C++ STL (Standard Template Library) Reference Guide

Table of Contents

Introduction

The Standard Template Library (STL) is a collection of template classes and functions that provide common data structures and algorithms. It consists of four main components:

- **Containers**: Data structures that store objects
- **Iterators**: Objects that traverse through containers
- **Algorithms**: Functions that perform operations on containers
- **Function Objects**: Objects that act like functions

Key Headers

cpp

```cpp
#include <vector>     // Dynamic arrays
#include <list>       // Doubly linked lists
#include <deque>      // Double-ended queues
#include <set>        // Ordered sets
#include <map>        // Associative arrays
#include <unordered_set>  // Hash sets
#include <unordered_map>  // Hash maps
#include <stack>      // LIFO container
#include <queue>      // FIFO container
#include <algorithm>  // Algorithms
#include <iterator>   // Iterator utilities
#include <string>     // String class
#include <memory>     // Smart pointers
```

Containers

Sequence Containers

Vector

Dynamic array that can grow and shrink.

Declaration:

```cpp
std::vector<int> v;
std::vector<int> v(10);       // Size 10, default initialized
std::vector<int> v(10, 5);    // Size 10, all elements = 5
std::vector<int> v{1, 2, 3};  // Initializer list
```

Common Operations:

```cpp
v.push_back(x);    // Add element at end
v.pop_back();      // Remove last element
v.size();          // Number of elements
v.capacity();      // Allocated space
v.empty();         // Check if empty
v.clear();         // Remove all elements
v.resize(n);       // Change size
v.reserve(n);      // Reserve capacity
v[i];              // Access element (no bounds check)
v.at(i);           // Access element (with bounds check)
v.front();         // First element
v.back();          // Last element
v.begin();         // Iterator to first element
v.end();           // Iterator past last element
```

List

Doubly linked list.

Declaration:

```cpp
std::list<int> l;
std::list<int> l{1, 2, 3, 4};
```

Common Operations:

```cpp
l.push_front(x);    // Add at beginning
l.push_back(x);     // Add at end
l.pop_front();      // Remove first element
l.pop_back();       // Remove last element
l.insert(it, x);    // Insert at iterator position
l.erase(it);        // Remove at iterator position
l.remove(x);        // Remove all elements equal to x
l.sort();           // Sort the list
l.unique();         // Remove consecutive duplicates
l.reverse();        // Reverse the list
```

Deque

Double-ended queue.

Declaration:

```cpp
std::deque<int> d;
std::deque<int> d{1, 2, 3};
```

Common Operations:

```cpp
d.push_front(x);    // Add at beginning
d.push_back(x);     // Add at end
d.pop_front();      // Remove first element
d.pop_back();       // Remove last element
d[i];               // Random access
d.at(i);            // Random access with bounds check
```

Associative Containers

Set

Ordered collection of unique elements.

Declaration:

cpp

```cpp
std::set<int> s;
std::set<int> s{1, 2, 3, 4};
std::set<int, std::greater<int>> s;  // Descending order
```

Common Operations:

cpp

```cpp
s.insert(x);       // Insert element
s.erase(x);        // Remove element
s.find(x);         // Find element (returns iterator)
s.count(x);        // Count occurrences (0 or 1 for set)
s.lower_bound(x);  // Iterator to first element >= x
s.upper_bound(x);  // Iterator to first element > x
s.equal_range(x);  // Pair of iterators [lower_bound, upper_bound)
```

Map

Associative array with unique keys.

Declaration:

cpp

```cpp
std::map<std::string, int> m;
std::map<std::string, int> m{{"key1", 1}, {"key2", 2}};
```

Common Operations:

cpp

```cpp
m[key] = value;    // Insert or update
m.insert({key, value});  // Insert pair
m.erase(key);      // Remove by key
m.find(key);       // Find by key
m.count(key);      // Check if key exists
m.at(key);         // Access with bounds check
```

Multiset and Multimap

Similar to set and map but allow duplicate keys.

cpp

```cpp
std::multiset<int> ms;
std::multimap<std::string, int> mm;
```

Unordered Containers (Hash Tables)

Unordered Set

Hash table implementation of set.

Declaration:

cpp

```cpp
std::unordered_set<int> us;
std::unordered_set<int> us{1, 2, 3, 4};
```

Common Operations:

cpp

```cpp
us.insert(x);      // Insert element
us.erase(x);       // Remove element
us.find(x);        // Find element
us.count(x);       // Count occurrences
us.bucket_count(); // Number of buckets
us.load_factor();  // Current load factor
```

Unordered Map

Hash table implementation of map.

