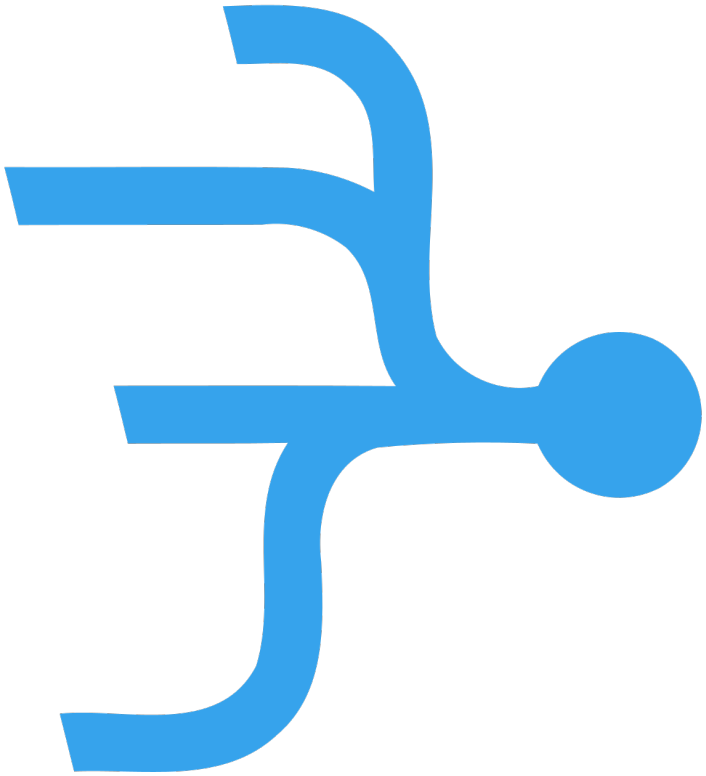
# C++ STL



readbytes

# C++ STL

## Guide to Containers, Algorithms, and Iterators

readbytes.github.io

2025-07-23

This page is intentionally left blank.

# Contents


**1 Introduction to the STL** **9**
- 1.1 What is the STL? History and Overview . . . 9
- 1.2 Key Components: Containers, Iterators, Algorithms, and Functors . . . 10
  - 1.2.1 Modular Design: Plug and Play . . . 11
- 1.3 Setting up Your Environment and Compiling STL Code . . . 11
- 1.4 Your First STL Program: Using `std::vector` . . . 13

**2 STL Containers: Sequence Containers Basics** **16**
- 2.1 Overview of Sequence Containers . . . 16
- 2.2 `std::vector`: Dynamic Arrays and Usage Examples . . . 17
- 2.3 `std::deque`: Double-Ended Queues . . . 18
- 2.4 `std::list`: Doubly Linked Lists . . . 20
- 2.5 Comparing Sequence Containers: Performance and Use Cases . . . 22
  - 2.5.1 Conclusion . . . 23

**3 STL Containers: Associative Containers** **25**
- 3.1 Overview of Associative Containers . . . 25
- 3.2 `std::set` and `std::multiset`: Ordered Unique and Duplicate Keys . . . 26
- 3.3 `std::map` and `std::multimap`: Key-Value Storage . . . 28
- 3.4 Insertion, Deletion, and Lookup Operations . . . 30
- 3.5 Practical Examples: Counting Unique Words, Phonebook . . . 31
  - 3.5.1 Summary . . . 33

**4 Unordered Containers (Hash-Based Containers)** **35**
- 4.1 Introduction to Hashing and Unordered Containers . . . 35
  - 4.1.1 Summary . . . 36
- 4.2 `std::unordered_set` and `std::unordered_multiset` . . . 36
- 4.3 `std::unordered_map` and `std::unordered_multimap` . . . 38
  - 4.3.1 Summary . . . 40
- 4.4 Comparing Ordered vs Unordered Containers: When to Use What . . . 40
- 4.5 Examples: Fast Lookup and Grouping Data . . . 41
  - 4.5.1 Why Choose Unordered Containers Here? . . . 43
  - 4.5.2 Summary . . . 43

**5 Iterators: The STLs Abstraction Layer** **45**
- 5.1 What Are Iterators? Types and Categories . . . 45
  - 5.1.1 Summary . . . 46
- 5.2 Using Iterators with Containers . . . 46
  - 5.2.1 Summary . . . 47
- 5.3 Iterator Operations and Validity Rules . . . 48
  - 5.3.1 Best Practices . . . 49
  - 5.3.2 Summary . . . 49
- 5.4 Reverse Iterators and Constant Iterators . . . 49

5.4.1 Summary . . . . . 51
5.5 Iterator Examples: Traversing and Modifying Containers . . . . . 51
5.5.1 Summary . . . . . 53

**6 STL Algorithms: Fundamentals 55**
6.1 Overview of STL Algorithms . . . . . 55
6.1.1 Summary . . . . . 56
6.2 Non-modifying Algorithms: `std::for_each`, `std::find`, `std::count` . . . . 56
6.2.1 Summary . . . . . 58
6.3 Modifying Algorithms: `std::copy`, `std::transform`, `std::replace` . . . . 58
6.3.1 Iterator Requirements and Result Handling . . . . . 60
6.3.2 Summary . . . . . 60
6.4 Sorting and Searching: `std::sort`, `std::binary_search` . . . . . 60
6.4.1 Summary . . . . . 62
6.5 Using Algorithm Examples in Real Scenarios . . . . . 62
6.5.1 Full Output . . . . . 63
6.5.2 Why Use STL Algorithms? . . . . . 63
6.5.3 Summary . . . . . 64

**7 Functors, Lambdas, and Predicates in STL 66**
7.1 What Are Functors and Why Use Them? . . . . . 66
7.1.1 Summary . . . . . 67
7.2 Writing Custom Functors . . . . . 67
7.2.1 Benefits of Custom Functors . . . . . 69
7.2.2 Summary . . . . . 69
7.3 Using Lambdas with STL Algorithms . . . . . 69
7.3.1 Why Use Lambdas? . . . . . 71
7.3.2 Summary . . . . . 71
7.4 Predicate Functions: Unary and Binary Predicates . . . . . 71
7.4.1 Summary . . . . . 72
7.5 Practical Examples: Filtering, Custom Sorting . . . . . 73
7.5.1 Summary . . . . . 74

