C++ STL

Last Updated 2016-07-09

Notation

- X ↦ A container
- T ↦ X::value_type
- K ↦ X::key_type
- C ↦ X::key_compare
- p ↦ Iterators to a.
- p1, p2 ↦ Iterator rng to a.
- i,i2,i3 ↦ Integral index
- NTS ↦ Null term string
- o1,o2,o3 ↦ Random objects
- pU ↦ 1 arg predicate
- fU ↦ 1 arg function
- CMP ↦ Binary compare (-1 for <, 0 for ==, 1 for >)
- n,n1,n2 ↦ Object of type X::size_type
- a, b ↦ Objects of type X
- t ↦ Object of type T
- k ↦ Object of type K
- c ↦ Object of type C
- q ↦ Iterators to b.
- q1, q2 ↦ Iterator rng to b.
- ch ↦ A char (charT)
- st ↦ Simple string
- N ↦ Number of elements
- bU ↦ binary predicate

Basic Attributes

- Assignable: o1 = o2 is a valid expression
- Default Constructible: A no-arg constructor exists
- Equality Comparable : o1 == o2 is a valid boolean-like expression
  - Derived:
    - o1 != o2 ⇔ !(o1 == o2)
  - Invariants:
    - Identity: &o1==&o2 ⇒ o1==o2
    - Reflexitivity: o1 == o2
    - Symmetry o1==o2 ⇔ o2==o1
    - Transitivity: o1==o2 && o2==o3 ⇒ o1==o3
- LessThan Comparable: o1<o2 is a valid boolean-like expression
  - Derived:
    - o1>o2 ⇔ o2<o1
    - o1<=o2 ⇔ !(o2<o1)
    - o1>=o2 ⇔ !(o1<o2)
    - o1==o2 ⇔ !(o1<o2) && !(o2<o1)
    - o1!=o2 ⇔ (o1<o2) || (o2<o1)
  - Invariants:
    - Irreflexivity: o1<o2 must be false
    - Antisymmetry: o1<o2 ⇒ !(o2<o1)
    - Transitivity: o1<o2 && o2<o3 ⇒ o1<o3

Container

𝒯: X::allocator_type 𝒯: X::value_type 𝒯: X::difference_type
𝒯: X::iterator 𝒯: X::const_iterator 𝒯: X::size_type
𝒯: X::pointer 𝒯: X::const_pointer
𝒯: X::reference 𝒯: X::const_reference
𝒞: X(b) ↦ Copy all elements from b. $\mathcal{O}$(b.size())
ℳ: a.begin() ↦ $\mathcal{O}$(1) ℳ: a.end() ↦ $\mathcal{O}$(1)
ℳ: a.size() ↦ $\mathcal{O}$(N) ℳ: a.max_size() ↦ $\mathcal{O}$(N)
ℳ: a.empty() ↦ $\mathcal{O}$(1) ℳ: a.swap(b) ↦ $\mathcal{O}$(N)

Forward Container

- If T is Equality Comparable, then X is too (elementwise, $\mathcal{O}$(N)).
  Note: a==b ⇒ (a.size()==b.size())
- If T is LessThan Comparable, then X is too (elementwise, $\mathcal{O}$(N)).
  Note: a<b == lexicographical_compare(a,b).
- Iterators for X are Forward Iterators.

Reversible Container

𝒯: reverse_iterator 𝒯: const_reverse_iterator
ℳ: a.rbegin() ↦ same as X::reverse_iterator(a.end())
ℳ: a.rend() ↦ same as X::reverse_iterator(a.begin())

- Iterators for X are Bidirectional Iterators.

Random Access Container

ℳ: a[n] ↦ Access to element n, 0<=n<N [reference if a is mutable]
ℳ: a.at(n) ↦ Access to element at n.

