Lecture 06

The Standard Template Library.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

04 Nov 2025

Outline

1. Containers
    - Sequence containers
    - Container adaptors
    - Associative containers
    - Special containers
2. Iterators
3. Algorithms
4. Evolution of the STL

An overview of the Standard Template Library (STL) in C++

History

In November 1993, Alexander Stepanov introduced the Standard Template Library (STL) to the ANSI/ISO committee for C++ standardization. This library was founded on the principles of generic programming and featured generic containers, iterators, and a comprehensive set of algorithms designed to operate on them.

The proposal was not only accepted but also paved the way for what is now known as the Standard Library. The Standard Library has evolved into a vast collection of tools that play an indispensable role in modern C++ development.

Notably, all functionalities provided by the Standard Library are encapsulated under the `std::` namespace or, in some cases, within sub-namespaces also contained within `std::`.

The Boost libraries

Many components added to the Standard Library over the years were originally proposed in the Boost C++ libraries, a collection of libraries designed to complement the Standard Library and cater to a wide range of applications.

Notably, one outstanding component within the Boost libraries is the Boost Graph Library, widely used by professionals working with graphs and networks.

All Boost libraries are open-source and can be installed individually or as a whole on various Linux platforms using package managers. For instance, on Ubuntu, you can use the command `sudo apt-get install libboost-all-dev`. Alternatively, you can download the source code and compile them manually.

An overview of the STL (1/2)

- **Porting of C libraries**: Several C libraries have been adapted to the `std` namespace. As an example, `<math.h>` becomes `<cmath>`, and they all begin with a 'c'.
- **Containers**: Generic containers and iterators.
- **Utilities**: Smart pointers, fixed-size collections of heterogeneous values like `pair` and `tuple`, clocks and timers, function wrappers, and predefined functors. Also, the `ratio` class for constant rationals.
- **Algorithms**: These operate on ranges of values, usually stored in standard containers, to perform specific actions like sorting, transformations, and copying. Some of them even support parallel execution.
- **Strings and text processing**: The `string` class and its derived classes, regular expressions, and efficient string manipulation tools.
- **Support for I/O**: I/O streams and related utilities.
- **Utilities for diagnostics**: Standard exception classes and exception handlers.

An overview of the STL (2/2)

- **Numerics**: `complex<T>`, numeric limits, random numbers and distributions, and basic mathematical operators.
- **Support for generic programming**: Type traits, `decltype`, `declval`, and `as_const`.
- **Support for reference and move semantics**: Reference wrappers, `move()`, and `forward<T>`.
- **Support for multithreading and concurrency**: Threads, mutexes, locks, and parallel algorithms.
- **Support for internationalization**: `locale` and `wide_char`.
- **Filesystem**: Tools for examining the file system.
- **Allocators**: Utilities that allow you to change how objects are allocated within containers.
- **Utilities for hybrid data**: `optional`, `variant`, and `any`.

A milestone: C++11

- **Standardization process**: C++11 marked the successful completion of a rigorous standardization process.
- **Modern features**: C++11 introduced modern language features like lambda expressions and range-based for loops.
- **Enhanced Standard Library**: New containers, algorithms, and utility classes.
- **Improved memory management**: Smart pointers were introduced in C++11.
- **Initializer lists**: C++11 introduced initializer lists, simplifying data structure initialization.
- **Simpler and safer code**: The addition of lambda expressions improved code readability and maintainability.
- **Standardized threads**: It brought a standardized threading library, enabling concurrent and parallel programming.
- **Improved performance**: Move semantics and rvalue references boosted resource management and program performance.

Containers

Sequence containers

Containers can be categorized based on how data is stored and handled internally. The categories include:

- **Sequence containers**: `std::vector<T>`, `std::array<T,N>`, `std::deque<T>`, `std::list<T>`, `std::forward_list<T>`.
  - Ordered collections of elements with their position independent of the element value.
  - In `std::vector` and `std::array`, elements are guaranteed to be contiguous in memory and can be accessed directly using the `[]` operator.
- **Adaptors**: These are built on top of other containers and provide special operations:
  - `std::stack<T>`, `std::queue<T>`, and `std::priority_queue<T>`.