**8 Advanced Algorithms and Utilities 76**
8.1 Set Algorithms: `std::set_union`, `std::set_intersection`, `std::set_difference` 76
8.1.1 Summary . . . . . 77
8.2 Numeric Algorithms: `std::accumulate`, `std::inner_product` . . . . . 77
8.2.1 Summary . . . . . 79
8.3 Permutations and Combinations: `std::next_permutation`, `std::prev_permutation` 79
8.3.1 Summary . . . . . 80
8.4 Algorithm Adaptors and Bindings . . . . . 81
8.4.1 Summary . . . . . 82
8.5 Case Studies: Applying Algorithms in Complex Scenarios . . . . . 82
8.5.1 Design Rationale . . . . . 84
8.5.2 Summary . . . . . 84

**9 Allocators and Memory Management** **86**
9.1 Understanding Allocators in STL . . . 86
9.2 Using Custom Allocators . . . 87
9.2.1 Using Custom Allocators . . . 87
9.3 Memory Considerations and Optimization . . . 89
9.4 Examples: Custom Pool Allocator Integration . . . 90
9.4.1 Step-by-Step Pool Allocator Example . . . 90
9.4.2 Explanation and Benefits . . . 91
9.4.3 When to Use This . . . 92

**10 String and Stream Utilities** **94**
10.1 `std::string` and `std::wstring`: Basic Usage and Operations . . . 94
10.1.1 Common Operations . . . 94
10.1.2 `std::wstring` Usage . . . 95
10.1.3 Conversion Between `std::string` and `std::wstring` . . . 95
10.1.4 Performance and Memory Considerations . . . 95
10.1.5 Summary . . . 96
10.2 String Manipulation Algorithms . . . 96
10.2.1 String Manipulation Algorithms . . . 96
10.2.2 Summary . . . 97
10.3 Streams and String Streams (`std::stringstream`) . . . 97
10.3.1 Summary . . . 99
10.4 Practical Examples: Parsing, Formatting, and Serialization . . . 99
10.4.1 Summary . . . 101

**11 Tuples and Pairs** **103**
11.1 Introduction to `std::pair` and `std::tuple` . . . 103
11.1.1 Summary . . . 104
11.2 Creating, Accessing, and Using Tuples . . . 104
11.2.1 Summary . . . 105
11.3 Structured Bindings with Tuples (C++17) . . . 106
11.3.1 Benefits of Structured Bindings . . . 107
11.4 Use Cases: Returning Multiple Values, Grouping Data . . . 107
11.4.1 Summary . . . 109

**12 Smart Pointers and Resource Management in STL** **111**
12.1 Overview of `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr` . . . 111
12.1.1 Summary . . . 112
12.2 Automatic Resource Management Using Smart Pointers . . . 113
12.2.1 Summary . . . 114
12.3 Custom Deleters and Use Cases . . . 115
12.3.1 Practical Use Cases . . . 116
12.3.2 Summary . . . 116
12.4 Examples: Managing Dynamic Memory Safely . . . 116
12.4.1 Why Use Smart Pointers? . . . 118

12.4.2 Summary . . . . . 119

**13 Concurrency Utilities in STL** **121**
13.1 Thread Support Library Basics . . . . . 121
13.1.1 Summary . . . . . 122
13.2 `std::thread`: Creating and Managing Threads . . . . . 122
13.3 Mutexes, Locks, and Condition Variables . . . . . 124
13.4 Atomic Operations . . . . . 127
13.5 Practical Examples: Parallel STL Algorithms and Synchronization . . . . . 129

**14 STL in Modern C (C11 and Beyond)** **133**
14.1 Range-based For Loops with STL . . . . . 133
14.2 Move Semantics and Emplace Functions (`emplace_back`, `emplace`) . . . . . 134
14.3 `std::optional`, `std::variant`, and `std::any` Overview . . . . . 136
14.3.1 Summary . . . . . 138
14.4 Using STL with Modern Language Features for Cleaner Code . . . . . 139
14.4.1 Combined Example . . . . . 140
14.4.2 Summary . . . . . 141

**15 Performance Considerations and Best Practices** **143**
15.1 Choosing the Right Container and Algorithm . . . . . 143
15.2 Avoiding Common Pitfalls and Inefficient Patterns . . . . . 144
15.2.1 Summary . . . . . 146
15.3 Measuring and Improving STL Code Performance . . . . . 146
15.4 Case Study: Optimizing a Real-World Application with STL . . . . . 147

**16 Real-World Projects and Examples** **151**
16.1 Building a Contact Manager Using STL Containers and Algorithms . . . . . 151
16.2 Implementing a Simple Cache with `std::unordered_map` . . . . . 153
16.3 Text Processing and Analysis Using STL Algorithms . . . . . 155
16.4 Interactive CLI To-Do List Application Using STL . . . . . 157
16.5 Advanced: Custom Container Implementation . . . . . 159
16.5.1 Key Components of an STL-Compatible Container . . . . . 159
16.5.2 Example: CustomVector A Minimal Dynamic Array . . . . . 160
16.5.3 Explanation . . . . . 161
16.5.4 Conclusion . . . . . 162

# Chapter 1.

## Introduction to the STL

1. What is the STL? History and Overview
2. Key Components: Containers, Iterators, Algorithms, and Functors
3. Setting up Your Environment and Compiling STL Code
4. Your First STL Program: Using `std::vector`

# 1 Introduction to the STL

## 1.1 What is the STL? History and Overview

The **Standard Template Library (STL)** is a powerful feature of C++ that provides a set of well-designed, reusable components for managing data and performing common operations like searching, sorting, and iterating. Think of the STL as a "toolbox" full of ready-to-use data structures (like arrays, lists, and maps) and algorithms (like sort and find), all designed to work seamlessly together.

### Origins and Purpose

The STL was developed in the early 1990s by **Alexander Stepanov**, whose goal was to create a flexible and efficient library based on the principles of **generic programming**. This means writing code that can work with any data type—without sacrificing performance. The STL became part of the C++ Standard in **1998** (C++98), marking a major milestone in the evolution of the language.

Before STL, developers often had to write their own data structures and algorithms from scratch. This not only led to repetitive work but also to bugs and inconsistencies. STL changed this by providing a **standardized** way to handle common programming tasks, making code easier to read, maintain, and reuse.

