# STL Algorithms

Contents

* Non-range algorithms
* Sequential search
* Comparing ranges
* General iteration
* Copying
* Replacing elements
* Filling ranges
* Removing elements
* Partitioning
* Permuting elements
* Sorting
* Merging
* Heap operations
* Binary search
* Set operations
* Numeric Algorithms
* Algorithms vs. Loops
* Algorithms vs. Member Functions

Except for the numeric algorithms, all algorithms are declared in the header <algorithm>

# Non-range algorithms

Most STL algorithms operate on iterator ranges. These are the exceptions.

swap (x, y);

* Swap two values.
  + Swaps by copying with the help of temporary.
  + Should be overloaded for types that can be swapped more efficiently.
  + namespace gadget {
  + struct widget {
  + void swap(widget&);
  + // ...
  + }
  + void swap (widget& x, widget& y) { x.swap(y); }
  + }

iter\_swap (p, q);

* Same as swap(\*p, \*q).

min (x, y);

min (x, y, binary\_predicate);

max (x, y);

max (x, y, binary\_predicate);

* Returns the smaller of the two or the first if neither is smaller.
  + The default binary\_predicate is operator<.
  + As all order comparisons in STL, it should be a strict weak ordering

# Sequential search

The STL algorithms below search a range for some element or a subrange.

* All iterators are input or forward iterators.
* The return value is an iterator to the element or to the beginning of the subrange if found, and the end iterator if not found.
  + count and count\_if return the number of elements found.
* The optional binary\_predicate of many algorithms is used for comparing two elements.
  + When omited, operator== is used as the default.
  + It does not have to be an equivalence relation. Any binary predicate is legal. For example:
* // find first decrease in values
* string s("abcdcba");
* string::iterator i;
* i = adjacent\_find(begin, end, greater<char>());
* assert(\*i=='d');

Here are the search algorithms:

find (begin, end, value);

find\_if (begin, end, unary\_predicate);

* Finds the first matching element.

count (begin, end, value);

count\_if (begin, end, unary\_predicate);

* Counts matching elements.

find\_first\_of (begin, end, set\_begin, set\_end);

find\_first\_of (begin, end, set\_begin, set\_end, binary\_predicate);

* Finds the first element that is in the set.

adjacent\_find (begin, end);

adjacent\_find (begin, end, binary\_predicate);

* Finds two adjacent elements that match each other.

search\_n (begin, end, n, value);

search\_n (begin, end, n, value, binary\_predicate);

* Finds a range of n elements all which match the value.

search (begin, end, begin2, end2);

search (begin, end, begin2, end2, binary\_predicate);

find\_end (begin, end, begin2, end2);

find\_end (begin, end, begin2, end2, binary\_predicate);

* Finds the first or the last matching subrange.
* string s("first matching subrange");
* string p("match");
* string::iterator i;
* i = search(s.begin(), s.end(), p.begin(), p.end());
* assert(\*i == 'm');

min\_element(begin, end);

min\_element(begin, end, order\_comparison);

max\_element(begin, end);

max\_element(begin, end, order\_comparison);

* Finds the (first) minimal or maximal element.

# Comparing ranges

The following algorithms compare the two ranges [begin1,end1) and [begin2,begin2+(end1-begin1)).

equal (begin1, end1, begin2);

equal (begin1, end1, begin2, binary\_predicate);

* Returns true if all elements match each other.

mismatch (begin1, end1, begin2);

mismatch (begin1, end1, begin2, binary\_predicate);

* Returns iterator to first position in [begin1,end1) that does not match.

lexicographic\_compare (begin1, end1, begin2);

lexicographic\_compare (begin1, end1, begin2, order\_comparison);

* Returns true if the first range is lexicographically smaller than the second.
  + The order comparison is a *strict weak ordering* and defines also the equivalence of elements.

# General iteration

for\_each is a powerful algorithm for iterating over all elements of a range doing something with them.

for\_each(begin, end, unary\_functor);

* Executes unary\_functor(x) for each element x.
  + The calls are made in sequential order always with the same copy of the functor.
  + Returns a copy of the functor.