Example: `std::vector`

```
std::vector<int> v{2,4,5};      // 2, 4, 5.
v.push_back(6);                 // 2, 4, 5, 6.
v.pop_back();                   // 2, 4, 5.
v[1] = 3;                       // 2, 3, 5.
std::cout << v[2] << std::endl; // Prints: 5
for (int x : v)
    std::cout << x << ' '; // Prints: 2 3 5
std::cout << std::endl;

v.reserve(8);               // Pre-allocate space for 8 elements.
v.resize(5, 0);             // Resize to 5, fill new elements with 0.
std::cout << v.capacity() << std::endl;  // Prints: 8
std::cout << v.size() << std::endl;      // Prints: 5
```

Example: `std::array`

```
std::array<int, 6> a{4,8,15,16,23,42};
std::cout << a.size() << std::endl;    // 6
std::cout << a[0] << std::endl;        // 4
std::cout << a[3] << std::endl;        // 16
std::cout << a.front() << std::endl;   // 4
std::cout << a.back() << std::endl;    // 42

std::array<int, 3> b{7,8,9};
// a = b;  // Compiler error: types don't match!
```

Source

C++ Standard Library Sequence Containers

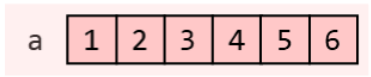
h/cpp hackingcpp.com

array<T,size>

fixed-size array

#include <array>

```
std::array<int,6> a {1,2,3,4,5,6};
cout << a.size();     // 6
cout << a[2];         // 3
a[0] = 7;             // 1st element ⇒ 7
```



contiguous memory; random access; fast linear traversal

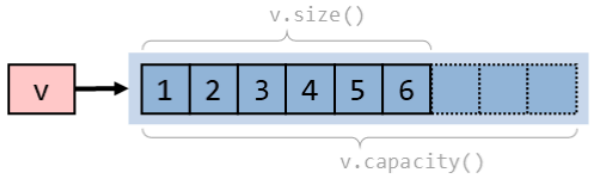
vector<T>

dynamic array

C++'s "default" container

#include <vector>

```
std::vector<int> v {1,2,3,4,5,6};
v.reserve(9);
cout << v.capacity();      // 9
cout << v.size();          // 6
v.push_back(7);            // appends '7'
v.insert(v.begin(), 0);    // prepends '0'
v.pop_back();              // removes last
v.erase(v.begin()+2);      // removes 3rd
v.resize(20, 0);           // size ⇒ 20
```



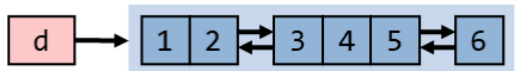
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

deque<T>

double-ended queue

#include <deque>

```
std::deque<int> d {1,2,3,4,5,6};
// same operations as vector
// plus fast growth/deletion at front
d.push_front(-1);   // prepends '-1'
d.pop_front();      // removes 1st
```



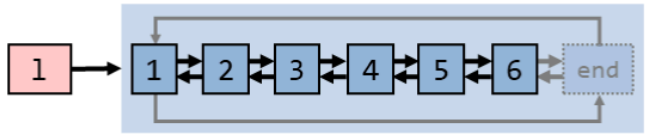
fast insertion/deletion at both ends

list<T>

doubly-linked list

#include <list>

```
std::list<int> l {1,5,6};
std::list<int> k {2,3,4};
// O(1) splice of k into l:
l.splice(l.begin()+1, std::move(k))
// some special member function algorithms:
l.reverse();
l.sort();
```



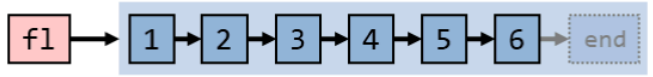
fast splicing; many operations without copy/move of elements

