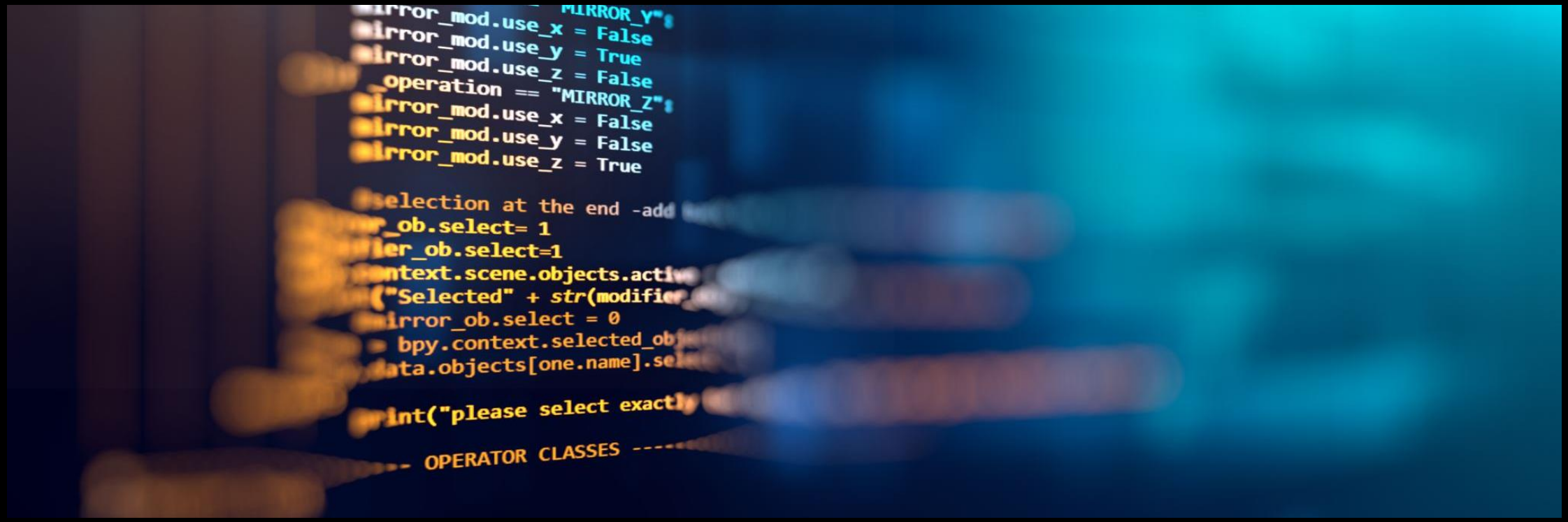


ALGORITHMS VIA C++ STL

Risman Adnan & Algorithm Mentors - Samsung R&D Indonesia (SRIN)

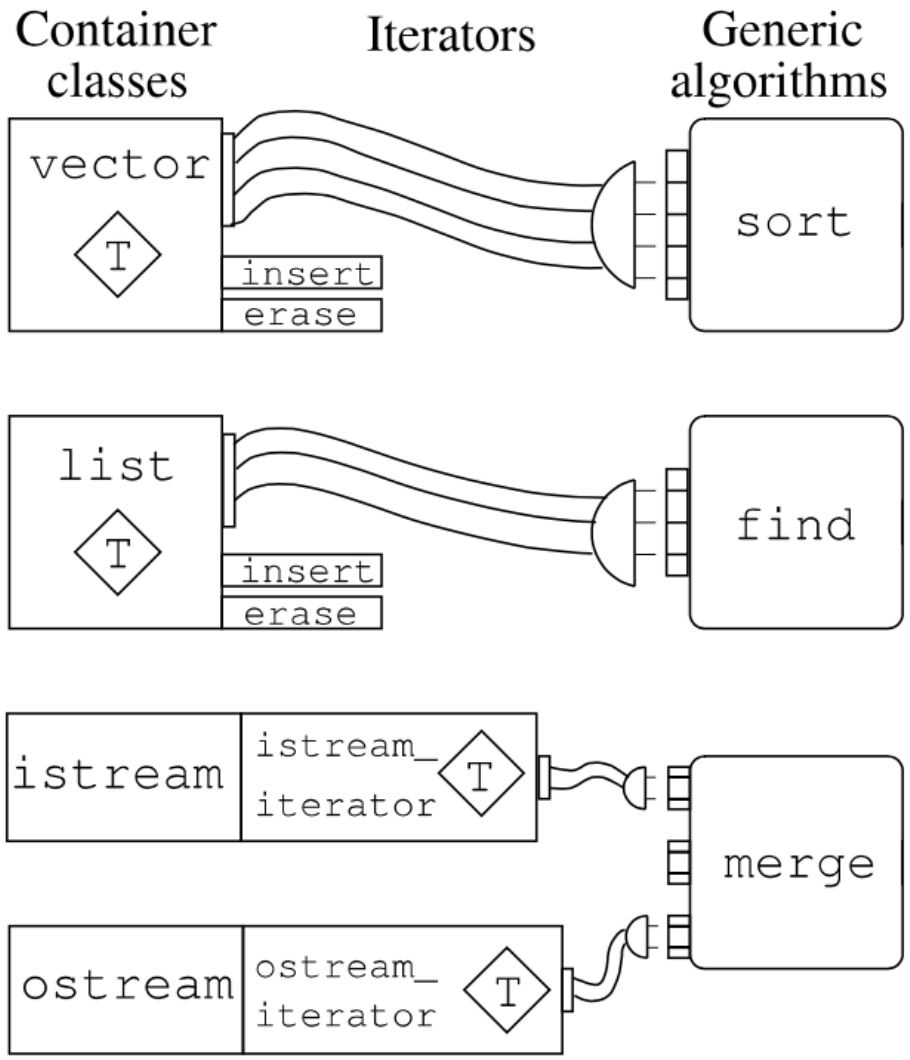
OUTLINE

What is STL

STL Containers

STL Iterators

STL Algorithms



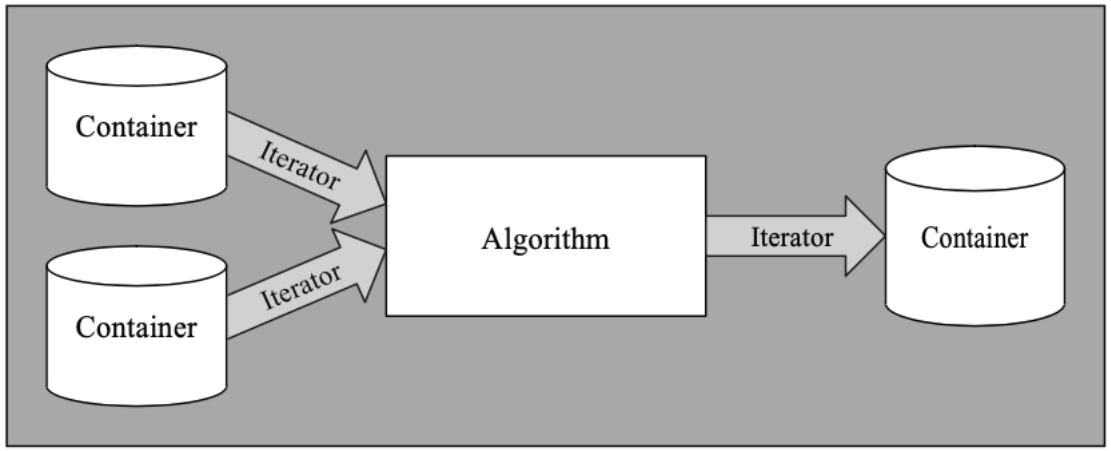
WHAT IS STL

- C++ Standard Template Library
- Widely Used in Modern C++
- Scalable and High Performance
- Key Ideas from Alexander Stepanov
  - Generic Programming
  - Abstractness without loss of efficiency
  - Von Neumann Computational Model
  - Value Semantic

WHAT IS STL

A generic library that provides solutions to managing collections of data with modern and efficient algorithms.

- **Containers** are used to manage collections of objects of a certain kind.
- **Iterators** common interface for all containers to step/traverse through the elements
- **Algorithms** are used to process the elements of collections, such as search, sort, modify



WHY USE STL IN SWC?

- **Reuse: "write less, do more"**
  - STL hides complex, tedious & error prone details
  - We can then focus on the problem at hand
  - Type-safe plug compatibility between STL components
- **Flexibility**
  - Iterators decouple algorithms from containers
  - Unanticipated combinations easily supported
- **Efficiency**
  - Templates avoid virtual function overhead
  - Strict attention to time complexity of algorithms

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main(){

    string greet = "Welcome to STL!";
    cout << greet << endl;

    vector<int> even;

    for (int i=0;i<100;i++){
        even.push_back(i*2);
    }

    for (int j=0; j<even.size();j++){
        cout << even[j] << " ";
    }
    cout << endl << endl;

    for (auto elem: even){
        cout << elem << " ";
    }

    return 0;
}
```

C++ STL CHOICES

Special Containers:
1. Stack
2. Queue
3. Priority Queue
4. Bitset

Common in SWC:
1. Linked List
2. Stack and Queue
3. Priority Queue
4. Hash Table

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing references, pointers	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

C++
DEMOS

1. STL Array – array<>
2. Dynamic Array – vector<>
3. Single Linked List – forward_list<>
4. Double Ended Queue – deque<>
5. Priority Queue – priority_queue<>

SWC > stl01.cpp > main()

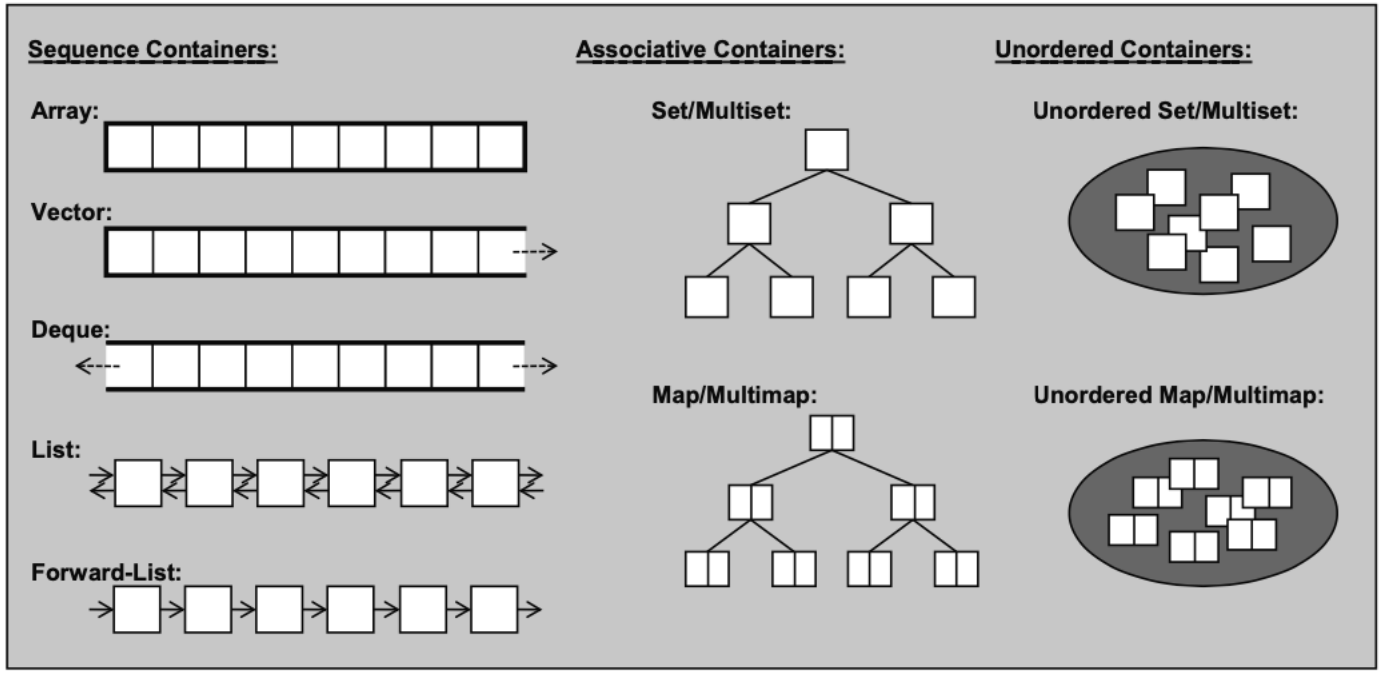
```cpp
#include <iostream>
#include <vector>
#include <forward_list>

using namespace std;

int main(){

    vector<int> a;
    forward_list<int> b;

    for (int i=0;i<=10;i++){
        a.push_back(2*i);
        b.push_front(3*i);
    }

