Modern C++ Containers, Algorithms and Concepts

Sequence Containers (containers which can be accessed sequentially)

Type	Description	Template Parameters	Search	Iterator Type	Value Type
`array`	static contiguous array	`class T, size_t N`	`std::find()`	`RandomAccessIterator`	`T`
`vector`	dynamic contiguous array	`class T, class Allocator = std::allocator<T>`	`std::find()`	`RandomAccessIterator`	`T`
`deque`	double-ended queue	`class T, class Allocator = std::allocator<T>`	`std::find()`	`RandomAccessIterator`	`T`
`forward_list`	singly-linked list	`class T, class Allocator = std::allocator<T>`	`std::find()`	`ForwardIterator`	`T`
`list`	doubly-linked list	`class T, class Allocator = std::allocator<T>`	`std::find()`	`BidirectionalIterator`	`T`

Associative Containers (sorted containers that can be quickly searched (O(log n) complexity))

Type	Description	Template Parameters	Search	Iterator Type	Value Type
`set`	collection of unique keys	`class Key, class Compare = std::less<Key>, class Allocator = std::allocator<Key>`	`.find()`	`BidirectionalIterator`	`Key`
`map`	collection of key-value pairs, unique keys	`class Key, class T, class Compare = std::less<Key>, class Allocator = std::allocator<std::pair<const Key, T>>`	`.find()`	`BidirectionalIterator`	`pair<const Key,T>`
`multiset`	collection of keys	`class Key, class Compare = std::less<Key>, class Allocator = std::allocator<Key>`	`.find()`	`BidirectionalIterator`	`Key`
`multimap`	collection of key-value pairs	`class Key, class T, class Compare = std::less<Key>, class Allocator = std::allocator<std::pair<const Key, T>>`	`.find()`	`BidirectionalIterator`	`pair<const Key,T>`

Unordered Associative Containers (unordered associative containers (hashed) that can be quickly searched (O(1) amortized, O(n) worst-case complexity))

Type	Description	Template Parameters	Search	Iterator Type	Value Type
`unordered_set`	collection of unique keys	`class Key, class Hash = std::hash<Key>, class KeyEqual = std::equal_to<Key>, class Allocator = std::allocator<Key>`	`.find()`	`ForwardIterator`	`pair<const Key,T>`
`unordered_multiset`	collection of keys	`class Key, class Hash = std::hash<Key>, class KeyEqual = std::equal_to<Key>, class Allocator = std::allocator<std::pair<const Key, T>>`	`.find()`	`ForwardIterator`	`Key`
`unordered_multimap`	collection of key-value pairs	`class Key, class T, class Hash = std::hash<Key>, class KeyEqual = std::equal_to<Key>, class Allocator = std::allocator<std::pair<const Key, T>>`	`.find()`	`ForwardIterator`	`pair<const Key,T>`

Iterator categories

Defined operations	ContiguousIterator	RandomAccessIterator	BidirectionalIterator	ForwardIterator	InputIterator	OutputIterator
read, increment (without multiple passes)	✓	✓	✓	✓	✓	
increment (with multiple passes)	✓	✓	✓	✓		
decrement	✓	✓	✓			
random access	✓	✓				
contiguous storage	✓					
write, increment (without multiple passes)						✓

"The standard library saves programmers from having to reinvent the wheel." – Bjarne Stroustrup

concrete

conceptual

Organization and content thanks to the C++17 standard and cppreference.com

Iterator adaptors

`reverse_iterator`	iterator adaptor for reverse-order traversal
`make_reverse_iterator` C++14	creates a `reverse_iterator` of type inferred from the argument
`move_iterator` C++11	iterator adaptor which dereferences to an rvalue reference
`make_move_iterator` C++11	creates a `move_iterator` of type inferred from the argument
`back_insert_iterator`	iterator adaptor for insertion at the end of a container
`back_inserter`	creates a `back_insert_iterator` of type inferred from the argument
`front_insert_iterator`	iterator adaptor for insertion at the front of a container
`front_inserter`	creates a `front_insert_iterator` of type inferred from the argument
`insert_iterator`	iterator adaptor for insertion into a container
`inserter`	creates a `insert_iterator` of type inferred from the argument