### Why STL Matters

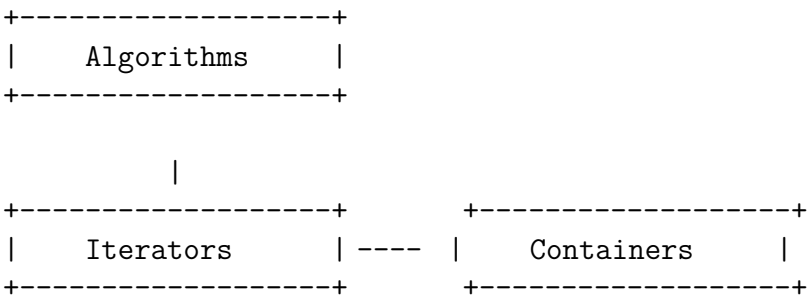
The STL is built around **type-safe templates**, which allow functions and classes to operate with generic types while maintaining strict compile-time type checking. This helps catch errors early and eliminates the need for messy type conversions or unsafe pointer arithmetic.

Some key benefits of using STL include:

- **Code reuse**: Write once, use with any data type.
- **Performance**: Efficient algorithms and data structures tuned for real-world use.
- **Consistency**: Unified interface across different components.
- **Interoperability**: Containers, algorithms, and iterators are designed to work together seamlessly.

### The STL Ecosystem

Imagine STL as a triangle made up of three pillars:



- **Containers** store data (e.g., `std::vector`, `std::map`).
- **Iterators** provide a uniform way to access container elements.
- **Algorithms** perform operations on data through iterators (e.g., `std::sort`, `std::find`).

Together, these components form a flexible, modular framework that simplifies many programming tasks. As you progress through this book, you'll see how the STL can help you write cleaner, faster, and more robust C++ programs.

## 1.2 Key Components: Containers, Iterators, Algorithms, and Functors

The Standard Template Library (STL) is built on four main components: **Containers**, **Iterators**, **Algorithms**, and **Functors**. Each plays a unique role, but together they form a cohesive, modular system that simplifies many common programming tasks in C++. Understanding these components is key to mastering the STL.

### Containers: Holding Your Data

**Containers** are data structures that store collections of objects. STL provides several types, such as:

- `std::vector` – dynamic array
- `std::list` – doubly linked list
- `std::map` – key-value pairs (sorted)
- `std::set` – unique sorted elements

Each container is implemented as a template class, allowing you to store any data type:

```
#include <vector>
std::vector<int> numbers = {1, 2, 3, 4};
```

Containers abstract away memory management and low-level data handling, allowing you to focus on logic rather than implementation.

### Iterators: Navigating Data

**Iterators** act like generalized pointers that allow you to traverse and manipulate the elements of a container. They unify access to different types of containers in a consistent way.

```
for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
    std::cout << *it << " ";
}
```

Modern C++ also supports range-based for loops, but understanding iterators is essential because STL algorithms rely on them.

### Algorithms: Reusable Data Operations

**Algorithms** are generic functions that perform operations on containers via iterators. The STL includes over 60 algorithms for tasks like searching, sorting, copying, and modifying data.

```
#include <algorithm>
std::sort(numbers.begin(), numbers.end());
```

Here, `std::sort` doesn't care what container it's operating on—it simply uses iterators. This generic design makes STL algorithms extremely flexible and reusable.

### Functors: Custom Behavior

**Functors** (short for function objects) are objects that behave like functions. They allow you to define custom operations in a way that integrates smoothly with algorithms.

```
struct IsEven {
    bool operator()(int x) const { return x % 2 == 0; }
};

#include <algorithm>
auto it = std::find_if(numbers.begin(), numbers.end(), IsEven());
```

You can also use **lambda expressions** as inline functors in modern C++:

```
auto it = std::find_if(numbers.begin(), numbers.end(), [](int x) { return x % 2 == 0; });
```

### 1.2.1 Modular Design: Plug and Play

The beauty of the STL lies in its **modular design**. Containers manage data, iterators give access to it, algorithms perform operations on it, and functors customize those operations. Since each component is designed independently but works together through generic interfaces, you can "mix and match" them effortlessly.

This design promotes **code reuse**, **flexibility**, and **type safety**, making the STL a robust foundation for efficient C++ programming.

## 1.3 Setting up Your Environment and Compiling STL Code

Before diving into STL programming, you'll need to set up a C++ development environment. Fortunately, the STL is part of the C++ Standard Library, so you don't need to install anything extra beyond a C++ compiler and an editor or IDE.

### Step 1: Install a C Compiler

Choose a compiler based on your platform:

- **Windows**
  - **MSVC (Microsoft Visual C++)** via Visual Studio
    - Install the "Desktop development with C++" workload.
  - **MinGW-w64 (g++)**: Lightweight GCC port for Windows (https://www.mingw-w64.org/)
- **Linux**
  - Install **GCC (g++)** or **Clang**:

```
sudo apt install g++
# or
sudo apt install clang
```

- **macOS**
  - Install **Xcode Command Line Tools**:

```
xcode-select --install
```

### Step 2: Choose an IDE or Editor

Popular options include:

- **Visual Studio** (Windows) – Full-featured IDE with MSVC
- **Visual Studio Code** – Lightweight and cross-platform, with C++ extension support
- **CLion** – Cross-platform IDE from JetBrains (commercial)
- **Qt Creator**, **Code::Blocks**, or **Eclipse CDT**

Use whatever you're comfortable with, but ensure it supports configuring and compiling C++ projects.

### Step 3: Create and Compile a Sample STL Program

**File: `main.cpp`**

Full runnable code:

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

int main() {
    std::vector<int> nums = {4, 2, 5, 1, 3};
    std::sort(nums.begin(), nums.end());

    for (int n : nums)
        std::cout << n << " ";
    return 0;
}
```

**Compiling (Command Line):**

- g++

```
g++ -std=c++11 -o app main.cpp
./app
```

- **clang++**

```
clang++ -std=c++11 -o app main.cpp
./app
```

- **MSVC (Developer Command Prompt)**

```
cl main.cpp
```

### Troubleshooting Tips

- Use `-std=c++11` or higher (e.g., `c++17`) to access modern STL features.
- Ensure environment variables (like `PATH` for g++) are correctly set.
- If using VS Code, install the C++ extension and configure tasks for build/run.

Once your environment is ready, you're all set to explore the STL!

## 1.4 Your First STL Program: Using `std::vector`

Let's write our first C++ STL program using `std::vector`, one of the most commonly used containers. A `vector` is a dynamic array that can grow or shrink in size at runtime. It supports fast access and flexible element management, making it a great starting point for STL programming.

### Step-by-Step Program

Create a file named `main.cpp` and add the following code:

Full runnable code:

```cpp
#include <iostream>     // For std::cout
#include <vector>       // For std::vector

int main() {
    // Step 1: Declare a vector of integers
    std::vector<int> numbers;

    // Step 2: Insert elements into the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);
    numbers.push_back(40);

    // Step 3: Print the vector size
    std::cout << "Vector size: " << numbers.size() << std::endl;

    // Step 4: Iterate and print elements using index
    std::cout << "Elements using index access:" << std::endl;
```

```
    for (size_t i = 0; i < numbers.size(); ++i) {
        std::cout << "numbers[" << i << "] = " << numbers[i] << std::endl;
    }

    // Step 5: Iterate using range-based for loop
    std::cout << "Elements using range-based for loop:" << std::endl;
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

### Explanation of Key Parts

- `#include <vector>`: Includes the STL vector class.
- `std::vector<int>`: Declares a vector that stores integers.
- `push_back()`: Adds elements to the end of the vector.
- `size()`: Returns the number of elements in the vector.
- Index-based and range-based for loops: Two ways to iterate through the vector.

