

# C++ STL

©1997-2004 Mitch Richling  
Last Updated 2004-09-08

## 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 operator: o1 != o2 equivalent to !(o1 == o2)
  - Invariants:
    - Identity: &o1==&o2 ⇒ o1==o2
    - Reflexivity: 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 binary operators:
    - o1>o2 equivalent to !(o2<o1)
    - o1<=o2 equivalent to !(o2<o1)
    - o1>=o2 equivalent to !(o1<o2)
  - Invariants:
    - Irreflexivity: o1<o2 must be false
    - Antisymmetry: o1<o2 ⇒ !(o2<o1)
    - Transitivity: o1<o2 && o2<o3 ⇒ o1<o3

## Container

- X: X::allocator\_type    K: X::value\_type    K: X::difference\_type
- X: iterator            K: X::const\_iterator    K: X::size\_type
- X: X::pointer         K: X::const\_pointer
- X: X::reference       K: X::const\_reference
- G: X(b) → Copy all elements from b. O(b.size())
- S: a.begin() → O(1)            S: a.end() → O(1)
- S: a.size() → O(N)            S: a.max\_size() → O(N)
- S: a.empty() → O(1)          S: a.swap(b) → O(N)

## Forward Container

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

## Reversible Container

- X: reverse\_iterator        K: const\_reverse\_iterator
- S: a.rbegin() → same as X::reverse\_iterator(a.end())
- S: a.rend() → same as X::reverse\_iterator(a.begin())
- Iterators for X are Bidirectional Iterators.

## Random Access Container

- S: a[n] → Access to element n, 0<=n<N [reference if a is mutable]
- S: a.at(n) → Access to element at n.
- Iterators for X are Random Access Iterators

## Sequence

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

## Front Insertion Sequence

- S: a.push\_front(t) → same as a.insert(a.begin(), t) AO(1)
- S: a.pop\_front() → same as a.erase(a.begin()). AO(1)

## Back Insertion Sequence

- S: a.back() → same as \*(--a.end()) AO(1)
- S: a.push\_back(t) → same as a.insert(a.end(), t) AO(1)
- S: a.pop\_back() → same as a.erase(--a.end()) AO(1)

## Associative Container

- G: X(c) → Construct with c as compare
- G: X(q1, q2, c) → Copy from [q1, q2) and use c as compare
- K: key\_type            K: value\_compare        K: compare\_type
- S: a.erase(k) → Destroy elements with key k (returns nothing)
- S: a.erase(p) → Destroy element at p (returns nothing) AO(1)
- S: a.erase(p1, p2) → Destroy elements in [p, q) (returns nothing)
- S: a.clear() → Same as a.erase(a.begin(), a.end())
- S: a.find(k) →
- S: a.count(k) → Return number of elements with key equal to k
- S: 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

- K: X::mapped\_type → Same as T    K: 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

- K: X::key\_compare    K: X::value\_compare
- S: a.lower\_bound(k) Return iterator to first key not less than k. O(logN)
- S: a.upper\_bound(k) Return iterator to first key greater less than k. O(logN)
- S: a.equal\_range(k) Return pair(lower\_bound(k), upper\_bound(k)). O(logN)
- The keys must be LessThan Comparable.
- Iterators are bidirectional (ordered induced by key order)
- find and count members are O(logN)
- The keys are const -- (or at least no order changing modifications)

## vector #include <vector>

- vector<typename T, typename Allocator=allocator<T> >
- S: a.capacity() → Return the current storage capacity. O(1)
- S: a.reserve(n) → Make space for n elements (invalidating iterators) O(N)
- Insert at the end is AO(1), while insert beginning is AO(N).
- Iterators can be invalidated by most any insert.
- Element access is O(1).

## deque (pronounced DECK) #include <deque>

- deque<typename T, typename Allocator=allocator<T> >
- Insert at the beginning or end is AO(1), others as bad as AO(N).
- Iterators can be invalidated by most any insert.
- Element access is O(1).

## list #include <list>

- list<typename T, typename Allocator=allocator<T> >
- S: a.splice(p, &b)
- S: a.splice(p, &b, q)
- S: a.splice(p, &b, q1, q2)
- S: a.remove(t) → Remove all occurrences of t. O(N)
- S: a.remove\_if(pU) → Remove if f(i) is true
- S: a.unique() → Keep first element of equal subsequences
- S: a.unique(pB)
- S: a.merge(b)
- S: a.merge(b, pB)
- S: a.reverse() → Reverse the list. O(N)
- S: a.sort() → Sort the list. O(NlogN)
- S: a.sort(pB)
- Inserting an element ANYPLACE is O(1).
- Insertion and removal do not invalidate iterators.

## set #include <set>

- set<Key, Compare=less<>, typename Allocator=allocator<T> >
- A map with, logically, key==value for every element.