Non-modifying sequence operations

`all_of` C++11	checks if a predicate is true for all of the elements in a range
`any_of` C++11	checks if a predicate is true for any of the elements in a range
`none_of` C++11	checks if a predicate is true for none of the elements in a range
`for_each`	applies a function to a range of elements
`for_each_n` C++17	applies a function object to the first n elements of a sequence
`count`	returns the number of elements satisfying specific criteria
`count_if`	returns the number of elements satisfying specific criteria
`mismatch`	finds the first position where two ranges differ
`find`	finds the first element satisfying specific criteria
`find_if`	finds the first element satisfying specific criteria
`find_if_not` C++11	finds the first element satisfying specific criteria
`find_end`	finds the last sequence of elements in a certain range
`find_first_of`	searches for any one of a set of elements
`adjacent_find`	finds the first two adjacent items that are equal (or satisfy a given predicate)
`search`	searches for a range of elements
`search_n`	searches for a number of consecutive copies of an element in a range

Modifying sequence operations

`copy`	copies a range of elements to a new location
`copy_if` C++11	copies a range of elements to a new location
`copy_n` C++11	copies a number of elements to a new location
`copy_backward`	copies a range of elements in backwards order
`move` C++11	moves a range of elements to a new location
`move_backward` C++11	moves a range of elements to a new location in backwards order
`fill`	copy-assigns the given value to every element in a range
`fill_n`	copy-assigns the given value to N elements in a range
`transform`	applies a function to a range of elements
`generate`	assigns the results of successive function calls to every element in a range
`generate_n`	assigns the results of successive function calls to N elements in a range
`remove`	removes elements satisfying specific criteria
`remove_if`	removes elements satisfying specific criteria
`remove_copy`	copies a range of elements omitting those that satisfy specific criteria
`remove_copy_if`	copies a range of elements omitting those that satisfy specific criteria
`replace`	replaces all values satisfying specific criteria with another value
`replace_if`	replaces all values satisfying specific criteria with another value
`replace_copy`	copies a range, replacing elements satisfying specific criteria with another value
`replace_copy_if`	copies a range, replacing elements satisfying specific criteria with another value
`swap`	swaps the values of two objects
`swap_ranges`	swaps two ranges of elements
`iter_swap`	swaps the elements pointed to by two iterators
`reverse`	reverses the order of elements in a range
`reverse_copy`	creates a copy of a range that is reversed
`rotate`	rotates the order of elements in a range
`rotate_copy`	copies and rotate a range of elements
`random_shuffle` until C++17	randomly re-orders elements in a range
`shuffle` C++11	randomly re-orders elements in a range
`sample` C++17	selects n random elements from a sequence
`unique`	removes consecutive duplicate elements in a range
`unique_copy`	creates a copy of some range of elements that contains no consecutive duplicates

Partitioning operations

`is_partitioned` C++11	determines if the range is partitioned by the given predicate
`partition`	divides a range of elements into two groups
`partition_copy` C++11	copies a range dividing the elements into two groups
`stable_partition`	divides elements into two groups while preserving their relative order
`partition_point` C++11	locates the partition point of a partitioned range

Permutation operations

`is_permutation`	determines if a sequence is a permutation of another sequence
`next_permutation`	generates the next greater lexicographic permutation of a range of elements
`prev_permutation`	generates the next smaller lexicographic permutation of a range of elements

Sorting operations

`is_sorted` C++11	checks whether a range is sorted into ascending order
`is_sorted_until` C++11	finds the largest sorted subrange
`sort`	sorts a range into ascending order
`partial_sort`	sorts the first N elements of a range
`partial_sort_copy`	copies and partially sorts a range of elements
`stable_sort`	sorts a range of elements while preserving order between equal elements
`nth_element`	partially sort a range such that it is partitioned by the given element

Binary search operations (on sorted ranges)

`lower_bound`	returns an iterator to the first element not less than the given value
`upper_bound`	returns an iterator to the first element greater than a certain value
`binary_search`	determines if an element exists in a certain range
`equal_range`	returns range of elements matching a specific key

Set operations (on sorted ranges)