### Compiling and Running

### For g++ or clang++:

```
g++ -std=c++11 -o vector_demo main.cpp
./vector_demo
```

### Expected Output:

```
Vector size: 4
Elements using index access:
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
Elements using range-based for loop:
10 20 30 40
```

### What You Learned

- How to declare and use a `std::vector`
- How to insert and access elements
- How to compile and run a basic STL program

This program demonstrates the foundational structure of STL usage: choose a container, populate it, and use simple iteration to process its elements. As we move forward, you'll explore more containers and powerful STL algorithms to manipulate them.

# Chapter 2.

# STL Containers: Sequence Containers Basics

1. Overview of Sequence Containers
2. `std::vector`: Dynamic Arrays and Usage Examples
3. `std::deque`: Double-Ended Queues
4. `std::list`: Doubly Linked Lists
5. Comparing Sequence Containers: Performance and Use Cases

# 2 STL Containers: Sequence Containers Basics

## 2.1 Overview of Sequence Containers

**Sequence containers** in the C++ Standard Template Library (STL) are designed to store elements in a **linear** arrangement—meaning the order in which elements are inserted is preserved. These containers support operations like iteration, insertion, and deletion and are ideal for handling lists, queues, buffers, or any other ordered collection of elements.

There are **three primary sequence containers** in STL:

- `std::vector`
- `std::deque`
- `std::list`

Each has unique strengths and trade-offs depending on the performance characteristics you need.

### Container Introductions

- **`std::vector`**: A dynamic array that provides fast **random access** and efficient **insertion/removal at the end**. It stores elements in a contiguous memory block, making it cache-friendly and compatible with raw arrays.
- **`std::deque`** (*double-ended queue*): Supports fast **insertion and deletion at both ends**. Unlike `vector`, it may not store elements contiguously but allows efficient expansion at both the front and back.
- **`std::list`**: A **doubly linked list** with fast **insertion and deletion at any position**, especially in the middle. However, it does **not support random access**, and traversing elements is slower due to pointer-based structure.

### Comparison Table

| Feature | `std::vector` | `std::deque` | `std::list` |
|---|---|---|---|
| Memory layout | Contiguous | Non-contiguous | Node-based |
| Random access | YES Fast (`O(1)`) | YES Fast (`O(1)`) | NO (`O(n)`) |
| Insert/remove at end | YES Fast (`O(1)` amort.) | YES Fast (`O(1)`) | YES Fast (`O(1)`) |
| Insert/remove at front | NO Slow (`O(n)`) | YES Fast (`O(1)`) | YES Fast (`O(1)`) |
| Insert/remove in middle | NO Slow (`O(n)`) | NO Moderate (`O(n)`) | YES Fast (`O(1)`) |
| Cache friendliness | YES High | WARNING Medium | NO Low |
| Best use cases | Arrays, buffers | Queues, sliding windows | Frequent mid-inserts |

### Choosing the Right Container

- Use **`vector`** when you need fast random access and frequent growth at the end.
- Use **`deque`** for **double-ended queues** or when you need fast front and back operations.
- Use **`list`** when your application involves **frequent insertions or deletions in the middle** of the collection.

Sequence containers are the building blocks for many STL programs. Choosing the right one based on your performance needs can significantly improve efficiency and clarity in your code. As we explore each container in detail, you'll see how their internal design affects real-world usage.

## 2.2 `std::vector`: Dynamic Arrays and Usage Examples

The `std::vector` container is one of the most widely used sequence containers in the C++ Standard Template Library. It acts like a **dynamic array**, automatically managing memory as elements are added or removed, while providing **fast random access** and **contiguous memory storage**.

### Key Features of `std::vector`

- **Dynamic resizing:** Unlike a regular array with fixed size, a vector can grow or shrink at runtime. When you add elements beyond its current capacity, it automatically allocates more memory.
- **Contiguous memory:** Elements in a vector are stored sequentially in memory, which makes vector very cache-friendly and allows efficient pointer arithmetic.
- **Random access:** You can access any element in constant time using the subscript operator `[]` or the `.at()` method.
- **Efficient end insertions:** Adding elements at the end (`push_back()`) is usually fast, with occasional reallocations that double capacity.
- **Size vs Capacity:**
  - **Size** is the number of elements currently stored.
  - **Capacity** is the amount of allocated memory available before resizing is needed.

### Basic Usage Example

Full runnable code:

```
#include <iostream>
#include <vector>

int main() {
    // Create an empty vector of integers
```

```cpp
    std::vector<int> numbers;

    // Insert elements at the end
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Print size and capacity
    std::cout << "Size: " << numbers.size() << ", Capacity: " << numbers.capacity() << "\n";

    // Access elements using random access
    std::cout << "Element at index 1: " << numbers[1] << std::endl;

    // Resize vector to contain 5 elements
    numbers.resize(5);  // New elements initialized to 0 by default

    std::cout << "After resizing to 5 elements:" << std::endl;
    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Explanation:**

- `push_back()` adds elements to the end, increasing size.
- `size()` returns current number of elements.
- `capacity()` shows allocated storage — may be larger than size.
- `resize()` changes the size. If increased, new elements are default-initialized.
- Accessing elements by index is fast and simple.

### When to Use `std::vector`

Vectors are ideal when:

- You need **fast random access** to elements.
- Your collection size may change but most insertions are at the **end**.
- Memory locality and performance matter (e.g., numerical computations, buffers).
- You want a simple, flexible container with low overhead.

By mastering `std::vector`, you gain a versatile tool that fits a wide range of programming needs. Next, we'll explore `std::deque`, which offers some additional flexibility for front-end operations.

## 2.3 `std::deque`: Double-Ended Queues

The **`std::deque`** (double-ended queue) is a versatile sequence container in the C++ Standard Library designed for efficient insertions and deletions at **both the front and the back** of

the container. This contrasts with `std::vector`, which offers fast insertions only at the back.

### How `std::deque` Differs from `std::vector`

While both `deque` and `vector` support random access and store elements in order, their **internal structures** differ:

- `std::vector` stores elements in a **single contiguous block** of memory, making it very cache-friendly but slow for insertions at the front (which require shifting elements).
- `std::deque` uses a **sequence of fixed-size memory blocks** linked together, allowing it to efficiently add or remove elements at both ends without relocating all elements.

This internal structure allows `deque` to grow dynamically on either side, providing more flexibility than `vector`.

### Conceptual Diagram of `std::deque`

