Declaration	Behavior
`template<typename T>` `constexpr const T& min(const T& a, const T& b );`	Returns the smaller of a and b
`template<typename T>` `constexpr const T& max( const T& a, const T& b );`	Returns the greater of a and b

Type	Container	Internal data structure
Sequential	`vector<T>`	A variable-size vector
	`list<T>`	A doubly-linked list
	`forward_list<T>`	A singly-linked list
	`deque<T>`	A double-ended queue
Associative	`set<T>`	A set (a map with just a key and no value)
	`multiset<T>`	A set in which a value can occur many times
	`map<K,V>`	An associative array
	`multimap<K,V>`	A map in which a key can occur many times
Unordered	`unordered_map<K,V>`	A map using a hashed lookup
	`unordered_multimap<K,V>`	A multimap using a hashed lookup
	`unordered_set<T>`	A set using a hashed lookup
	`unordered_multiset<T>`	A multiset using a hashed lookup

Function (STL containers)	Description
`T()`	create empty container (default constructor)
`T(const T&)`	copy container (copy constructor)
`T(T&&)`	move container (move constructor)
`~T()`	destroy container (including its elements)
`empty()`	test if container empty
`size()`	get number of elements in containe
`push_back(const T&)`	insert an element at end of container (sequential)
`insert(T&)`	insert an element (associative/unordered)
`clear()`	remove all elements from containe
`operator=()`	assign all elements of one container to other container
`operator[]()`	access element in container
`begin()`	returns an iterator to the beginning
`end()`	returns an iterator to the end

Type	Declaration

`std::pair`	`template<typename T1, typename T2>`
	`struct pair {`
	`  typedef T1 first_type;`
	`  typedef T2 second_type;`
	`  T1 first;`
	`  T2 second;`
	`  // … member functions`
	`}`

Function (STL algorithms)	Description
`p=find(b,e,x)`	p is the first p in [b:e) so that *p==x
`p=find_if(b,e,f)`	p is the first p in [b:e) so that f(*p)==true
`n=count(b,e,x)`	n is the number of elements *q in [b:e) so that *q==x
`n=count_if(b,e,f)`	n is the number of elements *q in [b:e) so that f(*q,x)
`replace(b,e,v,v2)`	Replace elements *q in [b:e) so that *q==v by v2
`replace_if(b,e,f,v2)`	Replace elements *q in [b:e) so that f(*q) by v2
`p=copy(b,e,out)`	Copy [b:e) to [out:p)
`p=copy_if(b,e,out,f)`	Copy elements *q from [b:e) so that f(*q) to [out:p)
`p=move(b,e,out)`	Move [b:e) to [out:p)
`p=unique_copy(b,e,out)`	Copy [b:e) to [out:p); don't copy adjacent duplicates
`sort(b,e)`	Sort elements of [b:e) using < as the sorting criterion
`sort(b,e,f)`	Sort elements of [b:e) using f as the sorting criterion
`for_each(b, e, f)`	Invoke function f() for every element in [b:e)
`p=transform(b, e, out, f)`	For elements *q in [b:e), put return value of function f() so that f(*q) to [out:p)

std::find	Description (partial)
`function prototype`	`template<typename InputIt, typename T>` `InputIt find(InputIt first, InputIt last, const T& value);`
`parameters`	**`first, last`** - the range of elements to examine **`value`** - value to compare the elements to
`Return value`	Iterator to the first element satisfying the condition or `last` if no such element is found.