forward_list<T>

singly-linked list

#include <forward_list>

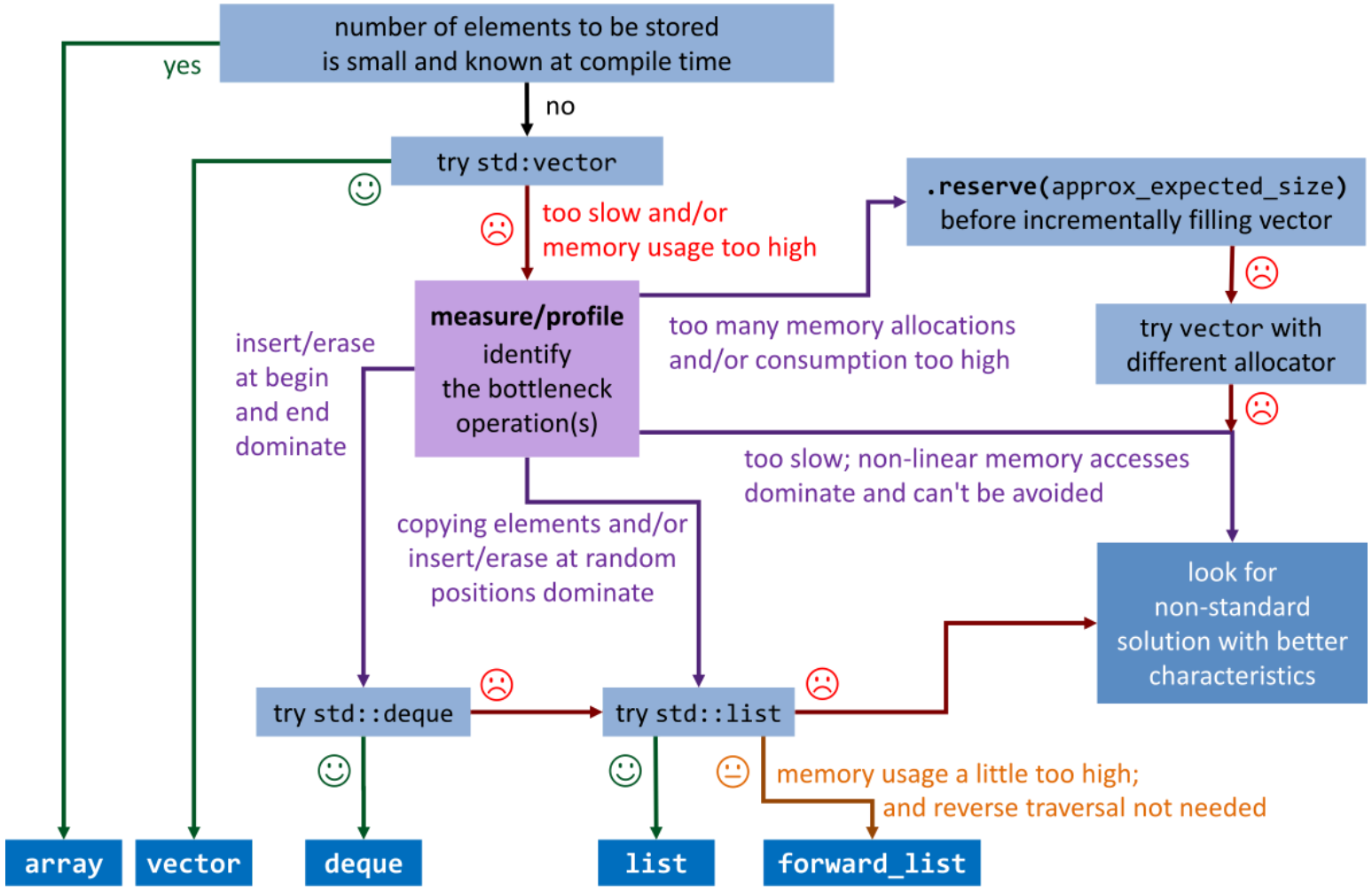
```
std::forward_list<int> fl {2,2,4,5,6};
fl.erase_after(begin(fl));
fl.insert_after(begin(fl), 3);
fl.insert_after(before_begin(fl), 1);
```



lower memory overhead than std::list; only forward traversal

v1.2

Which sequence container to choose?



Associative containers (1/3)

- **Associative containers**: These collections have elements whose position depends on their content. They are divided into:
  - **Maps**: Elements are key-value pairs.
  - **Sets**: Elements are just values (in sets, keys and values are considered the same).
  - Furthermore, they can be divided into **ordered** and **unordered**, depending on how the elements are stored, imposing different requirements on element types.

⚠️ **In a set, the terms "value" and "key" are used interchangeably since they are equivalent.**

Quick examples:

- **Map**: `std::map<std::string, int> ages = {{"Alice", 25}, {"Bob", 30}};`
- **Set**: `std::set<int> unique_nums = {1, 2, 3, 2};` stores only `{1, 2, 3}`

Associative containers (2/3)

Ordered associative containers

- `std::map<Key, T>`, `std::multimap<Key, T>`: Sorted associative containers that contain key-value pairs (maps) and allow multiple elements with the same key (multimaps).
- `std::set<Key>`, `std::multiset<Key>`: Sorted associative containers that contain unique keys (sets) and allow multiple elements with the same key (multisets).
- **Requirements**: Elements must be **comparable** (define `operator<` or provide a custom comparator).
- Typically implemented using **red-black trees** (self-balancing binary search trees).
- **Complexity**: Insertion, deletion, and search are all $O(\log n)$.