```
[Block 1] <-> [Block 2] <-> [Block 3] <-> ... <-> [Block N]
|           |    |           |    |           |            |
Elements    Elements        Elements         Elements
```

Each block holds a chunk of elements. This arrangement supports quick front and back operations.

### Basic Usage Example

Full runnable code:

```cpp
#include <iostream>
#include <deque>

int main() {
    std::deque<int> d;

    // Insert elements at the back
    d.push_back(10);
    d.push_back(20);

    // Insert elements at the front
    d.push_front(5);
    d.push_front(2);

    // Print all elements
    std::cout << "Deque elements: ";
    for (int n : d) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Remove elements from front and back
    d.pop_front();  // removes 2
    d.pop_back();   // removes 20

    std::cout << "After pop operations: ";
    for (int n : d) {
```

```
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Random access
    std::cout << "Element at index 0: " << d[0] << std::endl;

    return 0;
}
```

### Performance Considerations

| Operation | `std::vector` | `std::deque` |
|---|---|---|
| Insert/remove at back | Fast (`O(1)` amortized) | Fast (`O(1)`) |
| Insert/remove at front | Slow (`O(n)`) | Fast (`O(1)`) |
| Random access | Fast (`O(1)`) | Fast (`O(1)`) |
| Memory layout | Contiguous | Non-contiguous |
| Cache friendliness | High | Moderate |

Because `deque` is not contiguous in memory, iterating over it may be slightly slower than a vector, but its flexibility for front-end operations makes it an excellent choice for **queues**, **double-ended buffers**, or when you need efficient insertion/removal at both ends.

In summary, `std::deque` combines many benefits of `vector` and `list`, making it a flexible container when you need quick insertions/removals at either end without sacrificing random access.

## 2.4 `std::list`: Doubly Linked Lists

The **`std::list`** container implements a **doubly linked list**, a sequence of elements where each element points to both its previous and next neighbor. This structure makes it especially efficient for frequent insertions and deletions **anywhere** in the list—even in the middle—without needing to move other elements.

### Characteristics of `std::list`

- Unlike `vector` or `deque`, `std::list` **does not provide random access** (no direct indexing like `list[2]`).
- Accessing elements requires sequential traversal, which means accessing the $n$-th element takes time proportional to $n$ (`O(n)`).
- However, inserting or removing elements is very fast (`O(1)`) once you have an iterator to the position.

### When to Use `std::list`

Use `std::list` when:

- Your program requires **frequent insertions or deletions in the middle** of the sequence.
- You need stable iterators and pointers—elements won't be invalidated by insertions or removals elsewhere.
- Random access speed is less critical than flexibility in modifying the container.

### Basic Usage Example

Full runnable code:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> numbers = {10, 20, 30};

    // Insert at the front and back
    numbers.push_front(5);
    numbers.push_back(40);

    // Insert in the middle using an iterator
    auto it = numbers.begin();
    std::advance(it, 2);  // Move iterator to third element (value 20)
    numbers.insert(it, 15);  // Insert 15 before 20

    // Remove an element
    numbers.remove(30);  // Removes all occurrences of 30

    // Traverse and print
    std::cout << "List elements: ";
    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

### Why `std::list` Lacks Random Access

Because elements are stored as nodes linked via pointers, they are **not contiguous in memory**. This prevents direct access by index and requires stepping through each node sequentially. While this makes `list` slower for indexed access, it provides unmatched efficiency for modifying the list structure.

In summary, `std::list` is a powerful container when you need flexible insertion and removal at arbitrary positions in your sequence. For scenarios involving heavy middle insertions or deletions, it outperforms vectors and deques, but if you need fast indexed access, other containers like `vector` are preferable.

## 2.5 Comparing Sequence Containers: Performance and Use Cases

Choosing the right sequence container—`std::vector`, `std::deque`, or `std::list`—depends on your program's **performance needs** and **usage patterns**. Each container has different strengths and trade-offs related to insertion, deletion, memory layout, and access speed.

**Performance Summary**

| Operation | `std::vector` | `std::deque` | `std::list` |
|---|---|---|---|
| **Random access** | Fast (`O(1)`) | Fast (`O(1)`) | Slow (`O(n)`) |
| **Insertion at back** | Fast (`O(1)` amort.) | Fast (`O(1)`) | Fast (`O(1)`) |
| **Insertion at front** | Slow (`O(n)`) | Fast (`O(1)`) | Fast (`O(1)`) |
| **Insertion in middle** | Slow (`O(n)`) | Moderate (`O(n)`) | Fast (`O(1)`) |
| **Deletion at back** | Fast (`O(1)`) | Fast (`O(1)`) | Fast (`O(1)`) |
| **Deletion at front** | Slow (`O(n)`) | Fast (`O(1)`) | Fast (`O(1)`) |
| **Deletion in middle** | Slow (`O(n)`) | Moderate (`O(n)`) | Fast (`O(1)`) |
| **Memory layout** | Contiguous | Non-contiguous | Non-contiguous |
| **Cache friendliness** | High | Moderate | Low |

**When to Use Each Container**

`std::vector`

- Ideal when you need **fast random access** and expect most insertions/removals at the **end**.
- Perfect for numerical computations, buffers, and static-size collections that occasionally grow.
- Example: Storing a list of scores or processing a large array of sensor data.

`std::deque`

- Best when you need fast insertions/removals at **both ends** but still want efficient random access.
- Useful for implementing queues, sliding windows, or double-ended buffers.
- Example: Maintaining a list of recent events with frequent additions/removals at front and back.

`std::list`

- Suited for scenarios with frequent **insertions and deletions in the middle** of the sequence.
- Avoids costly element shifting but sacrifices random access speed.
- Example: Managing playlists or editing text buffers where items are frequently inserted or removed anywhere.

**Practical Scenario Comparison**

Imagine you need to maintain a collection of tasks:

| Scenario | Recommended Container | Reason |
|---|---|---|
| Fast lookup by position, occasional growth | `std::vector` | Contiguous storage, quick indexing |
| Tasks added/removed frequently at both ends | `std::deque` | Efficient front/back insertions |
| Frequent insertions/deletions in middle | `std::list` | Fast splicing without copying elements |

### 2.5.1 Conclusion

Understanding the strengths and weaknesses of these containers allows you to **choose the best tool for your data structure needs**. While `vector` often suffices due to its simplicity and speed, knowing when `deque` or `list` outperforms it is crucial for writing efficient, maintainable C++ code. This foundation will help you apply STL containers confidently in your projects.

# Chapter 3.

## STL Containers: Associative Containers

1. Overview of Associative Containers
2. `std::set` and `std::multiset`: Ordered Unique and Duplicate Keys
3. `std::map` and `std::multimap`: Key-Value Storage
4. Insertion, Deletion, and Lookup Operations
5. Practical Examples: Counting Unique Words, Phonebook

# 3 STL Containers: Associative Containers

## 3.1 Overview of Associative Containers

Associative containers are a core part of the C++ Standard Template Library (STL) designed to store **elements organized by keys**, enabling **fast and efficient lookup, insertion, and deletion**. Unlike sequence containers, which store elements in a linear order, associative containers automatically **keep their elements sorted** based on the keys, allowing quick searching through balanced tree structures.

### What Are Associative Containers?

Associative containers store data as **key-value pairs** or **unique keys**, depending on the container type. They manage their elements in a way that keeps the keys in sorted order internally—this guarantees efficient operations such as finding or inserting an element with logarithmic time complexity (`O(log n)`).

Because the containers keep the keys sorted automatically, you do **not** need to sort the data yourself before performing lookups or other operations.

### Main Types of Associative Containers

1. `std::set`
    - Stores **unique keys** (no duplicates).
    - The keys themselves are the elements.
    - Example: A collection of unique user IDs.
2. `std::multiset`
    - Stores **keys that may appear multiple times** (duplicates allowed).
    - Useful when you want to track counts or multiple instances of the same key.
3. `std::map`
    - Stores **key-value pairs** with **unique keys**.
    - Keys are used to access their associated values.
    - Example: A phonebook mapping names (keys) to phone numbers (values).
4. `std::multimap`
    - Similar to `map` but allows **duplicate keys**.
    - Multiple values can be associated with the same key.
    - Example: Storing multiple phone numbers for the same person.

### Unique Keys vs Duplicate Keys

- **Unique-key containers** (`set`, `map`) guarantee that each key appears only once.
- **Duplicate-key containers** (`multiset`, `multimap`) allow multiple elements with the same key.

This distinction helps choose the right container based on whether your data allows duplicates or not.

Associative containers are powerful tools when you need **fast, ordered retrieval** of data keyed by unique or repeating values. They underpin many practical applications like databases, indexing, and caching, where efficient search and sorted data management are essential. In the following sections, we will explore these containers in detail along with practical usage examples.

## 3.2 `std::set` and `std::multiset`: Ordered Unique and Duplicate Keys

The **`std::set`** and **`std::multiset`** containers are associative containers that store **sorted keys**, but they differ in how they handle duplicates.

### `std::set`: Unique Sorted Keys

A `std::set` stores **unique elements** in sorted order. If you try to insert an element that already exists, the set ignores it, ensuring no duplicates are present.

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> unique_numbers;

    unique_numbers.insert(5);
    unique_numbers.insert(1);
    unique_numbers.insert(3);
    unique_numbers.insert(5);  // Duplicate, will be ignored

    std::cout << "Set elements: ";
    for (int n : unique_numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Searching for an element
    if (unique_numbers.find(3) != unique_numbers.end()) {
        std::cout << "3 found in set." << std::endl;
    }

    return 0;
}
```