- Iterators for X are Random Access Iterators

Sequence

𝒞: X(n, t) ↦ Construct X with n copies of t $\mathcal{O}$(n)
𝒞: X(q1, q2) ↦ Copy range [q1, q2) into new object. $\mathcal{O}$(q2-q1)
ℳ: a.front() ↦ Same as *(a.first()) $\mathcal{AO}$(1)
ℳ: a.insert(p, t) ↦ Insert a copy of t before p.
ℳ: a.insert(p, n, t) ↦ Insert n copies of t before p.
ℳ: a.insert(p, q1, q2) ↦ Inserts stuff from [q1, q2) before p
ℳ: a.erase(p) ↦ Erases element at p, returns ++p
ℳ: a.erase(p1, p2) ↦ Erases [p1, p2). returns ++p
ℳ: a.clear() ↦ Same as a.erase(a.begin(), a.end())
ℳ: a.resize(n, t=T()) ↦ Resize adding copies of t if required

Front Insertion Sequence

ℳ: a.push_front(t) ↦ same as a.insert(a.begin(), t) $\mathcal{AO}$(1)
ℳ: a.pop_front() ↦ same as a.erase(a.begin()). $\mathcal{AO}$(1)

Back Insertion Sequence

ℳ: a.back() ↦ same as *(--a.end()) $\mathcal{AO}$(1)
ℳ: a.push_back(t) ↦ same as a.insert(a.begin(), t) $\mathcal{AO}$(1)
ℳ: a.pop_back() ↦ same as a.erase(--a.end()) $\mathcal{AO}$(1)

Associative Container

𝒞: X(c) ↦ Construct with c as compare
𝒞: X(q1, q2, c) ↦ Copy from [q1, q2) and use c as compare
𝒯: key_type 𝒯: value_compare 𝒯: compare_type
ℳ: a.erase(k) ↦ Destroy elements with key k (returns nothing)
ℳ: a.erase(p) ↦ Destroy element at p (returns nothing) $\mathcal{AO}$(1)
ℳ: a.erase(p1, p2) ↦ Destroy elements in [p, q) (returns nothing)
ℳ: a.clear() ↦ Same as a.erase(a.begin(), a.end())
ℳ: a.find(k)
ℳ: a.count(k) ↦ Return number of elements with key equal to k
ℳ: a.equal_range(k)

Simple Associative Container

- X::key_type and X::value_type must be the same
- X::iterator is the same as X::const_iterator (i.e. not mutable)

Pair Associative Container

𝒯: X::mapped_type ↦ Same as T 𝒯: X::value_type ↦ pair<const Key, T>

Unique Associative Container

- No two elements can have equivalent key.
- Equality is based on the order for sorted UACs.

Sorted Associative Container

𝒯: X::key_compare 𝒯: X::value_compare
ℳ: a.lower_bound(k) Return iterator to first key not less than k. $\mathcal{O}$(logN)
ℳ: a.upper_bound(k) Return iterator to first key greater less than k. $\mathcal{O}$(logN)
ℳ: a.equal_range(k) Return pair(lower_bound(k), upper_bound(k)). $\mathcal{O}$(logN)

- The keys must be LessThan Comparable.
- Iterators are bidirectional (ordered induced by key order)
- find and count members are $\mathcal{O}$(logN)
- The keys are const -- (or at least no order changing modifications)

vector #include <vector>

vector<T, Allocator=>
ℳ: a.capacity() ↦ Return the current storage capacity. $\mathcal{O}$(1)
ℳ: a.reserve(n) ↦ Make space for n elements (invalidating iterators) $\mathcal{O}$(N)

- Insert at the end is $\mathcal{AO}$(1), while insert beginning is $\mathcal{AO}$(N).
- Iterators can be invalidated by most any insert.
- Element access is $\mathcal{O}$(1).

deque (pronounced DECK) #include <deque>

deque<T, Allocator=>

- Insert at the beginning or end is $\mathcal{AO}$(1), others as bad as $\mathcal{AO}$(N).
- Iterators can be invalidated by most any insert.
- Element access is $\mathcal{O}$(1).