Associative containers (3/3)

Unordered associative containers

- `std::unordered_map<Key, T>`, `std::unordered_multimap<Key, T>`: Associative containers that contain key-value pairs (maps) and allow multiple elements with the same key (multimaps).
- `std::unordered_set<Key>`, `std::unordered_multiset<Key>`: Associative containers that contain unique keys (sets) and allow multiple elements with the same key (multisets).
- **Requirements**: Elements must be **hashable** (define `std::hash<Key>` or provide a custom hash function).
- Typically implemented using **hash tables**.
- **Complexity**: Average-case insertion, deletion, and search are all $O(1)$, but worst-case is $O(n)$ (due to hash collisions).

Example: `std::map`

```cpp
#include <map>

std::map<std::string, int> ages;
ages["Alice"] = 25;
ages["Bob"] = 30;
ages.insert({"Charlie", 35});

// Access elements.
std::cout << ages["Alice"] << std::endl; // 25
std::cout << ages.at("Bob") << std::endl; // 30

// Iterate over elements (sorted by key).
for (const auto& [name, age] : ages) {
    std::cout << name << ": " << age << std::endl;
}
```

Example: `std::set`

```cpp
#include <set>

std::set<int> numbers = {5, 2, 8, 2, 1};
// Automatically sorted and duplicates removed: {1, 2, 5, 8}

numbers.insert(3); // {1, 2, 3, 5, 8}
numbers.erase(2);  // {1, 3, 5, 8}

// Check if element exists.
if (numbers.find(5) != numbers.end()) {
    std::cout << "5 is in the set" << std::endl;
}

std::cout << numbers.count(3) << std::endl; // Either 1 or 0.

// Iterate over elements (sorted).
for (int num : numbers) {
    std::cout << num << " ";
}
```

Example: `std::unordered_map`

```cpp
#include <unordered_map>

std::unordered_map<std::string, int> scores;
scores["Alice"] = 95;
scores["Bob"] = 87;
scores["Charlie"] = 92;

// Fast average O(1) lookup.
std::cout << scores["Alice"] << std::endl; // 95

// Iterate (order not guaranteed).
for (const auto& [name, score] : scores) {
    std::cout << name << ": " << score << std::endl;
}
```

Performance comparison

Operation	`vector`	`list`	`deque`	`map`	`unordered_map`
Random access	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$ avg
Insert/delete at end	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$ avg
Insert/delete at beginning	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$ avg
Insert/delete in middle	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$ avg
Search	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$ avg

💡 **Note**: `list` and `map` have higher per-element memory overhead due to pointers/nodes.

Container adaptors

Container adaptors provide restricted interfaces to underlying containers:

- `std::stack<T>`: LIFO (Last In, First Out) structure.
  - Operations: `push()`, `pop()`, `top()`.
  - Default underlying container: `std::deque<T>`.
- `std::queue<T>`: FIFO (First In, First Out) structure.
  - Operations: `push()`, `pop()`, `front()`, `back()`.
  - Default underlying container: `std::deque<T>`.
- `std::priority_queue<T>`: Elements retrieved based on priority (largest element first by default).
  - Operations: `push()`, `pop()`, `top()`.
  - Default underlying container: `std::vector<T>`.

Example: `std::stack`

```cpp
#include <stack>

std::stack<int> s;
s.push(1);
s.push(2);
s.push(3);

std::cout << s.top() << std::endl; // 3
s.pop();
std::cout << s.top() << std::endl; // 2
```

Example: `std::priority_queue`

```cpp
#include <queue>

std::priority_queue<int> pq;
pq.push(30);
pq.push(10);
pq.push(50);
pq.push(20);

while (!pq.empty()) {
    std::cout << pq.top() << " "; // 50 30 20 10
    pq.pop();
}
```

Special containers (1/3)

- `std::pair<T1, T2>`: Stores two heterogeneous values.

```
std::pair<int, std::string> p{1, "one"};
std::cout << p.first << ", " << p.second << std::endl;
```

- `std::tuple<T1, T2, ...>`: Stores multiple heterogeneous values.

```
std::tuple<int, std::string, double> t{1, "one", 1.0};
std::cout << std::get<0>(t) << std::endl; // 1
```

- `std::optional<T>` (C++17): May or may not contain a value.