For example:

struct mean\_value : unary\_function<int,void>

{

mean\_value() : count(0), sum(0) {}

void operator() (int x) { ++count; sum+=x; }

operator double() {

return static\_cast<double>(sum) / static\_cast<double>(count);

}

private:

long count;

long sum;

};

vector<int> v;

// ...

double mean = for\_each(begin, end, mean\_value());

# Copying

The following algorithms copy a range into another.

copy (begin, end, result);

copy\_backward (begin, end, result);

The differences between the two are

* copy does the copying from the beginning to the end and copy\_backward from the end to the beginning.
* copy accepts input and output iterators but copy\_backward requires bidirectional iterators.
* For copy the target range is [result,result+(end-begin)), for copy\_backward it is [result-(end-begin),result).
* copy returns result+(end-begin). copy\_backward returns result-(end-begin).

Either algorithm can be used (but copy should be preferred) except when the two ranges overlap.

* Both algorithms work correctly except when result is in the range [begin,end).
* One of the two will always work.

Two ranges can also be swapped using element-by-element swaps.

swap\_ranges (begin1, end1, begin2);

Many of the following algorithms have a version with copy in the name.

* The non-copy version operates in-place modifying the range.
* The copy version writes the modified range into an output range leaving the original untouched.

copy and other algorithms writing to an output iterator range return an iterator to the end of the output range (result+(end-begin)). Writing can continue using that iterator.

vector<int> v1;

vector<int> v2;

// fill v1 and v2

vector<int> v3;

back\_insert\_iterator <vector<int> > i(v3);

i = copy(v1.begin(), v1.end(), i);

i = fill\_n(i, 10, 0);

i = copy(v2.begin(), v2.end(), i);

# Replacing elements

The following algorithms replace matching elements with a new value.

replace (begin, end, old\_value, new\_value);

replace\_if (begin, end, predicate, new\_value);

replace\_copy (begin, end, result, old\_value, new\_value);

replace\_copy\_if (begin, end, result, predicate, new\_value);

A more powerful algorithm for replacing elements is transform.

transform (begin, end, result, unary\_functor);

* For each element x read from [begin,end) writes unary\_function(x) into [result,result+(end-begin)).
* If result==begin, performs an in-place replacement.

There is also a variant that combines two ranges.

transform (begin1, end1, begin2, result, binary\_functor);

* Reads x1 from [begin1,end1) and x2 from [begin2,begin2+(end-begin)) and writes binary\_functor(x1,x2) into [result,result+(end-begin)).

# Filling ranges

fill (begin, end, value);

* Fills [begin,end) with copies of value.

fill\_n (begin, n, value);

* Fills [begin,begin+n) with copies of value.
  + Suitable for output iterators.
* vector<double> v(10, 1.0);
* fill\_n (back\_inserter(v), 10, 2.0);

A generalization of fill uses a generator (nullary functor) instead of a fixed value.

generate (begin, end, generator);

generate\_n (begin, n, generator);

* For example, the following fills a vector with the first 100 Fibonacci numbers.
* struct fibonacci\_generator {
* typedef int result\_type;
* fibonacci\_generator() : current(0), next(1) {}
* int operator() () {
* int tmp = current;
* current = next;
* next += tmp;
* return tmp;
* }
* private:
* int current;
* int next;
* };
* vector<int> v;
* v.reserve(100);
* generate\_n(back\_inserter(v), 100, fibonacci\_generator());

# Removing elements

The following algorithms remove all matching elements from a range.

remove (begin, end, value);

remove\_if (begin, end, predicate);

remove\_copy (begin, end, result, value);

remove\_copy\_if (begin, end, result, predicate);

* The copy versions copy the unremoved elements leaving the original range untouched.
  + There is no copy\_if algorithm but remove\_copy\_if can be used instead.
* The non-copy versions place the unremoved elements to the beginning of the range [begin,end).
  + The return value is the new\_end.
  + [begin,new\_end) contains the unremoved elements.
  + The contents of [new\_end,end) is unspecified.
* new\_end = remove\_if (begin, end, even());

The elements could not truely be erased, because the algorithm does not know enough about the container to do that. The container's erase member function can be called to perform the actual erasure using this standard idiom.

c.erase(remove(c.begin(), c.end(), value), c.end());

Here's another example of removing all 1's and 2's from a vector.

vector<int> v;

// fill v

vector<int>::iterator new\_end;

new\_end = remove(v.begin(), v.end(), 1);

new\_end = remove(v.begin(), new\_end, 2);

v.erase(new\_end, v.end());

There are also algorithms for removing duplicates:

unique (begin, end);

unique (begin, end, binary\_predicate);

unique\_copy (begin, end, result);

unique\_copy (begin, end, result, binary\_predicate);

* Remove all but the first from each consecutive group of equal or matching elements.
* Usually used on sorted ranges.
* Can be used on unsorted ranges, but only consecutive duplicates are removed.

