()).second << endl;
    ages.erase( ages.begin() );
}
```

When run, the above code displays:

```
Erasing: Bart, 11
Erasing: Homer, 38
Erasing: Lisa, 8
Erasing: Maggie, 1
Erasing: Marge, 37
```
find

Syntax:

```cpp
#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:

```cpp
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 << endl;
}
```

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:

```cpp
#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:

```cpp
map<string,int> theMap;
theMap.insert( make_pair( "Key 1", -1 ) );
theMap.insert( make_pair( "Another key!", 32 ) );
theMap.insert( make_pair( "Key the Three", 66667 ) );

map<string,int>::iterator iter;
for( iter = theMap.begin(); iter != theMap.end(); ++iter ) {
    cout << "Key: '" << iter->first << ", Value: " << iter->second << endl;
}
```

When run, the above code displays this output:

```
Key: 'Another key!', Value: 32
Key: 'Key 1', Value: -1
Key: 'Key the Three', Value: 66667
```

Note that because maps are sorted containers, the output is sorted by the key value. In this case, since the map key data type is `string`, the map is sorted alphabetically by key.
**key_comp**

Syntax:

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

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

`key_comp()` runs in **constant time**.

**Related topics:**
- `value_comp`

---

**lower_bound**

Syntax:

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

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

`lower_bound()` runs in **logarithmic time**.

**Related topics:**
- `upper_bound`

---

**Map Constructors & Destructors**
Syntax:

```cpp
#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:

```cpp
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;
age["Marge"] = 37;
age["Lisa"] = 8;
age["Maggie"] = 1;
age["Bart"] = 11;

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

Related topics:

Map Operators
Map operators

Syntax:

```cpp
#include <map>

TYPE& operator[]( const key_type& key );
map operator=(const map& c2);
bool operator==(const map& c1, const map& c2);
bool operator!=(const map& c1, const map& c2);
bool operator<(const map& c1, const map& c2);
bool operator>(const map& c1, const map& c2);
bool operator<=(const map& c1, const map& c2);
bool operator>=(const map& c1, const map& c2);
```

Maps can be compared and assigned with the standard comparison operators: `==, !=, <=, >=, <, >, and =`. Individual elements of a map can be examined with the `[]` operator.

Performing a comparison or assigning one map to another takes linear time.

Two maps are equal if:

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

Comparisons among maps are done lexicographically.

For example, the following code defines a map between strings and integers and loads values into the map using the `[]` operator:

```cpp
struct strCmp {
    bool operator()( const char* s1, const char* s2 ) const {
        return strcmp( s1, s2 ) < 0;
    }
};

map<const char*, int, strCmp> ages;
ages["Homer"] = 38;
ages["Marge"] = 37;
ages["Lisa"] = 8;
ages["Maggie"] = 1;
```
ages["Bart"] = 11;
cout << "Bart is " << ages["Bart"] << " years old" << endl;
cout << "In alphabetical order: " << endl;
for( map<const char*, int, strCmp>::iterator iter = ages.begin(); iter != ages.end(); iter++)
    cout << (*iter).first << " is " << (*iter).second << " years old" << endl;

When run, the above code displays this output:

Bart is 11 years old
In alphabetical order:
Bart is 11 years old
Homer is 38 years old
Lisa is 8 years old
Maggie is 1 years old
Marge is 37 years old

Related topics:
insert
Map Constructors & Destructors

max_size

Syntax:

```
#include <map>

size_type max_size() const;
```

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

Related topics:
size
rbegin

Syntax:

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

The rbegin() function returns a reverse_iterator to the end of the current map.

rbegin() runs in constant time.

Related topics:
begin
end
rend

rend

Syntax:

```
#include <map>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current map.

rend() runs in constant time.

Related topics:
begin
end
rbegin
size

Syntax:

```
#include <map>
size_type size() const;
```

The size() function returns the number of elements in the current map.

Related topics:
empty
max_size

swap

Syntax:

```
#include <map>
void swap( container& from );
```

The swap() function exchanges the elements of the current map with those of from. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:
**upper_bound**

*Syntax:*

```cpp
#include <map>
iterator upper_bound( const key_type& key );
```

The function `upper_bound()` returns an iterator to the first element in the map with a key greater than `key`.

*Related topics:*

- [lower_bound](#)

---

**value_comp**

*Syntax:*

```cpp
#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](#)
begin

Syntax:

```cpp
#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:

```cpp
// 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;
}
```

Related topics:
- end
- rbegin
- rend

---

clear

clear()

Syntax:

```cpp
#include <map>
void clear();
```

The function `clear()` deletes all of the elements in the multimap. `clear()` runs in
Related topics:
(C++ Lists) erase

---

**count**

Syntax:

```cpp
#include <map>
size_type count( const key_type& key );
```

The function count() returns the number of occurrences of `key` in the multimap. count() should run in logarithmic time.

---

**empty**

Syntax:

```cpp
#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:

```cpp
vector<int> v;
for( int i = 0; i < 5; i++ ) {
    v.push_back(i);
}
while( !v.empty() ) {
    cout << v.back() << endl;
    v.pop_back();
```
end

Syntax:

```cpp
#include <map>
iterator end();
const_iterator end() const;
```

The `end()` function returns an iterator just past the end of the multimap.

Note that before you can access the last element of the multimap using an iterator that you get from a call to `end()`, you'll have to decrement the iterator first.

For example, the following code uses `begin()` and `end()` to iterate through all of the members of a vector:

```cpp
vector<int> v1( 5, 789 );
vector<int>::iterator it;
for( it = v1.begin(); it != v1.end(); it++ ) {
    cout << *it << endl;
}
```

The iterator is initialized with a call to `begin()`. After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling `end()`. Since `end()` returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

`end()` runs in **constant time**.

Related topics:

- `begin`
- `rbegin`
equal_range

Syntax:

```cpp
#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()`:

```cpp
multimap<string, pair<int, int>> input_config;

// read configuration from file "input.conf" to input_config
readConfigFile( input_config, "input.conf" );

pair<multimap<string, pair<int, int>>::iterator, multimap<string, pair<int, int>>::iterator> ii;

ii = input_config.equal_range("key"); // keyboard key-bindings
// we can iterate over a range just like with begin() and end()
for( ii.first; ii.first != ii.second; ++ii ) {
    // add a key binding with this key and output
    bindkey(i->second.first, i->second.second);
}

ii = input_config.equal_range("joyb"); // joystick button key-bindings
for( ii.first; ii.first != ii.second; ++ii ) {
    // add a key binding with this joystick button and output
    bindjoyb(i->second.first, i->second.second);
}
```

erase

Syntax:
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:**

```cpp
#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:**

```cpp
#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:

```cpp
multimap<string,int> m;

int employeeID = 0;
m.insert( pair<string,int>("Bob Smith",employeeID++) );
m.insert( pair<string,int>("Bob Thompson",employeeID++) );
m.insert( pair<string,int>("Bob Smithey",employeeID++) );
m.insert( pair<string,int>("Bob Smith",employeeID++) );

cout << "Number of employees named 'Bob Smith': " << m.count("Bob Smith") << endl;
cout << "Number of employees named 'Bob Thompson': " << m.count("Bob Thompson") << endl;
cout << "Number of employees named 'Bob Smithey': " << m.count("Bob Smithey") << endl;

cout << "Employee list: " << endl;
for( multimap<string, int>::iterator iter = m.begin(); iter != m.end(); ++iter )
  cout << " Name: " << iter->first << " , ID #" << iter->second << endl;
```

When run, the above code produces the following output:

```
Number of employees named 'Bob Smith': 2
Number of employees named 'Bob Thompson': 1
Number of employees named 'Bob Smithey': 1
Employee list:
  Name: Bob Smith, ID #0
  Name: Bob Smith, ID #3
  Name: Bob Smithey, ID #2
  Name: Bob Thompson, ID #1
```

---

### key_comp

**Syntax:**

```cpp
#include <map>
key_compare key_comp() const;
```

The function key_comp() returns the function that compares keys.

key_comp() runs in **constant time**.
Related topics:
value_comp

lower_bound

Syntax:

```cpp
#include <map>
iterator lower_bound( const key_type & key );
```

The `lower_bound()` function returns an iterator to the first element which has a value greater than or equal to `key`.

`lower_bound()` runs in **logarithmic time**.

Related topics:
upper_bound

max_size

Syntax:

```cpp
#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:
size
Multimap constructors & destructors

Syntax:

```cpp
#include <map>
multimap();
multimap( const multimap & c );
multimap( iterator begin, iterator end,
         const key_compare & cmp = Compare(), const allocator & alloc = Allocator() );
~multimap();
```

Multimaps have several constructors:

- The default constructor takes no arguments, creates a new instance of that multimap, and runs in constant time.
- The default copy constructor runs in linear time and can be used to create a new multimap that is a copy of the given multimap `c`.
- Multimaps can also be created from a range of elements defined by `begin` and `end`. When using this constructor, an optional comparison function `cmp` and allocator `alloc` can also be provided.

The default destructor is called when the multimap should be destroyed.

The template definition of multimaps requires that both a key type and value type be supplied. For example, you can instantiate a multimap that maps strings to integers with this statement:

```cpp
multimap<string,int> m;
```

You can also supply a comparison function and an allocator in the template:

```cpp
multimap<string,int,myComp,myAlloc> m;
```

For example, the following code uses a multimap to associate a series of employee names with numerical IDs:

```cpp
multimap<string,int> m;
```
int employeeID = 0;
m.insert( pair<string,int>("Bob Smith",employeeID++) );
m.insert( pair<string,int>("Bob Thompson",employeeID++) );
m.insert( pair<string,int>("Bob Smithey",employeeID++) );
m.insert( pair<string,int>("Bob Smith",employeeID++) );

cout << "Number of employees named 'Bob Smith': " << m.count("Bob Smith") << endl;
cout << "Number of employees named 'Bob Thompson': " << m.count("Bob Thompson") << endl;
cout << "Number of employees named 'Bob Smithey': " << m.count("Bob Smithey") << endl;

cout << "Employee list: " << endl;
for( multimap<string,int>::iterator iter = m.begin(); iter != m.end(); ++iter )
    cout << " Name: " << iter->first << ", ID #" << iter->second << " " << endl;

When run, the above code produces the following output. Note that the employee list is displayed in alphabetical order, because multimaps are sorted associative containers:

Number of employees named 'Bob Smith': 2
Number of employees named 'Bob Thompson': 1
Number of employees named 'Bob Smithey': 1
Employee list:
 Name: Bob Smith, ID #0
 Name: Bob Smith, ID #3
 Name: Bob Smithey, ID #2
 Name: Bob Thompson, ID #1

Related topics:
  count  insert

Multimap operators

Syntax:

```cpp
#include <map>
multimap operator=(const multimap& c2);
bool operator==(const multimap& c1, const multimap& c2);
bool operator!=(const multimap& c1, const multimap& c2);
bool operator<(const multimap& c1, const multimap& c2);
bool operator>(const multimap& c1, const multimap& c2);
bool operator<=(const multimap& c1, const multimap& c2);
```
bool operator>=(const multimap& c1, const multimap& c2);

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !_, <=, >=, <, >, and =. Performing a comparison or assigning one multimap to another takes linear time.

Two multimaps are equal if:

1. Their size is the same, and
2. Each member in location i in one multimap is equal to the 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 reverse_iterator to the end of the current multimap.

rbegin() runs in constant time.

Related topics: begin end rend
Syntax:

```cpp
#include <map>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function `rend()` returns a `reverse_iterator` to the beginning of the current multimap.

`rend()` runs in **constant time**.

**Related topics:**
- `begin`
- `end`
- `rbegin`

---

**size**

Syntax:

```cpp
#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:
The swap() function exchanges the elements of the current multimap with those of \textit{from}. This function operates in \textit{constant time}.

For example, the following code uses the swap() function to exchange the values of two strings:

```cpp
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

\textit{Related topics:}
\begin{itemize}
\item \textit{(C++ Lists)} \texttt{splice}
\end{itemize}

\section*{upper\_bound}

\textbf{Syntax:}

```cpp
#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 \textit{key}.

\textit{Related topics:}
\begin{itemize}
\item \texttt{lower\_bound}
\end{itemize}

\section*{value\_comp}
Syntax:

```cpp
#include <map>
value_compare value_comp() const;
```

The `value_comp()` function returns the function that compares values.

`value_comp()` runs in constant time.

Related topics:
- key_comp
C++ 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:

```cpp
ifstream fin("data.txt");
string s;
while( fin >> s ) {
    cout << "Read from file: " << s << endl;
}
```

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 I/O getline() function.

```cpp
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;
```
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 C++ string getline() function to read lines into strings:

```cpp
ifstream fin("data.txt");
string s;
while( getline(fin,s) ) {
    cout << "Read from file: " << s << endl;
}
```
Checking For Errors

Simply evaluating an I/O object in a boolean context will return false if any errors have occurred:

```cpp
string filename = "data.txt";
ifstream fin( filename.c_str() );
if( !fin ) {
    cout << "Error opening " << filename << " for input" << endl;
    exit(-1);
}
```
bad

Syntax:

```cpp
#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

clear

Syntax:

```cpp
#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;

Related topics:
   eof
   fail
   good
   rdstate

---

close

Syntax:

```cpp
#include <fstream>
void close();
```

The close() function closes the associated file stream.

Related topics:
   I/O Constructors
   open

---

I/O Constructors

Syntax:
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 `iostream` mode flags. 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.