  Useful for dealing with datasets with missing data.

```
std::optional<int> opt = 42;
if (opt.has_value()) std::cout << *opt << std::endl;
```

Special containers (2/3)

- **`std::variant<T1, T2, ...>`** (C++17): Type-safe union.

```
std::variant<int, double, std::string> v = 42;
v = 3.14;
v = "hello";

// Check which type is currently held.
if (std::holds_alternative<std::string>(v)) {
  std::cout << "Holds string: " << std::get<std::string>(v) << std::endl;
}

// Safe access using std::get_if (returns nullptr if wrong type).
if (auto* ptr = std::get_if<double>(&value)) {
  std::cout << "Double: " << *ptr << std::endl;
}

// Use std::visit to handle all possible types.
std::visit([](const auto &arg) {
  std::cout << "Value: " << arg << std::endl;
}, v);
```

Special containers (3/3)

- `std::any` (C++17): Can hold any type.

```
std::any a = 42;
a = std::string("hello");

// Check type and extract value safely.
if (a.has_value()) {
  if (a.type() == typeid(std::string)) {
    std::cout << std::any_cast<std::string>(a) << std::endl;
 }
}

// Direct cast (throws exception if wrong type).
std::cout << std::any_cast<std::string>(a) << std::endl;
```

Iterators

What are iterators?

Iterators are objects that allow traversal through the elements of a container. They act as a bridge between containers and algorithms, providing a uniform interface for accessing elements.

Iterator categories (from least to most powerful):

1. **Input iterators**: Read-only, single-pass (e.g., reading from input stream).
2. **Output iterators**: Write-only, single-pass (e.g., writing to output stream).
3. **Forward iterators**: Read/write, multi-pass, forward direction only.
4. **Bidirectional iterators**: Forward + backward movement (e.g., `std::list`).
5. **Random access iterators**: Direct access to any element (e.g., `std::vector`).

Iterator example

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

// Using iterators.
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << " ";
}

// Using const iterators.
for (std::vector<int>::const_iterator it = vec.cbegin();
     it != vec.cend(); ++it) {
    std::cout << *it << " ";
}

// Reverse iterators.
for (auto rit = vec.rbegin(); rit != vec.rend(); ++rit) {
    std::cout << *rit << " "; // 5 4 3 2 1
}
```

Common iterator operations

```
std::vector<int> vec = {10, 20, 30, 40, 50};

auto it = vec.begin();
std::cout << *it << std::endl;          // 10 (dereference)

++it;                                   // Move to next element.
std::cout << *it << std::endl;          // 20

it += 2;                                // Jump forward (random access only).
std::cout << *it << std::endl;          // 40

auto distance = std::distance(vec.begin(), it); // 3
std::cout << distance << std::endl;

std::advance(it, 1);                    // Move forward by 1.
std::cout << *it << std::endl;          // 50
```

Iterator invalidation ⚠️

Modifying containers can invalidate iterators:

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

for (auto it = vec.begin(); it != vec.end(); ++it) {
    if (*it == 3) {
        vec.erase(it);  // ⚠️ Invalidates 'it'!
        // Correct: it = vec.erase(it);
    }
}
```

💡 **Rule**: After modifying a container, update or re-obtain iterators.

Algorithms

STL algorithms overview

The `<algorithm>` header provides a rich collection of functions for:

- **Searching**: `find`, `binary_search`, `lower_bound`, `upper_bound`
- **Sorting**: `sort`, `stable_sort`, `partial_sort`
- **Modifying**: `copy`, `move`, `transform`, `replace`, `fill`
- **Removing**: `remove`, `remove_if`, `unique`
- **Partitioning**: `partition`, `stable_partition`
- **Min/Max**: `min`, `max`, `min_element`, `max_element`
- **Numeric**: `accumulate`, `inner_product`, `partial_sum` (in `<numeric>`)

💡 Most algorithms work with iterator ranges `[begin, end)`.