Declaration:

cpp

```cpp
std::unordered_map<std::string, int> um;
std::unordered_map<std::string, int> um{{"key1", 1}, {"key2", 2}};
```

Container Adapters

Stack

LIFO (Last In, First Out) container.

Declaration:

cpp

```cpp
std::stack<int> st;
std::stack<int, std::vector<int>> st;  // Using vector as underlying container
```

Common Operations:

cpp

```cpp
st.push(x);       // Add element
st.pop();         // Remove top element
st.top();         // Access top element
st.empty();       // Check if empty
st.size();        // Number of elements
```

Queue

FIFO (First In, First Out) container.

Declaration:

cpp

```cpp
std::queue<int> q;
std::queue<int, std::list<int>> q;  // Using list as underlying container
```

Common Operations:

cpp

```cpp
q.push(x);       // Add element
q.pop();         // Remove front element
q.front();       // Access front element
q.back();        // Access back element
q.empty();       // Check if empty
q.size();        // Number of elements
```

Priority Queue

Heap-based priority queue.

Declaration:

cpp

```cpp
std::priority_queue<int> pq;                  // Max heap
std::priority_queue<int, std::vector<int>, std::greater<int>> pq;  // Min heap
```

Common Operations:

```cpp
pq.push(x);      // Add element
pq.pop();        // Remove top element
pq.top();        // Access top element
pq.empty();      // Check if empty
pq.size();       // Number of elements
```

Iterators

Iterators are objects that point to elements in containers and allow traversal.

Iterator Categories

1. **Input Iterator**: Read-only, single pass
2. **Output Iterator**: Write-only, single pass
3. **Forward Iterator**: Read/write, single pass
4. **Bidirectional Iterator**: Forward + backward movement
5. **Random Access Iterator**: Jump to any position

Common Iterator Operations

```cpp
// Basic operations
*it;            // Dereference
++it;           // Move to next element
it++;           // Post-increment
--it;           // Move to previous element (bidirectional+)
it--;           // Post-decrement

// Random access operations
it + n;         // Move n positions forward
it - n;         // Move n positions backward
it[n];          // Access element at offset n
it1 - it2;      // Distance between iterators

// Comparison
it1 == it2;     // Equality
it1 != it2;     // Inequality
it1 < it2;      // Less than (random access)
```

Iterator Functions

cpp

```cpp
#include <iterator>

std::advance(it, n);       // Move iterator n positions
std::distance(it1, it2);   // Distance between iterators
std::next(it, n);          // Return iterator n positions ahead
std::prev(it, n);          // Return iterator n positions back
```

Range-based for Loop

cpp

```cpp
std::vector<int> v{1, 2, 3, 4, 5};

// Read-only access
for (const auto& element : v) {
    std::cout << element << " ";
}

// Modify elements
for (auto& element : v) {
    element *= 2;
}
```

Algorithms

The `<algorithm>` header provides numerous algorithms that work with iterators.

Non-modifying Algorithms

cpp

```cpp
// Searching
std::find(begin, end, value);          // Find first occurrence
std::find_if(begin, end, predicate);   // Find first matching predicate
std::count(begin, end, value);         // Count occurrences
std::count_if(begin, end, predicate);  // Count matching predicate

// Checking conditions
std::all_of(begin, end, predicate);    // All elements match
std::any_of(begin, end, predicate);    // Any element matches
std::none_of(begin, end, predicate);   // No elements match

// Comparison
std::equal(begin1, end1, begin2);      // Compare two ranges
std::mismatch(begin1, end1, begin2);   // Find first difference
```

Modifying Algorithms

cpp

```cpp
// Copying
std::copy(begin, end, dest);           // Copy range
std::copy_if(begin, end, dest, pred);  // Copy elements matching predicate

// Filling
std::fill(begin, end, value);          // Fill range with value
std::fill_n(begin, n, value);          // Fill n elements with value

// Transforming
std::transform(begin, end, dest, func); // Apply function to each element

// Removing
std::remove(begin, end, value);        // Remove elements (logical removal)
std::remove_if(begin, end, predicate); // Remove elements matching predicate
std::unique(begin, end);               // Remove consecutive duplicates

// Replacing
std::replace(begin, end, old_val, new_val); // Replace values
std::replace_if(begin, end, pred, new_val); // Replace based on predicate
```