```cpp
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
    fout << temp + 2 << endl;
fout.close();
```

**Related topics:**
- `close`
- `open`

### eof

**Syntax:**

```cpp
#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:

```cpp
ifstream fin("data.txt");
string s;
while( fin >> s ) {
    cout << "Read from file: " << s << endl;
}
```

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 I/O getline() function.

```cpp
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;
}
```
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 C++ string getline() function to read lines into strings:

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream fin("data.txt");
    std::string s;
    while( getline(fin,s) ) {
        std::cout << "Read from file: " << s << std::endl;
    }
    return 0;
}
```
Checking For Errors

Simply evaluating an I/O object in a boolean context will return false if any errors have occurred:

```cpp
string filename = "data.txt";
ifstream fin( filename.c_str() );
if( !fin ) {
    cout << "Error opening " << filename << " for input" << endl;
    exit(-1);
}
```

---

fail

**Syntax:**

```cpp
#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:**

```cpp
#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:**

```cpp
#include <fstream>
fmtflags flags();
fmtflags flags(fmtflags f);
```

The `flags()` function either returns the io stream format flags for the current stream, or sets the flags for the current stream to be `f`.

*Related topics:*

- `setf`
- `unsetf`

---

### flush

**Syntax:**

```cpp
#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:**

```cpp
#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

get

**Syntax:**

```cpp
#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:

```cpp
char ch;
ifstream fin( "temp.txt" );
while( fin.get(ch) )
    cout << ch;
fin.close();
```

**Related topics:**
- `gcount`
- `getline`
- `(C++ Strings) getline`
- `ignore`
- `peek`
- `put`
- `read`

---

**getline**

**Syntax:**

```cpp
#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:

```cpp
#include <fstream>

bool good();

ifstream fin("tmp.dat");
int MAX_LENGTH = 100;
char line[MAX_LENGTH];

while( fin.getline(line, MAX_LENGTH) ) {
    cout << "read line: " << line << endl;
}
```

If you'd like to read lines from a file into strings instead of character arrays, consider using the string getline 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:

```cpp
#include <fstream>
bool good();
```

The function good() returns true if no errors have occurred with the current
ignore

Syntax:

```cpp
#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`

open

Syntax:
The function open() is used with file streams. It opens `filename` and associates it with the current stream. The optional `io stream mode flag 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:

```cpp
#include <fstream>

void open( const char *filename );
void open( const char *filename, openmode mode = default_mode );

ifstream inputStream;
inputStream.open("file.txt");
if( !inputStream ) {
    cerr << "Error opening input stream" << endl;
    return;
}
```

Related topics:
- I/O Constructors
- close

---

**peek**

*Syntax:*

```cpp
#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;
```

This code displays the following output:

```
314.16
```

Related topics:
fill
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:
flush
## putback

**Syntax:**

```cpp
#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:**

```cpp
#include <fstream>
iosstate rdstate();
```

The `rdstate()` function returns the `iostream state flags` of the current stream.

**Related topics:**
- bad
- clear
- eof
- fail
- good
Syntax:

```cpp
#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:

```cpp
struct {
    int height;
    int width;
} rectangle;

input_file.read( (char *)&rectangle, sizeof(rectangle) );
if( input_file.bad() ) {
    cerr << "Error reading data" << endl;
    exit( 0 );
}
```

Related topics:
<gcoun>  
<get>  
<getline>  
<write>

---

**seekg**

Syntax:

```cpp
#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 io stream format flags 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 io stream format flags.

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;

Note that the preceding code is functionally identical to:

```cpp
int number = 0x3FF;
cout << "Decimal: " << number << endl << hex << "Hexadecimal: " << number << endl;
```

thanks to io stream manipulators.

Related topics: flags unsetf

---

**sync_with_stdio**

Syntax:

```cpp
#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:

```cpp
#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..."_INTERNAL_);
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();
```

Related topics:
seekg
seekp
tellg

unsetf

Syntax:

```
#include <fstream>
void unsetf( fmtflags flags );
```
The function `unsetf()` uses flags to clear the io stream format flags associated with the current stream.

*Related topics:*

flags
setf

---

### width

**Syntax:**

```cpp
#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:

```cpp
cout.width(5);
cout << "2";
```

displays

```
  2
```

(that's four spaces followed by a '2')

*Related topics:*

fill
precision

---

### write

**Syntax:**
The `write()` function is used with output streams, and writes `num` bytes from `buffer` to the current output stream.

*Related topics:*
flush
put
read
empty

Syntax:

```cpp
#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:

```cpp
vector<int> v;
for( int i = 0; i < 5; i++ ) {
    v.push_back(i);
}
while( !v.empty() ) {
    cout << v.back() << endl;
    v.pop_back();
}
```

Related topics:
size

---

pop

Syntax:

```cpp
#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& c = Container() );
priority_queue( input_iterator start, input_iterator end, const Compare& cmp = Compare(), const Container& c = Container() );
```

Priority queues can be constructed with an optional compare function `cmp` and an optional container `c`. If `start` and `end` are specified, the priority queue will be constructed with the elements between `start` and `end`.

push

Syntax:

```
#include <queue>
void push( const TYPE& val );
```

The function `push()` adds `val` to the end of the current priority queue.

For example, the following code uses the `push()` function to add ten integers to the end of a queue:

```
queue<int> q;
for( int i=0; i < 10; i++ )
  q.push(i);
```

size

Syntax:
The `size()` function returns the number of elements in the current priority queue.

**Related topics:**
(C++ Strings) **capacity**
(C++ Strings) **empty**
(C++ Strings) **length**
(C++ Multimaps) **max_size**
(C++ Strings) **resize**

---

**top**

**Syntax:**

```c++
#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:

```c++
while( !s.empty() ) {
    cout << s.top() << " ";
    s.pop();
}
```

**Related topics:**
**pop**
**back**

*Syntax:*

```cpp
#include <queue>
TYPE& back();
const TYPE& back() const;
```

The `back()` function returns a reference to the last element in the queue.

For example:

```cpp
queue<int> q;
for( int i = 0; i < 5; i++ ) {
    q.push(i);
}
cout << "The first element is " << q.front() << " and the last element is " << q.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The `back()` function runs in constant time.

*Related topics:*

- `front` *(C++ Lists)*
- `pop_back`
For example, the following code uses empty() as the stopping condition on a `while` loop to clear a queue while displaying its contents:

```cpp
queue<int> q;
for( int i = 0; i < 5; i++ ) {
  q.push(i);
}
while( !q.empty() ) {
  cout << q.front() << endl;
  q.pop();
}
```

Related topics:
- `size`

### `front`

**Syntax:**

```cpp
#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:**

```cpp
#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:**

```cpp
#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:

```cpp
queue<int> q;
for( int i=0; i < 10; i++ ) {
    q.push(i);
}
```

*Related topics:*

pop

---

**Queue constructor**

**Syntax:**

```cpp
#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:

```cpp
queue<string> waiting_line;
while( waiting_line.size() < 5 ) {
    cout << "Welcome to the line, please enter your name: ";
    string s;
    getline( cin, s );
    waiting_line.push(s);
}
while( !waiting_line.empty() ) {
    cout << "Now serving: " << waiting_line.front() << endl;
    waiting_line.pop();
}
```

When run, the above code might produce this output:

```
Welcome to the line, please enter your name: Nate
Welcome to the line, please enter your name: lizzy
Welcome to the line, please enter your name: Robert B. Parker
Welcome to the line, please enter your name: ralph
Welcome to the line, please enter your name: Matthew
Now serving: Nate
Now serving: lizzy
Now serving: Robert B. Parker
Now serving: ralph
Now serving: Matthew
```

**size**

Syntax:

```cpp
#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
**empty**

Syntax:

```cpp
#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:

```cpp
stack<int> s;
for (int i = 0; i < 5; i++) {
    s.push(i);
}
while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
}
```

*Related topics:*

- size

---

**pop**

Syntax:

```cpp
#include <stack>
void pop();
```

The function `pop()` removes the top element of the stack and discards it.

*Related topics:*

- push
- top
**push**

*Syntax:*

```cpp
#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:

```cpp
stack<int> s;
for( int i=0; i < 10; i++ )
  s.push(i);
```

*Related topics:*

`pop`

---

**size**

*Syntax:*

```cpp
#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:

```cpp
#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:

```cpp
#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:

```cpp
while( !s.empty() ) {
    cout << s.top() << " ";
    s.pop();
}
```

Related topics:

* pop
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:

```cpp
string str = "Hello World";
str.append( 10, '!' );
cout << str << endl;
```

That code displays:

```
Hello World!!!!!!!!!!
```

In the next example, `append()` is used to concatenate a substring of one string onto another string:

```cpp
string str1 = "Eventually I stopped caring...";
string str2 = "but that was the '80s so nobody noticed.";
```
When run, the above code displays:

```
str1 is Eventually I stopped caring...nobody noticed.
```

### assign

**Syntax:**

```cpp
#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:

```cpp
string str1, str2 = "War and Peace";
str1.assign( str2, 4, 3 );
cout << str1 << endl;
```

displays
This function will destroy the previous contents of the string.

Related topics:
(C++ Lists) assign

at

Syntax:

```cpp
#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:

```cpp
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
    cout << "Element " << i << " is " << v[i] << endl;
}
```

This code overruns the end of the vector, producing potentially dangerous results. The following code would be much safer:

```cpp
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
    cout << "Element " << i << " is " << v.at(i) << endl;
}
```

Instead of attempting to read garbage values from memory, the at() function will realize that it is about to overrun the vector and will throw an exception.

Related topics:
(C++ Multimaps) Multimap operators
(C++ Double-ended Queues) Container operators
**begin**

*Syntax:*

```cpp
#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:

```cpp
// 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;
}
```

*Related topics:*

- end
- rbegin
- rend

---

**c_str**

*Syntax:*

```cpp
#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 \texttt{const}, the character data that \texttt{c\_str()} returns \textbf{cannot be modified}. Furthermore, you do not need to call \texttt{free()} or \texttt{delete} on this pointer.

\textit{Related topics:}  
\textbf{String operators}  
\textbf{data}

\section*{capacity}

\textbf{Syntax:}

\begin{verbatim}
#include <string>

size_type capacity() const;
\end{verbatim}

The \texttt{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:

\begin{verbatim}
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;
\end{verbatim}

When run, the above code produces the following output:

\begin{verbatim}
The capacity of v1 is 10
The capacity of v2 is 20
\end{verbatim}

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 \texttt{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 \textit{constant time}.

\textit{Related topics:}
\texttt{reserve} \texttt{resize} \texttt{size}

---

\section*{clear}

\textit{Syntax:}

\begin{verbatim}
#include <string>
void clear();
\end{verbatim}

The function clear() deletes all of the elements in the string. clear() runs in \textit{linear time}.

\textit{Related topics:}
\texttt{(C++ Lists) erase}

---

\section*{compare}

\textit{Syntax:}

\begin{verbatim}
#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, size_type length2 );
\end{verbatim}
The `compare()` function either compares `str` to the current string in a variety of ways, returning

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than zero</td>
<td><code>this &lt; str</code></td>
</tr>
<tr>
<td>zero</td>
<td><code>this == str</code></td>
</tr>
<tr>
<td>greater than zero</td>
<td><code>this &gt; str</code></td>
</tr>
</tbody>
</table>

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 each other:

```cpp
string names[] = {"Homer", "Marge", "3-eyed fish", "inanimate carbon rod"};
for( int i = 0; i < 4; i++ ) {
    for( int j = 0; j < 4; j++ ) {
        cout << names[i].compare( names[j] ) << " ";
    }
    cout << endl;
}
```

Data from the above code was used to generate this table, which shows how the various strings compare to each other:

<table>
<thead>
<tr>
<th></th>
<th>Homer</th>
<th>Marge</th>
<th>3-eyed fish</th>
<th>inanimate carbon rod</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Homer&quot;.compare( x )</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>
#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:

```c
char buf[30];
memset( buf, '\0', 30 );
string str = "Trying is the first step towards failure.";
str.copy( buf, 24 );
cout << buf << endl;
```

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**

**data**

*Syntax:*

```cpp
#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:*

```cpp
#include <string>
bool empty() const;
```

The `empty()` function returns true if the string has no elements, false otherwise.

For example:

```cpp
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;
```

When run, this code produces the following output:

```
true
true
```
end

Syntax:

```cpp
#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 `begin()` and `end()` to iterate through all of the members of a vector:

```cpp
vector<int> v1(5, 789);
vector<int>::iterator it;
for (it = v1.begin(); it != v1.end(); it++) {
    cout << *it << endl;
}
```

The iterator is initialized with a call to `begin()`. After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling `end()`. Since `end()` returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

`end()` runs in `constant time`.

Related topics:

- `begin`
- `rbegin`
- `rend`
erase

Syntax:

```cpp
#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:

```cpp
string s("So, you like donuts, eh? Well, have all the donuts in the world!");
cout << "The original string is " " << s " " "n" " endl;

s.erase( 50, 14);
cout << "Now the string is " " << s " " "n" " endl;

s.erase( 24 );
cout << "Now the string is " " << s " " "n" " endl;

s.erase();
cout << "Now the string is " " << s " " "n" " endl;
```

will display

```
The original string is 'So, you like donuts, eh? Well, have all the donuts in the world!'
Now the string is 'So, you like donuts, eh? Well, have all the donuts'
Now the string is 'So, you like donuts, eh?'
Now the string is ''
```
erase() runs in linear time.

Related topics:
insert

---

**find**

Syntax:

```cpp
#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 length );
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:

```cpp
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;
}
```

Related topics:
find_first_not_of
find_first_of
find_last_not_of
find_last_of
**find_first_not_of**

Syntax:

```cpp
#include <string>
size_type find_first_not_of( const string& str, size_type index = 0);
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, size_type size);
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 hyphen:

```cpp
string lower_case = "abcdefghijklmnopqrstuvwxyz,-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-CASE part";
string::size_type index = 0;
for ( ; index < str.size() ; ++index ) {
    if ( lower_case.find(str[index]) == string::npos ) {
        cout << "first non-lower-case letter in str at: " << index << endl;
        break;
    }
}
```

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_of

Syntax:

```cpp
#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(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:

```cpp
#include <string>
size_type find_last_not_of( const string& str, size_type index = npos );
size_type find_last_not_of( const char* str, size_type index = npos );
size_type find_last_not_of( const char* str, size_type index, size_type 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:

```cpp
string lower_case = "abcdefghijklmnopqrstuvwxyz";
string str = "abcdefgABCDEFGhijklmnop";
cout << "last non-lower-case letter in str at: " << str.find_last_not_of(lower_case);
```

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:

```cpp
#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_type size )
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`

getline

Syntax:

```cpp
#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:

```cpp
string s;
getline( cin, s );
cout << "You entered " << s << endl;
```

After getting a line of data in a string, you may find that string streams 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:

```cpp
// 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;
        }
    }
}
```

When run with a user supplying input, the above code might produce this output:

```plaintext
// 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
```
Related topics:
(C++ I/O) **get**
(C++ I/O) **getline**
**string streams**

## insert

**Syntax:**

```cpp
#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 index2 );
string& insert( size_type index, const char* str, size_type num );
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:

```cpp
#include <string>
size_type length() const;
```

The `length()` function returns the number of elements in the current string, performing the same role as the `size()` function.