# Partitioning

partition (begin, end, predicate);

stable\_partition (begin, end, predicate);

* Places all elements satisfying predicate in the beginning. If n is the number of satisfying elements, then after the call:
  + [begin,begin+n) contains the satisfying elements.
  + [begin+n,end) contains the other elements.
  + The return value is begin+n.
  + The stable version is slower but retains the original ordering within each partition.
* mid = partition (begin, end, even());

# Permuting elements

The following algorithms change the order of elements in the range.

reverse (begin, end);

reverse\_copy (begin, end, result);

* Reverses the order of the elements.

rotate (begin, middle, end);

rotate\_copy (begin, middle, end, result);

* Swaps [begin,middle) and [middle,end).

next\_permutation(begin, end);

prev\_permutation(begin, end);

next\_permutation(begin, end, order\_comparison);

prev\_permutation(begin, end, order\_comparison);

* Computes the next or previous permutation in the lexicograhical ordering of all permutations.

random\_suffle (begin, end);

random\_suffle (begin, end, random\_number\_generator);

# Sorting

sort (begin, end);

sort (begin, end, order\_comparison);

* Sorts the range.
  + Unstable: may change order of equal elements.

stable\_sort (begin, end);

stable\_sort (begin, end, order\_comparison);

* Sorts the range stably.

partial\_sort (begin, middle, end);

partial\_sort (begin, middle, end, order\_comparison);

* After the call [begin.middle) contains the smallest elements in ascending order. [middle,end) contains the largest elements in an arbitrary order.

partial\_sort\_copy (begin, end, result\_begin, result\_end);

partial\_sort\_copy (begin, end, result\_begin, result\_end,

order\_comparison);

* The smallest N=min(end-begin, result\_end-result\_begin) elements sre written to [result\_begin,result\_begin+N).

nth\_element (begin, middle, end);

nth\_element (begin, middle, end, order\_comparison);

* After the call:
  + All elements in [begin,middle) are smaller than or equal to \*middle.
  + All elements in [middle,end) are larger than or equal to \*middle.
  + For example, compute a median:
* iterator middle = begin + (end-begin) / 2;
* nth\_element(begin, middle, end);
* median = \*middle;

# Merging

merge (begin1, end1, begin2, end2, result);

merge (begin1, end1, begin2, end2, result, order\_comparison);

* Merge two sorted ranges placing the result into a third one.

inplace\_merge (begin, middle, end);

inplace\_merge (begin, middle, end, order\_comparison);

* Merge [begin,middle) and [middle,end) placing the result into [begin,end).

# Heap operations

These operations implement the standard binary heap.

make\_heap(begin, end);

make\_heap(begin, end, order\_comparison);

push\_heap(begin, end);

push\_heap(begin, end, order\_comparison);

pop\_heap(begin, end);

pop\_heap(begin, end, order\_comparison);

sort\_heap(begin, end);

sort\_heap(begin, end, order\_comparison);

# Binary search

The following algorithms perform a binary search on a sorted range.

binary\_search (begin, end, value);

binary\_search (begin, end, value, order\_comparison);

* Returns true if the value is in the range.

lower\_bound (begin, end, value);

lower\_bound (begin, end, value, order\_comparison);

* Returns an iterator i to the first element that is equal to or greater than value.
  + [begin,i) contains the elements that are smaller than value.
  + i is the first position, where value could be inserted without violating the order.

upper\_bound (begin, end, value);

upper\_bound (begin, end, value, order\_comparison);

* Returns an iterator i to the first element that is greater than value.
  + [i,end) contains the elements that are greater than value.
  + i is the last position, where value could be inserted without violating the order.

equal\_range (begin, end, value);

equal\_range (begin, end, value, order\_comparison);

* Returns an iterator pair p such that
  + [begin,p.first) contains the elements that are smaller than value.
  + [p.first,p.second) contains the elements that are equal to value.
  + [p.second,end) contains the elements that are greater than value.

