Modern C++ - STL reference

18. The Standard Template Library Components

Containers

There are three kind of containers in STL:

Sequential containers

Sequence containers implement data structures which can be accessed sequentially.

array (*since C++11*) A fixed-sized array of contigous elements.

vector A dynamically growing array of contigous elements.

deque A queue dynamically extensible on both end.

forward_list (*since C++11*) A singly-linked list of (not necessary contigous) elements.

list A doubly-linked list of (not necessary contigous) elements.

Associative containers

Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

map A collection of key-value pairs, sorted by unique key.

multimap A collection of key-value pairs, sorted by not necessary unique key.

set A collection of unique keys.

multiset A collection of not necessary unique keys.

Unordered associative containers (hash tables) (since C++11)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

unordered_map A hash-based collection of key-value pairs, sorted by unique key.

unordered_multimap A hash-based collection of key-value pairs, sorted by not necessary unique key.

unordered_set A hash-based collection of unique keys.

unordered_multiset A hash-based collection of not necessary unique keys.

Container adaptors

Container adaptors provide a different interface for sequential containers.

stack Adapts a container to provide a Last-in First-out interface.

queue Adapts a container to provide a First-in First-out interface.

prority_queue Adapts a container to provide a priority queue.

Algorithms

The algorithms define functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements defines **[begin, end)** where begin refers to the first element and end the one after the last element of the range.

All of these algorithms are function templates and defined in the **<algorithm>** library.

Non-modifying sequence algoritms

all_of (since C++11) check if a predicate is true for *all*,

any_of (since C++11) *any* or *none* of the elements in a range.

none_of (since C++11)

for_each applies a function to a range of elements.

for_each_n (since C++17) applies a function to the first n elements.

count returns the number of elements.

count_if returns the number of elements satisfying specific criteria.

mismatch finds the first position where two ranges differ.

equal determines if two sets of elements are the same.

find finds the first element satisfying specific criteria.

find_if

find_if_not (since C++11)

find_end finds the last sequence of elements in a certain range.

find_first_of searches for any one of a set of elements.

adjacent_find finds the first two adjacent items that are equal (or satisfy a given predicate).

search searches for a range of elements.

search_n searches for a number consecutive copies of an element in a range.

Modifying sequence algoritms

copy copies a range of elements to a new location.

copy_if (since C++11)

copy_n copies a number of elements to a new location.

copy_backward copies a range of elements in backwards order

move (since C++11) moves a range of elements to a new location

move_backward (since C++11) moves a range of elements to a new location in backwards order.

fill assigns a range of elements a certain value.

fill_n assigns a value to a number of elements.

transform applies a function to a range of elements.

generate saves the result of a function in a range.

generate_n saves the result of N applications of a function.

remove

remove_if removes elements satisfying specific criteria.

remove_copy

remove_copy_if copies a range of elements omitting those that satisfy specific criteria.

replace

replace_if replaces all values satisfying specific criteria with another value.

replace_copy

replace_copy_if copies a range, replacing elements satisfying specific criteria with another value.

swap swaps the values of two objects.

swap_ranges swaps two ranges of elements.

iter_swap swaps the elements pointed to by two iterators.

reverse reverses the order of elements in a range.

reverse_copy creates a copy of a range that is reversed.

rotate rotates the order of elements in a range.

rotate_copy copies and rotate a range of elements.

random_shuffle (until C++17)

shuffle (since C++11) randomly re-orders elements in a range.

sample (since C++17) selects n random elements from a sequence.

unique removes consecutive duplicate elements in a range.

unique_copy creates a copy of some range of elements that contains no consecutive duplicates.

Partitioning algoritms

is_partitioned (since C++11) determines if the range is partitioned by the given predicate.

partition divides a range of elements into two groups.

partition_copy (since C++11) copies a range dividing the elements. into two groups.

stable_partition divides elements into two groups while preserving their relative order.

partition_point (since C++11) locates the partition point of a partitioned range.

Sorting algoritms

is_sorted (since C++11) checks whether a range is sorted into ascending order.

is_sorted_until (since C++11) finds the largest sorted subrange.

sort sorts a range into ascending order.

partial_sort sorts the first N elements of a range.

partial_sort_copy copies and partially sorts a range of elements.

stable_sort sorts a range of elements while preserving order between equal elements.

nth_element partially sorts the given range making sure that it is partitioned by the given element.

Binary search algorithms

lower_bound returns an iterator to the first element not less than the given value.

upper_bound returns an iterator to the first element greater than a certain value.

binary_search determines if an element exists in a certain range.

equal_range returns range of elements matching a specific key.