    // use iterator and auto type
    for (auto elem:a) cout << elem << "-";
    cout << endl;
    for (auto elem:b) cout << elem << "++";


    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

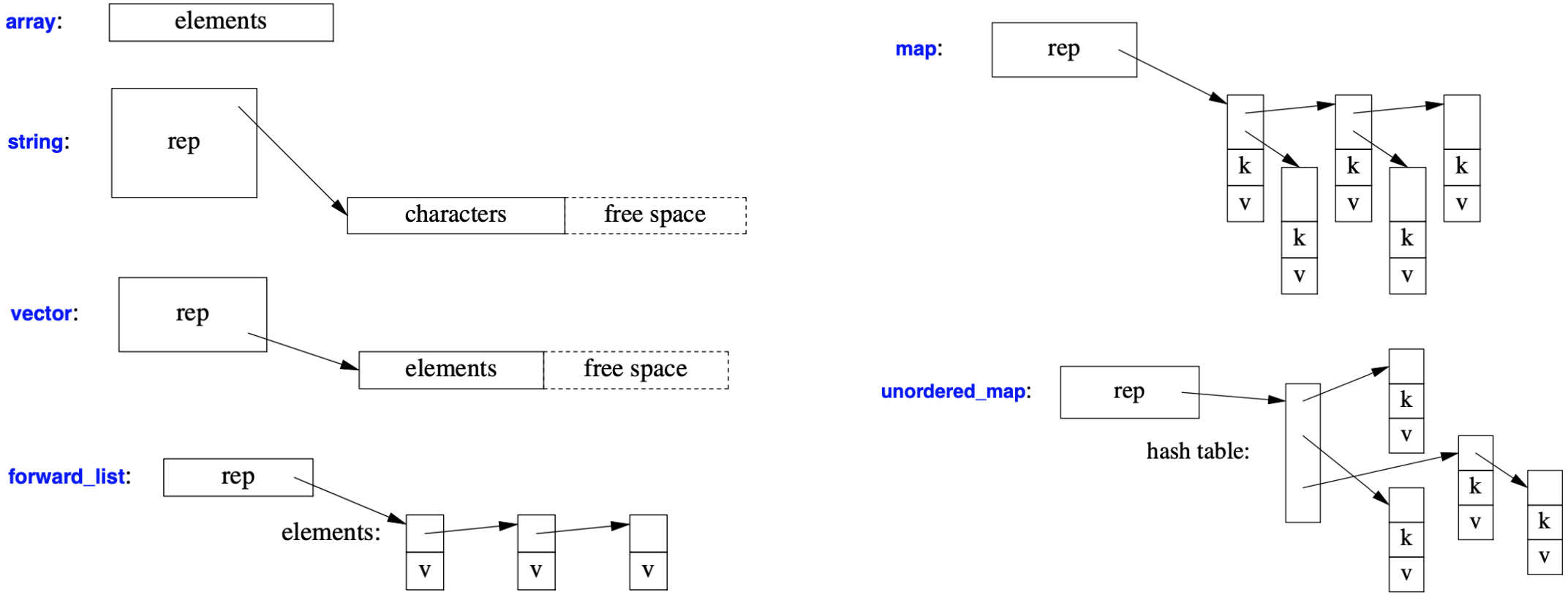
```
rismanadnan@Rismans-iMac SWC % c++ -std=c++11 -o stl01 stl01.cpp
rismanadnan@Rismans-iMac SWC % ./stl01
0-2-4-6-8-10-12-14-16-18-20-
30++27++24++21++18++15++12++9++6++3++0++%
rismanadnan@Rismans-iMac SWC %
```

STL CONTAINERS

- **Sequence**
  - Implemented as Arrays or Lists
- **Associative**
  - Implemented as Binary Trees
- **Unordered**
  - Implemented as Hash Tables
- **Adaptors**
  - Different way of access



STL CONTAINERS



STL CONTAINERS

The standard-library operations have complexity guarantees!

Sequence Containers	
vector<T,A>	A contiguously allocated sequence of Ts; the default choice of container
list<T,A>	A doubly-linked list of T; use when you need to insert and delete elements without moving existing elements
forward_list<T,A>	A singly-linked list of T; ideal for empty and very short sequences
deque<T,A>	A double-ended queue of T; a cross between a vector and a list; slower than one or the other for most uses

Ordered Associative Containers (§iso.23.4.2) C is the type of the comparison; A is the allocator type	
map<K,V,C,A>	An ordered map from K to V; a sequence of (K,V) pairs
multimap<K,V,C,A>	An ordered map from K to V; duplicate keys allowed
set<K,C,A>	An ordered set of K
multiset<K,C,A>	An ordered set of K; duplicate keys allowed

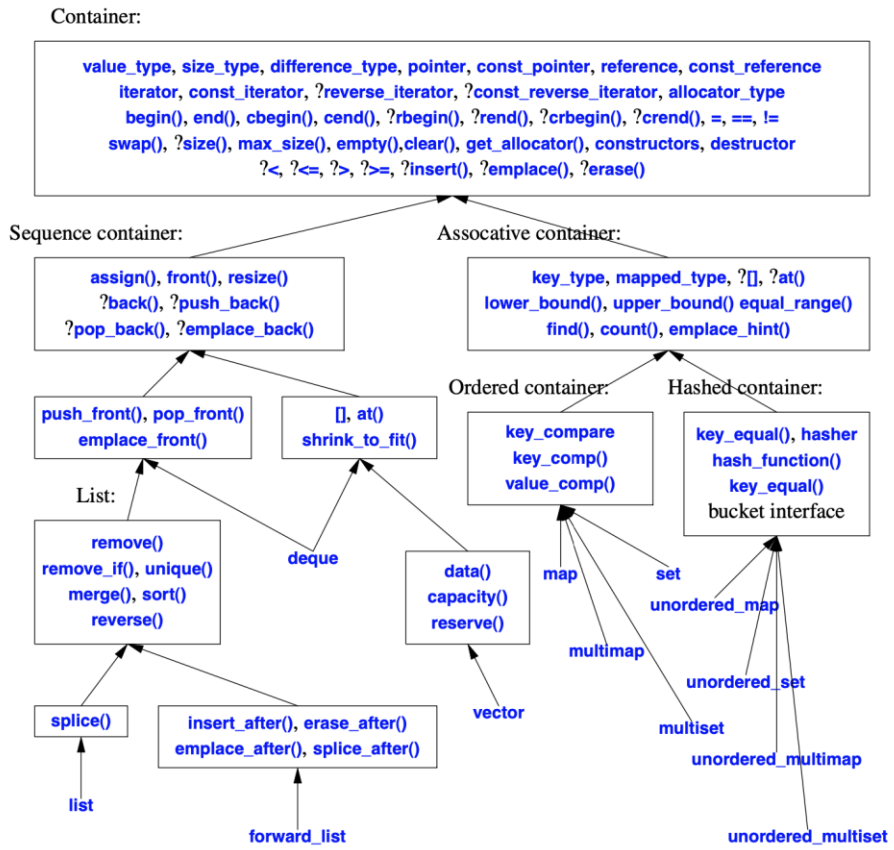
Unordered Associative Containers (§iso.23.5.2) H is the hash function type; E is the equality test; A is the allocator type	
unordered_map<K,V,H,E,A>	An unordered map from K to V
unordered_multimap<K,V,H,E,A>	An unordered map from K to V; duplicate keys allowed
unordered_set<K,H,E,A>	An unordered set of K
unordered_multiset<K,H,E,A>	An unordered set of K; duplicate keys allowed

Container Adaptors C is the container type	
priority_queue<T,C,Cmp>	Priority queue of Ts; Cmp is the priority function type
queue<T,C>	Queue of Ts with push() and pop()
stack<T,C>	Stack of Ts with push() and pop()

Standard Container Operation Complexity					
	[]	List	Front	Back	Iterators
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
built-in array	const				Ran
valarray	const				Ran
bitset	const				

STL CONTAINERS

- A **multi∗** associative container or a set does not provide **[]** or **at()**.
- A **forward_list** does not provide **insert()**, **erase()**, or **emplace()**; instead, it provides the **∗_after** operations.
- A **forward_list** does not provide **back()**, **push_back()**, **pop_back()**, or **emplace_back()**.
- A **forward_list** does not provide **reverse_iterator**, **const_reverse_iterator**, **rbegin()**, **rend()**, **crbegin()**, **crend()**, or **size()**.
- A **unordered_∗** associative container does not provide **<**, **<=**, **>**, or **>=**.



STL CONTAINERS

Member types	
value_type	Type of element
allocator_type	Type of memory manager
size_type	Unsigned type of container subscripts, element counts, etc.
difference_type	Signed type of difference between iterators
iterator	Behaves like value_type*
const_iterator	Behaves like const_value_type*
reverse_iterator	Behaves like value_type*
const_reverse_iterator	Behaves like const_value_type*
reference	value_type&
const_reference	const_value_type&
pointer	Behaves like value_type*
const_pointer	Behaves like const_value_type*
key_type	Type of key; associative containers only
mapped_type	Type of mapped value; associative containers only
key_compare	Type of comparison criterion; ordered containers only
hasher	Type of hash function; unordered containers only
key_equal	Type of equivalence function; unordered containers only
local_iterator	Type of bucket iterator; unordered containers only
const_local_iterator	Type of bucket iterator; unordered containers only

Constructors, Destructor, and Assignment (continues) C is a container; by default, a C uses the default allocator C::allocator_type{}	
C c {};	Default constructor: c is an empty container
C c {a};	Default construct c; use allocator a
C c(n);	c initialized with n elements with the value value_type{}; not for associative containers
C c(n,x);	Initialize c with n copies of x; not for associative containers
C c(n,x,a);	Initialize c with n copies of x; use allocator a; not for associative containers
C c {elem};	Initialize c from elem; if C has an initializer-list constructor, prefer that; otherwise, use another constructor
C c {c2};	Copy constructor: copy c2's elements and allocator into c
C c {move(c2)};	Move constructor: move c2's elements and allocator into c
C c {{elem},a};	Initialize c from the initializer_list {elem}; use allocator a
C c {b,e};	Initialize c with elements from [b:e)
C c {b,e,a};	Initialize c with elements from [b:e); use allocator a
c.~C()	Destructor: destroy c's elements and release all resources
c2=c	Copy assignment: copy c's elements into c2
c2=move(c)	Move assignment: move c's elements into c2
c={elem}	Assign to c from initializer_list {elem}
c.assign(n,x)	Assign n copies of x; not for associative containers
c.assign(b,e)	Assign to c from [b:e)
c.assign({elem})	Assign to c from initializer_list {elem}

STL CONTAINERS

- For **insert()** functions, the result, **q**, points to the last element inserted.
- For **erase()** functions, **q** points to the element that followed the last element erased.
- For containers with contiguous allocation, such as **vector** and **deque**, inserting and erasing an element can cause elements to be moved.
- The **emplace()** operation is used when it is notationally awkward or potentially inefficient to first create an object and then copy (or move) it into a container.

