Group 1: Sequence Containers (Linear)

Feature	std::vector	std::deque	std::list
Concept	Dynamic Array	Double-ended Queue	Doubly Linked List
Access	Fast Random Access (v[i])	Fast Random Access (d[i])	Slow sequential only (Iterators)
Insert/Delete	Fast at End only	Fast at Front & End	Fast Anywhere (if iterator known)
Memory	Contiguous (one block)	Chunks of memory	Scattered nodes
Best For	Default choice, resizing arrays	Queues where you add to front/back	Heavy insertion/deletion in middle

Function	Description	Vector	Deque	List
push_back(v)	Add to end			
pop_back()	Remove from end			
push_front(v)	Add to front			
pop_front()	Remove from front			
operator[i]	Access index i			
at(i)	Safe access index i			
insert(it, v)	Insert at iterator	(Slow)	(Slow)	(Fast)

Group 2: Associative Containers (Key-Based)

Feature	std::set	std::map	unordered_set	unordered_map
Concept	Unique Keys	Key-Value Pairs	Unique Keys (Hashed)	Key-Value (Hashed)
Ordering	Sorted (Always)	Sorted by Key	Random / Undefined	Random / Undefined
Search Speed	$O(\log n)$	$O(\log n)$	$O(1)$ (Average)	$O(1)$ (Average)
Duplicates?	No	Keys must be unique	No	Keys must be unique
Best For	Sorted list of unique items	Dictionary / Lookup table	Checking "Have I seen this?"	High-performance lookup

Function	Description	Set	Map	Unordered Versions
insert(v)	Add element			
erase(k)	Remove key k			
find(k)	Return iterator to k			
count(k)	1 if exists, 0 if not			
operator[k]	Access/Create value			(Map only)
lower_bound(k)	Find first $\ge k$			

Group 3: Container Adaptors (Restricted)

Feature	std::stack	std::queue	std::priority_queue
Concept	LIFO (Last In, First Out)	FIFO (First In, First Out)	Priority (Sorted Heap)
Analogy	Stack of plates	Line at a bank	Emergency Room Triage
Access	Top only	Front (read) & Back (write)	Top (Max/Min) only
Underlying	Uses deque by default	Uses deque by default	Uses vector by default

Function	Description	Stack	Queue	Priority Queue
push(v)	Add element			
pop()	Remove element	(Top)	(Front)	(Top)
top()	View top element			
front()	View first element			
back()	View last element			

Group 4: Multi-Containers (Allow Duplicates)

Feature	std::multiset	std::multimap
Concept	Sorted collection allowing duplicate values	Key-Value pairs allowing duplicate keys
Ordering	Sorted by value	Sorted by key
Search Speed	$O(\log n)$	$O(\log n)$
Best For	Keeping track of multiple scores or weights	Storing contact information where one person has multiple phone numbers

Function	Description	Example Use Case
insert(v)	Adds the element/pair, even if duplicates exist.	multiset.insert(10); multiset.insert(10); (Both 10s are stored)
count(k)	Returns the number of elements with key k.	If multimap has three entries for "Alice", this returns 3.
equal_range(k)	Returns a std::pair of iterators: the start and end of the range where the key k appears.	This is the standard way to iterate over all values associated with a duplicate key.

Group 5: Utility Components (pair, tuple, etc.)

Feature	std::pair	std::tuple
Concept	Holds exactly 2 items	Holds N items (2, 3, 4, etc.)
Types	Types can be different (e.g., int, string)	Types can be different
Access Style	Named members (.first, .second)	Template index (get<0>, get<1>)
Header	#include <utility>	#include <tuple>
Mutability	Mutable (can change values)	Mutable (can change values)
Best For	Key-Value returns, Dictionary entries	Grouping complex data rows without a class

Function	Description	Pair	Tuple
make_...	Helper to create without specifying types (make_pair, make_tuple)		
.first	Access the first element		
.second	Access the second element		
get<N>(x)	Access element at index N (must be a constant index)	(Modern C++	
swap(x)	Swaps content with another object of same type		
tie(...)	Unpacks values into separate variables		

Group 6: Smart Pointers (Memory Utilities)

Feature	std::unique_ptr	std::shared_ptr	std::weak_ptr
Concept	Exclusive Ownership	Shared Ownership	Non-owning Observer
Copy Policy	Move Only (Cannot copy)	Copyable (Increases ref count)	Copyable (No ref count increase)
Memory Freed	When the pointer goes out of scope.	When the last pointer goes out of scope.	Never (it doesn't own the memory).
Overhead	Zero (Same as raw pointer)	Low (Stores a reference count)	Low
Best For	90% of use cases. Class members, local variable	Graph nodes, shared resources, plugins.	Breaking circular dependencies, caching.

Function Category	Function	unique_ptr	shared_ptr	weak_ptr	Description
Access	*ptr (Dereference)				Access the object directly.
	ptr-> (Arrow)				Access a member (method/variable) of the object.
	get()				Returns the raw memory address (unsafe).
	lock()				Creates a temporary shared_ptr to access data safely.
Management	reset()				Releases/Deletes the current object and becomes nullptr.
	reset(new_ptr)				Deletes old object, takes ownership of new_ptr.
	release()				Gives up ownership without deleting. Returns raw pointer.
	swap(other)				Swaps managed objects with another pointer.
Status	operator bool				Checks if pointer is valid (if(ptr)...).
	use_count()				Returns number of shared_ptr owners.
	unique()				Returns true if use_count == 1 (Deprecated in C++20).
	expired()				Returns true if the object has already been deleted.
Creation	make_...	make_unique	make_shared		The standard, safe way to create them.