Algorithm categories

1. **Non-modifying algorithms**: Inspect but don't change elements
   - `find`, `count`, `all_of`, `any_of`, `none_of`
2. **Modifying algorithms**: Change elements or container structure
   - `copy`, `transform`, `replace`, `remove`, `reverse`
3. **Sorting algorithms**: Order elements
   - `sort`, `stable_sort`, `partial_sort`, `nth_element`
4. **Set algorithms**: Operate on sorted ranges
   - `set_union`, `set_intersection`, `set_difference`
5. **Numeric algorithms**: Mathematical operations (in `<numeric>`)
   - `accumulate`, `inner_product`, `adjacent_difference`

Example: Searching algorithms

```cpp
#include <algorithm>
#include <vector>

std::vector<int> vec = {1, 2, 3, 4, 5};

// find: Linear search.
auto it = std::find(vec.begin(), vec.end(), 3);
if (it != vec.end()) {
    std::cout << "Found: " << *it << std::endl;
}

// count: Count occurrences.
int count = std::count(vec.begin(), vec.end(), 3);
std::cout << "Count: " << count << std::endl;

// binary_search: Requires sorted range.
bool found = std::binary_search(vec.begin(), vec.end(), 3);
std::cout << "Binary search: " << found << std::endl;
```

Example: Sorting algorithms

```cpp
#include <algorithm>
#include <vector>

std::vector<int> vec = {5, 2, 8, 1, 9};

// sort: Average O(n log n).
std::sort(vec.begin(), vec.end());
// vec is now: {1, 2, 5, 8, 9}

// Sort in descending order.
std::sort(vec.begin(), vec.end(), std::greater<int>());
// vec is now: {9, 8, 5, 2, 1}

// Custom comparator.
std::sort(vec.begin(), vec.end(), [](int a, int b) {
    return std::abs(a) < std::abs(b);
});
```

Example: Modifying algorithms

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::vector<int> result(5);

// copy: Copy elements.
std::copy(vec.begin(), vec.end(), result.begin());

// transform: Apply function to each element.
std::transform(vec.begin(), vec.end(), result.begin(),
               [](int x) { return x * 2; });
// result is now: {2, 4, 6, 8, 10}

// replace: Replace all occurrences.
std::replace(vec.begin(), vec.end(), 3, 99);
// vec is now: {1, 2, 99, 4, 5}
```

Example: Removing algorithms

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

// remove: Moves elements to end, returns new logical end.
auto new_end = std::remove(vec.begin(), vec.end(), 2);
vec.erase(new_end, vec.end()); // Actually remove them.
// vec is now: {1, 3, 4, 5}

// remove_if: Remove elements satisfying predicate.
vec = {1, 2, 3, 4, 5, 6};
new_end = std::remove_if(vec.begin(), vec.end(),
                         [](int x) { return x % 2 == 0; });
vec.erase(new_end, vec.end());
// vec is now: {1, 3, 5}
```

Example: Numeric algorithms

```cpp
#include <numeric>
#include <vector>

std::vector<int> vec = {1, 2, 3, 4, 5};

// accumulate: Sum of elements.
int sum = std::accumulate(vec.begin(), vec.end(), 0);
std::cout << "Sum: " << sum << std::endl; // 15

// Product of elements.
int product = std::accumulate(vec.begin(), vec.end(), 1,
                              std::multiplies<int>());
std::cout << "Product: " << product << std::endl; // 120

// inner_product: Dot product.
std::vector<int> vec2 = {1, 2, 3, 4, 5};
int dot = std::inner_product(vec.begin(), vec.end(), vec2.begin(), 0);
std::cout << "Dot product: " << dot << std::endl; // 55
```

Why use STL algorithms?

Comparing manual loops vs STL algorithms:

```
// Manual loop (error-prone, verbose).
std::vector<int> vec = {1, 2, 3, 4, 5};
int sum = 0;
for (size_t i = 0; i < vec.size(); ++i) {
    sum += vec[i];
}

// STL algorithm (concise, expressive).
int sum = std::accumulate(vec.begin(), vec.end(), 0);
```

🔥 **Benefits**: More readable, less error-prone, potentially optimized, supports parallel execution (C++17).