List Operations	
q=c.insert(p,x)	Add x before p; use copy or move
q=c.insert(p,n,x)	Add n copies of x before p; if c is an associative container, p is a hint of where to start searching
q=c.insert(p,first,last)	Add elements from [first:last) before p; not for associative containers
q=c.insert(p,{elem})	Add elements from initializer_list {elem} before p; p is a hint of where to start searching for a place to put the new element; for ordered associative containers only
q=c.emplace(p,args)	Add element constructed from args before p; not for associative containers
q=c.erase(p)	Remove element at p from c
q=c.erase(first,last)	Erase [first:last) of c
c.clear()	Erase all elements of c

STL CONTAINERS

Element Access	
c.front()	Reference to first element of c; not for associative containers
c.back()	Reference to last element of c; not for forward_list or associative containers
c[i]	Reference to the ith element of c; unchecked access; not for lists or associative containers
c.at(i)	Reference to the ith element of c; throw an out_of_range if i is out of range; not for lists or associative containers
c[k]	Reference to the element with key k of c; insert (k,mapped_type{}) if not found; for map and unordered_map only
c.at(k)	Reference to the ith element of c; throw an out_of_range if k is not found; for map and unordered_map only

Stack Operations	
c.push_back(x)	Add x to c (using copy or move) after the last element
c.pop_back()	Remove the last element from c
c.emplace_back(args)	Add an object constructed from args to c after the last element

Comparisons and Swap	
c1==c2	Do all corresponding elements of c1 and c2 compare equal?
c1!=c2	!(c1==c2)
c1<c2	Is c1 lexicographically before c2?
c1<=c2	!(c2<c1)
c1>c2	c2<c1
c1>=c2	!(c1<c2)
c1.swap(c2)	Exchanges values of c1 and c2; noexcept
swap(c1,c2)	c1.swap(c2)

Size and Capacity	
x=c.size()	x is the number of elements of c
c.empty()	Is c empty?
x=c.max_size()	x is the largest possible number of elements of c
x=c.capacity()	x is the space allocated for c; vector and string only
c.reserve(n)	Reserve space for n elements for c; vector and string only
c.resize(n)	Change size of c to n; use the default element value for added elements; sequence containers only (and string)
c.resize(n,v)	Change size of c to n; use v for added elements; sequence containers only (and string)
c.shrink_to_fit()	Make c.capacity() equal to c.size(); vector, deque, and string only
c.clear()	Erase all elements of c

Iterators	
p=c.begin()	p points to first element of c
p=c.end()	p points to one-past-last element of c
cp=c.cbegin()	p points to constant first element of c
p=c.cend()	p points to constant one-past-last element of c
p=c.rbegin()	p points to first element of reverse sequence of c
p=c.rend()	p points to one-past-last element of reverse sequence of c
p=c.crbegin()	p points to constant first element of reverse sequence of c
p=c.crend()	p points to constant one-past-last element of reverse sequence of c

C++
DEMOS

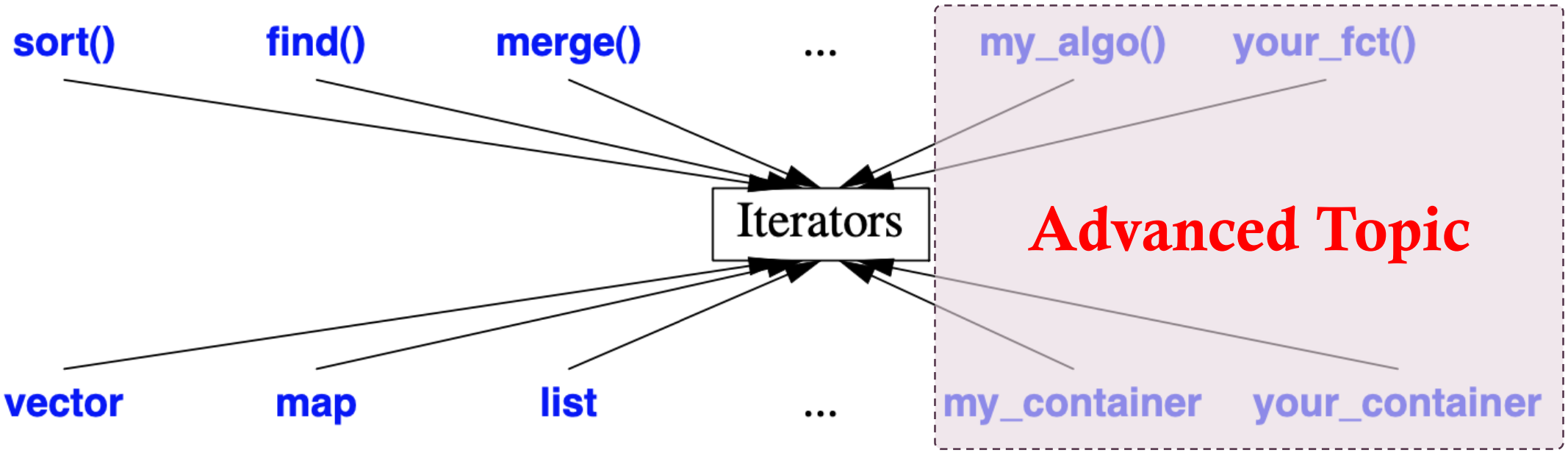
1. Sets and Multisets
2. Maps and Multimaps
3. Unordered Containers – Hash Tables

SWC > stl01.cpp > main()

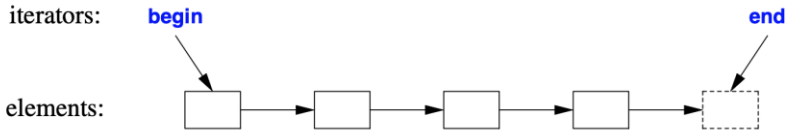
```cpp
#include <iostream>
#include <vector>
#include <forward_list>

using namespace std;

int main(){

    vector<int> a;
    forward_list<int> b;

    for (int i=0;i<=10;i++){
        a.push_back(2*i);
        b.push_front(3*i);
    }

    // use iterator and auto type
    for (auto elem:a) cout << elem << "-";
    cout << endl;
    for (auto elem:b) cout << elem << "++";


    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
rismanadnan@Rismans-iMac SWC % c++ -std=c++11 -o stl01 stl01.cpp
rismanadnan@Rismans-iMac SWC % ./stl01
0-2-4-6-8-10-12-14-16-18-20-
30++27++24++21++18++15++12++9++6++3++0++
rismanadnan@Rismans-iMac SWC %
```



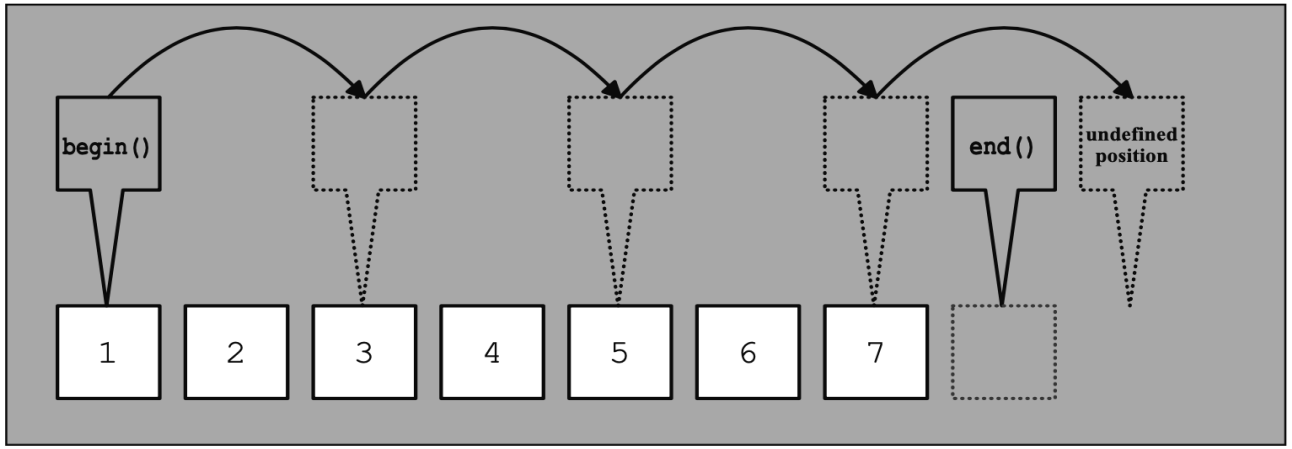
STL ITERATORS

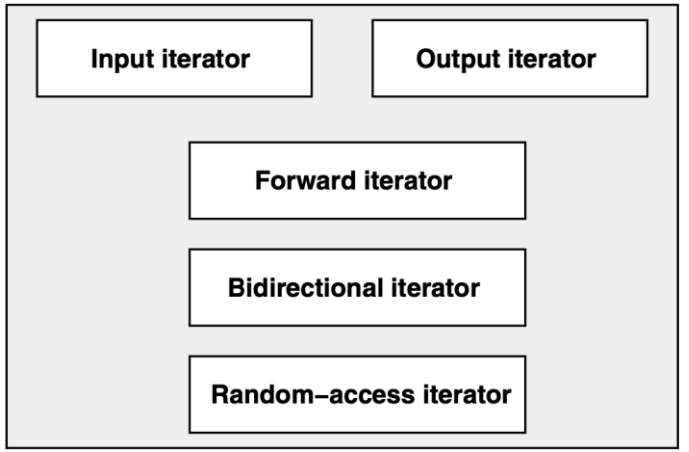


- Iterators are the glue that ties standard-library algorithms to their data. Iterators are the mechanism used to minimize an algorithm's dependence on the data structures.
- An iterator is akin to a pointer in that it provides operations for indirect access (e.g., ∗ for dereferencing) and for moving to point to a new element (e.g., ++ for moving to the next element).

STL ITERATORS

- All containers define their own iterator types, so you don't need a special header file for using iterators of containers. However, several definitions for special iterators, such as reverse iterators, and some auxiliary iterator functions are introduced by the **<iterator>** header file.





STL ITERATORS

Iterator Category	Ability	Providers
Output iterator	Writes forward	Ostream, inserter
Input iterator	Reads forward once	Istream
Forward iterator	Reads forward	Forward list, unordered containers
Bidirectional iterator	Reads forward and backward	List, set, multiset, map, multimap
Random-access iterator	Reads with random access	Array, vector, deque, string, C-style array

Expression	Effect
**iter = val*	Writes *val* to where the iterator refers
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
TYPE(iter)	Copies iterator (copy constructor)

Expression	Effect
**iter*	Provides read access to the actual element
iter ->member	Provides read access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward
iter1 == iter2	Returns whether two iterators are equal
iter1 != iter2	Returns whether two iterators are not equal
TYPE(iter)	Copies iterator (copy constructor)

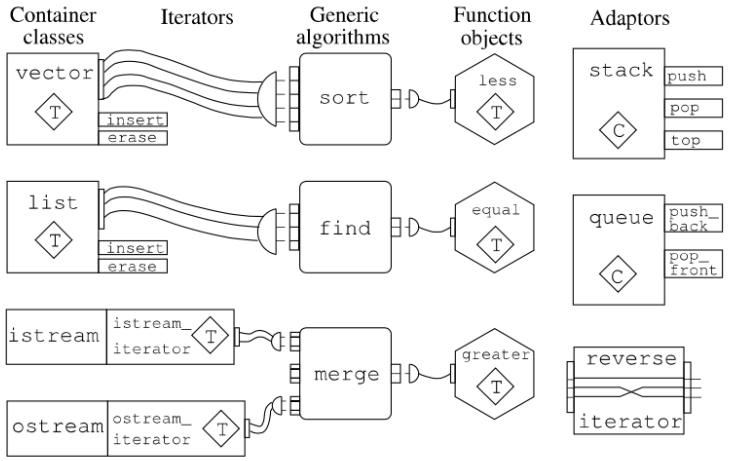
Expression	Effect
**iter*	Provides access to the actual element
iter->member	Provides access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
iter1 == iter2	Returns whether two iterators are equal
iter1 != iter2	Returns whether two iterators are not equal
TYPE()	Creates iterator (default constructor)
TYPE(iter)	Copies iterator (copy constructor)
iter1 = iter2	Assigns an iterator

Expression	Effect
--iter	Steps backward (returns new position)
iter--	Steps backward (returns old position)

Expression	Effect
iter[n]	Provides access to the element that has index *n*
iter+=n	Steps *n* elements forward (or backward, if *n* is negative)
iter-=n	Steps *n* elements backward (or forward, if *n* is negative)
iter+n	Returns the iterator of the *n*th next element
n+iter	Returns the iterator of the *n*th next element
iter-n	Returns the iterator of the *n*th previous element
iter1-iter2	Returns the distance between *iter1* and *iter2*
iter1<iter2	Returns whether *iter1* is before *iter2*
iter1>iter2	Returns whether *iter1* is after *iter2*
iter1<=iter2	Returns whether *iter1* is not after *iter2*
iter1>=iter2	Returns whether *iter1* is not before *iter2*

