STL Algorithms

Much of programming come down to sorting, searching, counting and selecting. The STL algorithms are function templates that perform general functions on nearly any kind of data.

General Concepts

Generally the `<algorithm>` header is required to get started, some other algorithms are under the `<numeric>` header.

Arguments usually include iterators that point to the beginning and end positions within a container, referred to as *beg* and *end*. Containers must provide their own iterator.

```
#include <algorithm>
// ...
int arr[8] = {100, 1234, 4, 18, 1, 89, 43, 2};
sort(arr, arr + 8);
```

Algorithms that take functions as arguments can be passed them as:

- The address
- A function object
- A lambda function

STL container bounds can always be determined.

Using Lambda Functions (C++11)

```
[] (int n) { cout << n; }
```

Lambdas also let you capture variables from the surrounding scope. For example

```
int sum = 0;
[&sum] (int n) {
    sum += n;
    cout << n << " ";
}
```

The return type can be specified via

```
[] (args) -> return_type { //... }
```

Algorithms and Iterators

Each algorithm works with a specific type of iterator - random access iterators are supported by all algorithms.

Insert Iterators

Are used primarily with STL algorithms and STL containers. To use them include the `<iterator>` header.

Iterator functions include

- `back_inserter(con)`: calls `con.push_back(val)` and supported by most STL containers but not arrays
- `front_inserter(con)`: calls `con.push_front(val)` and supported by most STL containers but not vectors
- `inserter(con)`: calls `con.insert(val)` and supported by most STL containers

Summary of Algorithms

STL supports many algorithms, broadly grouped into:

- Read-only
- Modifying
- Sorting
- Heap (specialised)
- Numeric, which apply a mathematical operation

Summary Read-only

- `adjacent_find`: finds the first occurrence of two consecutive elements having the same value
- `all_of, any_of, none_of`: tests a container if all, any, or none of the elements fulfil a specified condition
- `binary_search`: searches an ordered container for a specified value
- `count, count_if`: returns the number of elements matching a specified value or condition
- `equal`: compares two containers or sub-ranges for equality, i.e. each corresponding element must be equal
- `for_each`: perform a specified action for each element
- `find, find_if`: find the first element matching a specified value or condition
- `find_if_not`: finds the first element not fulfilling a specified condition
- `find_end`: finds the last occurrence of a subrange within a larger range
- `find_first_of`: find the first element matching any element that is also an element of a specified range
- `includes`: determines whether *every* element in a range matches some element in another container or subrange
- `lower_bound, upper_bound, equal_range`: returns the iterators representing the lower and upper bounds for a range specified
- `max`: returns the larger of two values
- `min`: returns the smaller of two values

- `max_element, min_element`: find the position of the max / min element
- `minmax_element`: returns the positions of the min and max element
- `mismatch`: compares two ranges and reports the first mismatch
- `search_n`: search fora series of consecutive elements all meeting some condition
- `search`: finds the first occurrence of a complete subrange within another range

Modifying Algorithms

- `copy, copy_backward`: copy elements in a range into a destination range
- `copy_n, copy_if`: copy n elements in a range or if a condition is met
- `fill, fill_n`: set all the elements in a specified range to a particular value
- `generate, generate_n`: set the elements in a range by calling a generator function
- `iter_swap`: swaps two elements
- `merge, inplace_merge`: merge two sorted ranges together into a single sorted range
- `move, move_backward`: move all the elements of one range into another, not a copy action - possibly destructive
- `remove, remove_if, remove_copy, remove_copy_if`: remove one or more elements that meet a specific condition
- `replace, replace_if, replace_copy`: replace the value of one or more elements that meet a specific condition
- `swap`: exchanges any two values having the same base type
- `swap_ranges`: swaps two ranges of elements
- `transform`: performs an operation on a range of elements and places the result in a destination

Sorting and Re-ordering algorithms

- `is_permutation`: compares two ranges and checks if one is a permutation of the other
- `is_sorted, is_sorted_until`: Returns `true` if the range is already sorted - or the position of the element breaking the sort
- `partial_sort, partial_sort_copy`: sort part of a range
- `is_partitioned`: returns `true` if the range is ordered as a partition
- `partition_point`: find the element that divides the contained into a partition
- `prev_permutation, next_permutation`: rearrange elements so that they form the previous or next permutation
- `random_shuffle`: randomly shuffle the contents of a container or subrange
- `reverse, reverse_copy`: rearrange a series of elements so that their order is the reverse
- `rotate, rotate_copy`: move elements forward or backward in a container

through rotation

- `set_difference`: finds the difference between two ordered ranges
- `set)intersetion`: creates the intersection between two ordered ranges copying the result to a destination range
- `set_symmetric_difference`: finds the symmetric difference between two ordered ranges
- `set_union`: creates a union between two ordered ranges, copying the result to a destination range
- `sort`: sorts a container or subrange
- `stable_sort`: sort a container preserving the relative position of elements with equal values
- `unique, unique_copy`: remove consecutive elements having the same value

Heap Algorithms

The term "heap" is used in a specialised way, these algorithms reorder a random-access container (array, vector, deque) to treat it as a virtual top-down binary tree.

The "highest" element is the element at the root of the tree. Each child of a node is less than or equal to the node itself

Elements in a heap can be inserted more quickly into larger collections than they can with linear searches. The heap search time grows logarithmically, i.e. search a container with millions of elements takes twice as long as it does for a collection with thousand

- `make_heap`: reorders a range within a random access container so that it fulfils the condition of being a heap
- `is_heap`: returns `true` if the range is a arranged as a heap
- `is_heap_until`: returns the first position of the container that breaks the requirements of being a heap
- `pop_heap`: removes the first element of the heap and rearranges the remaining elements s.t. they are a heap
- `push_heap`: insert a new element into a heap
- `sort_heap`: sorts a range ordered as a heap

Numeric Algorithms

The numeric algorithm are part of the `<numeric>` header, they involve mathematical manipulation or calculation.

- `accumulate`: adds up the contents of a range and returns the result
- `adjacent_difference`: finds the difference between adjacent elements
- `intter_product`: accumulates the inner product
- `iota`: sets the first element in a range to a specified value and then each other element to 1 plus the previous value

- `partial_sum`: for each element find the partial sum formed by adding it to all preceding elements

Numeric algorithms work with functions and function objects. There are a number of templates that can return function objects performing addition, subtraction and so on. These templates require the `<functional>` header. These are useful in confection with numeric algorithms

- `plus<type>()`: returns the function object that adds together its two arguments
- `minus<type>()`: returns the function object that subtracts the second arg from the first
- `negate<type>()`: returns the function object that produces the arithmetic negation
- `multiplies<type>()`: returns the function object that multiplies its two arguments
- `divides<type>()`: returns the function object that divides the first arguments by the second
- `modulus<type>()`: returns the function object that performs the modulus
- `equal_to<type>()`: returns the function object that tests two args for equality
- `less<type>()`: returns the function object that performs a < b
- `greater<type>()`: returns the function object that performs a > b
- `less_equal<type>()`: returns the function object that performs a <= b
- `greater_equal<type>()`: returns the function object that performs a >= b
- `logical_not<type>()`: returns the function object that produces the logical negation
- `logical_and<type>()`: returns the function object that produces the logical AND
- `logical_or<type>()`: returns the function object that produces the logical OR
- `bit_and<type>()`: returns the function object that performs the bitwise AND
- `bit_or<type>()`: returns the function object that performs the bitwise OR
- `bit_nor<type>()`: returns the function object that performs the bitwise exclusive OR