Output:

```
Set elements: 1 3 5
3 found in set.
```

### `std::multiset`: Allowing Duplicate Keys

Unlike `set`, a `std::multiset` allows **multiple identical keys**. Elements are still stored in sorted order, but duplicates are preserved.

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <set>

int main() {
    std::multiset<int> numbers;

    numbers.insert(5);
    numbers.insert(1);
    numbers.insert(3);
    numbers.insert(5);  // Duplicate allowed

    std::cout << "Multiset elements: ";
    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output:

```
Multiset elements: 1 3 5 5
```

### Typical Use Cases

- Use `std::set` when you need to **maintain a collection of unique elements**, such as unique user IDs, distinct words, or configuration options.
- Use `std::multiset` when you need to **track frequencies or allow duplicates**, like counting occurrences of words or storing multiple values that can be equal.

### Performance Characteristics

Both `set` and `multiset` typically implement balanced binary search trees (like red-black trees), providing:

- **Insertion, removal, and lookup** in `O(log n)` time.
- Automatic sorting of elements.
- No direct random access (you must iterate or search via keys).

In summary, `std::set` and `std::multiset` are powerful tools when you want sorted collections with or without duplicates, offering efficient search and ordered iteration.

## 3.3 `std::map` and `std::multimap`: Key-Value Storage

The **`std::map`** and **`std::multimap`** containers are associative containers designed to store **key-value pairs** in sorted order based on their keys. They differ mainly in how they handle duplicate keys: `map` allows only **unique keys**, while `multimap` permits **multiple entries with the same key**.

### `std::map`: Unique Keys with Associated Values

A `std::map` stores pairs where each **key is unique** and mapped to a single value. It maintains the keys in sorted order internally, enabling efficient lookup, insertion, and deletion, all with average logarithmic time complexity (`O(log n)`).