`merge`	merges two sorted ranges
`inplace_merge`	merges two ordered ranges in-place
`includes`	returns true if one set is a subset of another
`set_difference`	computes the difference between two sets
`set_intersection`	computes the intersection of two sets
`set_symmetric_difference`	computes the symmetric difference between two sets
`set_union`	computes the union of two sets

Heap operations

`is_heap` C++11	checks if the given range is a max heap
`is_heap_until` C++11	finds the largest subrange that is a max heap
`make_heap`	creates a max heap out of a range of elements
`push_heap`	adds an element to a max heap
`pop_heap`	removes the largest element from a max heap
`sort_heap`	turns a max heap into a range of elements sorted in ascending order

Minimum/maximum operations

`max` C++11	returns the greater of the given values
`max_element` C++11	returns the largest element in a range
`min`	returns the smaller of the given values
`min_element`	returns the smallest element in a range
`minmax` C++11	returns the smaller and larger of two elements
`minmax_element` C++11	returns the smallest and the largest elements in a range
`clamp` C++17	tclamps a value between a pair of boundary values

Comparison operations

`equal`	determines equality of two sets of elements
`lexicographical_compare`	returns true if one range is lexicographically less than another
`compare_3way` C++20	compares two values using 3-way comparison
`lexicographical_compare_3way` C++20	compares two ranges using 3-way comparison

Numeric operations

`iota` C++11	fills a range with successive increments of the starting value
`accumulate`	sums up a range of elements
`inner_product`	computes the inner product of two ranges of elements
`adjacent_difference`	computes the differences between adjacent elements in a range
`partial_sum`	computes the partial sum of a range of elements
`reduce` C++17	similar to `accumulate`, except out of order
`exclusive_scan` C++17	similar to `partial_sum`, excludes the ith input element from the ith sum
`inclusive_scan` C++17	similar to `partial_sum`, includes the ith input element in the ith sum
`transform_reduce` C++17	applies a functor, then reduces out of order
`transform_exclusive_scan` C++17	applies a functor, then calculates exclusive scan
`transform_inclusive_scan` C++17	applies a functor, then calculates inclusive scan

Operations on uninitialized memory

`uninitialized_copy`	copies a range of objects to an uninitialized area of memory
`uninitialized_copy_n` C++11	copies a number of objects to an uninitialized area of memory
`uninitialized_fill`	copies an object to an uninitialized area of memory, defined by a range
`uninitialized_fill_n`	copies an object to an uninitialized area of memory, defined by a start and a count
`uninitialized_move` C++17	moves a range of objects to an uninitialized area of memory
`uninitialized_move_n` C++17	moves a number of objects to an uninitialized area of memory
`uninitialized_default_construct` C++17	constructs objects by default-initialization in an uninitialized area of memory, defined by a range
`uninitialized_default_construct_n` C++17	constructs objects by default-initialization in an uninitialized area of memory, defined by a start and a count
`uninitialized_value_construct` C++17	constructs objects by value-initialization in an uninitialized area of memory, defined by a range
`uninitialized_value_construct_n` C++17	constructs objects by value-initialization in an uninitialized area of memory, defined by a start and a count
`destroy_at` C++17	destroys an object at a given address
`destroy` C++17	destroys a range of objects
`destroy_n` C++17	destroys a number of objects in a range

Smart pointers

`unique_ptr` C++11	smart pointer with unique object ownership semantics
`shared_ptr` C++11	smart pointer with shared object ownership semantics
`weak_ptr` C++11	weak reference to an object managed by `shared_ptr`
`auto_ptr` removed in C++17	smart pointer with strict object ownership semantics

Library layout concepts

`TriviallyCopyable` C++11	class with trivial copy, assignment and destructor
`TrivialType` C++11	class with trivial constructors, assignment and destructor
`StandardLayoutType` C++11	non-virtual class containing only other StandardLayout members, all with the same access control
`PODType`	POD (Plain Old Data) structure, compatible with C struct