STL ITERATORS

Iterator Operations (§iso.24.2.2)	
++p	Pre-increment (advance one element): make **p** refer to the next element or to the one-beyond-the-last element; the resulting value is the incremented value
p++	Post-increment (advance one element): make **p** refer to the next element or to the one-beyond-the-last element; the resulting value is **p**'s value before the increment
***p**	Access (dereference): ***p** refers to the element pointed to by **p**
--p	Pre-decrement (go back one element): make **p** point to the previous element; the resulting value is the decremented value
p--	Post-decrement (go back one element): make **p** point to the previous element; the resulting value is **p**'s value before the decrement
p[n]	Access (subscripting): **p[n]** refers to the element pointed to by **p+n**; equivalent to ***(p+n)**
p->n	Access (member access): equivalent to **(*p).m**
p==q	Equality: Do **p** and **q** point to the same element or do both point to the one-beyond-the-last element?
p!=q	Inequality: **!(p==q)**
p<q	Does **p** point to an element before the one **q** points to?
p<=q	**p<q || p==q**
p>q	Does **p** point to an element after the one **q** points to?
p>=q	**p>q || p==q**
p+=n	Advance **n**: make **p** point to the **n**th element after the one to which it points
p-=n	Advance **-n**: make **p** point to the **n**th element before the one to which it points
q=p+n	**q** points to the **n**th element after the one **p** points to
q=p-n	**q** points to the **n**th element before the one **p** points to



advance(p)	Like **p+=n**; **p** must be at least an input iterator
x=distance(p,q)	Like **x=q-p**; **p** must be at least an input iterator
q=next(p,n)	Like **q=p+n**; **p** must be at least a forward iterator
q=next(p)	**q=next(p,1)**
q=prev(p,n)	Like **q=p-n**; **p** must be at least a bidirectional iterator
q=prev(p)	**q=prev(p,1)**

Iterator Adaptors	
reverse_iterator	Iterate backward
back_insert_iterator	Insert at end
front_insert_iterator	Insert at beginning
insert_iterator	Insert anywhere
move_iterator	Move rather than copy
raw_storage_iterator	Write to uninitialized storage

C++
DEMOS

1. Using Common Iterators
2. Writing Custom Iterators*

SWC > stl01.cpp > main()

```cpp
#include <iostream>
#include <vector>
#include <forward_list>

using namespace std;

int main(){

    vector<int> a;
    forward_list<int> b;

    for (int i=0;i<=10;i++){
        a.push_back(2*i);
        b.push_front(3*i);
    }

    // use iterator and auto type
    for (auto elem:a) cout << elem << "-";
    cout << endl;
    for (auto elem:b) cout << elem << "++";


    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

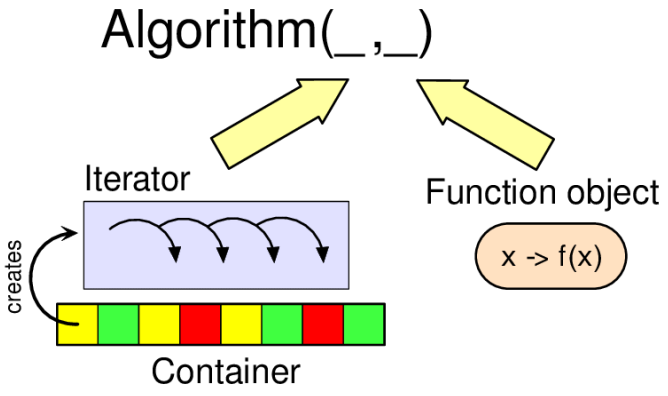
```
rismanadnan@Rismans-iMac SWC % c++ -std=c++11 -o stl01 stl01.cpp
rismanadnan@Rismans-iMac SWC % ./stl01
0-2-4-6-8-10-12-14-16-18-20-
30++27++24++21++18++15++12++9++6++3++0++%
rismanadnan@Rismans-iMac SWC %
```

Algorithm Complexity

(1)	swap(), iter_swap()
(log(n))	lower_bound(), upper_bund(), equal_range(), binary_search(), push_he
(n∗log(n))	inplace_merge() (worst case), stable_partition() (worst case), sort(), stable_sort(), partial_sort(), partial_sort_copy(), sort_heap()
(n∗n)	find_end(), find_first_of(), search(), search_n()
(n)	All the rest

STL ALGORITHMS

- Non Modifying algorithms
- Modifying algorithms
- Removing algorithms
- Mutating algorithms
- Sorting algorithms
- Sorted-range algorithms
- Numeric algorithms



STL ALGORITHMS

Name	Effect
`for_each()`	Performs an operation for each element
`copy()`	Copies a range starting with the first element
`copy_if()`	Copies elements that match a criterion (since C++11)
`copy_n()`	Copies *n* elements (since C++11)
`copy_backward()`	Copies a range starting with the last element
`move()`	Moves elements of a range starting with the first element (since C++11)
`move_backward()`	Moves elements of a range starting with the last element (since C++11)
`transform()`	Modifies (and copies) elements; combines elements of two ranges
`merge()`	Merges two ranges
`swap_ranges()`	Swaps elements of two ranges
`fill()`	Replaces each element with a given value
`fill_n()`	Replaces *n* elements with a given value
`generate()`	Replaces each element with the result of an operation
`generate_n()`	Replaces *n* elements with the result of an operation
`iota()`	Replaces each element with a sequence of incremented values (since C++11)
`replace()`	Replaces elements that have a special value with another value
`replace_if()`	Replaces elements that match a criterion with another value
`replace_copy()`	Replaces elements that have a special value while copying the whole range
`replace_copy_if()`	Replaces elements that match a criterion while copying the whole range

Search for	String Function	STL Algorithm
First occurrence of one element	`find()`	`find()`
Last occurrence of one element	`rfind()`	`find()` with reverse iterators
First occurrence of a subrange	`find()`	`search()`
Last occurrence of a subrange	`rfind()`	`find_end()`
First occurrence of several elements	`find_first_of()`	`find_first_of()`
Last occurrence of several elements	`find_last_of()`	`find_first_of()` with reverse iterators
First occurrence of *n* consecutive elements		`search_n()`

Name	Effect
`remove()`	Removes elements with a given value
`remove_if()`	Removes elements that match a given criterion
`remove_copy()`	Copies elements that do not match a given value
`remove_copy_if()`	Copies elements that do not match a given criterion
`unique()`	Removes adjacent duplicates (elements that are equal to their predecessor)
`unique_copy()`	Copies elements while removing adjacent duplicates

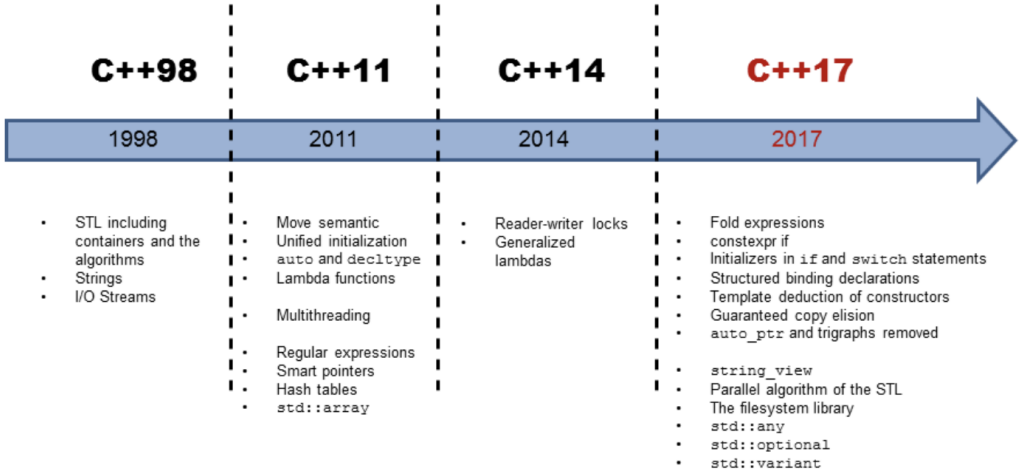
Name	Effect
`reverse()`	Reverses the order of the elements
`reverse_copy()`	Copies the elements while reversing their order
`rotate()`	Rotates the order of the elements
`rotate_copy()`	Copies the elements while rotating their order
`next_permutation()`	Permutates the order of the elements
`prev_permutation()`	Permutates the order of the elements
`shuffle()`	Brings the elements into a random order (since C++11)
`random_shuffle()`	Brings the elements into a random order
`partition()`	Changes the order of the elements so that elements that match a criterion are at the front
`stable_partition()`	Same as `partition()` but preserves the relative order of matching and nonmatching elements
`partition_copy()`	Copies the elements while changing the order so that elements that match a criterion are at the front

Name	Effect
`for_each()`	Performs an operation for each element
`count()`	Returns the number of elements
`count_if()`	Returns the number of elements that match a criterion
`min_element()`	Returns the element with the smallest value
`max_element()`	Returns the element with the largest value
`minmax_element()`	Returns the elements with the smallest and largest value (since C++11)
`find()`	Searches for the first element with the passed value
`find_if()`	Searches for the first element that matches a criterion
`find_if_not()`	Searches for the first element that matches a criterion not (since C++11)
`search_n()`	Searches for the first *n* consecutive elements with certain properties
`search()`	Searches for the first occurrence of a subrange
`find_end()`	Searches for the last occurrence of a subrange
`find_first_of()`	Searches the first of several possible elements
`adjacent_find()`	Searches for two adjacent elements that are equal (by some criterion)
`equal()`	Returns whether two ranges are equal
`is_permutation()`	Returns whether two unordered ranges contain equal elements (since C++11)
`mismatch()`	Returns the first elements of two sequences that differ
`lexicographical_compare()`	Returns whether a range is lexicographically less than another range
`is_sorted()`	Returns whether the elements in a range are sorted (since C++11)
`is_sorted_until()`	Returns the first unsorted element in a range (since C++11)
`is_partitioned()`	Returns whether the elements in a range are partitioned in two groups according to a criterion (since C++11)
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)
`is_heap()`	Returns whether the elements in a range are sorted as a heap (since C++11)
`is_heap_until()`	Returns the first element in a range not sorted as a heap (since C++11)
`all_of()`	Returns whether all elements match a criterion (since C++11)
`any_of()`	Returns whether at least one element matches a criterion (since C++11)
`none_of()`	Returns whether none of the elements matches a criterion (since C++11)

STL ALGORITHMS

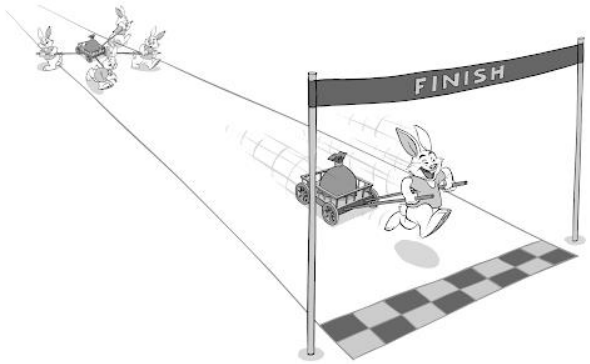
Name	Effect
`sort()`	Sorts all elements
`stable_sort()`	Sorts while preserving order of equal elements
`partial_sort()`	Sorts until the first *n* elements are correct
`partial_sort_copy()`	Copies elements in sorted order
`nth_element()`	Sorts according to the *n*th position
`partition()`	Changes the order of the elements so that elements that match a criterion are at the beginning
`stable_partition()`	Same as `partition()` but preserves the relative order of matching and nonmatching elements
`partition_copy()`	Copies the elements while changing the order so that elements that match a criterion are at the beginning
`make_heap()`	Converts a range into a heap
`push_heap()`	Adds an element to a heap
`pop_heap()`	Removes an element from a heap
`sort_heap()`	Sorts the heap (it is no longer a heap after the call)

Name	Effect
`is_sorted()`	Returns whether the elements in a range are sorted (since C++11)
`is_sorted_until()`	Returns the first unsorted element in a range (since C++11)
`is_partitioned()`	Returns whether the elements in a range are partitioned in two groups according to a criterion (since C++11)
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)
`is_heap()`	Returns whether the elements in a range are sorted as a heap (since C++11)
`is_heap_until()`	Returns the first element in a range not sorted as a heap (since C++11)



Name	Effect
`binary_search()`	Returns whether the range contains an element
`includes()`	Returns whether each element of a range is also an element of another range
`lower_bound()`	Finds the first element greater than or equal to a given value
`upper_bound()`	Finds the first element greater than a given value
`equal_range()`	Returns the range of elements equal to a given value
`merge()`	Merges the elements of two ranges
`set_union()`	Processes the sorted union of two ranges
`set_intersection()`	Processes the sorted intersection of two ranges
`set_difference()`	Processes a sorted range that contains all elements of a range that are not part of another range
`set_symmetric_difference()`	Processes a sorted range that contains all elements that are in exactly one of two ranges
`inplace_merge()`	Merges two consecutive sorted ranges
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)

Name	Effect
`accumulate()`	Combines all element values (processes sum, product, and so forth)
`inner_product()`	Combines all elements of two ranges
`adjacent_difference()`	Combines each element with its predecessor
`partial_sum()`	Combines each element with all its predecessors



STL ALGORITHMS

- Find an algorithm call you wish to optimize with parallelism in your program. Good candidates are algorithms which do more than O(n) work like sort, and show up as taking reasonable amounts of time when profiling your application.
- Verify that code you supply to the algorithm is safe to parallelize.
- Choose a parallel execution policy. (Execution policies are described below.)
- If you aren't already, **#include <execution>** to make the parallel execution policies available.
- Add one of the execution policies as the first parameter to the algorithm call to parallelize.
- Benchmark the result to ensure the parallel version is an improvement. Parallelizing is not always faster, particularly for non-random-access iterators, or when the input size is small, or when the additional parallelism creates contention on external resources like a disk.

Standard library algorithms for which parallelized versions are provided [Collapse]

- std::adjacent_difference
- std::adjacent_find
- std::all_of
- std::any_of
- std::copy
- std::copy_if
- std::copy_n
- std::count
- std::count_if
- std::equal
- std::fill
- std::fill_n
- std::find
- std::find_end
- std::find_first_of
- std::find_if
- std::find_if_not
- std::generate
- std::generate_n
- std::includes
- std::inner_product
- std::inplace_merge
- std::is_heap
- std::is_heap_until
- std::is_partitioned
- std::is_sorted
- std::is_sorted_until
- std::lexicographical_compare
- std::max_element
- std::merge
- std::min_element
- std::minmax_element
- std::mismatch
- std::move
- std::none_of
- std::nth_element
- std::partial_sort
- std::partial_sort_copy
- std::partition
- std::partition_copy
- std::remove
- std::remove_copy
- std::remove_copy_if
- std::remove_if
- std::replace
- std::replace_copy
- std::replace_copy_if
- std::replace_if
- std::reverse
- std::reverse_copy
- std::rotate
- std::rotate_copy
- std::search
- std::search_n
- std::set_difference
- std::set_intersection
- std::set_symmetric_difference
- std::set_union
- std::sort
- std::stable_partition
- std::stable_sort
- std::swap_ranges
- std::transform
- std::uninitialized_copy
- std::uninitialized_copy_n
- std::uninitialized_fill
- std::uninitialized_fill_n
- std::unique
- std::unique_copy

C++
DEMOS

1. Brief of STL Algorithms
2. STL Parallel Algorithms

SWC > stl01.cpp > main()

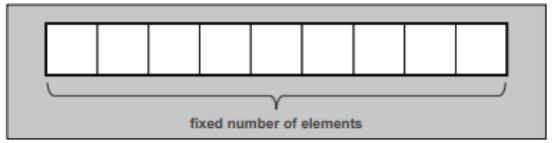
```cpp
#include <iostream>
#include <vector>
#include <forward_list>