Related topics:

`size`

max_size

Syntax:

```cpp
#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:

`size`

push_back

Syntax:

```cpp
#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:

```cpp
string the_string;
for (int i = 0; i < 10; i++)
    the_string.push_back( i+'a' );
```

When displayed, the resulting string would look like this:

```
abcdefghij
```

`push_back()` runs in **constant time**.

*Related topics:*
- `assign`
- `insert`

---

**rbegin**

*Syntax:*

```cpp
#include <string>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The `rbegin()` function returns a `reverse_iterator` to the end of the current string.

`rbegin()` runs in **constant time**.

*Related topics:*
- `begin`
- `end`
- `rend`

---

**rend**
Syntax:

```cpp
#include <string>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function `rend()` returns a `reverse_iterator` to the beginning of the current string.

`rend()` runs in `constant time`.

**Related topics:**
- `begin`
- `end`
- `rbegin`

---

replace

Syntax:

```cpp
#include <string>
string& replace(size_type index, size_type num, const string& str);
string& replace(size_type index1, size_type num1, const string& str);
string& replace(size_type index, size_type num, const char* str);
string& replace(size_type index, size_type num1, const char* str);
string& replace(iterator start, iterator end, const string& str);
string& replace(iterator start, iterator end, const char* str);
string& replace(iterator start, iterator end, size_type num, char ch);
```

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 \textit{num1} characters in the current string (beginning at \textit{index1}) with \textit{num2} characters from \textit{str} beginning at \textit{index2},
- replaces up to \textit{num1} characters in the current string (beginning at \textit{index}) with \textit{num2} copies of \textit{ch},
- replaces the characters in the current string from \textit{start} to \textit{end} with \textit{str},
- replaces characters in the current string from \textit{start} to \textit{end} with \textit{num} characters from \textit{str},
- or replaces the characters in the current string from \textit{start} to \textit{end} with \textit{num} copies of \textit{ch}.

For example, the following code displays the string "They say he carved it himself...find your soul-mate, Homer."

```cpp
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;
```

\textit{Related topics: insert}

---

**reserve**

\textit{Syntax:}

```cpp
#include <string>
void reserve( size_type size );
```

The \texttt{reserve()} function sets the capacity of the string to at least \texttt{size}.

\texttt{reserve()} runs in \texttt{linear time}.

\textit{Related topics: capacity}

---

**resize**
Syntax:

```cpp
#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) [Multimap constructors & destructors](#)
[capacity](#)
[size](#)

---

### rfind

**Syntax:**

```cpp
#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.
```cpp
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;
```

**Related topics:**
- find
- find_first_not_of
- find_first_of
- find_last_not_of
- find_last_of

## size

**Syntax:**

```cpp
#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:**

```cpp
#include <string>
string();
string( const string& s );
string( size_type length, const char& ch );
```
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,

```cpp
string str1( 5, 'c' );
string str2( "Now is the time..." );
string str3( str2, 11, 4 );
cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;
```

displays

```
cccccc
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:

```cpp
#include <string>
bool operator==(const string& c1, const string& c2);
bool operator!=(const string& c1, const string& c2);
```
### C++ strings

C++ strings can be compared and assigned with the standard comparison operators: `==`, `!=`, `<=`, `>=`, `<`, `>`, and `=`. Performing a comparison or assigning one string to another takes linear time.

Two strings are equal if:

1. Their size is the same, and
2. Each member in location `i` in one string is equal to the member in location `i` in the other string.

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:

```cpp
string s1 = "Now is the time...";
string s2 = "for all good men...";
string s3 = s1 + s2;
cout << "s3 is " << s3 << endl;
```

Furthermore, strings can be assigned values that are other strings, character arrays, or even single characters. The following code is perfectly valid:

```cpp
char ch = 'N';
string s;
s = ch;
```
Individual characters of a string can be examined with the [] operator, which runs in constant time.

Related topics:
c_str
compare
data

---

**substr**

Syntax:

```cpp
#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:

```cpp
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;
```

displays

```
The original string is What we have here is a failure to communicate
The substring is a failure to communicate
```

Related topics:
copy

---

**swap**
Syntax:

```cpp
#include <string>
void swap( container& from );
```

The `swap()` function exchanges the elements of the current string with those of `from`. This function operates in constant time.

For example, the following code uses the `swap()` function to exchange the values of two strings:

```cpp
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

Related topics:

(C++ Lists) **splice**
accumulate

Syntax:

```cpp
#include <numeric>
TYPE accumulate( iterator start, iterator end, TYPE val );
TYPE accumulate( iterator start, iterator end, TYPE val, BinaryFunction f );
```

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:

```cpp
#include <numeric>
iterator adjacent_difference( iterator start, iterator end, iterator result );
iterator adjacent_difference( iterator start, iterator end, iterator result, BinaryFunction f );
```

The `adjacent_difference` function calculates the differences between adjacent elements in the range `[start,end)` and stores the result starting at `result`. If a binary function `f` is given, it is used instead of the `-` operator to compute the differences.
adjacent_difference() runs in linear time.

Related topics:
accumulate
count
inner_product
partial_sum

adjacent_find

Syntax:

```cpp
#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:

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back(i);
    // add a duplicate 7 into v1
    if( i == 7 ) {
        v1.push_back(i);
    }
}

vector<int>::iterator result;
result = adjacent_find( v1.begin(), v1.end() );
if( result == v1.end() ) {
```
cout << "Did not find adjacent elements in v1" << endl;
}
else {
    cout << "Found matching adjacent elements starting at " << *result << endl;
}

Related topics:
find
find_end
find_first_of
find_if
unique
unique_copy

**binary_search**

**Syntax:**
```
#include <algorithm>
bool binary_search( iterator start, iterator end, const TYPE& val
bool binary_search( iterator start, iterator end, const TYPE& val
```

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 ) ) {
```
```cpp
    cout << "nums[] contains " << i << endl;
} else {
    cout << "nums[] DOES NOT contain " << i << endl;
}
```

When run, this code displays the following output:

```
nums[] contains 0
nums[] DOES NOT contain 1
nums[] DOES NOT contain 2
nums[] DOES NOT contain 3
nums[] DOES NOT contain 4
nums[] contains 5
nums[] DOES NOT contain 6
nums[] DOES NOT contain 7
nums[] contains 8
nums[] contains 9
```

**Related topics:**
- `equal_range`
- `is_sorted`
- `lower_bound`
- `partial_sort`
- `partial_sort_copy`
- `sort`
- `stable_sort`
- `upper_bound`

## copy

**Syntax:**

```cpp
#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,

```cpp
*dest == *start
*(dest+1) == *(start+1)
```
The return value is an iterator to the last element copied. `copy()` runs in **linear**
time.

For example, the following code uses `copy()` to copy the contents of one vector
to another:

```cpp
vector<int> from_vector;
for( int i = 0; i < 10; i++ ) {
    from_vector.push_back( i );
}

vector<int> to_vector(10);

copy( from_vector.begin(), from_vector.end(), to_vector.begin() );

cout << "to_vector contains: ";
for( unsigned int i = 0; i < to_vector.size(); i++ ) {
    cout << to_vector[i] << " ";
}

cout << endl;
```

**Related topics:**
- `copy_backward`
- `copy_n`
- `generate`
- `remove_copy`
- `swap`
- `transform`

## `copy_backward`

**Syntax:**

```cpp
#include <algorithm>
iterator copy_backward( iterator start, iterator end, iterator dest );
```

copy_backward() is similar to (C++ Strings) `copy()`, in that both functions copy
elements from \textit{start} to \textit{end} to \textit{dest}. The \texttt{copy_backward()} function, however, starts depositing elements at \textit{dest} and then works backwards, such that:

\begin{verbatim}
*(dest-1) == *(end-1)
*(dest-2) == *(end-2)
*(dest-3) == *(end-3)
...
*(dest-N) == *(end-N)
\end{verbatim}

The following code uses \texttt{copy_backward()} to copy 10 integers into the end of an empty vector:

\begin{verbatim}
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;
\end{verbatim}

The above code produces the following output:

\begin{verbatim}
to_vector contains: 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9
\end{verbatim}

Related topics:

- \texttt{copy}
- \texttt{copy\_n}
- \texttt{swap}

\section*{copy\_n}

\textbf{Syntax:}

\begin{verbatim}
#include <algorithm>
iterator copy_n( iterator from, size_t num, iterator to );
\end{verbatim}
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:

```cpp
#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:

```cpp
vector<int> v;
for( int i = 0; i < 10; i++ ) {
    v.push_back( i );
}

int target_value = 3;
int num_items = count( v.begin(), v.end(), target_value );
cout << "v contains " << num_items << " items matching " << target_value << endl;
```

The above code displays the following output:

```
v contains 1 items matching 3
```
Related topics:
accumulate
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:

```
int nums[] = { 0, 1, 2, 3, 4, 5, 9, 3, 13 };
int start = 0;
int end = 9;

int target_value = 3;
int num_items = count_if( nums+start, 
                           nums+end, 
                           bind2nd(equal_to<int>(), target_value) );

cout << "nums[] contains " << num_items << " items matching " << target_value << endl;
```

When run, the above code displays the following output:

```
nums[] contains 2 items matching 3
```
equal

Syntax:

```cpp
#include <algorithm>
bool equal( iterator start1, iterator end1, iterator start2 );
bool equal( iterator start1, iterator end1, iterator start2, BinPred p );
```

The `equal()` function returns true if the elements in two ranges are the same. The first range of elements are those between `start1` and `end1`. The second range of elements has the same size as the first range but starts at `start2`.

If the binary predicate `p` is specified, then it is used instead of `==` to compare each pair of elements.

For example, the following code uses `equal()` to compare two vectors of integers:

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

vector<int> v2;
for( int i = 0; i < 10; i++ ) {
    v2.push_back( i );
}

if( equal( v1.begin(), v1.end(), v2.begin() ) ) {
    cout << "v1 and v2 are equal" << endl;
} else {
    cout << "v1 and v2 are NOT equal" << endl;
}
```

Related topics:
- find_if
- lexicographical_compare
- mismatch
- search
equal_range

Syntax:

```cpp
#include <algorithm>
pair<iterator,iterator> equal_range( iterator first, iterator last, const pair<iterator,iterator> &val );
```

The equal_range() function returns the range of elements between `first` and `last` that are equal to `val`. This function assumes that the elements between `first` and `last` are in order according to `comp`, if it is specified, or the `<` operator otherwise.

equal_range() can be thought of as a combination of the lower_bound() and `upper_bound1`() functions, since the first of the pair of iterators that it returns is what lower_bound() returns and the second iterator in the pair is what `upper_bound1`() returns.

For example, the following code uses equal_range() to determine all of the possible places that the number 8 can be inserted into an ordered vector of integers such that the existing ordering is preserved:

```cpp
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );

pair<vector<int>::iterator, vector<int>::iterator> result;
int new_val = 8;

result = equal_range( nums.begin(), nums.end(), new_val );

cout << "The first place that " << new_val << " could be inserted is before " << *result.first << ", and the last place that it could be inserted is before " << *result.second << endl;
```

The above code produces the following output:

```
The first place that 8 could be inserted is before 8,
```
and the last place that it could be inserted is before 11

Related topics:
- binary_search
- lower_bound
- upper_bound

**fill**

Syntax:

```cpp
#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:

```cpp
type<type> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

cout << "Before, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
    cout << v1[i] << " ";
}
cout << endl;

fill( v1.begin(), v1.end(), -1 );

cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
    cout << v1[i] << " ";
}
cout << endl;
```

When run, the above code displays:

```
Before, v1 is: 0 1 2 3 4 5 6 7 8 9
```
After, v1 is: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Related topics:
fill_n
generate
transform

**fill_n**

**Syntax:**

```cpp
#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:

```cpp
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
    v1.push_back( i );
}

cout << "Before, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
    cout << v1[i] << " ";
}
cout << endl;
fill_n( v1.begin(), v1.size()/2, -1 );
cout << "After, v1 is: ";
for( unsigned int i = 0; i < v1.size(); i++ ) {
    cout << v1[i] << " ";
}
cout << endl;
```

When run, this code displays:
Before, v1 is: 0 1 2 3 4 5 6 7 8 9
After, v1 is: -1 -1 -1 -1 -1 5 6 7 8 9

Related topics:
fill

find

Syntax:

```cpp
#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:

```cpp
int num_to_find = 3;

vector<int> v1;
for( int i = 0; i < 10; i++ ){
    v1.push_back(i);
}

vector<int>::iterator result;
result = find( v1.begin(), v1.end(), num_to_find );

if( result == v1.end() ){
    cout << "Did not find any element matching " << num_to_find << endl;
}
else {
    cout << "Found a matching element: " << *result << endl;
}
```

In the next example, shown below, the `find()` function is used on an array of integers. This example shows how the C++ Algorithms can be used to manipulate arrays and pointers in the same manner that they manipulate
containers and iterators:

```cpp
int nums[] = { 3, 1, 4, 1, 5, 9 };

int num_to_find = 5;
int start = 0;
int end = 2;
int* result = find( nums + start, nums + end, num_to_find );

if( result == nums + end ) {
    cout << "Did not find any number matching " << num_to_find << endl;
} else {
    cout << "Found a matching number: " << *result << endl;
}
```

Related topics:
- adjacent_find
- find_end
- find_first_of
- find_if
- mismatch
- search

**find_end**

Syntax:

```cpp
#include <algorithm>

iterator find_end( iterator start, iterator end, iterator seq_start, iterator seq_end );
```

The `find_end()` function searches for the sequence of elements denoted by `seq_start` and `seq_end`. If such a sequence is 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:

```cpp
int nums[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4 };  // Target sequence
int* result;  // Pointer to store result
int start = 0;
int end = 11;

int target1[] = { 1, 2, 3 };  // Target sequence
result = find_end( nums + start, nums + end, target1 + 0, target1 + 2 );
if( *result == nums[end] ) {
    cout << "Did not find any subsequence matching \{ 1, 2, 3 \}" << endl;
} else {
    cout << "The last matching subsequence is at: " << *result << endl;
}

int target2[] = { 3, 2, 3 };  // Target sequence
result = find_end( nums + start, nums + end, target2 + 0, target2 + 2 );
if( *result == nums[end] ) {
    cout << "Did not find any subsequence matching \{ 3, 2, 3 \}" << endl;
} else {
    cout << "The last matching subsequence is at: " << *result << endl;
}
```

Related topics:
- `adjacent_find`
- `find`
- `find_first_of`
- `find_if`
- `search_n`

### **find_first_of**

Syntax:

```cpp
#include <algorithm>

iterator find_first_of( iterator start, iterator end, iterator find_start, iterator find_end, BinPred bp );
```

The `find_first_of()` function searches for the first occurrence 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:

```cpp
int nums[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int* result;
int start = 0;
int end = 10;

int targets[] = { 9, 4, 7 };
result = find_first_of( nums + start, nums + end, targets + 0, targets + 2);
if(*result == nums[end]) {
    cout << "Did not find any of { 9, 4, 7 }" << endl;
} else {
    cout << "Found a matching target: " << *result << endl;
}
```

Related topics:
- adjacent_find
- find
- find_end
- find_if
- (Standard C String and Character) strpbrk

## find_if

**Syntax:**

```cpp
#include <algorithm>
iterator find_if( iterator start, iterator end, UnPred up );
```

The find_if() function searches for the first element between start and end for which the unary predicate up returns true.

If such an element is found, an iterator pointing to that element is returned. Otherwise, an iterator pointing to end is returned.

For example, the following code uses find_if() and a "greater-than-zero" unary predicate to the first positive, non-zero number in a list of numbers:
int nums[] = { 0, -1, -2, -3, -4, 342, -5 };  
int* result;  
int start = 0;  
int end = 7;  

result = find_if( nums + start, nums + end, bind2nd(greater<int>(), 0));  
if( *result == nums[end] ) {  
    cout << "Did not find any number greater than zero" << endl;  
} else {  
    cout << "Found a positive non-zero number: " << *result << endl;  
}

Related topics:
adjacent_find
equal
find
find_end
find_first_of
search_n

for_each

Syntax:

```cpp
#include <algorithm>
UnaryFunction for_each( iterator start, iterator end, UnaryFunction f );
```

The for_each() algorithm applies the function f to each of the elements between start and end. The return value of for_each() is f.

For example, the following code snippets define a unary function then use it to increment all of the elements of an array:

```cpp
template<class TYPE> struct increment : public unary_function<TYPE, void>  
    {  
    void operator() (TYPE& x) {  
        x++;  
    }  
};  

...  

int nums[] = {3, 4, 2, 9, 15, 267};
```
The above code displays the following output:

<table>
<thead>
<tr>
<th>Before, nums[] is: 3 4 2 9 15 267</th>
</tr>
</thead>
<tbody>
<tr>
<td>After, nums[] is: 4 5 3 10 16 268</td>
</tr>
</tbody>
</table>

**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
Syntax:

```cpp
#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:

```cpp
#include <algorithm>
bool includes( iterator start1, iterator end1, iterator start2, iterator end2 );
bool includes( iterator start1, iterator end1, iterator start2, iterator end2, StrictWeakOrdering cmp );
```

The `includes()` algorithm returns true if every element in `[start2, end2)` is also in `[start1, end1)`. Both of the given ranges must be sorted in ascending order.

By default, the `<` operator is used to compare elements. If the strict weak ordering function object `cmp` is given, then it is used instead.

`includes()` runs in **linear time**.

Related topics:

- `set_difference`
- `set_intersection`
- `set_symmetric_difference`
- `set_union`

---

**inner_product**

Syntax:
The `inner_product()` function computes the inner product of \([start1, end1]\) and a range of the same size starting at \(start2\).

`inner_product()` runs in **linear time**.

**Related topics:**
- `accumulate`
- `adjacent_difference`
- `count`
- `partial_sum`

---

### inplace_merge

**Syntax:**

```cpp
#include <algorithm>
inline void inplace_merge( iterator start, iterator middle, iterator end );
inline void inplace_merge( iterator start, iterator middle, iterator end, StrictWeakOrdering cmp );
```

The `inplace_merge()` function is similar to the `merge()` function, but instead of creating a new sorted range of elements, `inplace_merge()` alters the existing ranges to perform the merge in-place.

**Related topics:**
- `merge`

---

### is_heap

**Syntax:**

```cpp
#include <algorithm>
bool is_heap( iterator start, iterator end );
bool is_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```
The \texttt{is_heap()} function returns true if the given range \([\textit{start,end}]\) is a heap.

If the strict weak ordering comparison function object \textit{cmp} is given, then it is used instead of the \texttt{<} operator to compare elements.

\texttt{is_heap()} runs in \texttt{linear time}.

\textit{Related topics:}
\begin{itemize}
    \item \texttt{make_heap}
    \item \texttt{pop_heap}
    \item \texttt{push_heap}
    \item \texttt{sort_heap}
\end{itemize}

---

\textbf{is_sorted}

\textit{Syntax:}

\begin{verbatim}
#include <algorithm>
bool is_sorted( iterator start, iterator end );
bool is_sorted( iterator start, iterator end, StrictWeakOrdering cmp );
\end{verbatim}

The \texttt{is_sorted()} algorithm returns true if the elements in the range \([\textit{start,end}]\) are sorted in ascending order.

By default, the \texttt{<} operator is used to compare elements. If the strict weak order function object \textit{cmp} is given, then it is used instead.

\texttt{is_sorted()} runs in \texttt{linear time}.

\textit{Related topics:}
\begin{itemize}
    \item \texttt{binary_search}
    \item \texttt{partial_sort}
    \item \texttt{partial_sort_copy}
    \item \texttt{sort}
    \item \texttt{stable_sort}
\end{itemize}
**iter_swap**

Syntax:

```cpp
#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

```cpp
swap(*a, *b);
```

would.

*Related topics:*

- swap
- swap_ranges

---

**lexicographical_compare**

Syntax:

```cpp
#include <algorithm>
bool lexicographical_compare(iterator start1, iterator end1, iterator start2, iterator end2);
bool lexicographical_compare(iterator start1, iterator end1, iterator start2, iterator end2, BinPred p);
```

The `lexicographical_compare()` function returns true if the range of elements `[start1,end1)` is lexicographically less than the range of elements `[start2,end2)`. If you're confused about what lexicographic means, it might help to know that dictionaries are ordered lexicographically.

`lexicographical_compare()` runs in linear time.

*Related topics:*

- equal
- lexicographical_compare_3way
- mismatch
**lexicographical_compare_3way**

*Syntax:*

```cpp
#include <algorithm>
int lexicographical_compare_3way( iterator start1, iterator end1,
                                iterator start2, iterator end2 );
```

The `lexicographical_compare_3way()` function compares the first range, defined by `[start1,end1)` to the second range, defined by `[start2,end2)`.

If the first range is lexicographically less than the second range, this function returns a negative number. If the first range is lexicographically greater than the second, a positive number is returned. Zero is returned if neither range is lexicographically greater than the other.

`lexicographical_compare_3way()` runs in *linear time*.

*Related topics:*

- [lexicographical_compare](#)

---

**lower_bound**

*Syntax:*

```cpp
#include <algorithm>
iterator lower_bound( iterator first, iterator last, const TYPE& val );
iterator lower_bound( iterator first, iterator last, const TYPE& val,
                     CompFn f );
```

The `lower_bound()` function is a type of `binary_search()`. This function searches for the first place that `val` can be inserted into the ordered range defined by `first` and `last` that will not mess up the existing ordering.

The return value of `lower_bound()` is an iterator that points to the location where `val` can be safely inserted. Unless the comparison function `f` is specified, the `<
operator is used for ordering.

For example, the following code uses lower_bound() to insert the number 7 into an ordered vector of integers:

```cpp
vector<int> nums;
nums.push_back( -242 );
nums.push_back( -1 );
nums.push_back( 0 );
nums.push_back( 5 );
nums.push_back( 8 );
nums.push_back( 8 );
nums.push_back( 11 );

cout << "Before nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
    cout << nums[i] << " ";
}
cout << endl;

vector<int>::iterator result;
int new_val = 7;
result = lower_bound( nums.begin(), nums.end(), new_val );
nums.insert( result, new_val );

cout << "After, nums is: ";
for( unsigned int i = 0; i < nums.size(); i++ ) {
    cout << nums[i] << " ";
}
cout << endl;
```

The above code produces the following output:

```
Before nums is: -242 -1 0 5 8 8 11
After, nums is: -242 -1 0 5 7 8 8 11
```

Related topics:
- [binary_search](#)
- [equal_range](#)
- [make_heap](#)
Syntax:

```cpp
#include <algorithm>
void make_heap( iterator start, iterator end );
void make_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The `make_heap()` function turns the given range of elements \([start,end]\) into a heap.

If the strict weak ordering comparison function object `cmp` is given, then it is used instead of the `<` operator to compare elements.

`make_heap()` runs in **linear time**.

**Related topics:**
- `is_heap`
- `pop_heap`
- `push_heap`
- `sort_heap`

---

**max**

Syntax:

```cpp
#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:

```cpp
cout << "Max of 1 and 9999 is " << max( 1, 9999 ) << endl;
```
cout << "Max of 'a' and 'b' is " << max('a', 'b') << endl;
cout << "Max of 3.14159 and 2.71828 is " << max(3.14159, 2.71828) << endl;

When run, this code displays:

Max of 1 and 9999 is 9999
Max of 'a' and 'b' is b
Max of 3.14159 and 2.71828 is 3.14159

Related topics:
max_element
min
min_element

max_element

Syntax:

```cpp
#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:

```cpp
int array[] = { 3, 1, 4, 1, 5, 9 };
unsigned int array_size = 6;
cout << "Max element in array is " << *max_element( array, array+array_size ) << endl;

vector<char> v;
v.push_back('a'); v.push_back('b'); v.push_back('c'); v.push_back('d');
cout << "Max element in the vector v is " << *max_element( v.begin(), v.end() ) << endl;
```
When run, the above code displays this output:

<table>
<thead>
<tr>
<th>Max element in array is 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max element in the vector v is d</td>
</tr>
</tbody>
</table>

Related topics: `max` `min` `min_element`

## merge

Syntax:

```cpp
#include <algorithm>
iterator merge(iterator start1, iterator end1, iterator start2, iterator end2, iterator result);
iterator merge(iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp);
```

The `merge()` function combines two sorted ranges `[start1,end1)` and `[start2,end2)` into a single sorted range, stored starting at `result`. The return value of this function is an iterator to the end of the merged range.

If the strict weak ordering function object `cmp` is given, then it is used in place of the `<` operator to perform comparisons between elements.

`merge()` runs in linear time.

Related topics: `inplace_merge` `set_union` `sort`

## min

Syntax:

```cpp
#include <algorithm>
```
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:

```cpp
#include <algorithm>
iterator min_element( iterator start, iterator end );
iterator min_element( iterator start, iterator end, BinPred p );
```

The `min_element()` function returns an iterator to the smallest element in the range `[start,end)`.

If the binary predicate `p` is given, then it will be used instead of the `<` operator to determine the smallest element.

**Related topics:**
- `max`
- `max_element`
- `min`

---

**mismatch**

Syntax:

```cpp
#include <algorithm>
```
The mismatch() function compares the elements in the range defined by \([start1, end1]\) to the elements in a range of the same size starting at \(start2\). The return value of mismatch() is the first location where the two ranges differ.

If the optional binary predicate \(p\) is given, then it is used to compare elements from the two ranges.

The mismatch() algorithm runs in linear time.

**Related topics:**
- equal
- find
- lexicographical_compare
- search

---

**next_permutation**

**Syntax:**

```cpp
#include <algorithm>
bool next_permutation( iterator start, iterator end );
bool next_permutation( iterator start, iterator end, StrictWeakOrdering cmp );
```

The next_permutation() function attempts to transform the given range of elements \([start, end]\) into the next lexicographically greater permutation of elements. If it succeeds, it returns true, otherwise, it returns false.

If a strict weak ordering function object \(cmp\) is provided, it is used in lieu of the \(<\) operator when comparing elements.

**Related topics:**
- prev_permutation
- random_sample
- random_sample_n
- random_shuffle
### nth_element

**Syntax:**

```cpp
#include <algorithm>

void nth_element( iterator start, iterator middle, iterator end )
void nth_element( iterator start, iterator middle, iterator end, StrictWeakOrdering cmp );
```

The `nth_element()` function semi-sorts the range of elements defined by `[start,end)`. It puts the element that `middle` points to in the place that it would be if the entire range was sorted, and it makes sure that none of the elements before that element are greater than any of the elements that come after that element.

`nth_element()` runs in **linear time** on average.

**Related topics:**
- `partial_sort`

### partial_sort

**Syntax:**

```cpp
#include <algorithm>

void partial_sort( iterator start, iterator middle, iterator end )
void partial_sort( iterator start, iterator middle, iterator end, StrictWeakOrdering cmp );
```

The `partial_sort()` function arranges the first N elements of the range `[start,end)` in ascending order. N is defined as the number of elements between `start` and `middle`.

By default, the `<` operator is used to compare two elements. If the strict weak ordering comparison function `cmp` is given, it is used instead.

**Related topics:**
- `binary_search`
- `is_sorted`
- `nth_element`
**partial_sort_copy**

Syntax:

```cpp
#include <algorithm>
iterator partial_sort_copy(iterator start, iterator end, iterator result_start, iterator result_end);
iterator partial_sort_copy(iterator start, iterator end, iterator result_start, iterator result_end, StrictWeakOrdering cmp);
```

The `partial_sort_copy()` algorithm behaves like `partial_sort()`, except that instead of partially sorting the range in-place, a copy of the range is created and the sorting takes place in the copy. The initial range is defined by `[start,end)` and the location of the copy is defined by `[result_start,result_end)`.

`partial_sort_copy()` returns an iterator to the end of the copied, partially-sorted range of elements.

Related topics:
- `binary_search`
- `is_sorted`
- `partial_sort`
- `sort`
- `stable_sort`

---

**partial_sum**

Syntax:

```cpp
#include <numeric>
iterator partial_sum(iterator start, iterator end, iterator result);
iterator partial_sum(iterator start, iterator end, iterator result, BinOp p);
```

The `partial_sum()` function calculates the partial sum of a range defined by
[\text{start, end}], storing the output at \text{result}.

- \text{start} is assigned to *\text{result}, the sum of *\text{start} and *(\text{start} + 1) is assigned to *(\text{result} + 1), etc.

\text{partial\_sum()} runs in \text{linear time}.

\textit{Related topics:}
\begin{itemize}
  \item \texttt{accumulate}
  \item \texttt{adjacent\_difference}
  \item \texttt{count}
  \item \texttt{inner\_product}
\end{itemize}

\section*{partition}

\begin{itemize}
  \item Syntax:
  \begin{verbatim}
  \#include <algorithm>
  iterator partition( iterator start, iterator end, Predicate p );
  \end{verbatim}
\end{itemize}

The partition() algorithm re-orders the elements in [\text{start, end}] such that the elements for which the predicate \text{p} returns true come before the elements for which \text{p} returns false.

In other words, partition() uses \text{p} to divide the elements into two groups.

The return value of partition() is an iterator to the first element for which \text{p} returns false.

partition() runs in \text{linear time}.

\textit{Related topics:}
\begin{itemize}
  \item \texttt{stable\_partition}
\end{itemize}

\section*{pop\_heap}
### pop_heap

Syntax:
```
#include <algorithm>
void pop_heap( iterator start, iterator end );
void pop_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The `pop_heap()` function removes the largest element (defined as the element at the front of the heap) from the given heap.

If the strict weak ordering comparison function object `cmp` is given, then it is used instead of the `<` operator to compare elements.

`pop_heap()` runs in **logarithmic time**.

**Related topics:**
- `is_heap`
- `make_heap`
- `push_heap`
- `sort_heap`

---

### prev_permutation

Syntax:
```
#include <algorithm>
bool prev_permutation( iterator start, iterator end );
bool prev_permutation( iterator start, iterator end, StrictWeakOrdering cmp );
```

The `prev_permutation()` function attempts to transform the given range of elements `[start,end)` into the next lexicographically smaller permutation of elements. If it succeeds, it returns true, otherwise, it returns false.

If a strict weak ordering function object `cmp` is provided, it is used instead of the `<` operator when comparing elements.

**Related topics:**
- `next_permutation`
- `random_sample`
- `random_sample_n`
random_shuffle

push_heap

Syntax:

```cpp
#include <algorithm>
void push_heap( iterator start, iterator end );
void push_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The `push_heap()` function adds an element (defined as the last element before `end`) to a heap (defined as the range of elements between `[start,end-1)`).

If the strict weak ordering comparison function object `cmp` is given, then it is used instead of the `<` operator to compare elements.

`push_heap()` runs in logarithmic time.

Related topics:
- is_heap
- make_heap
- pop_heap
- sort_heap

random_sample

Syntax:

```cpp
#include <algorithm>
iterator random_sample( iterator start1, iterator end1, iterator start2, iterator end2 );
iterator random_sample( iterator start1, iterator end1, iterator end1, iterator end1, iterator end2 );
```

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 \textit{linear time}.

\textit{Related topics:}

\texttt{next_permutation}
\texttt{prev_permutation}
\texttt{random_sample_n}
\texttt{random_shuffle}

---

**random_sample_n**

\textit{Syntax:}

\begin{verbatim}
#include <algorithm>
iterator random_sample_n( iterator start, iterator end, iterator result, size_t N );
iterator random_sample_n( iterator start, iterator end, iterator result, size_t N, RandomNumberGenerator& rnd );
\end{verbatim}

The \texttt{random_sample_n()} algorithm randomly copies \( N \) elements from \([\text{start},\text{end})\) to \textit{result}. Elements are chosen with uniform probability and elements from the input range will appear at most once in the output range. \textbf{Element order is preserved} from the input range to the output range.

If a random number generator function object \textit{rnd} is supplied, then it will be used instead of an internal random number generator.

The return value of \texttt{random_sample_n()} is an iterator to the end of the output range.

\texttt{random_sample_n()} runs in \textit{linear time}.

\textit{Related topics:}

\texttt{next_permutation}
\texttt{prev_permutation}
\texttt{random_sample}
random_shuffle

random_shuffle

Syntax:

```c++
#include <algorithm>
void random_shuffle( iterator start, iterator end );
void random_shuffle( iterator start, iterator end, RandomNumberGenerator& rnd );
```

The random_shuffle() function randomly re-orders the elements in the range [start,end). If a random number generator function object rnd is supplied, it will be used instead of an internal random number generator.

Related topics:
- next_permutation
- prev_permutation
- random_sample
- random_sample_n

remove

Syntax:

```c++
#include <algorithm>
iterator remove( iterator start, iterator end, const TYPE& val );
```

The remove() algorithm removes all of the elements in the range [start,end) that are equal to val.

The return value of this function is an iterator to the last element of the new sequence that should contain no elements equal to val.

The remove() function runs in linear time.

Related topics:
remove_copy
remove_copy_if
remove_if
unique
unique_copy

remove_copy

Syntax:

```cpp
#include <algorithm>
iterator remove_copy( iterator start, iterator end, iterator result,
```

The `remove_copy()` algorithm copies the range `[start,end)` to `result` but omits any elements that are equal to `val`.

`remove_copy()` returns an iterator to the end of the new range, and runs in linear time.

Related topics:
copy
table
remove
remove_copy_if
remove_if

remove_copy_if

Syntax:

```cpp
#include <algorithm>
iterator remove_copy_if( iterator start, iterator end, iterator resu
```

The `remove_copy_if()` function copies the range of elements `[start,end)` to `result`, omitting any elements for which the predicate function `p` returns true.

The return value of `remove_copy_if()` is an iterator the end of the new range.
remove_copy_if() runs in **linear time**.

*Related topics:*
- remove
- remove_copy
- remove_if

---

**remove_if**

**Syntax:**

```
#include <algorithm>
iterator remove_if( iterator start, iterator end, Predicate p );
```

The `remove_if()` function removes all elements in the range `[start,end)` for which the predicate `p` returns true.

The return value of this function is an iterator to the last element of the pruned range.

remove_if() runs in **linear time**.

*Related topics:*
- remove
- remove_copy
- remove_copy_if

---

**replace**

**Syntax:**

```
#include <algorithm>
void replace( iterator start, iterator end, const TYPE& old_value, const TYPE& new_value );
```

The `replace()` function sets every element in the range `[start,end)` that is equal to `old_value` to have `new_value` instead.
replace() runs in linear time.

Related topics:
replace_copy
replace_copy_if
replace_if

---

**replace_copy**

Syntax:

```
#include <algorithm>
iterator replace_copy( iterator start, iterator end, iterator result, const
```

The replace_copy() function copies the elements in the range \([start,end]\) to the destination \(result\). Any elements in the range that are equal to \(old_value\) are replaced with \(new_value\).

Related topics:
replace

---

**replace_copy_if**

Syntax:

```
#include <algorithm>
iterator replace_copy_if( iterator start, iterator end, iterator result,
```

The replace_copy_if() function copies the elements in the range \([start,end]\) to the destination \(result\). Any elements for which the predicate \(p\) is true are replaced with \(new_value\).

Related topics:
replace
replace_if

Syntax:

```cpp
#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:

```cpp
#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:

```cpp
#include <algorithm>
iterator reverse_copy( iterator start, iterator end, iterator result
```

The `reverse_copy()` algorithm copies the elements in the range \([start, end]\) to
result such that the elements in the new range are in reverse order.

The return value of the reverse_copy() function is an iterator the end of the new range.

Related topics:
reverse

rotate

Syntax:
```cpp
#include <algorithm>
inline iterator rotate( iterator start, iterator middle, iterator end );
```

The rotate() algorithm moves the elements in the range [start,end) such that the middle element is now where start used to be, (middle+1) is now at (start+1), etc.

The return value of rotate() is an iterator to start + (end-middle).

rotate() runs in linear time.

Related topics:
rotate_copy

rotate_copy

Syntax:
```cpp
#include <algorithm>
iterator rotate_copy( iterator start, iterator middle, iterator end, iterator result );
```

The rotate_copy() algorithm is similar to the rotate() algorithm, except that the range of elements is copied to result before being rotated.
search

Syntax:

```cpp
#include <algorithm>
iterator search(iterator start1, iterator end1, iterator start2, iterator end2);
iterator search(iterator start1, iterator end1, iterator start2, iterator end2, BinPred p);
```

The `search()` algorithm looks for the elements `[start2,end2)` in the range `[start1,end1)`. If the optional binary predicate `p` is provided, then it is used to perform comparisons between elements.

If `search()` finds a matching subrange, then it returns an iterator to the beginning of that matching subrange. If no match is found, an iterator pointing to `end1` is returned.

In the worst case, `search()` runs in quadratic time, on average, it runs in **linear time**.

Related topics:
- `equal`
- `find`
- `lexicographical_compare`
- `mismatch`
- `search_n`

search_n

Syntax:

```cpp
#include <algorithm>
iterator search_n(iterator start, iterator end, size_t num, const iterator search_n(iterator start, iterator end, size_t num, const
```
The search_n() function looks for num occurrances 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:

```cpp
#include <algorithm>
iterator set_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result);
iterator set_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp);
```

The set_difference() algorithm computes the difference between two sets defined by [start1,end1) and [start2,end2) and stores the difference starting at result.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_difference() is an iterator to the end of the result range.

If the strict weak ordering comparison function object cmp is not specified, set_difference() will use the < operator to compare elements.

Related topics: includes set_intersection set_symmetric_difference set_union
set_intersection

Syntax:

```cpp
#include <algorithm>

iterator set_intersection( iterator start1, iterator end1, iterator start2, iterator end2, iterator result);

iterator set_intersection( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp);
```

The `set_intersection()` algorithm computes the intersection of the two sets defined by `[start1,end1)` and `[start2,end2)` and stores the intersection starting at `result`.

Both of the sets, given as ranges, must be sorted in ascending order.

The return value of `set_intersection()` is an iterator to the end of the intersection range.

If the strict weak ordering comparison function object `cmp` is not specified, `set_intersection()` will use the `<` operator to compare elements.

Related topics:
- `includes`
- `set_difference`
- `set_symmetric_difference`
- `set_union`

set_symmetric_difference

Syntax:

```cpp
#include <algorithm>

iterator set_symmetric_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result);

iterator set_symmetric_difference( iterator start1, iterator end1, iterator start2, iterator end2, iterator result, StrictWeakOrdering cmp);
```

The `set_symmetric_difference()` algorithm computes the symmetric difference of the two sets defined by `[start1,end1)` and `[start2,end2)` and stores the difference starting at `result`. 
Both of the sets, given as ranges, must be sorted in ascending order.

The return value of set_symmetric_difference() is an iterator to the end of the result range.

If the strict weak ordering comparison function object \textit{cmp} is not specified, set_symmetric_difference() will use the \textless operator to compare elements.

\textit{Related topics:}

\texttt{includes}
\texttt{set_difference}
\texttt{set_intersection}
\texttt{set_union}

---

\textbf{set_union}

\textit{Syntax:}

\begin{verbatim}
#include <algorithm>
iterator set_union( iterator start1, iterator end1, iterator start2, iterator end2, iterator result

iterator set_union( iterator start1, iterator end1, iterator end2, iterator result,
StrictWeakOrdering cmp
\end{verbatim}

The set_union() algorithm computes the union of the two ranges \([start1,end1)\) and \([start2,end2)\) and stores it starting at \textit{result}.

The return value of set_union() is an iterator to the end of the union range.

set_union() runs in linear time.

\textit{Related topics:}

\texttt{includes}
\texttt{merge}
\texttt{set_difference}
\texttt{set_intersection}
\texttt{set_symmetric_difference}
**sort**

Syntax:

```cpp
#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:

```cpp
vector<int> v;
    v.push_back( 23 );
    v.push_back( -1 );
    v.push_back( 9999 );
    v.push_back( 0 );
    v.push_back( 4 );

cout << "Before sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
    cout << v[i] << " ";
}
cout << endl;

sort( v.begin(), v.end() );

cout << "After sorting: ";
for( unsigned int i = 0; i < v.size(); i++ ) {
    cout << v[i] << " ";
}
cout << endl;
```
When run, the above code displays this output:

<table>
<thead>
<tr>
<th>Before sorting: 23 -1 9999 0 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>After sorting: -1 0 4 23 9999</td>
</tr>
</tbody>
</table>

Alternatively, the following code uses the `sort()` function to sort a normal array of integers, and displays the same output as the previous example:

```c++
int array[] = { 23, -1, 9999, 0, 4 };
unsigned int array_size = 5;

cout << "Before sorting: ";
for( unsigned int i = 0; i < array_size; i++ ) {
    cout << array[i] << " ";
} cout << endl;
sort( array, array + array_size );
cout << "After sorting: ";
for( unsigned int i = 0; i < array_size; i++ ) {
    cout << array[i] << " ";
} cout << endl;
```

This next example shows how to use `sort()` with a user-specified comparison function. The function `cmp` is defined to do the opposite of the `<` operator. When `sort()` is called with `cmp` used as the comparison function, the result is a list sorted in descending, rather than ascending, order:

```c++
bool cmp( int a, int b ) {
    return a > b;
}

vector<int> v;
for( int i = 0; i < 10; i++ ) {
    v.push_back(i);
}
cout << "Before: ";
for( int i = 0; i < 10; i++ ) {
    cout << v[i] << " ";
} cout << endl;
```
sort( v.begin(), v.end(), cmp );

cout << "After: ";
for( int i = 0; i < 10; i++ ) {
    cout << v[i] << " ";
}
cout << endl;

Related topics:
binary_search
is_sorted
merge
partial_sort
partial_sort_copy
stable_sort
(Other Standard C Functions) qsort

sort_heap

Syntax:

```
#include <algorithm>
void sort_heap( iterator start, iterator end );
void sort_heap( iterator start, iterator end, StrictWeakOrdering cmp );
```

The sort_heap() function turns the heap defined by [start,end) into a sorted range.