Library basic concepts

`DefaultConstructible`	specifies that an object of the type can be default constructed
`MoveConstructible` C++11	specifies that an object of the type can be constructed from rvalue
`CopyConstructible`	specifies that an object of the type can be constructed from lvalue
`MoveAssignable` C++11	specifies that an object of the type can be assigned from rvalue
`CopyAssignable`	specifies that an object of the type can be assigned from lvalue
`Destructible`	specifies that an object of the type can be destroyed

Library-wide concepts

`EqualityComparable`	`operator ==` is an equivalence relation
`LessThanComparable`	`operator <` is a strict weak ordering relation
`Swappable` C++11	can be swapped with an unqualified non-member function `swap()`
`ValueSwappable` C++11	an `Iterator` that dereferences to a `Swappable` type
`NullablePointer` C++11	a pointer-like type supporting a null value
`Hash` C++11	a `FunctionObject` that for inputs with different values has a low probability of giving the same output
`Allocator`	a class type that contains allocation information
`FunctionObject`	an object that can be called with the function call syntax
`Callable`	a type for which the invoke operation is defined
`Predicate`	a `FunctionObject` that returns a value convertible to bool for one argument without modifying it
`BinaryPredicate`	a `FunctionObject` that returns a value convertible to bool for two arguments without modifying them
`Compare`	a `BinaryPredicate` that establishes an ordering relation

Container concepts

`Container`	data structure that allows element access using iterators
`ReversibleContainer`	container using bidirectional iterators
`AllocatorAwareContainer` C++11	container using an allocator
`SequenceContainer`	container with elements stored linearly
`ContiguousContainer` C++11	container with elements stored at adjacent memory addresses
`AssociativeContainer`	container that stores elements by associating them to keys
`UnorderedAssociativeContainer` C++11	container that stores elements stored in buckets by associating them to keys

Container element concepts

`DefaultInsertable` C++11	element can be default-constructed in uninitialized storage
`CopyInsertable` C++11	element can be copy-constructed in uninitialized storage
`MoveInsertable` C++11	element can be move-constructed in uninitialized storage
`EmplaceConstructible` C++11	element can be constructed in uninitialized storage
`Erasable` C++11	element can be destroyed using an allocator

Iterator concepts

`Iterator`	general concept to access data within some data structure
`InputIterator`	iterator that can be used to read data
`OutputIterator`	iterator that can be used to write data
`ForwardIterator`	iterator that can be used to read data multiple times
`BidirectionalIterator`	iterator that can be both incremented and decremented
`RandomAccessIterator`	iterator that can be advanced in constant time
`ContiguousIterator` C++17	iterator to contiguously-allocated elements

Concurrency concepts

`BasicLockable` C++11	provides exclusive ownership semantics for execution agents (i.e. threads)
`Lockable` C++11	extends the `BasicLockable` concept to include attempted locking
`TimedLockable` C++11	a `Lockable` that supports timed lock acquisition
`Mutex` C++11	extends the `Lockable` concept to include inter-thread synchronization
`TimedMutex` C++11	extends the `TimedLockable` concept to include inter-thread synchronization
`SharedMutex` C++17	a `Mutex` that supports shared ownership semantics
`SharedTimedMutex` C++14	a `TimedMutex` that supports shared ownership semantics

Stream I/O function concepts

`UnformattedInputFunction`	a stream input function that does not skip leading whitespace and counts the processed characters
`FormattedInputFunction`	a stream input function that skips leading whitespace
`UnformattedOutputFunction`	a basic stream output function
`FormattedOutputFunction`	a stream output function that sets failbit on errors and returns a reference to the stream

Random number generation concepts

`SeedSequence` C++11	consumes a sequence of integers and produces a sequence of 32-bit unsigned values
`UniformRandomBitGenerator` C++11	returns uniformly distributed random unsigned integers
`RandomNumberEngine` C++11	a deterministic `UniformRandomBitGenerator`, defined by the seed
`RandomNumberEngineAdaptor` C++11	a `RandomNumberEngine` that transforms the output of another `RandomNumberEngine`
`RandomNumberDistribution` C++11	returns random numbers distributed according to a given mathematical probability density function

dwm
@mcplex.net

"This pattern is common to all great programmers I know: they're not experts in something as much as experts in becoming experts in something."
– Andrei Alexandrescu