using namespace std;

int main(){

    vector<int> a;
    forward_list<int> b;

    for (int i=0;i<=10;i++){
        a.push_back(2*i);
        b.push_front(3*i);
    }

    // use iterator and auto type
    for (auto elem:a) cout << elem << "-";
    cout << endl;
    for (auto elem:b) cout << elem << "++";


    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
rismanadnan@Rismans-iMac SWC % c++ -std=c++11 -o stl01 stl01.cpp
rismanadnan@Rismans-iMac SWC % ./stl01
0-2-4-6-8-10-12-14-16-18-20-
30++27++24++21++18++15++12++9++6++3++0++%
rismanadnan@Rismans-iMac SWC %
```

#include<array>: An ordered collection with random access. It is a sequence of elements with constant size.

Initialization:



- Default initialized
  - array<int,4> x; //weks
  - array<int,4> x {}; //OK
  - array<int,4> x {0,1,2,3}; //OK
  - array<int,4> x = {1}; //OK
  - array<int,4> x = {1,2,3,4}; //OK
- Swap Initialization
  - swap(x,y); //OK
- Move Initialization
  - y = move(y); //OK

Constructors:

Operation	Effect
`array<Elem,N> c`	Default constructor; creates an array with default-initialized elements
`array<Elem,N> c(c2)`	Copy constructor; creates a copy of another array of the same type (all elements are copied)
`array<Elem,N> c = c2`	Copy constructor; creates a copy of another array of the same type (all elements are copied)
`array<Elem,N> c(rv)`	Move constructor; creates a new array taking the contents of the rvalue *rv* (since C++11)
`array<Elem,N> c = rv`	Move constructor; creates a new array, taking the contents of the rvalue *rv* (since C++11)
`array<Elem,N> c = initlist`	Creates an array initialized with the elements of the initializer list

Basic Operations:

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Operation	Effect
`c = c2`	Assigns all elements of *c2* to *c*
`c = rv`	Move assigns all elements of the rvalue *rv* to *c* (since C++11)
`c.fill(val)`	Assigns *val* to each element in array *c*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

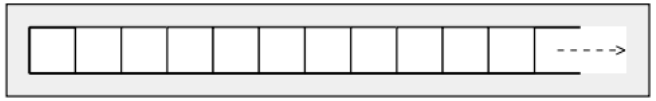
Access Operations:

Operation	Effect
`c[idx]`	Returns the element with index *idx* (*no* range checking)
`c.at(idx)`	Returns the element with index *idx* (throws range-error exception if *idx* is out of range)
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)

Operation	Effect
`c.begin()`	Returns a random-access iterator for the first element
`c.end()`	Returns a random-access iterator for the position after the last element
`c.cbegin()`	Returns a constant random-access iterator for the first element (since C++11)
`c.cend()`	Returns a constant random-access iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

#include<vector>: An ordered collection with random access. It is a dynamic array. Append and delete element at the end.

Initialization:



- Default initialized
  - vector<int> x; //OK
  - vector<int> x(5); //OK
  - vector<int> x {0,1,2,3}; //OK
  - vector<int> x = {1,2,3,4}; //OK
- Reserve Memory
  - vector<int> x; //OK
  - v.reserve(80); // reserve memory for 80 elements
  - v.shrink_to_fit(); // shrink memory (C++11)

Constructors:

Operation	Effect
`vector<Elem> c`	Default constructor; creates an empty vector without any elements
`vector<Elem> c(c2)`	Copy constructor; creates a new vector as a copy of *c2* (all elements are copied)
`vector<Elem> c = c2`	Copy constructor; creates a new vector as a copy of *c2* (all elements are copied)
`vector<Elem> c(rv)`	Move constructor; creates a new vector, taking the contents of the rvalue *rv* (since C++11)
`vector<Elem> c = rv`	Move constructor; creates a new vector, taking the contents of the rvalue *rv* (since C++11)
`vector<Elem> c(n)`	Creates a vector with *n* elements created by the default constructor
`vector<Elem> c(n,elem)`	Creates a vector initialized with *n* copies of element *elem*
`vector<Elem> c(beg,end)`	Creates a vector initialized with the elements of the range [*beg*,*end*)
`vector<Elem> c(initlist)`	Creates a vector initialized with the elements of initializer list *initlist* (since C++11)
`vector<Elem> c = initlist`	Creates a vector initialized with the elements of initializer list *initlist* (since C++11)
`c.~vector()`	Destroys all elements and frees the memory

Basic Operations:

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c.capacity()`	Returns the maximum possible number of elements without reallocation
`c.reserve(num)`	Enlarges capacity, if not enough yet[6]
`c.shrink_to_fit()`	Request to reduce capacity to fit number of elements (since C++11)[6]
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Operation	Effect
`c = c2`	Assigns all elements of *c2* to *c*
`c = rv`	Move assigns all elements of the rvalue *rv* to *c* (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to *c* (since C++11)
`c.assign(n,elem)`	Assigns *n* copies of element *elem*
`c.assign(beg,end)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(initlist)`	Assigns all the elements of the initializer list *initlist*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

Operation	Effect
`c.push_back(elem)`	Appends a copy of *elem* at the end
`c.pop_back()`	Removes the last element (does not return it)
`c.insert(pos,elem)`	Inserts a copy of *elem* before iterator position *pos* and returns the position of the new element
`c.insert(pos,n,elem)`	Inserts *n* copies of *elem* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert(pos,beg,end)`	Inserts a copy of all elements of the range [*beg*,*end*) before iterator position *pos* and returns the position of the first new element (or pos if there is no new element)
`c.insert(pos,initlist)`	Inserts a copy of all elements of the initializer list *initlist* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element; since C++11)
`c.emplace(pos,args...)`	Inserts a copy of an element initialized with *args* before iterator position *pos* and returns the position of the new element (since C++11)
`c.emplace_back(args...)`	Appends a copy of an element initialized with *args* at the end (returns nothing; since C++11)
`c.erase(pos)`	Removes the element at iterator position *pos* and returns the position of the next element
`c.erase(beg,end)`	Removes all elements of the range [*beg*,*end*) and returns the position of the next element
`c.resize(num)`	Changes the number of elements to *num* (if `size()` grows new elements are created by their default constructor)
`c.resize(num,elem)`	Changes the number of elements to *num* (if `size()` grows new elements are copies of *elem*)
`c.clear()`	Removes all elements (empties the container)

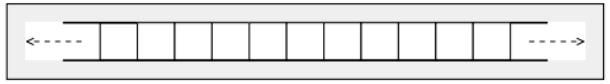
Access Operations:

Operation	Effect
`c[idx]`	Returns the element with index *idx* (*no* range checking)
`c.at(idx)`	Returns the element with index *idx* (throws range-error exception if *idx* is out of range)
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)

Operation	Effect
`c.begin()`	Returns a random-access iterator for the first element
`c.end()`	Returns a random-access iterator for the position after the last element
`c.cbegin()`	Returns a constant random-access iterator for the first element (since C++11)
`c.cend()`	Returns a constant random-access iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

#include<deque>: An ordered collection with dynamic array (vector-like)) and random access. Deque is open at both sides.

Initialization:

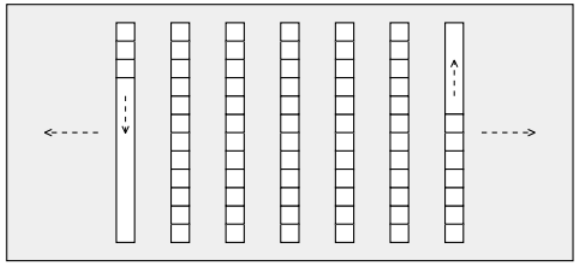


- Default initialized
  - deque<int> x; //OK
  - deque<int> x(5); //OK
  - deque<int> x {0,1,2,3}; //OK
  - deque<int> x = {1}; //OK
  - deque<int> x = {1,2,3,4}; //OK
- Inserting and removing elements is fast at both the beginning and the end (for vectors, it is fast only at the end). Deque element access and iterator movement are usually a bit slower.

Constructors:

Operation	Effect
`deque<Elem> c`	Default constructor; creates an empty deque without any elements
`deque<Elem> c(c2)`	Copy constructor; creates a new deque as a copy of *c2* (all elements are copied)
`deque<Elem> c = c2`	Copy constructor; creates a new deque as a copy of *c2* (all elements are copied)
`deque<Elem> c(rv)`	Move constructor; creates a new deque, taking the contents of the rvalue *rv* (since C++11)
`deque<Elem> c = rv`	Move constructor; creates a new deque, taking the contents of the rvalue *rv* (since C++11)
`deque<Elem> c(n)`	Creates a deque with *n* elements created by the default constructor
`deque<Elem> c(n,elem)`	Creates a deque initialized with *n* copies of element *elem*
`deque<Elem> c(beg,end)`	Creates a deque initialized with the elements of the range [*beg*,*end*)
`deque<Elem> c(initlist)`	Creates a deque initialized with the elements of initializer list *initlist* (since C++11)
`deque<Elem> c = initlist`	Creates a deque initialized with the elements of initializer list *initlist* (since C++11)
`c.~deque()`	Destroys all elements and frees the memory

Operations:

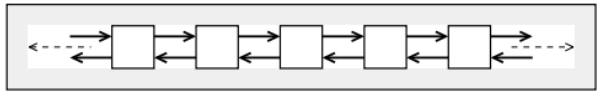


Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c.shrink_to_fit()`	Request to reduce capacity to fit number of elements (since C++11)[8]
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)
`c[idx]`	Returns the element with index *idx* (*no* range checking)
`c.at(idx)`	Returns the element with index *idx* (throws range-error exception if *idx* is out of range)
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)
`c.begin()`	Returns a random-access iterator for the first element
`c.end()`	Returns a random-access iterator for the position after the last element
`c.cbegin()`	Returns a constant random-access iterator for the first element (since C++11)
`c.cend()`	Returns a constant random-access iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

Operation	Effect
`c = c2`	Assigns all elements of *c2* to `c`
`c = rv`	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
`c.assign(n,elem)`	Assigns *n* copies of element *elem*
`c.assign(beg,end)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(initlist)`	Assigns all the elements of the initializer list *initlist*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*
`c.push_back(elem)`	Appends a copy of *elem* at the end
`c.pop_back()`	Removes the last element (does not return it)
`c.push_front(elem)`	Inserts a copy of *elem* at the beginning
`c.pop_front()`	Removes the first element (does not return it)
`c.insert(pos,elem)`	Inserts a copy of *elem* before iterator position *pos* and returns the position of the new element
`c.insert(pos,n,elem)`	Inserts *n* copies of *elem* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert(pos,beg,end)`	Inserts a copy of all elements of the range [*beg*,*end*) before iterator position *pos* and returns the position of the first new element (or `pos` if there is no new element)
`c.insert(pos,initlist)`	Inserts a copy of all elements of the initializer list *initlist* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element; since C++11)
`c.emplace(pos,args...)`	Inserts a copy of an element initialized with *args* before iterator position *pos* and returns the position of the new element (since C++11)
`c.emplace_back(args...)`	Appends a copy of an element initialized with *args* at the end (returns nothing; since C++11)
`c.emplace_front(args...)`	Inserts a copy of an element initialized with *args* at the beginning (returns nothing; since C++11)