list #include <list>

list<T, Allocator=>
ℳ: a.splice(p, &b)
ℳ: a.splice(p, &b, q)
ℳ: a.splice(p, &b, q1, q2)
ℳ: a.remove(t) ↦ Remove all occurrences of t. $\mathcal{O}$(N)
ℳ: a.remote_if(pU) ↦ Remove if f(*i) is true
ℳ: a.unique() ↦ Keep first element of equal subsequences
ℳ: a.unique(pB)
ℳ: a.merge(b)
ℳ: a.merge(b, pB)
ℳ: a.reverse() ↦ Reverse the list. $\mathcal{O}$(N)
ℳ: a.sort() ↦ Sort the list. $\mathcal{O}$(NlogN)
ℳ: a.sort(pB)

- Inserting an element ANYPLACE is $\mathcal{O}$(1).
- Insertion and removal do not invalidate iterators.

map & multimap #include <map>

map<Key, T, Compare=, Allocator=>
ℳ: a[k] ↦ Access the reference to data part of elements. Insert pair(k,T()) if missing.

- Inserts do not invalidate iterators, pointers, or references.
- Element erasure invalidates iterators-like things pointing to the thing being eliminated.

map<Key, T, Compare=, Allocator=>

- Just like a map, except that keys do not have to be unique.

set & multiset #include <set>

set<Key, Compare=, Allocator=>

- A map with, logically, key==value for every element.

multiset<Key, Compare=, Allocator=>

- Just like a set, except that keys do not have to be unique.

unordered_map & unordered_multimap #include <unordered_map>

unordered_map<Key, T, Hash=, EqPred=, Allocator=>
ℳ: a[k] ↦ Access the reference to data part of elements. Insert pair(k,T()) if missing.

- Inserts do not invalidate iterators, pointers, or references.
- Element erasure invalidates iterators-like things pointing to the thing being eliminated.

unordered_multimap<Key, T, Hash=, EqPred=, Allocator=>

- Just like a unordered_map, except that keys do not have to be unique.

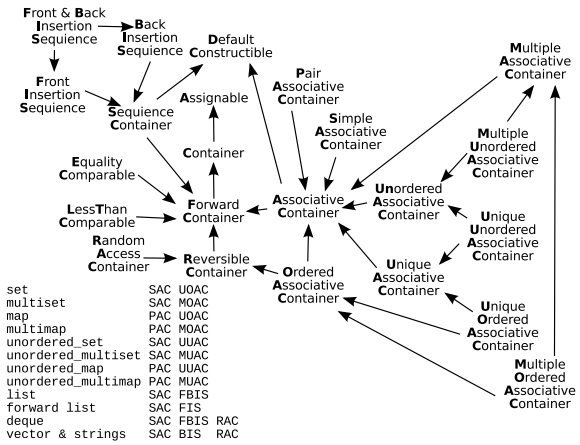
unordered_set & unordered_multiset #include <unordered_set>

unordered_set<Key, Hash=, EqPred=, Allocator=>

- A unordered_map with, logically, key==value for every element.

unordered_multiset<Key, Hash=, EqPred=, Allocator=>

- Just like a unordered_set, except that keys do not have to be unique.



stack (LIFO) #include <stack>

stack<T, Container=dqueue<T>> -- Container adapter
ℳ: stack(X a=X()) ↦ Copy content of a into stack $\mathcal{O}$(a.size())
𝒯: value_type 𝒯: size_type 𝒯: container_type
ℳ: empty() ↦ ℳ: top() ↦ the back $\mathcal{O}$(1)
ℳ: pop() ↦ on back $\mathcal{O}$(1) ℳ: push() ↦ off back $\mathcal{O}$(1)
ℳ: size() ↦ This one can be $\mathcal{O}$(1), but can be as bad as $\mathcal{O}$(N)

priority queue #include <queue>

priority_queue<T, Container=dqueue<T>, Compare=less<T>> -- Container adapter
𝒞: priority_queue(X a=X()) ↦ Copy content of a into stack
𝒯: value_type 𝒯: size_type 𝒯: container_type
ℳ: empty() ↦ $\mathcal{O}$(1) ℳ: top() ↦ $\mathcal{O}$(1)
ℳ: pop() ↦ from back $\mathcal{O}$(?) ℳ: push() ↦ sorted $\mathcal{O}$(?)
ℳ: size() ↦ This one can be $\mathcal{O}$(1), but can be as bad as $\mathcal{O}$(N)

queue (FIFO)

queue<T, Container=dqueue<T>> -- Container adapter

$\mathcal{C}$: queue(X a=X()) ↦ Copy content of a into stack $\mathcal{O}$(a.size())