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> phonebook;

    // Inserting key-value pairs
    phonebook["Alice"] = 12345;
    phonebook["Bob"] = 67890;
    phonebook.insert({"Charlie", 54321});

    // Lookup by key
    std::string name = "Bob";
    if (phonebook.find(name) != phonebook.end()) {
        std::cout << name << "'s number is " << phonebook[name] << std::endl;
    }

    // Iterating over all entries (sorted by key)
    std::cout << "Phonebook entries:\n";
    for (const auto& entry : phonebook) {
        std::cout << entry.first << ": " << entry.second << "\n";
    }

    return 0;
}
```

Output:

```
Bob's number is 67890
Phonebook entries:
Alice: 12345
Bob: 67890
Charlie: 54321
```

### `std::multimap`: Allowing Duplicate Keys

`std::multimap` allows multiple key-value pairs with the **same key**, making it useful when a key may correspond to multiple values.

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <map>

int main() {
    std::multimap<std::string, int> phonebook;

    phonebook.insert({"Alice", 12345});
    phonebook.insert({"Bob", 67890});
    phonebook.insert({"Bob", 11111});  // Duplicate key allowed

    std::cout << "Bob's numbers:\n";
    auto range = phonebook.equal_range("Bob");
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->second << "\n";
    }

    return 0;
}
```

Output:

```
Bob's numbers:
67890
11111
```

### Underlying Structure

Both `map` and `multimap` are typically implemented as **balanced binary search trees** (e.g., red-black trees). This structure keeps keys sorted and ensures:

- Efficient **insertion**, **deletion**, and **lookup** in `O(log n)` time.
- Ordered iteration by key.
- No direct random access (elements must be accessed via iterators or by key search).

### Common Use Cases

- **`std::map`** is ideal for implementing dictionaries, symbol tables, or any scenario where each key maps to a single unique value.
- **`std::multimap`** suits cases like a phonebook where a person might have multiple phone numbers or when multiple values naturally share the same key.

In summary, `map` and `multimap` provide powerful, ordered key-value storage with predictable performance, making them essential tools for associative data management in C++.

## 3.4 Insertion, Deletion, and Lookup Operations

Associative containers like `std::set`, `std::multiset`, `std::map`, and `std::multimap` provide powerful methods to **insert**, **delete**, and **find** elements efficiently. Understanding how to use these operations correctly is crucial for working effectively with these containers.

### Insertion

- **`insert()`** is the primary method to add elements.
- For **`set` and `map`**, `insert()` returns a pair containing an iterator to the inserted element (or the existing element if it was a duplicate) and a bool indicating success.
- For **`multiset` and `multimap`**, duplicates are allowed, so all inserts succeed.

**Example:**

```
std::set<int> s;
auto result = s.insert(10);  // Inserts 10
if (!result.second) {
    std::cout << "Element already exists\n";
}

std::multiset<int> ms;
ms.insert(10);
ms.insert(10);  // Allowed duplicate
```

### Deletion

- Use **`erase()`** to remove elements.
- You can erase by:
  - **Key:** `container.erase(key)` removes all matching elements (for `set` and `map`, this is one element since keys are unique).
  - **Iterator:** `container.erase(iterator)` removes the element at that position.
  - **Range:** `container.erase(startIterator, endIterator)` removes a range of elements.

**Example:**

```
std::set<int> s = {1, 2, 3, 4};
s.erase(2);  // Removes key 2

auto it = s.find(3);
if (it != s.end()) {
    s.erase(it);  // Removes element at iterator it
}
```

### Lookup

- **`find(key)`** returns an iterator to the element if found, or `end()` if not.
- For `multimap` and `multiset`, `equal_range(key)` returns a pair of iterators representing the range of matching elements.

**Example:**

```cpp
std::map<std::string, int> m = {{"apple", 5}, {"banana", 3}};
auto it = m.find("banana");
if (it != m.end()) {
    std::cout << "Found banana with value " << it->second << "\n";
} else {
    std::cout << "Banana not found\n";
}
```

### Iterator Invalidation Rules

- For associative containers, **insertion and deletion do not invalidate iterators** to other elements, except for iterators to erased elements.
- This means you can safely iterate while inserting or deleting other elements.

### Common Mistakes to Avoid

- **Ignoring return values from `insert()`** — especially with `set` and `map`, since duplicates won't be inserted.
- **Erasing elements by invalid iterators** — always check if an iterator is valid before erasing.
- **Assuming random access** — associative containers don't support indexing like arrays or vectors.
- Using **`find()` result without checking** if it equals `end()` — always verify before dereferencing.

Mastering these basic operations will make your use of associative containers robust and efficient. In the next section, we'll explore practical examples to deepen your understanding.

## 3.5 Practical Examples: Counting Unique Words, Phonebook

To see associative containers in action, let's explore two common real-world applications: **counting word frequencies** in a text, and building a simple **phonebook**. We will use `std::map` (and briefly mention `std::unordered_map`) for efficient key-value storage.

### Example 1: Counting Unique Words

Suppose you want to count how many times each word appears in a paragraph. `std::map<std::string, int>` is perfect for this because it stores words as keys and their counts as values, automatically sorted by the word.

Full runnable code:

```cpp
#include <iostream>
#include <map>
#include <sstream>

int main() {
    std::string text = "hello world hello STL hello map world";
```

```
    std::map<std::string, int> wordCount;

    std::istringstream iss(text);
    std::string word;

    // Read each word and update count
    while (iss >> word) {
        ++wordCount[word];  // Increment count for the word
    }

    // Print the results
    std::cout << "Word frequencies:\n";
    for (const auto& pair : wordCount) {
        std::cout << pair.first << ": " << pair.second << "\n";
    }

    return 0;
}
```

**Explanation:**

- We use an input string stream (`std::istringstream`) to split the text into words.
- Each word is used as a key in the map, and the associated value is incremented.
- Finally, we iterate over the map to display words in alphabetical order with their counts.

  **Note:** If you don't require the sorted order, `std::unordered_map` offers faster average insertion and lookup (`O(1)`), but the order of keys is not maintained.

### Example 2: Simple Phonebook Application

A phonebook stores names and associated phone numbers. Using `std::map<std::string, std::string>`, we can easily manage the entries with fast lookup.

Full runnable code:

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, std::string> phonebook;

    // Insert entries
    phonebook["Alice"] = "555-1234";
    phonebook["Bob"] = "555-5678";
    phonebook.insert({"Charlie", "555-8765"});

    // Lookup a number
    std::string name = "Bob";
    auto it = phonebook.find(name);
    if (it != phonebook.end()) {
        std::cout << name << "'s phone number is " << it->second << "\n";
    } else {
        std::cout << name << " not found in the phonebook.\n";
    }