If the strict weak ordering comparison function object cmp is given, then it is used instead of the < operator to compare elements.

Related topics:
is_heap
make_heap
pop_heap
push_heap
stable_partition

Syntax:

```cpp
#include <algorithm>
iterator stable_partition(iterator start, iterator end, Predicate p);
```

The stable_partition() function behaves similarly to `partition()`. The difference between the two algorithms is that stable_partition() will preserve the initial ordering of the elements in the two groups.

Related topics:

`partition`

stable_sort

Syntax:

```cpp
#include <algorithm>
void stable_sort(iterator start, iterator end);
void stable_sort(iterator start, iterator end, StrictWeakOrdering cmp);
```

The stable_sort() algorithm is like the `sort()` algorithm, in that it sorts a range of elements into ascending order. Unlike `sort()`, however, stable_sort() will preserve the original ordering of elements that are equal to each other.

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:

```cpp
#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:

```cpp
#include <algorithm>
iterator swap_ranges( iterator start1, iterator end1, iterator start2);
```

The swap_ranges() function exchanges the elements in the range `[start1,end1)` with the range of the same size starting at `start2`.

The return value of swap_ranges() is an iterator to `start2 + (end1-start1)`.

*Related topics:*
- `iter_swap`
- `swap`
transform

Syntax:

```
#include <algorithm>
iterator transform( iterator start, iterator end, iterator result, UnaryFunction f );
iterator transform( iterator start1, iterator end1, iterator start2, iterator result, BinaryFunction f );
```

The transform() algorithm applies the function $f$ to some range of elements, storing the result of each application of the function in $result$.

The first version of the function applies $f$ to each element in $[start, end)$ and assigns the first output of the function to $result$, the second output to $(result+1)$, etc.

The second version of the transform() works in a similar manner, except that it is given two ranges of elements and calls a binary function on a pair of elements.

Related topics:
copy
fill
generate

unique

Syntax:

```
#include <algorithm>
iterator unique( iterator start, iterator end );
iterator unique( iterator start, iterator end, BinPred p );
```

The unique() algorithm removes all consecutive duplicate elements from the range $[start, end)$. If the binary predicate $p$ is given, then it is used to test to test two elements to see if they are duplicates.

The return value of unique() is an iterator to the end of the modified range.
unique() runs in linear time.

Related topics:
adjacent_find
remove
unique_copy

独特性副本

语法：
```
#include <algorithm>
iterator unique_copy(iterator start, iterator end, iterator result);
iterator unique_copy(iterator start, iterator end, iterator result, BinPred p);
```

unique_copy() 函数复制范围 [start,end) 到 result，移除所有连续的重复元素。如果提供二元谓词 p，则用于测试两个元素是否为重复。

unique_copy() 返回值是结果范围的结束迭代器。
unique_copy() 运行时间是线性时间。

相关主题：
adjacent_find
remove
unique

最后界

语法：
```
#include <algorithm>
iterator upper_bound(iterator start, iterator end, const TYPE& val);
iterator upper_bound(iterator start, iterator end, const TYPE& val, StrictWeakOrdering cmp);
```

upper_bound() 算法在有序范围 [start,end) 中搜索最后一个大于等于 val 的元素。

upper_bound() 运行时间是线性时间。
相关主题：
adjacent_find
remove
unique

最后界
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:

```cpp
#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:**

```cpp
#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;
}
String Stream Constructors

Syntax:

```cpp
#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 `io stream mode flags`.

An ostringstream object can be used to write to a string. This is similar to the C `sprintf()` function. For example:

```cpp
ostringstream s1;
int i = 22;
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;
```

An istringstream object can be used to read from a string. This is similar to the C `sscanf()` function. For example:

```cpp
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25
```
You can also specify the input string in the istringstream constructor as in this example:

```cpp
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl; // displays 25
```

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:**

```cpp
#include <sstream>
operator<<
operator>>
```

Like C++ I/O Streams, 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:

```cpp
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:

```cpp
stream1 >> i;
```

This code reads a value from stream1 and assigns the variable i that value.

Related topics:
C++ I/O Streams
rdbuf

Syntax:

```cpp
#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:

```cpp
#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:

```cpp
ostringstream stream1;
stream1 << "Testing!" << endl;
cout << stream1.str();
```

Second, `str()` can be used to copy a string into the stream. This is most useful with input strings. For example:

```cpp
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:

```cpp
istringstream stream1;
float num;

// use it once
string string1 = "25 1 3.235\n111111\n222222";
stream1.str(string1);
while( stream1 >> num ) cout << "num: " << num << endl; // displays numbers, one per line

// 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; // displays numbers, one per line
```

Related topics:
- rdbuf()
- C++ I/O Streams
#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 ) ) {
    ...
}
```

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 its 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:

```c
#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 );
}
```

Related topics:

`#`, `##`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#undef`

---

### #error

**Syntax:**

```c
#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:**

```c
#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.

```c
#if expression

If the value of expression is true, then the code that immediately follows the command will be compiled.
```

```c
#endif
```

```c
#ifndef macro

If the macro has not been defined by a #define statement, then the code immediately following the command will be compiled.
```

```c
#endif
```

```c
#else

If the macro has been defined by a #define statement, then the code immediately following the command will be compiled.
```

```c
#elif

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:

```c
#ifndef DEBUG
    cout << "This is the test version, i=" << i << endl;
#else
    cout << "This is the production version!" << endl;
#endif
```
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 `#define` 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

```c
#define to_string(s) # s
```

will make the compiler turn this command

```c
cout << to_string( Hello World! ) << endl;
```

into

```c
cout << "Hello World!" << endl;
```

Here is an example of the ## command:

```c
#define concatenate(x, y) x ## y
...
int xy = 10;
...
```

This code will make the compiler turn

```c
cout << concatenate(x, y) << endl;
```

into

```c
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 `#define`.

*Related topics:*

`#define`
**abort**

*Syntax:*

```c
#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:*

```c
#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:*

```c
#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

bsearch

Syntax:

```c
#include <cstdlib>
void *bsearch( const void *key, const void *buf, size_t num, size_t size,
int (*compare)(const void *, const void *) );
```

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:

```c
#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:

```c
#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:

```c
#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 **setjmp**(). **envbuf** is usually set through a call to **setjmp**(). **status** becomes the return value of **setjmp**() and can be used to figure out where longjmp() came from. **status** should not be set to zero.

Related topics:
**setjmp**
qsort

Syntax:

```c
#include <cstdlib>
void qsort( void *buf, size_t num, size_t size, int (*compare)(const void *, const void *) );
```

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:

```c
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:

<table>
<thead>
<tr>
<th>Before sorting:</th>
<th>-2 99 0 -743 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>After sorting:</td>
<td>-743 -2 0 2 3 4 99</td>
</tr>
</tbody>
</table>

Related topics:
- bsearch
- (C++ Algorithms) sort

raise

Syntax:
```c
#include <csignal>
int raise( int signal );
```

The `raise()` function sends the specified `signal` to the program. Some signals:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Termination error</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Floating pointer error</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Bad instruction</td>
</tr>
<tr>
<td>SIGINT</td>
<td>User presed CTRL-C</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Illegal memory access</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Terminate program</td>
</tr>
</tbody>
</table>

The return value is zero upon success, nonzero on failure.

Related topics:
- signal

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`

---

**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 `longjmp()`. When you first call `setjmp()`, its return value is zero. Later, when you call `longjmp()`, the second argument of `longjmp()` is what the return value of `setjmp()` will be. Confused? Read about `longjmp()`.

**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 received by your program. *func* can be a custom signal handler, or one of these macros (defined in the csignal header file):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIG_DFL</td>
<td>default signal handling</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>ignore the signal</td>
</tr>
</tbody>
</table>

Some basic signals that you can attach a signal handler to are:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGTERM</td>
<td>Generic stop signal that can be caught.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Interrupt program, normally ctrl-c.</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Interrupt program, similar to SIGINT.</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Stops the program. Cannot be caught.</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>Reports a disconnected terminal.</td>
</tr>
</tbody>
</table>

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:

```cpp
typedef void(*endFunc)(void);
vector<endFunc> endFuncs;

void f1() {
    cout << "calling f1()..." << endl;
}
void f2() {
    cout << "calling f2()..." << endl;
}
```
void cleanUp( int dummy ) {
    for( unsigned int i = 0; i < endFuncs.size(); i++ ) {
        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:
raise

srand

Syntax:

#include <cstdlib>
void srand( unsigned seed );

The function srand() is used to seed the random sequence generated by rand(). 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() );
system

Syntax:

```c
#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
getenv

va_arg

Syntax:

```c
#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:

```c
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:

```c
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
abs

Syntax:

```c
#include <cstdlib>
int abs(int num);
```

The `abs()` function returns the absolute value of `num`. For example:

```c
int magic_number = 10;
cout << "Enter a guess: ";
cin >> x;
cout << "Your guess was " << abs(magic_number - x) << " away from the magic number."
```

Related topics:
- fabs
- labs

acos

Syntax:

```c
#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
- sin
**asin**

Syntax:

```cpp
#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`
- `cos`
- `cosh`
- `sin`
- `sinh`
- `tan`
- `tanh`

---

**atan**

Syntax:

```cpp
#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:

```cpp
#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
atan
cos
cosh
sin
sinh
tan	anh

---

ceil
Syntax:

```cpp
#include <cmath>
double ceil( double num );
```

The ceil() function returns the smallest integer no less than num. For example,

```cpp
y = 6.04;
x = ceil( y );
```

would set x to 7.0.

Related topics:
- floor
- fmod

---

**COS**

Syntax:

```cpp
#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:

```cpp
#include <cmath>
double cosh( double arg );
```

The function cosh() returns the hyperbolic cosine of arg.

Related topics:
acos
asin
atan
atan2
cos
sin
sinh
tan
tanh

div

Syntax:

```cpp
#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:

```cpp
int quot; // The quotient
int rem;  // The remainder
```

For example, the following code displays the quotient and remainder of x/y:

```cpp
div_t temp;
temp = div( x, y );
printf( "%d divided by %d yields %d with a remainder of %d\n",
```
exp

Syntax:

```c
#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:

```c
#include <cmath>
double fabs( double arg );
```

The function `fabs()` returns the absolute value of $arg$.

Related topics:
- abs
- fmod
- labs

div
Syntax:

```cpp
#include <cmath>
double floor( double arg );
```

The function `floor()` returns the largest integer not greater than `arg`. For example,

```cpp
y = 6.04;
x = floor( y );
```

would result in `x` being set to 6.0.

Related topics:
- `ceil`
- `fmod`

---

**fmod**

Syntax:

```cpp
#include <cmath>
double fmod( double x, double y );
```

The `fmod()` function returns the remainder of `x/y`.

Related topics:
- `ceil`
- `fabs`
- `fabsf`
- `floor`

---

**frexp**

Syntax:

```cpp
#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 `labs()` function returns the absolute value of `num`.

**Related topics:**
- `abs`
- `fabs`

---

## ldexp

**Syntax:**

```
#include <cmath>
double ldexp( double num, int exp );
```

The `ldexp()` function returns `num * (2 ^ exp)`. And get this: if an overflow occurs, `HUGE_VAL` is returned.

**Related topics:**
- `frexp`
- `modf`
**ldiv**

**Syntax:**

```c
#include <cstdlib>
ldiv_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:

```c
long quot; // the quotient
long rem;  // the remainder
```

**Related topics:**
- `div`

---

**log**

**Syntax:**

```c
#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:

```c
double answer = log(x) / log(b);
```

**Related topics:**
- `exp`
- `log10`
- `pow`
log10

Syntax:

```c
#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:

```c
#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:

```c
#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

sin

Syntax:

```c
#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:

```cpp
#include <cmath>
double sinh( double arg );
```

The function sinh() returns the hyperbolic sine of `arg`.