# Set operations

Set operations on sorted ranges representing sets:

includes (begin1, end1, begin2, end2);

includes (begin1, end1, begin2, end2, order\_comparison);

set\_union (begin1, end1, begin2, end2, result);

set\_union (begin1, end1, begin2, end2, result, order\_comparison);

set\_intersection (begin1, end1, begin2, end2, result);

set\_intersection (begin1, end1, begin2, end2, result, order\_comparison);

set\_difference (begin1, end1, begin2, end2, result);

set\_difference (begin1, end1, begin2, end2, result, order\_comparison);

set\_symmetric\_difference (begin1, end1, begin2, end2, result);

set\_symmetric\_difference (begin1, end1, begin2, end2, result, order\_comparison);

Can be used on associative containers.

# Numeric Algorithms

The following algorithms are defined in the header <numeric>.

accumulate (begin, end, init);

accumulate (begin, end, init, binary\_functor);

* Computes the sum of the elements.
  + init + \*begin + \*(begin+1) + ... + \*(end-1)
  + Allows using binary\_functor in place of operator+.
* struct size\_sum : binary\_function<size\_t, string, size\_t> {
* size\_t operator() (size\_t sum, const string& s) const {
* return sum + s.size();
* }
* };
* vector<string> v;
* // ...
* size\_t total\_length = accumulate(v.begin(), v.end(), 0, size\_sum());

inner\_product (begin1, end1, begin2, init);

inner\_product (begin1, end1, begin2, init,

binary\_function1, binary\_function2);

* Computes inner product (sum over the elementwise product).
  + binary\_function1 generalizes operator+
  + binary\_function2 generalizes operator\*

partial\_sum (begin, end, result);

partial\_sum (begin, end, result, binary\_function);

* Computes cumulative sums.

adjacent\_difference (begin, end, result);

adjacent\_difference (begin, end, result, binary\_function);

* Computes differences of adjacent elements.
  + Inverse operation to partial\_sum.
* // [ 1 2 3 4 ]
* partial\_sum (begin, end, begin);
* // [ 1 3 6 10 ]
* adjacent\_difference (begin, end, begin);
* // [ 1 2 3 4 ]

# Algorithms vs. Loops

Often a call to an algorithms could be replaced with a simple loop.

iterator i = find (c.begin(), c.end(), value);

// is the same as

iterator i;

for ( i=c.begin(); i!=c.end(); ++i ) {

if (\*i==value) break;

}

Algorithms have several advantages over loops:

* Clarity: Algorithm call is (usually) clearer and more self-documenting.
* Correctness: Loops are more prone to errors.
  + Particularly, with a safe STL implementation.
* Efficiency
  + Call c.end() just once.
  + Carefully tuned implementations.
  + Special optimizations:
    - Implement copy with memcpy when possible.
    - Two level iteration over deque.

Sometimes a loop may be clearer. For example, to replace

iterator i;

for ( i=begin; i!=end; ++i ) {

if ( x<\*i && \*i<y ) break;

}

with find\_if there are several possibilities:

* Use bind from TR1/Boost:
* iterator i;
* i = find\_if (begin, end,
* bind(logical\_and<bool>(),
* bind(less<int>(), x, \_1),
* bind(less<int>(), \_1, y)));

which looks confusing.

* Write a separate functor:
* struct between\_values : unary\_function<int,bool> {
* between\_values(int x, int y) : lo(x), hi(y) {}
* bool operator() (int n) const { lo<n && n<hi; }
* private:
* int lo;
* int hi;
* };
* // ...
* iterator i;
* i = find\_if (begin, end, between\_values(x,y));

which moves the details away from the call point.

* Use boost::lambda
* iterator i;
* i = find\_if (begin, end, x<\_1 && \_1<y);

which uses a non-standard library and an exotic technique (expression templates) that may confuse some.

# Algorithms vs. Member Functions

The standard containers do not contain member functions for operations that can be implemented just as well using standard algorithms:

* string is an exception due to historical reasons.

The member functions that exist are there for a reason:

* More efficient than algorithms:
  + Associative container search operations.
    - Binary searching on sorted vector is still faster, though.
* Safer:
  + Associative container search operations use the correct order comparison automatically.
* Work differently from algorithms:
  + Many list members manipulate pointers between nodes rather than move elements.