`c.erase(pos)`	Removes the element at iterator position *pos* and returns the position of the next element
`c.erase(beg,end)`	Removes all elements of the range [*beg*,*end*) and returns the position of the next element
`c.resize(num)`	Changes the number of elements to *num* (if `size()` grows new elements are created by their default constructor)
`c.resize(num,elem)`	Changes the number of elements to *num* (if `size()` grows new elements are copies of *elem*)
`c.clear()`	Removes all elements (empties the container)

#include<list>: An ordered container as STL implementation of double linked list. It has two pointers as anchors at first and end.

Initialization:

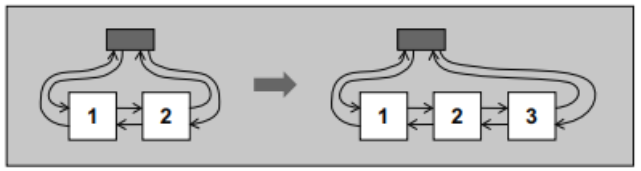


- Default initialized
  - list<int> x; //OK
  - list<int> x(5); //OK
  - list<int> x {0,1,2,3}; //OK
  - list<int> x = {1}; //OK
  - list<int> x = {1,2,3,4}; //OK
- List has no index hence not supporting random access
- Insert and remove elements are fast at both anchors.
- Support exception handling

Constructors:

Operation	Effect
`list<Elem> c`	Default constructor; creates an empty list without any elements
`list<Elem> c(c2)`	Copy constructor; creates a new list as a copy of *c2* (all elements are copied)
`list<Elem> c = c2`	Copy constructor; creates a new list as a copy of *c2* (all elements are copied)
`list<Elem> c(rv)`	Move constructor; creates a new list, taking the contents of the rvalue *rv* (since C++11)
`list<Elem> c = rv`	Move constructor; creates a new list, taking the contents of the rvalue *rv* (since C++11)
`list<Elem> c(n)`	Creates a list with *n* elements created by the default constructor
`list<Elem> c(n,elem)`	Creates a list initialized with *n* copies of element *elem*
`list<Elem> c(beg,end)`	Creates a list initialized with the elements of the range [*beg*,*end*)
`list<Elem> c(initlist)`	Creates a list initialized with the elements of initializer list *initlist* (since C++11)
`list<Elem> c = initlist`	Creates a list initialized with the elements of initializer list *initlist* (since C++11)
`c.~list()`	Destroys all elements and frees the memory

Non Modifying Operations:

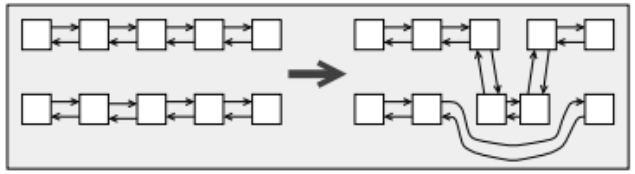


Operation	Effect
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Modifying Operations:

Operation	Effect
`c.push_back(elem)`	Appends a copy of *elem* at the end
`c.pop_back()`	Removes the last element (does not return it)
`c.push_front(elem)`	Inserts a copy of *elem* at the beginning
`c.pop_front()`	Removes the first element (does not return it)
`c.insert(pos,elem)`	Inserts a copy of *elem* before iterator position *pos* and returns the position of the new element
`c.insert(pos,n,elem)`	Inserts *n* copies of *elem* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert(pos,beg,end)`	Inserts a copy of all elements of the range [*beg*,*end*) before iterator position *pos* and returns the position of the first new element (or `pos` if there is no new element)
`c.insert(pos,initlist)`	Inserts a copy of all elements of the initializer list *initlist* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element; since C++11)
`c.emplace(pos,args...)`	Inserts a copy of an element initialized with *args* before iterator position *pos* and returns the position of the new element (since C++11)
`c.emplace_back(args...)`	Appends a copy of an element initialized with *args* at the end (returns nothing; since C++11)
`c.emplace_front(args...)`	Inserts a copy of an element initialized with *args* at the beginning (returns nothing; since C++11)
`c.erase(pos)`	Removes the element at iterator position *pos* and returns the position of the next element
`c.erase(beg,end)`	Removes all elements of the range [*beg*,*end*) and returns the position of the next element
`c.remove(val)`	Removes all elements with value `val`
`c.remove_if(op)`	Removes all elements for which *op*`(elem)` yields `true`
`c.resize(num)`	Changes the number of elements to *num* (if `size()` grows new elements are created by their default constructor)
`c.resize(num,elem)`	Changes the number of elements to *num* (if `size()` grows new elements are copies of *elem*)
`c.clear()`	Removes all elements (empties the container)

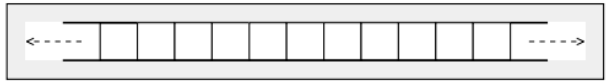


Operation	Effect
`c = c2`	Assigns all elements of *c2* to *c*
`c = rv`	Move assigns all elements of the rvalue *rv* to *c* (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to *c* (since C++11)
`c.assign(n,elem)`	Assigns *n* copies of element *elem*
`c.assign(beg,end)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(initlist)`	Assigns all the elements of the initializer list *initlist*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

Operation	Effect
`c.begin()`	Returns a bidirectional iterator for the first element
`c.end()`	Returns a bidirectional iterator for the position after the last element
`c.cbegin()`	Returns a constant bidirectional iterator for the first element (since C++11)
`c.cend()`	Returns a constant bidirectional iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

#include<forward_list>: An ordered container as STL implementation of single linked list. It only has one pointer as anchor.

Initialization:

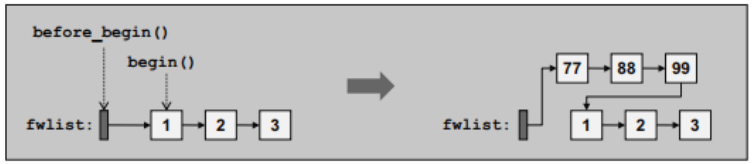


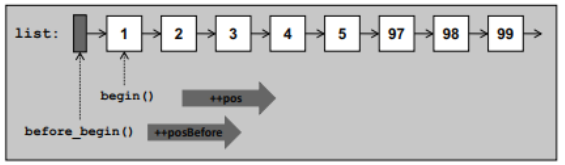
- Default initialized
  - forward_list<int> x; //OK
  - forward_list<int> x(5); //OK
  - forward_list<int> x {0,1,2,3}; //OK
  - forward_list<int> x = {1}; //OK
  - forward_list<int> x = {1,2,3,4}; //OK
- No random access provided
- Only provide forward iterator, not bidirectional
- The anchor has no pointer in last element
- Insert and remove element is fast if you are there

Constructors:

Operation	Effect
`forward list<Elem> c`	Default constructor; creates an empty forward list without any elements
`forward list<Elem> c(c2)`	Copy constructor; creates a new forward list as a copy of *c2* (all elements are copied)
`forward list<Elem> c = c2`	Copy constructor; creates a new forward list as a copy of *c2* (all elements are copied)
`forward list<Elem> c(rv)`	Move constructor; creates a new forward list, taking the contents of the rvalue *rv* (since C++11)
`forward list<Elem> c = rv`	Move constructor; creates a new forward list, taking the contents of the rvalue *rv* (since C++11)
`forward list<Elem> c(n)`	Creates a forward list with *n* elements created by the default constructor
`forward list<Elem> c(n,elem)`	Creates a forward list initialized with *n* copies of element *elem*
`forward list<Elem> c(beg,end)`	Creates a forward list initialized with the elements of the range [*beg*,*end*)
`forward list<Elem> c(initlist)`	Creates a forward list initialized with the elements of initializer list *initlist* (since C++11)
`forward list<Elem> c = initlist`	Creates a forward list initialized with the elements of initializer list *initlist* (since C++11)
`c.~forward list()`	Destroys all elements and frees the memory

Operations:





Operation	Effect
`c.empty()`	Returns whether the container is empty
`c.max_size()`	Returns the maximum number of elements possible
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Operation	Effect
`c = c2`	Assigns all elements of *c2* to `c`
`c = rv`	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
`c.assign(n,elem)`	Assigns *n* copies of element *elem*
`c.assign(beg,end)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(initlist)`	Assigns all the elements of the initializer list *initlist*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

Operation	Effect
`c.begin()`	Returns a bidirectional iterator for the first element
`c.end()`	Returns a bidirectional iterator for the position after the last element
`c.cbegin()`	Returns a constant bidirectional iterator for the first element (since C++11)
`c.cend()`	Returns a constant bidirectional iterator for the position after the last element (since C++11)
`c.before_begin()`	Returns a forward iterator for the position before the first element
`c.cbefore_begin()`	Returns a constant forward iterator for the position before the first element

Operation	Effect
`c.front()`	Returns the first element (*no* check whether a first element exists)

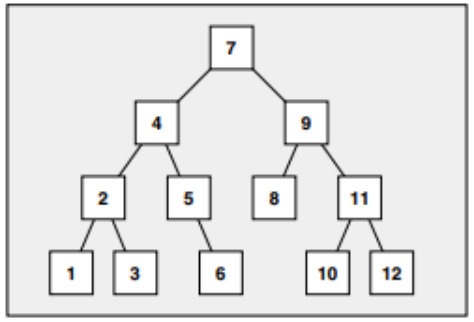
Operation	Effect
`c.unique()`	Removes duplicates of consecutive elements with the same value
`c.unique(op)`	Removes duplicates of consecutive elements, for which *op*() yields `true`
`c.splice_after(pos,c2)`	Moves all elements of *c2* to `c` right behind the iterator position *pos*
`c.splice_after(pos,c2,c2pos)`	Moves the element behind *c2pos* in *c2* right after *pos* of forward list `c` (`c` and *c2* may be identical)
`c.splice_after(pos,c2, c2beg,c2end)`	Moves all elements between *c2beg* and *c2end* (both not included) in *c2* right after *pos* of forward list `c` (`c` and *c2* may be identical)
`c.sort()`	Sorts all elements with operator `<`
`c.sort(op)`	Sorts all elements with *op*()
`c.merge(c2)`	Assuming that both containers contain the elements sorted, moves all elements of *c2* into `c` so that all elements are merged and still sorted
`c.merge(c2,op)`	Assuming that both containers contain the elements sorted by the sorting criterion *op*(), moves all elements of *c2* into `c` so that all elements are merged and still sorted according to *op*()
`c.reverse()`	Reverses the order of all elements

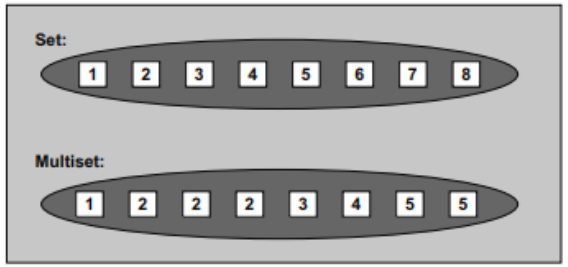
Operation	Effect
`c.push_front(elem)`	Inserts a copy of *elem* at the beginning
`c.pop_front()`	Removes the first element (does not return it)
`c.insert_after(pos,elem)`	Inserts a copy of *elem* after iterator position *pos* and returns the position of the new element
`c.insert_after(pos,n,elem)`	Inserts *n* copies of *elem* after iterator position *pos* and returns the position of the first new element (or `pos` if there is no new element)
`c.insert_after(pos,beg,end)`	Inserts a copy of all elements of the range [*beg*,*end*) after iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert_after(pos,initlist)`	Inserts a copy of all elements of the initializer list *initlist* after iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.emplace_after(pos,args...)`	Inserts a copy of an element initialized with *args* after iterator position *pos* and returns the position of the new element (since C++11)
`c.emplace_front(args...)`	Inserts a copy of an element initialized with *args* at the beginning (returns nothing; since C++11)
`c.erase_after(pos)`	Removes the element after iterator position *pos* (returns nothing)
`c.erase_after(beg,end)`	Removes all elements of the range [*beg*,*end*) (returns nothing)
`c.remove(val)`	Removes all elements with value *val*
`c.remove_if(op)`	Removes all elements for which *op*(*elem*) yields `true`
`c.resize(num)`	Changes the number of elements to *num* (if `size()` grows new elements are created by their default constructor)
`c.resize(num,elem)`	Changes the number of elements to *num* (if `size()` grows new elements are copies of *elem*)
`c.clear()`	Removes all elements (empties the container)

#include<set> or #include<multiset>: An sorted set implementation in STL. Multiset allows duplicate elements.

Initialization:

- Sorted set
- Input sorting criteria
- Binary tree
- No direct access