**Related topics:**
- `acos`
- `asin`
- `atan`
- `atan2`
- `cos`
- `cosh`
- `sin`
- `tan`
- `tanh`

---

**sqrt**

Syntax:

```cpp
#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`

---

**tan**

Syntax:
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:**
- \texttt{acos}
- \texttt{asin}
- \texttt{atan}
- \texttt{atan2}
- \texttt{cos}
- \texttt{cosh}
- \texttt{sin}
- \texttt{sinh}
- \texttt{tanh}

---

**tanh**

**Syntax:**

```cpp
#include <cmath>
double tanh( double arg );
```

The function tanh() returns the hyperbolic tangent of \( arg \).

**Related topics:**
- \texttt{acos}
- \texttt{asin}
- \texttt{atan}
- \texttt{atan2}
- \texttt{cos}
- \texttt{cosh}
- \texttt{sin}
- \texttt{sinh}
- \texttt{tanh}
asctime

Syntax:

```c
#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
clock
time

---

clock

Syntax:

```c
#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 CLOCKKS_PER_SEC. (Note: if your compiler is POSIX compliant, then CLOCKKS_PER_SEC is always defined as 1000000.)
ctime

Syntax:
```c
#include <ctime>
char *ctime( const time_t *time );
```

The `ctime()` function converts the calendar time `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`

difftime

Syntax:
```c
#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:**

```cpp
#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
cftime
difftime
dftime
localtime
mktime
strftime
time

---

**localtime**

**Syntax:**

```cpp
#include <ctime>
struct tm *localtime( const time_t *time );
```

The function `localtime()` converts calendar time `time` into local time. Watch out for the **static return**.
**mktime**

Syntax:

```c
#include <ctime>

#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`
- `difftime`
- `gmtime`
- `strftime`
- `time`

**setlocale**

Syntax:

```c
#include <clocale>

#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: 
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_ALL</td>
<td>All of the locale</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>Date and time formatting</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>Number formatting</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>String collation and regular expression matching</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>For monetary formatting</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>For natural language messages</td>
</tr>
</tbody>
</table>

Related topics:
(Standard C String and Character) `strcoll`

**strftime**

**Syntax:**
```
#include <ctime>
size_t strftime( char *str, size_t maxsize, const char *fmt, struct tm *time);
```

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>abbreviated weekday name (e.g. Fri)</td>
</tr>
<tr>
<td>%A</td>
<td>full weekday name (e.g. Friday)</td>
</tr>
<tr>
<td>%b</td>
<td>abbreviated month name (e.g. Oct)</td>
</tr>
<tr>
<td>%B</td>
<td>full month name (e.g. October)</td>
</tr>
<tr>
<td>%c</td>
<td>the standard date and time string</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>%d</td>
<td>day of the month, as a number (1-31)</td>
</tr>
<tr>
<td>%H</td>
<td>hour, 24 hour format (0-23)</td>
</tr>
<tr>
<td>%I</td>
<td>hour, 12 hour format (1-12)</td>
</tr>
<tr>
<td>%j</td>
<td>day of the year, as a number (1-366)</td>
</tr>
<tr>
<td>%m</td>
<td>month as a number (1-12). Note: some versions of Microsoft Visual C++ may use values that range from 0-11.</td>
</tr>
<tr>
<td>%M</td>
<td>minute as a number (0-59)</td>
</tr>
<tr>
<td>%p</td>
<td>locale's equivalent of AM or PM</td>
</tr>
<tr>
<td>%S</td>
<td>second as a number (0-59)</td>
</tr>
<tr>
<td>%U</td>
<td>week of the year, (0-53), where week 1 has the first Sunday</td>
</tr>
<tr>
<td>%W</td>
<td>week of the year, (0-53), where week 1 has the first Monday</td>
</tr>
<tr>
<td>%x</td>
<td>standard date string</td>
</tr>
<tr>
<td>%X</td>
<td>standard time string</td>
</tr>
<tr>
<td>%y</td>
<td>year in decimal, without the century (0-99)</td>
</tr>
<tr>
<td>%Y</td>
<td>year in decimal, with the century</td>
</tr>
<tr>
<td>%Z</td>
<td>time zone name</td>
</tr>
<tr>
<td>%%</td>
<td>a percent sign</td>
</tr>
</tbody>
</table>

The strftime() function returns the number of characters put into str, or zero if an error occurs.

*Related topics:*

gmtime
localtime
time

**time**
Syntax:

```c
#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`
atof

Syntax:

```cpp
#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,

```cpp
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:

```cpp
#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:

```cpp
int i;
i = atoi( "512" );
i = atoi( "512.035" );
```


```c
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:

```c
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:**

```c
#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,

```c
x = atol( "1024.0001" );
```

results in $x$ being set to 1024L.

**Related topics:**
- `atof`
- `atoi`
- *(Standard C I/O)* `sprintf`
- `strtol`
**isalnum**

*Syntax:*

```c
#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.

```c
char c;
scanf( "%c", &c );
if( isalnum(c) )
    printf( "You entered the alphanumeric character %c\n", c );
```

**Related topics:**
- `isalpha`
- `iscntrl`
- `isdigit`
- `isgraph`
- `isprint`
- `ispunct`
- `isspace`
- `isxdigit`

---

**isalpha**

*Syntax:*

```c
#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.

```c
char c;
scanf( "%c", &c );
```
if ( isalpha(c) )
    printf( "You entered a letter of the alphabet\n" );

Related topics:
isalnum
iscntrl
isdigit
isgraph
isprint
ispunct
isspace
isxdigit

iscntrl

Syntax:

```c
#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:
isalnum
isalpha
isdigit
isgraph
isprint
ispunct
isspace
isxdigit

isdigit

Syntax:
```c
#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.

```c
char c;
scanf( "%c", &c );
if( isdigit(c) )
    printf( "You entered the digit %c\n", c );
```

**Related topics:**
isalnum
isalpha
iscntrl
isgraph
isprint
ispunct
isspace
isxdigit

---

**isgraph**

**Syntax:**
```c
#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:**
isalnum
isalpha
iscntrl
isgraph
isdigit
isprint
islower

Syntax:

```c
#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:
isupper

isprint

Syntax:

```c
#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
isalpha
iscntrl
isdigit
isgraph
ispunct
isspace
ispunct

Syntax:

```c
#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:
islalnum
isalpha
iscntrl
isdigit
isgraph
isprint
isspace
isxdigit

isspace

Syntax:

```c
#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:
islalnum
isalpha
iscntrl
isdigit
**isupper**

*Syntax:*

```c
#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:*

```c
#include <cctype>
int isxdigit( int ch );
```

The function `isxdigit()` returns non-zero if its argument is a hexadecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

*Related topics:*

- isalnum
- isalpha
- iscntrl
- isdigit
- isgraph
- ispunct
memchr

Syntax:

```c
#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:

```c
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:

```c
#include <cstring>
int memcmp( const void *buffer1, const void *buffer2, size_t count );
```

The function `memcmp()` compares the first `count` characters of `buffer1` and `buffer2`. The return values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than 0</td>
<td>buffer1 is less than buffer2</td>
</tr>
<tr>
<td>equal to 0</td>
<td>buffer1 is equal to buffer2</td>
</tr>
<tr>
<td>greater than 0</td>
<td>buffer1 is greater than buffer2</td>
</tr>
</tbody>
</table>

Related topics:
- memchr
- memcpy
- memset
- strcmp

**memcpy**

**Syntax:**

```c
#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
- strncpy
- strlen
- strncpy

**memmove**

**Syntax:**

```c
#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:
memcpymemset

---

### memset

**Syntax:**

```c
#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 initializing a section of memory to some value. For example, this command:

```c
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:

<table>
<thead>
<tr>
<th>Input size</th>
<th>Initialized with a for-loop</th>
<th>Initialized with memset()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.016</td>
<td>0.017</td>
</tr>
<tr>
<td>10000</td>
<td>0.055</td>
<td>0.013</td>
</tr>
<tr>
<td>100000</td>
<td>0.443</td>
<td>0.029</td>
</tr>
<tr>
<td>1000000</td>
<td>4.337</td>
<td>0.291</td>
</tr>
</tbody>
</table>

Related topics:
memcmpmemcpy
**strcat**

**Syntax:**

```c
#include <cstring>
char *strcat( char *str1, const char *str2 );
```

The `strcat()` function concatenates `str2` onto the end of `str1`, and returns `str1`. For example:

```c
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 `strlcpy` and `strlcat`.

---

**strchr**

**Syntax:**

```c
#include <cstring>
char *strchr( const char *str, int ch );
```
The function strchr() returns a pointer to the first occurrence of \textit{ch} in \textit{str}, or \texttt{NULL} if \textit{ch} is not found.

\textit{Related topics:}
\begin{itemize}
  \item \texttt{strcat}
  \item \texttt{strcmp}
  \item \texttt{strcpy}
  \item \texttt{strlen}
  \item \texttt{strncat}
  \item \texttt{strncpy}
  \item \texttt{strpbrk}
  \item \texttt{strspn}
  \item \texttt{strstr}
  \item \texttt{strtok}
\end{itemize}

\section*{\texttt{strcmp}}

\textbf{Syntax:}

\begin{verbatim}
#include <cstring>
int strcmp( const char *str1, const char *str2 );
\end{verbatim}

The function \texttt{strcmp()} compares \textit{str1} and \textit{str2}, then returns:

\begin{center}
\begin{tabular}{|c|c|}
\hline
Return value & Explanation  \\
\hline
less than 0 & "str1" is less than "str2"  \\
\hline
equal to 0 & "str1" is equal to "str2"  \\
\hline
greater than 0 & "str1" is greater than "str2"  \\
\hline
\end{tabular}
\end{center}

For example:

\begin{verbatim}
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 ) {
    printf( "Hello, Dr. Mary!\n" );
\}
\end{verbatim}
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 `strncpy()`.

Related topics:
- `memcmp`
- `strcat`
- `strchr`
- `strcoll`
- `strcpy`
- `strlen`
- `strncpy`
- `strxfrm`

---

### `strcoll`

**Syntax:**

```c
#include <cstring>
int strcoll( const char *str1, const char *str2 );
```

The `strcoll()` function compares `str1` and `str2`, much like `strcmp()`. However, `strcoll()` performs the comparison using the locale specified by the (Standard C Date & Time) `setlocale()` function.

Related topics:
- (Standard C Date & Time) `setlocale`
- `strcmp`
- `strxfrm`

---

### `strcpy`

**Syntax:**
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 `strncpy()`.

*Related topics:*
- `memcpy`
- `strcat`
- `strchr`
- `strcspn`
- `strcmpl`
- `strncpy`

Another set of related (but non-standard) functions are `strlcpy` and `strlcat`.

---

**strcspn**

*Syntax:*

```c
#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`
- `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`

---

**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:

<table>
<thead>
<tr>
<th>Return value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than 0</td>
<td>&quot;str1&quot; is less than &quot;str2&quot;</td>
</tr>
<tr>
<td>equal to 0</td>
<td>&quot;str1&quot; is equal to &quot;str2&quot;</td>
</tr>
<tr>
<td>greater than 0</td>
<td>&quot;str1&quot; is greater than str2</td>
</tr>
</tbody>
</table>

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
strlen
strncat
strncpy
**strncpy**

*Syntax:*

```c
#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:*

```c
#include <cstring>
char* strpbrk( const char* str1, const char* str2 );
```

The function `strpbrk()` returns a pointer to the first occurrence 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:*

```c
#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:*

```c
#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:

```c
#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:

```c
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, str1 );
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

---

**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:**

```c
#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:

```c
char str[] = "now # is the time for all # good men to come to the#
char delims[] = ">#\nchar *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:

```c
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
strstr
strrchr
strspn
strstr

---

**strtol**

Syntax:

```c
#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

---

**strtoul**

Syntax:

```c
#include <cstdlib>
unsigned long strtoul( const char *start, char **end, int base );
```

The function `strtoul()` behaves exactly like `strtol()`, except that it returns an unsigned long rather than a mere long.

Related topics:
strtol
**strxfrm**

Syntax:

```c
#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:

```c
#include <cctype>
int tolower( int ch );
```

The function `tolower()` returns the lowercase version of the character `ch`.

Related topics:

- `isupper`
- `toupper`

---

**toupper**

Syntax:

```c
#include <cctype>
int toupper( int ch );
```
The toupper() function returns the uppercase version of the character \textit{ch}.

\textit{Related topics:}

\texttt{tolower}
calloc

Syntax:

```c
#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

free

Syntax:

```c
#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 `malloc()`, `calloc()`, or `realloc()`. An example:

```c
typedef struct data_type {
    int age;
    char name[20];
} data;

data *willy;
willy = (data*) malloc( sizeof(*willy) );
...
free( willy );
```
malloc

Syntax:

```c
#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:

```c
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 );
```
 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`
clearerr

Syntax:

```c
#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 `perror()` to figure out which error actually occurred.

*Related topics:* `feof` `ferror` `perror`

fclose

Syntax:

```c
#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:

```c
#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`

---

### ferror

Syntax:

```c
#include <cstdio>
int ferror( FILE *stream );
```

The `ferror()` function looks for errors with `stream`, returning zero if no errors have occurred, and non-zero if there is an error. In case of an error, use `perror()` to determine which error has occurred.

Related topics:
- `clearerr`
- `feof`
- `perror`

---

### fflush

Syntax:

```c
#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.

```c
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

**fgets**

Syntax:

```c
#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:
fwrite
fputc
fread
fopen
getc
putc
**fgetpos**

Syntax:

```c
#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`

**fgets**

Syntax:

```c
#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.
**fopen**

**Syntax:**

```c
#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)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Open a text file for reading</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Create a text file for writing</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Append to a text file</td>
</tr>
<tr>
<td>&quot;rb&quot;</td>
<td>Open a binary file for reading</td>
</tr>
<tr>
<td>&quot;wb&quot;</td>
<td>Create a binary file for writing</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>Append to a binary file</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>Open a text file for read/write</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>Create a text file for read/write</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>Open a text file for read/write</td>
</tr>
<tr>
<td>&quot;rb+&quot;</td>
<td>Open a binary file for read/write</td>
</tr>
<tr>
<td>&quot;wb+&quot;</td>
<td>Create a binary file for read/write</td>
</tr>
<tr>
<td>&quot;ab+&quot;</td>
<td>Open a binary file for read/write</td>
</tr>
</tbody>
</table>
An example:

```c
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

Related topics:
- fclose
- fflush
- fgetc
- fputc
- fread
- freopen
- fseek
- fwrite
- getc
- getchar
- setbuf

---

**fprintf**

Syntax:

```c
#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 `printf()` 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:

```c
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
    fprintf( out, "Hello %s\n", name );
```

Related topics:
- fputc
**fputc**

Syntax:

```c
#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`

---

**fputs**

Syntax:

```c
#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.
**fread**

Syntax:

```c
#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 `feof()` or `ferror()` to figure out if an error occurs.

**Related topics:**
- `fflush`
- `fgetc`
- `fopen`
- `fputc`
- `fscanf`
- `fwrite`
- `getc`

---

**freopen**

Syntax:

```c
#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:

```c
#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

---

**fseek**

Syntax:

```c
#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):

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_SET</td>
<td>Seek from the start of the file</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>Seek from the current location</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>Seek from the end of the file</td>
</tr>
</tbody>
</table>

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:**

```c
#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:

```c
#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:

```c
#include <cstdio>
int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

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`
- `freadf`
- `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 fgetc(). 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
fopen
fputc
fread
fwrite
putc
ungetc

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`
- `fgets`
- `fputs`
- `puts`

## gets

**Syntax:**

```c
#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 `fgets()`.