## map #include <map>

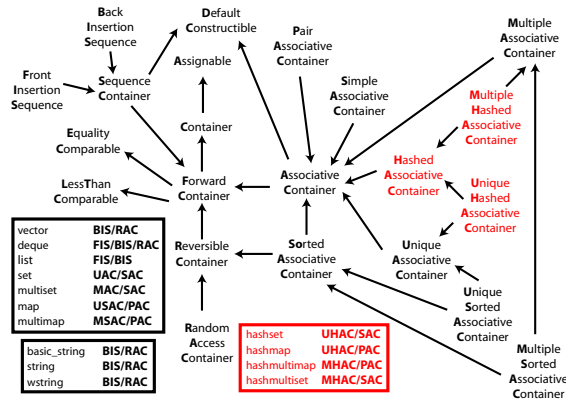
- map<typename Key, typename T, Compare=less<>, typename Allocator=allocator<T> >
- S: a[k] → Access the reference to data part of elements. Insert pair(k,T()) if missing.
- Inserts do not invalidate iterators, pointers, or references.
- Element erasure invalidates iterators-like things pointing to the thing being eliminated.

## multiset #include <multiset>

- multiset<typename Key, Compare=less<>, typename Allocator=allocator<T> >
- Just like a set, except that keys do not have to be unique.

## multimap #include <multimap>

- map<typename Key, typename T, Compare=less<>, typename Allocator=allocator<T> >
- Just like a map, except that keys do not have to be unique.



## stack (LIFO) #include <stack>

- stack<T, Container=deque<T> > -- Container adapter
- S: stack(X a=X()) → Copy content of a into stack O(a.size())
- K: value\_type            K: size\_type            K: container\_type
- Mf: empty() →
- Mf: pop() → on back O(1)    Mf: top() → the back O(1)
- Mf: push() → off back O(1)
- Mf: size() → This one can be O(1), but can be as bad as O(N)

## priority queue #include <queue>

- priority\_queue<T, Container=deque<T>, Compare=less<T> > -- Container adapter
- G: priority\_queue(X a=X()) → Copy content of a into stack
- K: value\_type            K: size\_type            K: container\_type
- Mf: empty() → O(1)            Mf: top() → O(1)
- Mf: pop() → from back O(?)    Mf: push() → sorted O(?)
- Mf: size() → This one can be O(1), but can be as bad as O(N)

## queue (FIFO)

- queue<T, Container=deque<T> > -- Container adapter
- G: queue(X a=X()) → Copy content of a into stack O(a.size())
- K: value\_type            K: size\_type            K: container\_type
- Mf: back() → O(1)            Mf: empty() → O(1)            Mf: top() → O(1)
- Mf: pop() → from back O(1)    Mf: push() → on front O(1)
- Mf: size() → This one can be O(1), but can be as bad as O(N)

## pair #include <utility>

- pair<K, T> -- Contain a pair of objects
- G: pair(k, t)            G: pair()            G: pair(pair<K,T>)
- K: first\_type            K: second\_type
- Mf: first()            Mf: second()            Mf: 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  $\mapsto$  Preincrement  
 $\mathcal{E}$ : (void)p++  $\mapsto$  Postincrement  
 $\mathcal{E}$ : \*p++  $\mapsto$  Postincrement and dereference  
 $\mathcal{E}$ : \*p  $\mapsto$  dereference. Can't read any element more than once.  
 $\mathcal{E}$ : p->m  $\mapsto$  member access

## Output Iterator

$\mathcal{E}$ : ++p  $\mapsto$  Preincrement  
 $\mathcal{E}$ : (void)p++  $\mapsto$  Postincrement  
 $\mathcal{E}$ : \*p=t  $\mapsto$  Dereference assignment. Only one time per element.

## Forward Iterator

- Elements may be dereferenced multiple times and they can 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--  $\mapsto$  Postdecrement  
 $\mathcal{E}$ : --p  $\mapsto$  Predecrement

## Random Access Iterator

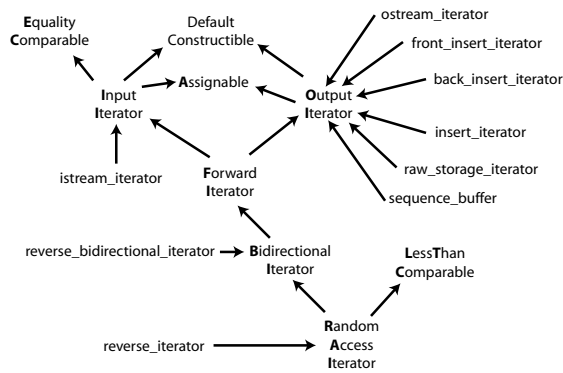
$\mathcal{E}$ : p += n  $\mapsto$  Increment n times  
 $\mathcal{E}$ : p -= n  $\mapsto$  Decrement n times  
 $\mathcal{E}$ : p+n  $\mapsto$  same as p+=n (n+p is OK too)  
 $\mathcal{E}$ : p - n  $\mapsto$  same as p-=n  
 $\mathcal{E}$ : p1 - p2  $\mapsto$  Returns integral distance  
 $\mathcal{E}$ : p[n]  $\mapsto$  Get n-th value  
 $\mathcal{E}$ : p[n]=t  $\mapsto$  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