```
namespace std {
    template <typename T,
              typename Compare = less<T>,
              typename Allocator = allocator<T> >
             class set;

    template <typename T,
              typename Compare = less<T>,
              typename Allocator = allocator<T> >
             class multiset;
}
```

Constructors:

set	Effect
`set<Elem>`	A set that by default sorts with `less<>` (operator <)
`set<Elem,Op>`	A set that by default sorts with `Op`
`multiset<Elem>`	A multiset that by default sorts with `less<>` (operator <)
`multiset<Elem,Op>`	A multiset that by default sorts with *Op*

Operation	Effect
set `c`	Default constructor; creates an empty set/multiset without any elements
set `c(op)`	Creates an empty set/multiset that uses *op* as the sorting criterion
set `c(c2)`	Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied)
set `c = c2`	Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied)
set `c(rv)`	Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue *rv* (since C++11)
set `c = rv`	Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue *rv* (since C++11)
set `c(beg,end)`	Creates a set/multiset initialized by the elements of the range [*beg*,*end*)
set `c(beg,end,op)`	Creates a set/multiset with the sorting criterion *op* initialized by the elements of the range [*beg*,*end*)
set `c(initlist)`	Creates a set/multiset initialized with the elements of initializer list *initlist* (since C++11)
set `c = initlist`	Creates a set/multiset initialized with the elements of initializer list *initlist* (since C++11)
`c.~set()`	Destroys all elements and frees the memory

Operations:

Operation	Effect
`c.key_comp()`	Returns the comparison criterion
`c.value_comp()`	Returns the comparison criterion for values as a whole (same as `key_comp()`)
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Operation	Effect
`c.begin()`	Returns a bidirectional iterator for the first element
`c.end()`	Returns a bidirectional iterator for the position after the last element
`c.cbegin()`	Returns a constant bidirectional iterator for the first element (since C++11)
`c.cend()`	Returns a constant bidirectional iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

Operation	Effect
`c = c2`	Assigns all elements of *c2* to *c*
`c = rv`	Move assigns all elements of the rvalue *rv* to *c* (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to *c* (since C++11)
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

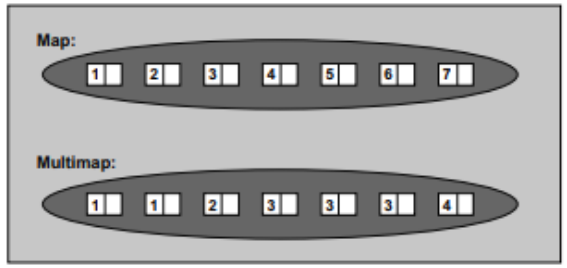
Operation	Effect
`c.count(val)`	Returns the number of elements with value *val*
`c.find(val)`	Returns the position of the first element with value *val* (or `end()` if none found)
`c.lower_bound(val)`	Returns the first position, where *val* would get inserted (the first element >= *val*)
`c.upper_bound(val)`	Returns the last position, where *val* would get inserted (the first element > *val*)
`c.equal_range(val)`	Returns a range with all elements with a value equal to *val* (i.e., the first and last position, where *val* would get inserted)

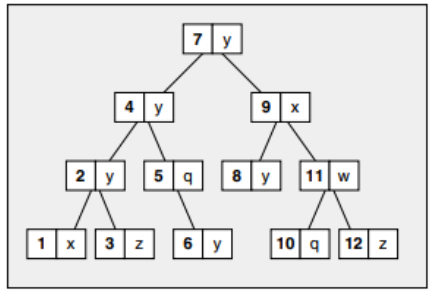
Operation	Effect
`c.insert(val)`	Inserts a copy of *val* and returns the position of the new element and, for sets, whether it succeeded
`c.insert(pos,val)`	Inserts a copy of *val* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
`c.insert(beg,end)`	Inserts a copy of all elements of the range [*beg*,*end*) (returns nothing)
`c.insert(initlist)`	Inserts a copy of all elements in the initializer list *initlist* (returns nothing; since C++11)
`c.emplace(args...)`	Inserts a copy of an element initialized with *args* and returns the position of the new element and, for sets, whether it succeeded (since C++11)
`c.emplace_hint(pos,args...)`	Inserts a copy of an element initialized with *args* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
`c.erase(val)`	Removes all elements equal to *val* and returns the number of removed elements
`c.erase(pos)`	Removes the element at iterator position *pos* and returns the following position (returned nothing before C++11)
`c.erase(beg,end)`	Removes all elements of the range [*beg*,*end*) and returns the following position (returned nothing before C++11)
`c.clear()`	Removes all elements (empties the container)

#include<map> or #include<multimap>: An sorted key-value implementation in STL. Multiset allows duplicate elements.

Initialization:

- Sorted map
- Associated array
- Balanced binary tree
- No direct access






```
namespace std {
    template <typename Key, typename T,
              typename Compare = less<Key>,
              typename Allocator = allocator<pair<const Key,T> > >
             class map;

    template <typename Key, typename T,
              typename Compare = less<Key>,
              typename Allocator = allocator<pair<const Key,T> > >
             class multimap;
}
```

Constructors:

Operation	Effect
map c	Default constructor; creates an empty map/multimap without any elements
map c(*op*)	Creates an empty map/multimap that uses *op* as the sorting criterion
map c(*c2*)	Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)
map c = *c2*	Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)
map c(*rv*)	Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue *rv* (since C++11)
map c = *rv*	Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue *rv* (since C++11)
map c(*beg*,*end*)	Creates a map/multimap initialized by the elements of the range [*beg*,*end*)
map c(*beg*,*end*,*op*)	Creates a map/multimap with the sorting criterion *op* initialized by the elements of the range [*beg*,*end*)
map c(*initlist*)	Creates a map/multimap initialized with the elements of initializer list *initlist* (since C++11)
map c = *initlist*	Creates a map/multimap initialized with the elements of initializer list *initlist* (since C++11)
c.~*map*()	Destroys all elements and frees the memory

Operations:

map	Effect
map<*Key*,*Val*>	A map that by default sorts keys with less<> (operator <)
map<*Key*,*Val*,*Op*>	A map that by default sorts keys with *Op*
multimap<*Key*,*Val*>	A multimap that by default sorts keys with less<> (operator <)
multimap<*Key*,*Val*,*Op*>	A multimap that by default sorts keys with *Op*

Operation	Effect
c.count(*val*)	Returns the number of elements with key *val*
c.find(*val*)	Returns the position of the first element with key *val* (or end() if none found)
c.lower_bound(*val*)	Returns the first position where an element with key *val* would get inserted (the first element with a key >= *val*)
c.upper_bound(*val*)	Returns the last position where an element with key *val* would get inserted (the first element with a key > *val*)
c.equal_range(*val*)	Returns a range with all elements with a key equal to *val* (i.e., the first and last positions, where an element with key *val* would get inserted)

Operation	Effect
c.begin()	Returns a bidirectional iterator for the first element
c.end()	Returns a bidirectional iterator for the position after the last element
c.cbegin()	Returns a constant bidirectional iterator for the first element (since C++11)
c.cend()	Returns a constant bidirectional iterator for the position after the last element (since C++11)
c.rbegin()	Returns a reverse iterator for the first element of a reverse iteration
c.rend()	Returns a reverse iterator for the position after the last element of a reverse iteration
c.crbegin()	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
c.crend()	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

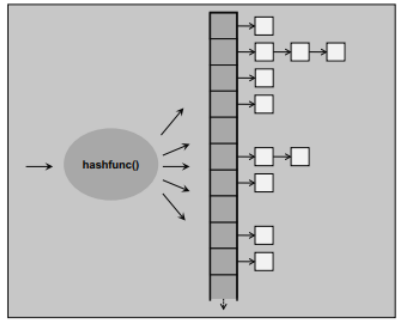
Operation	Effect
c.key_comp()	Returns the comparison criterion
c.value_comp()	Returns the comparison criterion for values as a whole (an object that compares the key in a key/value pair)
c.empty()	Returns whether the container is empty (equivalent to size()==0 but might be faster)
c.size()	Returns the current number of elements
c.max_size()	Returns the maximum number of elements possible
c1 == *c2*	Returns whether *c1* is equal to *c2* (calls == for the elements)
c1 != *c2*	Returns whether *c1* is not equal to *c2* (equivalent to !(*c1*==*c2*))
c1 < *c2*	Returns whether *c1* is less than *c2*
c1 > *c2*	Returns whether *c1* is greater than *c2* (equivalent to *c2*<*c1*)
c1 <= *c2*	Returns whether *c1* is less than or equal to *c2* (equivalent to !(*c2*<*c1*))
c1 >= *c2*	Returns whether *c1* is greater than or equal to *c2* (equivalent to !(*c1*<*c2*))

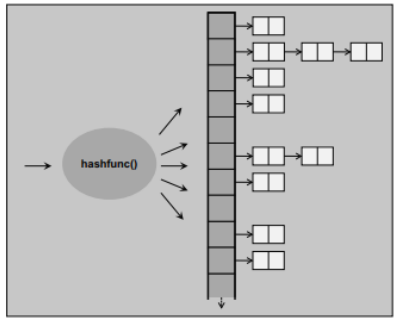
Operation	Effect
c = *c2*	Assigns all elements of *c2* to c
c = *rv*	Move assigns all elements of the rvalue *rv* to c (since C++11)
c = *initlist*	Assigns all elements of the initializer list *initlist* to c (since C++11)
c1.swap(*c2*)	Swaps the data of *c1* and *c2*
swap(*c1*,*c2*)	Swaps the data of *c1* and *c2*

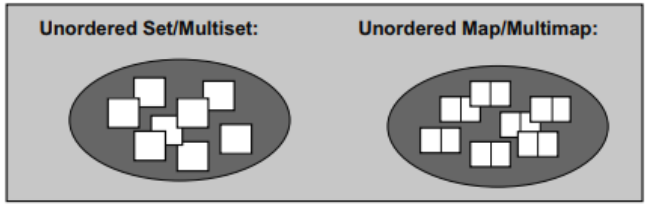
Operation	Effect
c.insert(*val*)	Inserts a copy of *val* and returns the position of the new element and, for maps, whether it succeeded
c.insert(*pos*,*val*)	Inserts a copy of *val* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
c.insert(*beg*,*end*)	Inserts a copy of all elements of the range [*beg*,*end*) (returns nothing)
c.insert(*initlist*)	Inserts a copy of all elements in the initializer list *initlist* (returns nothing; since C++11)
c.emplace(*args*...)	Inserts a copy of an element initialized with *args* and returns the position of the new element and, for maps, whether it succeeded (since C++11)
c.emplace_hint(*pos*,*args*...)	Inserts a copy of an element initialized with *args* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
c.erase(*val*)	Removes all elements equal to *val* and returns the number of removed elements
c.erase(*pos*)	Removes the element at iterator position *pos* and returns the following position (returned nothing before C++11)
c.erase(*beg*,*end*)	Removes all elements of the range [*beg*,*end*) and returns the following position (returned nothing before C++11)
c.clear()	Removes all elements (empties the container)

#include<unordered_set>; #include<unordered_map>; : STL Implementation of Hash Function (Map / Set)

Initialization:








```
namespace std {
    template <typename T,
              typename Hash = hash<T>,
              typename EqPred = equal_to<T>,
              typename Allocator = allocator<T> >
             class unordered_set;

    template <typename T,
              typename Hash = hash<T>,
              typename EqPred = equal_to<T>,
              typename Allocator = allocator<T> >
             class unordered_multiset;

    template <typename Key, typename T,
              typename Hash = hash<T>,
              typename EqPred = equal_to<T>,
              typename Allocator = allocator<pair<const Key, T> > >
             class unordered_map;

    template <typename Key, typename T,
              typename Hash = hash<T>,
              typename EqPred = equal_to<T>,
              typename Allocator = allocator<pair<const Key, T> > >
             class unordered_multimap;
}
```

Iterators:

Operation	Effect
`c.begin()`	Returns a forward iterator for the first element
`c.end()`	Returns a forward iterator for the position after the last element
`c.cbegin()`	Returns a constant forward iterator for the first element (since C++11)
`c.cend()`	Returns a constant forward iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

Operations:

Operation	Effect
Unord `c`	Default constructor; creates an empty unordered container without any elements
Unord `c(`*bnum*`)`	Creates an empty unordered container that internally uses at least *bnum* buckets
Unord `c(`*bnum*`,`*hf*`)`	Creates an empty unordered container that internally uses at least *bnum* buckets and *hf* as hash function
Unord `c(`*bnum*`,`*hf*`,`*cmp*`)`	Creates an empty unordered container that internally uses at least *bnum* buckets, *hf* as hash function, and *cmp* as predicate to identify equal values
Unord `c(`*c2*`)`	Copy constructor; creates a copy of another unordered container of the same type (all elements are copied)
Unord `c = `*c2*	Copy constructor; creates a copy of another unordered container of the same type (all elements are copied)
Unord `c(`*rv*`)`	Move constructor; creates an unordered container, taking the contents of the rvalue *rv* (since C++11)
Unord `c = `*rv*	Move constructor; creates an unordered container, taking the contents of the rvalue *rv* (since C++11)
Unord `c(`*beg*`,`*end*`)`	Creates an unordered container initialized by the elements of the range [*beg*,*end*)
Unord `c(`*beg*`,`*end*`,`*bnum*`)`	Creates an unordered container initialized by the elements of the range [*beg*,*end*) that internally uses at least `bnum` buckets
Unord `c(`*beg*`,`*end*`,`*bnum*`,`*hf*`)`	Creates an unordered container initialized by the elements of the range [*beg*,*end*) that internally uses at least *bnum* buckets and *hf* as hash function
Unord `c(`*beg*`,`*end*`,`*bnum*`,`*hf*`,`*cmp*`)`	Creates an unordered container initialized by the elements of the range [*beg*,*end*) that internally uses at least *bnum* buckets, *hf* as hash function, and *cmp* as predicate to identify equal values
Unord `c(`*initlist*`)`	Creates an unordered unordered container initialized by the elements of the initializer list *initlist*
Unord `c = `*initlist*	Creates an unordered unordered container initialized by the elements of the initializer list *initlist*
`c.~`***Unord***`()`	Destroys all elements and frees the memory

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
c1 `==` *c2*	Returns whether *c1* is equal to *c2*
c1 `!=` *c2*	Returns whether *c1* is not equal to *c2* (equivalent to `!(`*c1*`==`*c2*`)`)

Operation	Effect
`c.count(`*val*`)`	Returns the number of elements with value *val*
`c.find(`*val*`)`	Returns the position of the first element with value *val* (or `end()` if none found)
`c.equal_range(`*val*`)`	Returns a range with all elements with a value equal to *val* (i.e., the first and last positions, where `val` would get inserted)

Operation	Effect
`c =` *c2*	Assigns all elements of *c2* to `c`
`c =` *rv*	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c =` *initlist*	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
c1`.swap(`*c2*`)`	Swaps the data of *c1* and *c2*
`swap(`*c1*`,`*c2*`)`	Swaps the data of *c1* and *c2*

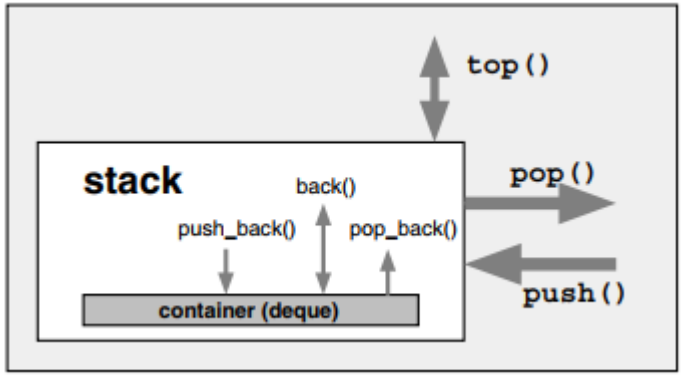
Unord	Effect
`unordered_set<`*Elem*`>`	An unordered set that by default hashes with `hash<>` and compares `equal_to<>` (operator `==`)
`unordered_set<`*Elem*`,`*Hash*`>`	An unordered set that by default hashes with *Hash* and compares `equal_to<>` (operator `==`)
`unordered_set<`*Elem*`,`*Hash*`,`*Cmp*`>`	An unordered set that by default hashes with *Hash* and compares with *Cmp*
`unordered_multiset<`*Elem*`>`	An unordered multiset that by default hashes with `hash<>` and compares `equal_to<>` (operator `==`)
`unordered_multiset<`*Elem*`,`*Hash*`>`	An unordered multiset that by default hashes with *Hash* and compares `equal_to<>` (operator `==`)
`unordered_multiset<`*Elem*`,`*Hash*`,`*Cmp*`>`	An unordered multiset that by default hashes with *Hash* and compares with *Cmp*
`unordered_map<`*Key*`,`*T*`>`	An unordered map that by default hashes with `hash<>` and compares `equal_to<>` (operator `==`)
`unordered_map<`*Key*`,`*T*`,`*Hash*`>`	An unordered map that by default hashes with *Hash* and compares `equal_to<>` (operator `==`)
`unordered_map<`*Key*`,`*T*`,`*Hash*`,`*Cmp*`>`	An unordered map that by default hashes with *Hash* and compares with *Cmp*
`unordered_multimap<`*Key*`,`*T*`>`	An unordered multimap that by default hashes with `hash<>` and compares `equal_to<>` (operator `==`)
`unordered_multimap<`*Key*`,`*T*`,`*Hash*`>`	An unordered multimap that by default hashes with *Hash* and compares `equal_to<>` (operator `==`)
`unordered_multimap<`*Key*`,`*T*`,`*Hash*`,`*Cmp*`>`	An unordered multimap that by default hashes with *Hash* and compares with *Cmp*

Operation	Effect
`c.insert(`*val*`)`	Inserts a copy of *val* and returns the position of the new element and, for unordered containers, whether it succeeded
`c.insert(`*pos*`,`*val*`)`	Inserts a copy of *val* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
`c.insert(`*beg*`,`*end*`)`	Inserts a copy of all elements of the range [*beg*,*end*) (returns nothing)
`c.insert(`*initlist*`)`	Inserts a copy of all elements in the initializer list *initlist* (returns nothing; since C++11)
`c.emplace(`*args...*`)`	Inserts a copy of an element initialized with *args* and returns the position of the new element and, for unordered containers, whether it succeeded (since C++11)
`c.emplace_hint(`*pos*`,`*args...*`)`	Inserts a copy of an element initialized with *args* and returns the position of the new element (*pos* is used as a hint pointing to where the insert should start the search)
`c.erase(`*val*`)`	Removes all elements equal to *val* and returns the number of removed elements
`c.erase(`*pos*`)`	Removes the element at iterator position *pos* and returns the following position (returned nothing before C++11)
`c.erase(`*beg*`,`*end*`)`	Removes all elements of the range [*beg*,*end*) and returns the following position (returned nothing before C++11)
`c.clear()`	Removes all elements (empties the container)

#include<stack>; #include<queue>; #include<priority_queue>; #include<bitsets>

Stack:

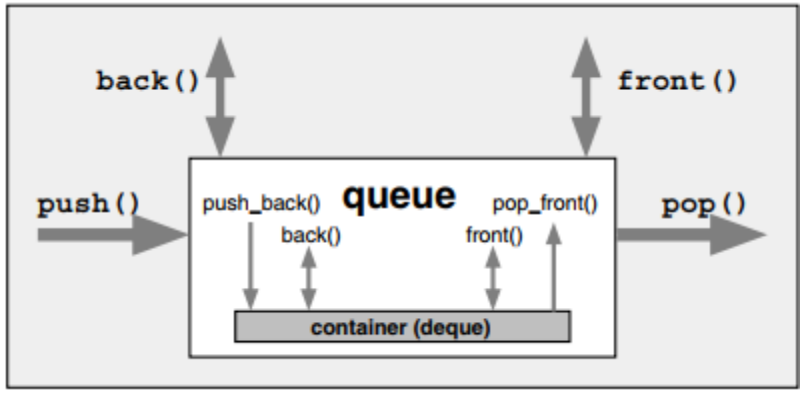
- push()
- pop()
- top()




```
namespace std {
    template <typename T,
              typename Container = deque<T>>
             class stack;
}
```

Queue:

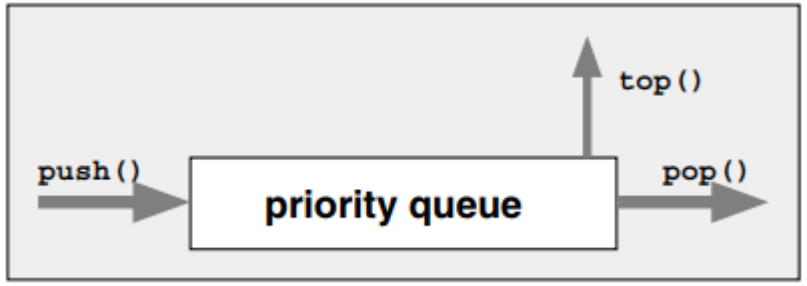
- push()
- front()
- back()
- pop()




```
namespace std {
    template <typename T,
              typename Container = deque<T>>
             class queue;
}
```

Stack:

- push()
- pop()
- top()

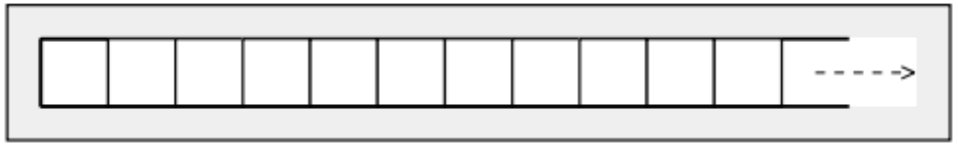



```
namespace std {
   template <typename T,
             typename Container = vector<T>,
             typename Compare = less<typename Container::value_type>>
            class priority_queue;
}
```

Bitsets:

- Array of bits
- We cant change the number
- Use vector<bool>

```
namespace std {
    template <size_t Bits>
    class bitset;
}
```



http://www.cppstdlib.com/ Get the source code samples

REFERENCES

Disclaimer: Only for Serious C++ Developers (Download Here)