$\mathcal{T}$x: value_type $\mathcal{T}$x: size_type $\mathcal{T}$x: container_type

$\mathcal{M}_f$: back() ↦ $\mathcal{O}$(1) $\mathcal{M}_f$: empty() ↦ $\mathcal{O}$(1) $\mathcal{M}_f$: top() ↦ $\mathcal{O}$(1)

$\mathcal{M}_f$: pop() ↦ from back $\mathcal{O}$(1) $\mathcal{M}_f$: push() ↦ on front $\mathcal{O}$(1)

$\mathcal{M}_f$: size() ↦ This one can be $\mathcal{O}$(1), but can be as bad as $\mathcal{O}$(N)

pair #include <utility>

pair<K, T> -- Contain a pair of objects

$\mathcal{C}$: pair(k, t) $\mathcal{C}$: pair() $\mathcal{C}$: pair(pair<K,T>)

$\mathcal{T}$x: first_type $\mathcal{T}$x: second_type

$\mathcal{M}_f$: first() $\mathcal{M}_f$: second() $\mathcal{M}_f$: make_pair(k, t)

- If K and T are equality comparable then so is the pair
- If K and T are LessThan comparable then so is the pair

Input Iterator

Expressions:

$\mathcal{E}$: ++p ↦ Preincrement

$\mathcal{E}$: (void)p++ ↦ Postincrement

$\mathcal{E}$: *p++ ↦ Postincrement and dereference

$\mathcal{E}$: *p ↦ deference. Can't read any element more than once.

$\mathcal{E}$: p->m ↦ member access

Output Iterator

$\mathcal{E}$: ++p ↦ Preincrement

$\mathcal{E}$: (void)p++ ↦ Postincrement

$\mathcal{E}$: *p=t ↦ Dereference assignment. Only one time per element.

Forward Iterator

- Elements may be dereferenced multiple times and they can be be assigned to (via a dereference assignment) multiple times.
- May be used any place an Output iterator or Input iterator can.

Bidirectional Iterator

$\mathcal{E}$: p-- ↦ Postdecrement

$\mathcal{E}$: --p ↦ Predecrement

Random Access Iterator

$\mathcal{E}$: p += n ↦ Increment n times

$\mathcal{E}$: p -= n ↦ Decrement n times

$\mathcal{E}$: p+n ↦ same as p+=n (n+p is OK too)

$\mathcal{E}$: p - n ↦ same as p-=n

$\mathcal{E}$: p1 - p2 ↦ Returns integral distance

$\mathcal{E}$: p[n] ↦ Get n-th value

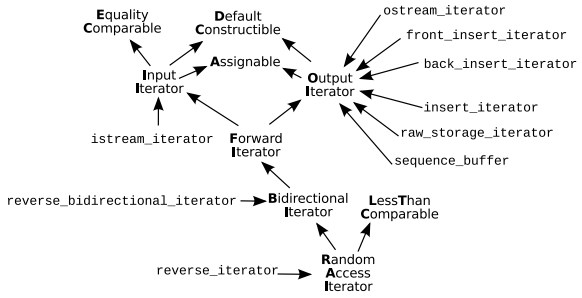
$\mathcal{E}$: p[n]=t ↦ Set the n-th value

- A pointer IS a random access iterator and may be used in just about any place a RAI can be used.