Set algorithms (working on sorted ranges)

merge merges two sorted ranges.

inplace_merge merges two ordered ranges in-place.

includes returns true if one set is a subset of another.

set_difference computes the difference between two sets.

set_intersection computes the intersection of two sets.

set_symmetric_difference computes the symmetric difference between two sets.

set_union computes the union of two sets.

Heap operations

is_heap (since C++11) checks if the given range is a max heap.

is_heap_until (since C++11) finds the largest subrange that is a max heap.

make_heap creates a max heap out of a range of elements.

push_heap adds an element to a max heap.

pop_heap removes the largest element from a max heap.

sort_heap turns a max heap into a range of elements sorted in ascending order.

Minimum/maximum operations

max returns the greater of the given values.

max_element returns the largest element in a range.

min returns the smaller of the given values.

min_element returns the smallest element in a range.

minmax (since C++11) returns the smaller and larger of two elements.

minmax_element (since C++11) returns the smallest and the largest elements in a range.

clamp (since C++17) clamps a value between a pair of boundary values.

lexicographical_compare returns true if one range is lexicographically less than another.

is_permutation (since C++11) determines if a sequence is a permutation of another sequence.

next_permutation generates the next greater lexicographic permutation of a range of elements.

prev_permutation generates the next smaller lexicographic permutation of a range of elements.

Numeric operations

Defined in the header **<numeric>**

iota (since C++11) fills a range with successive increments of the starting value.

accumulate sums up a range of elements.

inner_product computes the inner product of two ranges of elements.

adjacent_difference computes the differences between adjacent elements in a range.

partial_sum computes the partial sum of a range of elements.

reduce (since C++17) similar to std::accumulate, except out of order.

exclusive_scan (since C++17) similar to std::partial_sum, excludes the ith input element from the ith sum.

inclusive_scan (since C++17) similar to std::partial_sum, includes the ith input element in the ith sum.

transform_reduce (since C++17) applies a functor, then reduces out of order.

transform_exclusive_scan (since C++17) applies a functor, then calculates exclusive scan.

transform_inclusive_scan (since C++17) applies a functor, then calculates inclusive scan.

Operations on uninitialized memory

Defined in the header **<memory>**

uninitialized_copy copies a range of objects to an uninitialized area of memory.

uninitialized_copy_n (since C++11) copies a number of objects to an uninitialized area of memory.

uninitialized_fill copies an object to an uninitialized area of memory, defined by a range.

uninitialized_fill_n copies an object to an uninitialized area of memory, defined by a start and a count.

C library

Defined in the header **<cstdlib>**

qsort sorts a range of elements with unspecified type.

bsearch searches an array for an element of unspecified type.

Iterators

The iterators povide definitions for five kinds of iterators as well as iterator traits, adapters, and utility functions.

Usage of types declared in a container and iterators makes you be able to write generic code:

```
template<class C> typename C::value_type sum(const C& c)
{
    typename C::value_type s = 0;
    typename C::const_iterator p = c.begin();   // start at the beginning
    while (p!=c.end()) {                        // continue until the end
```

```
        s += *p;        // get value of element
        ++p;            // make p point to next element
    }
    return s;
}
```

Iterators are given in const and non-const form:

```
auto begin( C& c ) -> decltype(c.begin()); (since C++11)
                                           (until C++17)
constexpr auto begin( C& c ) -> decltype(c.begin()); (since C++17)

auto begin( const C& c ) -> decltype(c.begin()); (since C++11)
                                                 (until C++17)
auto begin( const C& c ) -> decltype(c.begin()); (since C++17)

T* begin( T (&array)[N] );             (since C++11)
                                       (until C++14)
constexpr T* begin( T (&array)[N] );                            (since C++14)
constexpr auto cbegin( const C& c ) -> decltype(std::begin(c)); (since C++14)
```

...and the same for **end()** and **cend()** functions.

Also there are *reverse iterators*

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
  typename C::reverse_iterator p = c.rbegin();  // view sequence in reverse
  while (p!=c.rend()) {
    if (*p==v) {
      typename C::iterator i = p.base();
      return --i;
    }
    ++p;            // note: increment, not decrement (--)
  }
  return c.end(); // use c.end() to indicate "not found"
}
```

Be careful on iterator -> reverse iterator conversion:

```
i = ri.base();

          rend()      ri   rbegin()
            |          |       |
            V          V       V
              ___________________ _ _
             |   |   |   |   |   |   :
             | 1 | 2 | 3 | 4 | 5 |   :
             |___|___|___|___|___|_ _:
               ^           ^       ^
               |           |       |
             begin()       i     end()
```