$\mathcal{N}$ : count(p1, p2, t)  $\mapsto$  Number of times in [p1, p2) element ==t  
 $\mathcal{N}$ : count\_if(p1, p2, pU)  $\mapsto$  Number of times in [p1, p2) pU is true  
 $\mathcal{N}$ : for\_each(p1, p2, fU)  $\mapsto$  Evaluates fU on each thing in [p1, p2)  
 $\mathcal{N}$ : find(p1, p2, t)  
 $\mathcal{N}$ : find\_if(p1, p2, pB)

- (\*)  $\mapsto$  (o1, o2), (o1, o2, c)  $\mathcal{N}$ : min(\*)  $\mathcal{N}$ : max(\*)
- (\*)  $\mapsto$  (p1, p2), (p1, p2, c)  $\mathcal{N}$ : min\_element(\*)  $\mathcal{N}$ : max\_element(\*)
- (\*)  $\mapsto$  (p1, p2, t), (p1, p2, t, c)  $\mathcal{N}$ : equal\_reverse(\*)  $\mathcal{N}$ : lower\_bound(\*)  $\mathcal{N}$ : upper\_bound(\*)
- (\*)  $\mapsto$  (p1, p2, q), (p1, p2, q, pB)  $\mathcal{N}$ : equal(\*)  $\mathcal{N}$ : mismatch(\*)
- (\*)  $\mapsto$  (p1, p2, q1, q2), (p1, p2, q1, q2, pB)  $\mathcal{N}$ : find\_end(\*)  $\mathcal{N}$ : find\_first\_of(\*)  $\mathcal{N}$ : search(\*)
- (\*)  $\mapsto$  (p1, p2), (p1, p2, pB)  $\mathcal{N}$ : adjacent\_find(\*)
- (\*)  $\mapsto$  (p1, p2, n, t), (p1, p2, n, pB)  $\mathcal{N}$ : search\_n(\*)
- (\*)  $\mapsto$  (p1, p2, t), (p1, p2, c)  $\mathcal{N}$ : binary\_search(\*)
- (\*)  $\mapsto$  (p1, p2, q1, q2), (p1, p2, q1, q2, c)  $\mathcal{N}$ : lexicographical\_compare(\*)

## Algorithms: Modifying

- (\*)  $\mapsto$  (p1, p2, q)  $\mathcal{N}$ : copy(\*)  $\mathcal{N}$ : copy\_backward(\*)  $\mathcal{N}$ : uninitialized\_copy(\*)  
 $\mathcal{N}$ : swap\_ranges(\*)  
 $\mathcal{N}$ : reverse\_copy(\*)  $\mathcal{N}$ : unique\_copy(\*)
- (\*)  $\mapsto$  (p1, p2)  $\mathcal{N}$ : stable\_sort(\*)  $\mathcal{N}$ : unique(\*)  $\mathcal{N}$ : sort(\*)  
 $\mathcal{N}$ : iter\_swap(\*)  $\mathcal{N}$ : random\_shuffle(\*)  $\mathcal{N}$ : reverse(\*)
- (\*)  $\mapsto$  (p1, p2, pB)  $\mathcal{N}$ : partition(\*)  $\mathcal{N}$ : unique(\*)  $\mathcal{N}$ : stable\_partition(\*)
- (\*)  $\mapsto$  (p1, p2, c)  $\mathcal{N}$ : sort(\*)  $\mathcal{N}$ : stable\_sort(\*)
- (\*)  $\mapsto$  (p1, p2, t)  $\mathcal{N}$ : fill  $\mathcal{N}$ : remove(\*)  $\mathcal{N}$ : uninitialized\_fill(\*)
- (\*)  $\mapsto$  (p1, p2, pU, t)  $\mathcal{N}$ : replace\_copy\_if(\*)  $\mathcal{N}$ : replace\_if(\*)
- (\*)  $\mapsto$  (first, middle, last), (first, middle, last, q)  $\mathcal{N}$ : rotate(\*)  $\mathcal{N}$ : rotate\_copy(\*)
- (\*)  $\mapsto$  (p1, p2, q, unryOp), (p1, p2, p3, q, binOp)  $\mathcal{N}$ : transform(\*)  $\mathcal{N}$ : transform(\*)
- (\*)  $\mapsto$  (first, middle, last), (first, middle, last, c)  $\mathcal{N}$ : partial\_sort(\*)  
 $\mathcal{N}$ : partial\_sort\_copy(first, last, result\_first, result\_last)  
 $\mathcal{N}$ : partial\_sort\_copy(first, last, result\_first, result\_last, c)
- $\mathcal{N}$ : remove\_copy(p1, p2, q, t)  
 $\mathcal{N}$ : remove\_copy\_if(p1, p2, q, pU)  
 $\mathcal{N}$ : generate(p1, p2, gen)  
 $\mathcal{N}$ : generate\_n(p1, n, gen)  
 $\mathcal{N}$ : fill\_n(p1, n, t)  
 $\mathcal{N}$ : random\_shuffle(p1, p2, rand)  
 $\mathcal{N}$ : remove\_if(p1, p2, pU)  
 $\mathcal{N}$ : replace(p1, p2, old\_val, new\_val)  
 $\mathcal{N}$ : replace\_copy(p1, p2, q, old\_val, new\_val)  
 $\mathcal{N}$ : unique\_copy(p1, p2, q, pB)  
 $\mathcal{N}$ : uninitialized\_copy(p1, n, t)