Parallel algorithms (C++17)

Many algorithms support execution policies for parallel/vectorized execution:

```cpp
#include <execution>
#include <algorithm>

std::vector<int> vec(1'000'000);
std::iota(vec.begin(), vec.end(), 0);

// Sequential execution.
std::sort(std::execution::seq, vec.begin(), vec.end()); // Same as: std::sort(vec.begin(), vec.end()).

// Parallel execution.
std::sort(std::execution::par, vec.begin(), vec.end());

// Parallel + vectorized execution.
std::sort(std::execution::par_unseq, vec.begin(), vec.end());
```

Top 10 must-know algorithms

For this course, prioritize mastering these algorithms:

1. `std::sort`: Sorting elements
2. `std::find` / `std::find_if`: Searching for elements
3. `std::copy`: Copying elements between containers
4. `std::transform`: Applying functions to ranges
5. `std::accumulate`: Summing or reducing elements
6. `std::count` / `std::count_if`: Counting elements
7. `std::remove` / `std::remove_if`: Removing elements
8. `std::for_each`: Applying function to each element
9. `std::max_element` / `std::min_element`: Finding extrema
10. `std::reverse`: Reversing element order

Common STL pitfalls

1. **Iterator invalidation**: Modifying containers while iterating

```
for (auto it = vec.begin(); it != vec.end(); ++it) {
    vec.push_back(42); // ⚠️ Invalidates iterators!
}
```

2. **Copying large containers**: Use references or move semantics

```
void process(std::vector<int> v);        // ❌ Copies entire vector.
void process(const std::vector<int>& v); // ✅ No copy.
```

3. **Using `[]` on maps**: Creates element if key doesn't exist

```
std::map<int, std::string> m;
std::cout << m[1]; // ⚠️ Creates entry with key 1.
// Use m.at(1) or m.find(1) for safer access.
```

Common STL pitfalls (continued)

4. **Forgetting to erase after remove**

```
vec.remove(vec.begin(), vec.end(), 5); // ❌ Doesn't actually remove.
vec.erase(std::remove(vec.begin(), vec.end(), 5), vec.end()); // ✅
```

5. **Comparing iterators from different containers**

```
std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = {1, 2, 3};
if (v1.begin() == v2.begin()) { /* ⚠️ Undefined behavior! */ }
```

6. **Not checking return values**

```
auto it = map.find(key);
std::cout << it->second; // ⚠️ Check 'if (it != map.end())' first!
```

Evolution of the STL

Essential vs advanced features

Essential (must master):

- Containers: `vector`, `map`, `set`, `array`
- Iterators: basic usage and iteration patterns
- Algorithms: `sort`, `find`, `transform`, `copy`, `accumulate`
- Range-based for loops
- Smart pointers: `unique_ptr`, `shared_ptr`

Advanced:

- `forward_list`, `deque`
- Custom allocators
- Parallel algorithms
- Ranges library (C++20)
- Concepts (C++20)

Useful references

- cppreference.com - Comprehensive C++ reference
- Hacking C++ - Many visual cheat sheets
- ISO C++ - Official C++ standards site
- C++ Core Guidelines - As the name says
- C++20/17/14/11 - List of modern features for each standard version, with examples
- Compiler Explorer - Test and analyze C++ code online

C++11

1. Move semantics
2. Variadic templates
3. Rvalue references
4. Lambda expressions
5. `auto`
6. `nullptr`
7. Range-based for loops
8. Smart pointers
9. Type traits
10. ...

C++14

1. Binary literals
2. Generic lambdas
3. Return type deduction
4. `decltype(auto)`
5. Variable templates
6. User-defined literals for standard library types
7. ...

C++17

1. Template argument deduction for class templates
2. Fold expressions
3. Lambda capture `this` by value
4. Structured bindings
5. `constexpr if`
6. UTF-8 character literals
7. New library features like `std::variant`, `std::optional`, and `std::any`
8. ...

C++20

1. Coroutines
2. Concepts
3. Ranges
4. Modules
5. Designated initializers
6. Template syntax for lambdas
7. `constexpr` virtual functions
8. New library features, including `std::span` and math constants
9. ...

C++23

1. **Concepts in STL**: Further adoption of concepts in STL algorithms and containers.
2. **Improved parallelism**: Expanding parallel algorithms and enhancing support for parallel execution.
3. **Reflection**: Potential support for reflection, making it easier to inspect and manipulate types at runtime.
4. **Networking library**: The Networking library might become part of the standard, adding networking capabilities.
5. **Enhanced ranges**: Expanding and refining the ranges library with new utilities.
6. ...