Iterator Adapters

- front_insert_iterator
- back_insert_iterator
- insert_iterator
- reverse_iterator
- reverse_bidirectional_iterator

Iterator Diagram



Algorithms: Non-modifying

$\mathcal{F}$n: count(p1, p2, t) ↦ Number of times in [p1, p2) element ==t

$\mathcal{F}$n: count_if(p1, p2, pU) ↦ Number of times in [p1, p2) pU is true

$\mathcal{F}$n: for_each(p1, p2, fU) ↦ Evaluates fU on each thing in [p1, p2)

$\mathcal{F}$n: find(p1, p2, t)

$\mathcal{F}$n: find_if(p1, p2, pB)

- (*) ↦ (o1, o2), (o1, o2, c)
  - $\mathcal{F}$n: min(*) $\mathcal{F}$n: max(*)
- (*) ↦ (p1, p2), (p1, p2, c)
  - $\mathcal{F}$n: min_element(*) $\mathcal{F}$n: max_element(*)
- (*) ↦ (p1, p2, t), (p1, p2, t, c)
  - $\mathcal{F}$n: equal_reverse(*) $\mathcal{F}$n: lower_bound(*) $\mathcal{F}$n: upper_bound(*)
- (*) ↦ (p1, p2, q), (p1, p2, q, pB)
  - $\mathcal{F}$n: equal(*) $\mathcal{F}$n: mismatch(*)
- (*) ↦ (p1, p2, q1, q2), (p1, p2, q1, q2, pB)
  - $\mathcal{F}$n: find_end(*) $\mathcal{F}$n: find_first_of(*) $\mathcal{F}$n: search(*)
- (*) ↦ (p1, p2), (p1, p2, pB)
  - $\mathcal{F}$n: adjacent_find(*)
- (*) ↦ (p1, p2, n, t), (p1, p2, n, pB)
  - $\mathcal{F}$n: search_n(*)
- (*) ↦ (p1, p2, t), (p1, p2, c)
  - $\mathcal{F}$n: binary_search(*)
- (*) ↦ (p1, p2, q1, q2), (p1, p2, q1, q2, c)
  - $\mathcal{F}$n: lexicographical_compare(*)

Algorithms: Modifying

- (*) ↦ (p1, p2, q)
  - $\mathcal{F}$n: copy(*) $\mathcal{F}$n: copy_backward(*) $\mathcal{F}$n: uninitialized_copy(*)
  - $\mathcal{F}$n: swap_ranges(*)
  - $\mathcal{F}$n: reverse_copy(*) $\mathcal{F}$n: unique_copy(*)
- (*) ↦ (p1, p2)
  - $\mathcal{F}$n: stable_sort(*) $\mathcal{F}$n: unique(*) $\mathcal{F}$n: sort(*)
  - $\mathcal{F}$n: iter_swap(*) $\mathcal{F}$n: random_shuffle(*) $\mathcal{F}$n: reverse(*)
- (*) ↦ (p1, p2, pB)
  - $\mathcal{F}$n: partition(*) $\mathcal{F}$n: unique(*) $\mathcal{F}$n: stable_partition(*)
- (*) ↦ (p1, p2, c)
  - $\mathcal{F}$n: sort(*) $\mathcal{F}$n: stable_sort(*)
- (*) ↦ (p1, p2, t)
  - $\mathcal{F}$n: fill $\mathcal{F}$n: remove(*) $\mathcal{F}$n: uninitialized_fill(*)
- (*) ↦ (p1, p2, pU, t)
  - $\mathcal{F}$n: replace_copy_if(*) $\mathcal{F}$n: replace_if(*)
- (*) ↦ (first, middle, last), (first, middle, last, q)
  - $\mathcal{F}$n: rotate(*) $\mathcal{F}$n: rotate_copy(*)
- (*) ↦ (p1, p2, q, unryOp), (p1, p2, p3, q, binOp)
  - $\mathcal{F}$n: transform(*) $\mathcal{F}$n: transform(*)
- (*) ↦ (first, middle, last), (first, middle, last, c)
  - $\mathcal{F}$n: partial_sort(*)

$\mathcal{F}$n: partial_sort_copy(first, last, result_first, result_last)

$\mathcal{F}$n: partial_sort_copy(first, last, result_first, result_last, c)

$\mathcal{F}$n: remove_copy(p1, p2, q, t)

$\mathcal{F}$n: remove_copy_if(p1, p2, q, pU)

$\mathcal{F}$n: generate(p1, p2, gen),

$\mathcal{F}$n: generate_n(p1, n, gen)

$\mathcal{F}$n: fill_n(p1, n, t)

$\mathcal{F}$n: random_shuffle(p1, p2, rand)