Sorting Algorithms

cpp

```cpp
// Sorting
std::sort(begin, end);                  // Sort in ascending order
std::sort(begin, end, comparator);      // Sort with custom comparator
std::stable_sort(begin, end);           // Stable sort
std::partial_sort(begin, middle, end);  // Sort first n elements

// Searching in sorted ranges
std::binary_search(begin, end, value);  // Check if value exists
std::lower_bound(begin, end, value);    // First position where value could be inserted
std::upper_bound(begin, end, value);    // Last position where value could be inserted
std::equal_range(begin, end, value);    // Pair of lower_bound and upper_bound

// Merging
std::merge(begin1, end1, begin2, end2, dest);  // Merge two sorted ranges
```

Heap Algorithms

cpp

```cpp
std::make_heap(begin, end);          // Create heap
std::push_heap(begin, end);          // Add element to heap
std::pop_heap(begin, end);           // Remove max element from heap
std::sort_heap(begin, end);          // Sort heap (destroys heap property)
```

Permutation Algorithms

cpp

```cpp
std::next_permutation(begin, end);     // Generate next permutation
std::prev_permutation(begin, end);     // Generate previous permutation
```

Function Objects

Function objects (functors) are objects that can be called like functions.

Predefined Function Objects

cpp

```cpp
#include <functional>

// Arithmetic operations
std::plus<int>()        // Addition
std::minus<int>()       // Subtraction
std::multiplies<int>()  // Multiplication
std::divides<int>()     // Division

// Comparison operations
std::equal_to<int>()    // Equality
std::not_equal_to<int>() // Inequality
std::greater<int>()     // Greater than
std::less<int>()        // Less than
std::greater_equal<int>() // Greater than or equal
std::less_equal<int>()  // Less than or equal

// Logical operations
std::logical_and<bool>() // Logical AND
std::logical_or<bool>()  // Logical OR
std::logical_not<bool>() // Logical NOT
```

Lambda Expressions

cpp

```cpp
// Basic lambda
auto lambda = [](int x) { return x * 2; };

// Lambda with capture
int multiplier = 3;
auto lambda2 = [multiplier](int x) { return x * multiplier; };

// Lambda with reference capture
int sum = 0;
auto lambda3 = [&sum](int x) { sum += x; };

// Lambda with mixed capture
auto lambda4 = [=, &sum](int x) { sum += x * multiplier; };
```

Using Function Objects with Algorithms

cpp

```cpp
std::vector<int> v{1, 2, 3, 4, 5};

// Sort in descending order
std::sort(v.begin(), v.end(), std::greater<int>());

// Find first even number
auto it = std::find_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });

// Transform elements
std::transform(v.begin(), v.end(), v.begin(), [](int x) { return x * x; });
```

Utility Classes

Pair

cpp

```cpp
#include <utility>

std::pair<int, std::string> p;
std::pair<int, std::string> p(1, "hello");
std::pair<int, std::string> p{1, "hello"};

// Accessing elements
p.first;        // First element
p.second;       // Second element

// Creating pairs
auto p2 = std::make_pair(1, "hello");
```

Tuple

cpp

```cpp
#include <tuple>

std::tuple<int, std::string, double> t;
std::tuple<int, std::string, double> t(1, "hello", 3.14);
std::tuple<int, std::string, double> t{1, "hello", 3.14};

// Accessing elements
std::get<0>(t);    // First element
std::get<1>(t);    // Second element
std::get<2>(t);    // Third element

// Creating tuples
auto t2 = std::make_tuple(1, "hello", 3.14);

// Unpacking tuples
int a;
std::string b;
double c;
std::tie(a, b, c) = t;
```

String Class

Declaration and Initialization

cpp

```cpp
#include <string>

std::string s;              // Empty string
std::string s("hello");     // From C-string
std::string s{"hello"};     // From C-string
std::string s(5, 'a');      // 5 'a' characters
std::string s1(s2);         // Copy constructor
```

Common Operations

cpp

```cpp
// Size and capacity
s.size();          // Number of characters
s.length();        // Same as size()
s.empty();         // Check if empty
s.capacity();      // Allocated space
s.reserve(n);      // Reserve space

// Access
s[i];              // Character at position i
s.at(i);           // Character at position i (bounds checked)
s.front();         // First character
s.back();          // Last character

// Modifying
s.push_back(c);    // Add character at end
s.pop_back();      // Remove last character
s.append(str);     // Append string
s.insert(pos, str); // Insert string at position
s.erase(pos, len);  // Remove substring
s.replace(pos, len, str);  // Replace substring

// Searching
s.find(str);       // Find first occurrence
s.rfind(str);      // Find last occurrence
s.find_first_of(chars);    // Find first of any character
s.find_last_of(chars);     // Find last of any character
s.find_first_not_of(chars); // Find first not of any character

// Substring
s.substr(pos, len); // Extract substring

// Comparison
s1 == s2;          // Equality
s1 != s2;          // Inequality
s1 < s2;           // Lexicographical comparison
s1.compare(s2);    // Three-way comparison
```