Related topics:
- `fgetc`
- `fgets`
- `fputs`
- `puts`

## perror

**Syntax:**

```c
#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:

```c
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:**
- clearerr
- feof
- ferror

---

### printf

**Syntax:**

```c
#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

```c
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>character</td>
</tr>
<tr>
<td>%d</td>
<td>signed integers</td>
</tr>
<tr>
<td>%i</td>
<td>signed integers</td>
</tr>
<tr>
<td>%e</td>
<td>scientific notation, with a lowercase &quot;e&quot;</td>
</tr>
<tr>
<td>%E</td>
<td>scientific notation, with an uppercase &quot;E&quot;</td>
</tr>
<tr>
<td>%f</td>
<td>floating point</td>
</tr>
<tr>
<td>%g</td>
<td>use %e or %f, whichever is shorter</td>
</tr>
<tr>
<td>%G</td>
<td>use %E or %f, whichever is shorter</td>
</tr>
<tr>
<td>%o</td>
<td>octal</td>
</tr>
<tr>
<td>%s</td>
<td>a string of characters</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>%x</td>
<td>unsigned hexadecimal, with lowercase letters</td>
</tr>
<tr>
<td>%X</td>
<td>unsigned hexadecimal, with uppercase letters</td>
</tr>
<tr>
<td>%p</td>
<td>a pointer</td>
</tr>
<tr>
<td>%n</td>
<td>the argument shall be a pointer to an integer into which is placed the number of characters written so far</td>
</tr>
<tr>
<td>%%</td>
<td>a '%%' sign</td>
</tr>
</tbody>
</table>

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 modifier simply acts as a maximum field 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 hexadecimal 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 '+'. 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

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;
inpu = 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
**putchar**

*Syntax:*

```c
#include <cstdio>
int putchar( int ch );
```

The `putchar()` function writes `ch` to `stdout`. The code

```c
putchar( ch );
```

is the same as

```c
putc( ch, stdout );
```

The return value of `putchar()` is the written character, or `EOF` if there is an error.

*Related topics:*

`putc`

---

**puts**

*Syntax:*

```c
#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:*
**remove**

*Syntax:*

```c
#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:*

```c
#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:*

```c
```
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:**

```c
#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 `printf()`. 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:

<table>
<thead>
<tr>
<th>Control Character</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%c</code></td>
<td>a single character</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>a decimal integer</td>
</tr>
<tr>
<td><code>%i</code></td>
<td>an integer</td>
</tr>
<tr>
<td><code>%e</code>, <code>%f</code>, <code>%g</code></td>
<td>a floating-point number</td>
</tr>
<tr>
<td><code>%lf</code></td>
<td>a double</td>
</tr>
<tr>
<td><code>%o</code></td>
<td>an octal number</td>
</tr>
<tr>
<td><code>%s</code></td>
<td>a string</td>
</tr>
<tr>
<td><code>%x</code></td>
<td>a hexadecimal number</td>
</tr>
<tr>
<td><code>%p</code></td>
<td>a pointer</td>
</tr>
</tbody>
</table>
\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{\%n} & an integer equal to the number of characters read so far \\
\hline
\textbf{\%u} & an unsigned integer \\
\hline
\textbf{\%[]} & a set of characters \\
\hline
\textbf{\%\%} & a percent sign \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item 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.
\item \textbf{Example code:}
\item 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:
\begin{verbatim}
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 );
\end{verbatim}
\item Related topics:
\begin{itemize}
\item \textbf{fgets}
\item \textbf{fscanf}
\item \textbf{printf}
\item \textbf{sscanf}
\end{itemize}
\end{itemize}
**setbuf**

Syntax:

```c
#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:

```c
#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`
sprintf

Syntax:

```
#include <cstdio>
int sprintf( char *buffer, const char *format, ... );
```

The `sprintf()` function is just like `printf()`, 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 `atoi()` -- where `atoi()` 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) `atof`
(Standard C String and Character) `atoi`
(Standard C String and Character) `atol`
`fprintf`
printf

sscanf

Syntax:

```c
#include <cstdio>
int sscanf( const char *buffer, const char *format, ... );
```

The function `sscanf()` is just like `scanf()`, except that the input is read from `buffer`.

Related topics:
`fscanf`
`scanf`

tmpfile

Syntax:

```c
#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`

tmpnam

Syntax:

```c
#include <cstdio>
```
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:**
```c
#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:**
```c
#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 `printf()`, `fprintf()`, and `sprintf()`. 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) `va_arg()`. For example:

```c
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 );
}
**asm**

Syntax:

```c
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

```c
asm {
    instruction-sequence
}
```

or

```c
asm( instruction );
```

**auto**

The keyword `auto` is used to declare local variables, and is purely optional.

*Related topics:*

`register`

**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.

```cpp
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 `do`, `for`, or `while` loop. It is also used to finish each clause of a `switch` statement, keeping the program from "falling through" to the next case in the code. An example:

```cpp
while( x < 100 ) {
   if( x < 0 )
      break;
   cout << x << endl;
   x++;
}
```

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 `goto` statement to break out of the loop.
case

The case keyword is used to test a variable against a certain value in a `switch` statement.

Related topics:
- `default`
- `switch`

catch

The catch statement handles exceptions generated by the `throw` statement.

Related topics:
- `throw`
- `try`

char

The char keyword is used to declare character variables. For more information about variable types, see the `data types` page.

Related topics:
class

Syntax:

```c
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:

```c
class Date {
    int Day;
    int Month;
    int Year;
    public:
        void display();
};
```

Related topics:

friend
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:*

```cpp
cast_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 **const**.

*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:

```cpp
for( int i = 0; i < 21; i++ ) {
    if( i == 10 ) {
        continue;
    }
    cout << i << " ";
}
```

Related topics:
break
dodo
for
while

default

A default case in the switch statement.

Related topics:
case
caseswitch

delete

Syntax:

```cpp
delete p;
```
The delete operator frees the memory pointed to by \( p \). The argument should have been previously allocated by a call to \texttt{new}. The second form of delete should be used to delete an array.

\textit{Related topics:}
(Standard C Memory) \texttt{free}
(Standard C Memory) \texttt{malloc}
\texttt{new}

---

\textbf{do}

\textit{Syntax:}

\begin{verbatim}
do {
    statement-list;
} while( condition );
\end{verbatim}

The \texttt{do} construct evaluates the given \texttt{statement-list} repeatedly, until \texttt{condition} becomes false. Note that every \texttt{do} loop will evaluate its statement list at least once, because the terminating condition is tested at the end of the loop.

\textit{Related topics:}
\texttt{break}
\texttt{continue}
\texttt{for}
\texttt{while}

---

\textbf{double}

The \texttt{double} keyword is used to declare double precision floating-point variables. Also see the \texttt{data types} page.

\textit{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 **if** 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:

```cpp
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
...
ColorT c1 = indigo;
if( c1 == indigo ) {
    cout << "c1 is indigo" << endl;
}
```

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:

```cpp
enum ColorT { red = 10, blue = 15, green };  
...
ColorT c = green;
cout << "c is " << c << endl;
```

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:

```cpp
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:

```cpp
struct foo {
    explicit foo( int a )
    : a_( a )
    {}

    int a_; };

int bar( const foo & f ) {
    return f.a_; }

bar( 1 ); // fails because an implicit conversion from int to foo
         // is forbidden by explicit.

bar( foo( 1 ) ); // works -- explicit call to explicit constructor

bar( foo( 1.0 ) ); // works -- explicit call to explicit constructor
                   // with automatic conversion from float to int.
```

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 [data types](#) page.

*Related topics:*

- **bool**
- **char**
- **double**
- **int**
- **long**
- **short**
- **signed**
for

Syntax:

```c
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:

```c
for( int i = 0; i < 10; i++ ) {
    cout << "i is " << i << endl;
}
int j, k;
for( j = 0, k = 10;
    j < k;
    j++, k-- ) {
    cout << "j is " << j << " and k is " << k << endl;
}
for( ; ; ) {
    // loop forever!
}
```

Related topics:

break
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 considered harmful, it can occasionally be useful. For example, it may be cleaner to use a goto to break out of a deeply-nested for loop, compared to the space and time that extra break logic would consume.

Related topics:
break

if

Syntax:
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 else statement is present, then the statement list within the else block will be executed. All of the else 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.
The `int` keyword is used to declare integer variables. Also see the data types page.

Related topics: `bool`, `char`, `double`, `float`, `long`, `short`, `signed`, `unsigned`, `void`, `wchar_t`

---

The `long` keyword is a data type modifier that is used to declare long integer variables. For more information on long, see the data types page.

Related topics: `bool`, `char`, `double`, `float`, `int`, `short`, `signed`, `void`
mutable

The mutable keyword overrides any enclosing const statement. A mutable member of a const object can be modified.

Related topics:
const

namespace

Syntax:

```cpp
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 using keyword.

Example code:

```cpp
namespace CartoonNameSpace {
    int HomersAge;
    void incrementHomersAge() {
        HomersAge++;
    }
}

int main() {
    ...
    CartoonNameSpace::HomersAge = 39;
    CartoonNameSpace::incrementHomersAge();
    cout << CartoonNameSpace::HomersAge << endl;
    ...
}
```

Related topics:
using
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:

```cpp
class Base {
public:
    Base() {}

    void *operator new( unsigned int size, string str ) {
        cout << "Logging an allocation of " << size << " bytes for new \n        return malloc( size );
    }

    int var;
    double var2;
};

Base* b = new ("Base instance 1") Base;
```

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) free
Standard C Memory) malloc

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:
this

private

Private data of a class can only be accessed by members of that class, except when friend is used. The private keyword can also be used to inherit a base class
privately, which causes all public and protected 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 public and protected members of the base class to become protected members of the derived class.

Related topics:
- class
- 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 protected members of the base class to become public and protected members of the derived class.

Related topics:
- class
- private
- 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:

\[
\text{reinterpret\_cast<type> (object);}
\]

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:

\[
\text{return;}
\text{return( value );}
\]

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 data types 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 data types page.

Related topics:
bool
char
double
float
int
long
short
unsigned
void
wchar_t
sizeof

The sizeof 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 <climits> header file.

For example, the following code uses sizeof to display the sizes of a number of variables:

```cpp
struct EmployeeRecord {
    int ID;
    int age;
    double salary;
    EmployeeRecord* boss;
};
...

cout << "sizeof(int): " << sizeof(int) << endl
    << "sizeof(float): " << sizeof(float) << endl
    << "sizeof(double): " << sizeof(double) << endl
    << "sizeof(char): " << sizeof(char) << endl
    << "sizeof(EmployeeRecord): " << sizeof(EmployeeRecord) << endl;

int i;
float f;
double d;
char c;
EmployeeRecord er;

cout << "sizeof(i): " << sizeof(i) << endl
    << "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 sizeof 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 sizeof with a variable type (e.g. sizeof(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 `class`, all instantiations of that class share one copy of the variable.

---

**static_cast**

**Syntax:**

```cpp
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:

```plaintext
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 **public** rather than **private**. In C, structs can only contain data and are not permitted to have inheritance lists. For example:

```plaintext
struct Date {
    int Day;
    int Month;
    int Year;
};
```

Related topics:
- **class**
- **union**

switch

Syntax:

```plaintext
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 `if()...else if()...else if()...` statements. `break` statements are required between each `case` statement, otherwise execution will "fall-through" to the next `case` statement. The `default` case is optional. If provided, it will match any case not explicitly covered by the preceding cases in the switch statement. For example:

```c
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**
Syntax:

```cpp
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:

```cpp
template<class X> void genericSwap( X &a, X &b ) {
    X tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

```cpp
int main(void) {
    ...  
    int num1 = 5;
    int num2 = 21;
    cout << "Before, num1 is " << num1 << " and num2 is " << num2 << endl;
    genericSwap( num1, num2 );
    cout << "After, num1 is " << num1 << " and num2 is " << num2 << endl;
    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](#)
The this keyword is a pointer to the current object. All member functions of a class have a this pointer.

Related topics:
class
operator

## throw

**Syntax:**

```cpp
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 try and catch statements, the C++ exception handling system gives programmers an elegant mechanism for error recovery.

You will generally use a try 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 try block and into one of the catch blocks. For example:

```cpp
try {
    cout << "Before throwing exception" << endl;
    throw 42;
    cout << "Shouldn't ever see this" << endl;
}
catch( int error ) {
    cout << "Error: caught exception " << error << endl;
}
```
true

The Boolean value of "true".

Related topics:
- bool
- false

try

The try statement attempts to execute exception-generating code. See the throw statement for more details.

Related topics:
- catch
- throw

typedef

Syntax:

```plaintext
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:

```cpp
typedef unsigned int* pui_t;

// data1 and data2 have the same type
pui_t data1;
unsigned int* data2;
```

typeid

**Syntax:**

```cpp
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 class keyword in a template declaration.

*Related topics:*

- class
- template

union

**Syntax:**

```cpp
union union-name {
public-members-list;
private:
private-members-list;
```
A union is like a class, except that all members of a union share the same memory location and are by default public rather than private. For example:

```c
union Data {
    int i;
    char c;
};
```

Related topics:
class
struct

unsigned

The unsigned keyword is a data type modifier that is usually used to declare unsigned int variables. See the data types page.

Related topics:
bool
char
double
float
int
short
signed
void
wchar_t

using

The using keyword is used to import a namespace (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 preceeding "std::".

```cpp
using namespace std;
```

Alternatively, the next code snippet just imports a single element of the std namespace into the current namespace:

```cpp
using std::cout;
```

Related topics:

namespace

---

### virtual

Syntax:

```cpp
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:

```cpp
class Base {
public:
    void nonVirtualFunc() {
        cout << "Base: non-virtual function" << endl;
    }
    virtual void virtualFunc() {
        cout << "Base: virtual function" << endl;
    }
};

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
- 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 data types 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 data types page.

Related topics:
- bool
- char
- double
- float
**while**

Syntax:

```plaintext
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:

```plaintext
bool done = false;
while( !done ) {
    ProcessData();
    if( StopLooping() ) {
        done = true;
    }
}
```

*Related topics:*
- `break`
- `continue`
- `do`
- `for`
- `if`