$\mathcal{F}$n: remove_if(p1, p2, pU)

$\mathcal{F}$n: replace(p1, p2, old_val, new_val)

$\mathcal{F}$n: replace_copy(p1, p2, q, old_val, new_val)

$\mathcal{F}$n: unique_copy(p1, p2, q, pB)

$\mathcal{F}$n: uninitialized_copy(p1, n, t)

Algorithms: Set Stuff

- includes(p1, p2, q1, q2, c=less<>) ↦ true if [q1,q2) is a subset of [p1, p2)
- (*) ↦ (p1, p2, q1, q2, w) (p1, p2, q1, q2, w, c) [output is placed at w]
  - $\mathcal{F}$n: set_difference $\mathcal{F}$n: set_intersection
  - $\mathcal{F}$n: set_symmetric_difference $\mathcal{F}$n: set_union

Algorithms: Random Stuff

- (*) ↦ (p1, p2, init_val), (p1, p2, init_val, binOp)
  - $\mathcal{F}$n: accumulate
- (*) ↦ (p1, p2, p3, init_val), (p1, p2, p3, init_val, binOp1, binOp2)
  - $\mathcal{F}$n: inner_product
- (*) ↦ (p1, p2, q), (p1, p2, q, binOp)
  - $\mathcal{F}$n: adjacent_difference $\mathcal{F}$n: partial_sum
- (*) ↦ (p1, p2), (p1, p2, c)
  - $\mathcal{F}$n: next_permutation $\mathcal{F}$n: prev_permutation

$\mathcal{F}$n: swapI(t1, t2)

Strings

- typedefs:
  - typedef basic_string<char> string
  - typedef basic_string<wchar_t> wstring
- Strings are ALMOST containers!
  - The string name behaves like a random access iterator
  - Don't support pop_back, back, front

$\mathcal{E}$: a = o1 ↦ Construct out of o1 (a charT or NTS)

$\mathcal{E}$: a += o1 ↦ call append with o1 as argument

$\mathcal{E}$: a.c_str() ↦ Return pointer null terminated, c-style, string

$\mathcal{E}$: a.data() ↦ Return pointer array of charT (not null terminated)

$\mathcal{M}_f$: copy(charT* dst, n, pos=0) ↦ Copy n chrs of a starting at pos to dst.

$\mathcal{M}_f$: length() ↦ returns size()

$\mathcal{M}_f$: reserve(n=0)

$\mathcal{M}_f$: resize(n, ch=charT())

- (*) ↦ (pos=0, n=npos)
  - $\mathcal{M}_f$: substr(*) $\mathcal{M}_f$: erase(*)
- (*) ↦ (bs, pos=0), (nts, pos, n), (nts, pos=0), (ch, pos=0)
  - $\mathcal{M}_f$: find(*) $\mathcal{M}_f$: find_first_not_of(*) $\mathcal{M}_f$: find_first_of(*)
  - $\mathcal{M}_f$: rfind(*) $\mathcal{M}_f$: find_last_not_of(*) $\mathcal{M}_f$: find_last_of(*)
- (*) ↦ (bs), (bs, pos, n), (nts), (nts, n), (n, ch), (q1, q2)
  - $\mathcal{M}_f$: assign(*) $\mathcal{M}_f$: append(*)
- (*) ↦ (pos1,n2,bs,pos2,n2), (pos1,n1,bs), (pos1,n1,nts), (pos1,n1,nts,n2), (pos1,n1,n2,ch) (p1,p2,q1,q2), (p1,p2,bs), (p1,p2,nts), (p1,p2,nts,n), (p1,p2,n,ch)
  - $\mathcal{M}_f$: replace(*)
- (*) ↦ (pos, bs), (pos, bs, pos2, n), (pos, nts), (pos, nts, n), (pos, n, ch)
  - $\mathcal{M}_f$: insert(*)
- (*) ↦ (bs), (pos1,n,bs), (nts), (pos1,n1,bs,pos2,n2), (pos1,n1,nts,n2), (pos1,n1,nts)
  - $\mathcal{M}_f$: compare(*)