String Conversion

cpp

```cpp
// Convert to numbers
std::stoi(s);      // String to int
std::stol(s);      // String to long
std::stof(s);      // String to float
std::stod(s);      // String to double

// Convert from numbers
std::to_string(123);    // int to string
std::to_string(3.14);   // double to string
```

Smart Pointers

Smart pointers provide automatic memory management.

unique_ptr

cpp

```cpp
#include <memory>

// Creation
std::unique_ptr<int> ptr(new int(42));
std::unique_ptr<int> ptr = std::make_unique<int>(42);

// Access
*ptr;             // Dereference
ptr.get();        // Get raw pointer
ptr.reset();      // Delete object and reset to nullptr
ptr.reset(new int(10));  // Delete object and reset to new object
ptr.release();    // Release ownership (returns raw pointer)

// Move semantics (unique_ptr cannot be copied)
std::unique_ptr<int> ptr2 = std::move(ptr);
```

shared_ptr

cpp

```cpp
// Creation
std::shared_ptr<int> ptr(new int(42));
std::shared_ptr<int> ptr = std::make_shared<int>(42);

// Access
*ptr;              // Dereference
ptr.get();         // Get raw pointer
ptr.reset();       // Decrease reference count
ptr.use_count();   // Get reference count

// Copying (reference count increases)
std::shared_ptr<int> ptr2 = ptr;
```

weak_ptr

cpp

```cpp
// Creation (from shared_ptr)
std::shared_ptr<int> sptr = std::make_shared<int>(42);
std::weak_ptr<int> wptr = sptr;

// Access
wptr.expired();    // Check if object still exists
wptr.lock();       // Get shared_ptr if object exists
wptr.use_count();  // Get reference count
```

Common Patterns

Container Iteration

cpp

```cpp
std::vector<int> v{1, 2, 3, 4, 5};

// Range-based for loop
for (const auto& element : v) {
    std::cout << element << " ";
}

// Iterator-based loop
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}

// Algorithm-based
std::for_each(v.begin(), v.end(), [](int x) {
    std::cout << x << " ";
});
```

Container Manipulation

cpp

```cpp
std::vector<int> v{1, 2, 3, 2, 4, 2, 5};

// Remove all occurrences of 2
v.erase(std::remove(v.begin(), v.end(), 2), v.end());

// Remove duplicates
std::sort(v.begin(), v.end());
v.erase(std::unique(v.begin(), v.end()), v.end());

// Find and replace
std::replace(v.begin(), v.end(), 3, 30);
```

Custom Comparators

cpp

```cpp
// For sorting
std::vector<std::string> words{"apple", "banana", "cherry"};
std::sort(words.begin(), words.end(), [](const std::string& a, const std::string& b) {
    return a.length() < b.length();
});

// For containers
std::set<std::string, std::greater<std::string>> descending_set;
std::map<std::string, int, std::less<std::string>> ascending_map;
```

Error Handling

cpp

```cpp
// Using exceptions
try {
    std::vector<int> v{1, 2, 3};
    int x = v.at(10);  // Throws std::out_of_range
} catch (const std::out_of_range& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}

// Using find instead of direct access
std::map<std::string, int> m;
auto it = m.find("key");
if (it != m.end()) {
    // Key found
    int value = it->second;
}
```

Performance Tips

1. **Choose the right container**: Use `vector` for random access, `list` for frequent insertions/deletions, `deque` for double-ended operations.
2. **Reserve capacity**: Use `reserve()` for vectors when you know the approximate size.
3. **Use const references**: Pass containers by const reference to avoid unnecessary copying.
4. **Prefer algorithms**: Use STL algorithms instead of hand-written loops for better performance and readability.
5. **Use move semantics**: Use `std::move()` to avoid unnecessary copying of large objects.
6. **Choose appropriate iterators**: Use the most specific iterator type needed for your algorithm.

This reference guide covers the essential components of the C++ STL. For more detailed information, consult the official C++ documentation.