Source

Example: the evolution of `for` loops (1/5)

1. Traditional `for` loop (pre-C++11)

```
std::vector<int> vec = {1, 2, 3, 4, 5};
for (size_t i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << " ";
}
```

- **Usage**: Commonly used before C++11.
- **Explanation**: This form uses an integer index to access elements. It works with any indexable container (like `std::vector` or `std::array`).
- **Pros**: Allows direct access to both the index and the element, which is useful when you need the position of elements.
- **Cons**: Can be error-prone with boundary conditions, and may be inefficient if the container recalculates `size()` each time.

Example: the evolution of `for` loops (2/5)

2. `for` loop with iterators (pre-C++11)

```
std::vector<int> vec = {1, 2, 3, 4, 5};
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << " ";
}
```

- **Usage**: A step toward more generic and flexible loops.
- **Explanation**: Uses iterators instead of an index, making it adaptable to all STL containers, including those without random-access iterators (like `std::list`).
- **Pros**: Supports non-indexable containers and is more generic.
- **Cons**: Verbose syntax and the need for dereferencing (`*it`) can make code harder to read.

Example: the evolution of `for` loops (3/5)

3. Range-based `for` loop (C++11)

```
std::vector<int> vec = {1, 2, 3, 4, 5};
for (int value : vec) { // Or: for (const auto &value : vec)
    std::cout << value << " ";
}
```

- **Usage**: Introduced in C++11, this syntax is much more concise and readable.
- **Explanation**: Automatically loops over elements in the container without the need for iterators or indices.
- **Pros**: Simple, concise, and avoids off-by-one errors. Great for most use cases where you just need the element value.
- **Cons**: Limited flexibility - doesn't provide access to the index, and modifying elements requires using a reference (`for (int &value : vec)`).

Example: the evolution of `for` loops (4/5)

4. Structured bindings (C++17)

```
std::tuple<int, std::string, double> my_tuple{1, "a string", 2.5};
const auto [i, s, d] = my_tuple; // Unpack tuple.

std::map<int, std::string> my_map = {{1, "one"}, {2, "two"}};
for (const auto& [key, value] : my_map) {
    std::cout << key << ": " << value << "\n";
}
```

- **Usage**: Simplifies code when looping over key-value pairs in associative containers.
- **Explanation**: Introduced in C++17, structured bindings allow unpacking of key-value pairs directly within the loop.
- **Pros**: Very readable and works well with `std::map` and `std::unordered_map`.
- **Cons**: Limited to associative containers; doesn't add much benefit when used with containers like `std::vector`.

Example: the evolution of `for` loops (5/5)

5. Range-based `for` with `std::ranges` (C++20)

```
#include <ranges>

for (int value : vec | std::views::reverse) {
    std::cout << value << " ";
}
```

- **Usage**: Introduced with the ranges library in C++20.
- **Explanation**: Adds flexibility by allowing modifications (like `reverse`, `filter`, `transform`) directly in the loop using range adaptors.
- **Pros**: Makes the loop more expressive and reduces the need for external functions to modify sequences.
- **Cons**: Requires understanding of range adaptors and may not be necessary for simpler loops.

Hands-on challenge

Try refactoring this code to use STL algorithms:

```
// Before: Manual loop
std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<int> evens;
for (int n : nums) {
    if (n % 2 == 0) {
        evens.push_back(n * 2);
    }
}
```

Hint

Use `std::copy_if` with `std::back_inserter` and `std::transform`.
Or, if you have a C++20-ready compiler, use `std::ranges::copy_if` with `std::views::filter` and `std::views::transform`.

Conclusion

The STL is a fundamental part of the C++ standard library, offering a rich set of data structures, algorithms, and utilities that make C++ a powerful and expressive language. To fully harness the power of the STL:

1. **Algorithm usage**: Algorithms are the backbone of the STL. Utilize them to simplify and optimize common operations, enhancing code readability and maintainability.
2. **Container selection**: Choose the appropriate container type (e.g., `std::vector`, `std::map`, `std::queue`) based on your specific needs. This decision greatly impacts your code's efficiency.
3. **Smart pointers**: Smart pointers like `std::shared_ptr` and `std::unique_ptr` are crucial for effective memory management, preventing memory leaks and resource leaks.
4. **Newer features**: Stay up-to-date with the latest C++ standards (e.g., C++20, C++23) and incorporate new features like ranges, concepts, and structured bindings to write cleaner and more efficient code.

➡️ Smart pointers, move semantics, utilities from the STL.