## Algorithms: Set Stuff

- includes(p1, p2, q1, q2, c=less<>)  $\mapsto$  true if [q1,q2) is a subset of [p1, p2)
- (\*)  $\mapsto$  (p1, p2, q1, q2, w) (p1, p2, q1, q2, w, c) [output is placed at w]  
 $\mathcal{N}$ : set\_difference  $\mathcal{N}$ : set\_intersection  
 $\mathcal{N}$ : set\_symmetric\_difference  $\mathcal{N}$ : set\_union

## Algorithms: Random Stuff

- (\*)  $\mapsto$  (p1, p2, init\_val), (p1, p2, init\_val, binOp)  
 $\mathcal{N}$ : accumulate
- (\*)  $\mapsto$  (p1, p2, p3, init\_val), (p1, p2, p3, init\_val, binOp1, binOp2)  
 $\mathcal{N}$ : inner\_product
- (\*)  $\mapsto$  (p1, p2, q), (p1, p2, q, binOp)  
 $\mathcal{N}$ : adjacent\_difference  $\mathcal{N}$ : partial\_sum
- (\*)  $\mapsto$  (p1, p2), (p1, p2, c)  $\mathcal{N}$ : next\_permutation  $\mathcal{N}$ : prev\_permutation
- $\mathcal{N}$ : 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  $\mapsto$  Construct out of o1 (a charT or NTS)
- $\mathcal{E}$ : a += o1  $\mapsto$  call append with o1 as argument
- $\mathcal{E}$ : a.c\_str()  $\mapsto$  Return pointer null terminated, c-style, string
- $\mathcal{E}$ : a.data()  $\mapsto$  Return pointer array of charT (not null terminated)
- $\mathcal{M}$ : copy(charT\* dst, n, pos=0)  $\mapsto$  Copy n chrs of a starting at pos to dst.
- $\mathcal{M}$ : length()  $\mapsto$  returns size()
- $\mathcal{M}$ : reserve(n=0)
- $\mathcal{M}$ : resize(n, ch=charT())
- (\*)  $\mapsto$  (pos=0, n=npos)  $\mathcal{M}$ : substr(\*)  $\mathcal{M}$ : erase(\*)
- (\*)  $\mapsto$  (bs, pos=0), (nts, pos, n), (nts, pos=0), (ch, pos=0)  
 $\mathcal{M}$ : find(\*)  $\mathcal{M}$ : find\_first\_not\_of(\*)  $\mathcal{M}$ : find\_first\_of(\*)  
 $\mathcal{M}$ : rfind(\*)  $\mathcal{M}$ : find\_last\_not\_of(\*)  $\mathcal{M}$ : find\_last\_of(\*)
- (\*)  $\mapsto$  (bs), (bs, pos, n), (nts), (nts, n), (n, ch), (q1, q2)  
 $\mathcal{M}$ : assign(\*)  $\mathcal{M}$ : append(\*)
- (\*)  $\mapsto$  (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)  
 $\mathcal{M}$ : replace(\*)
- (\*)  $\mapsto$  (pos, bs), (pos, bs, pos2, n), (pos, nts), (pos, nts, n), (pos, n, ch)  
 $\mathcal{M}$ : insert(\*)
- (\*)  $\mapsto$  (bs), (pos1,n,bs), (nts), (pos1,n1,bs,pos2,n2), (pos1,n1,nts,n2), (pos1,n1,nts)  
 $\mathcal{M}$ : compare(\*)