    // Iterate and print all entries
```

```
    std::cout << "\nFull phonebook:\n";
    for (const auto& entry : phonebook) {
        std::cout << entry.first << ": " << entry.second << "\n";
    }

    return 0;
}
```

**Explanation:**

- We add entries using both `operator[]` and `insert()`.
- `find()` locates a key and returns an iterator; if the key doesn't exist, it returns `end()`.
- Iteration displays all contacts sorted by name.

### 3.5.1 Summary

These examples demonstrate how associative containers simplify managing collections keyed by strings or other types. Counting word frequencies or building lookup tables like phonebooks are common tasks that benefit from their **automatic sorting**, **fast search**, and **easy insertion/deletion**. Mastering these patterns will empower you to solve many practical programming challenges efficiently.

# Chapter 4.

# Unordered Containers (Hash-Based Containers)

1. Introduction to Hashing and Unordered Containers
2. `std::unordered_set` and `std::unordered_multiset`
3. `std::unordered_map` and `std::unordered_multimap`
4. Comparing Ordered vs Unordered Containers: When to Use What
5. Examples: Fast Lookup and Grouping Data

# 4 Unordered Containers (Hash-Based Containers)

## 4.1 Introduction to Hashing and Unordered Containers

Hashing is a powerful technique used to **quickly locate data** in a large collection by transforming keys into indices using a special function called a **hash function**. The C++ Standard Template Library (STL) provides **unordered containers** that use hashing to offer **average constant time complexity** (`O(1)`) for operations like insertion, lookup, and deletion.

### What Is Hashing?

At its core, hashing involves:

- Taking a **key** (like a string or integer).
- Applying a **hash function** to convert that key into a numerical value, often called a **hash code**.
- Using this hash code to find the **bucket** where the element should be stored.

Because this process directly maps keys to buckets, it avoids the need to search through the whole collection, enabling fast access.

### Hash Table Structure and Buckets

Hash tables organize elements into **buckets**—containers where multiple elements with hash codes mapping to the same bucket are stored.

Here's a conceptual diagram of a hash table with buckets:

```
Hash Table:

Index: 0   1   2   3   4   5
Buckets: [A] [C,D] [ ] [B] [ ] [E,F]
```

- Keys A, B, C, D, E, and F have been hashed.
- Elements C and D share the same bucket (index 1), illustrating a **collision**.

### Collisions and How They Are Handled

Since many keys may hash to the same bucket, **collisions** are inevitable. Common collision resolution techniques include:

- **Chaining**: Storing multiple elements in the same bucket using linked lists or similar structures (used by STL unordered containers).
- **Open addressing**: Searching for another empty bucket within the table (not used in STL containers).

Chaining allows unordered containers to maintain efficient lookup times even with collisions by only searching within the small bucket.

**Unordered Containers in STL**

STL provides unordered alternatives to ordered associative containers:

- `std::unordered_set` and `std::unordered_multiset` store **unique and duplicate keys**, respectively.
- `std::unordered_map` and `std::unordered_multimap` store **key-value pairs** with unique or duplicate keys.

Because these containers do **not maintain order**, iteration over elements can appear random, but the trade-off is much faster average lookups.

**Trade-offs of Using Unordered Containers**

| Advantage | Disadvantage |
|---|---|
| Average `O(1)` lookup time | No ordering of elements |
| Fast insertion and deletion | Performance degrades with poor hash function or many collisions |
| Ideal for large datasets needing fast access | Slightly more memory overhead due to buckets and chains |

### 4.1.1 Summary

Hashing and hash tables form the foundation of unordered containers in the STL. By sacrificing ordering, these containers provide lightning-fast average access times, making them ideal for applications like caching, symbol tables, or any scenario requiring quick membership tests. Understanding the structure and behavior of hash-based containers is essential for efficient C++ programming.

## 4.2 `std::unordered_set` and `std::unordered_multiset`

The containers **`std::unordered_set`** and **`std::unordered_multiset`** are part of the STL's hash-based collection family designed for fast storage and retrieval of elements **without any guaranteed order**.

**What Are They?**

- **`std::unordered_set`** stores **unique elements** — no duplicates allowed.
- **`std::unordered_multiset`** allows **multiple identical elements** (duplicates).

Both provide **average constant time complexity (`O(1)`)** for operations like insertion, lookup, and deletion, thanks to their underlying hash table implementation.

### Basic Usage Example

Full runnable code:

```cpp
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> unique_numbers = {10, 20, 30};

    // Insert elements
    unique_numbers.insert(40);
    unique_numbers.insert(20);  // Duplicate ignored

    // Lookup
    if (unique_numbers.find(30) != unique_numbers.end()) {
        std::cout << "30 found in unordered_set\n";
    }

    // Iterate and print elements (order is unspecified)
    std::cout << "unordered_set elements: ";
    for (int n : unique_numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // unordered_multiset example
    std::unordered_multiset<int> numbers = {10, 20, 20, 30};
    numbers.insert(20);

    std::cout << "unordered_multiset elements: ";
    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

### Possible output:

```
30 found in unordered_set
unordered_set elements: 40 10 20 30
unordered_multiset elements: 20 30 10 20 20
```

### Typical Use Cases

- **`unordered_set`** is ideal for **membership tests**, like checking if an element exists quickly (e.g., a blacklist or dictionary lookup).
- **`unordered_multiset`** is useful when counting or storing multiple instances of elements, such as tracking item frequencies or duplicate entries.

### Differences from Ordered Counterparts

- Both unordered containers **do not keep elements sorted**, unlike `set` and `multiset`.
- Iteration order in unordered containers is **unspecified and can change** as elements

are added or removed.
- Unordered containers typically offer **faster average lookup and insertion** compared to ordered containers (`O(1)` vs `O(log n)`), but with potential performance variation due to hashing collisions.

In summary, `std::unordered_set` and `std::unordered_multiset` are excellent choices when you need **fast, unordered storage** of unique or duplicate elements, especially for large datasets where lookup speed is critical.

## 4.3 `std::unordered_map` and `std::unordered_multimap`

The containers **`std::unordered_map`** and **`std::unordered_multimap`** are hash-based associative containers designed to store **key-value pairs** with **fast average lookup times** using hashing. They provide efficient insertion, searching, and updating, making them ideal for tasks such as caching, indexing, and quick lookups.

### `std::unordered_map`: Unique Keys with Fast Access

`std::unordered_map` stores **unique keys**, each mapped to a single value. Internally, it uses a hash table to convert keys into bucket indices, enabling average **constant-time (`O(1)`)** complexity for operations like insertion and search.

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> scores;

    // Insert key-value pairs
    scores["Alice"] = 95;
    scores["Bob"] = 82;

    // Update a value
    scores["Alice"] = 97;  // Overwrites previous value

    // Search for a key
    std::string name = "Bob";
    auto it = scores.find(name);
    if (it != scores.end()) {
        std::cout << name << "'s score is " << it->second << std::endl;
    } else {
        std::cout << name << " not found.\n";
    }

    return 0;
}
```

Output:

```
Bob's score is 82
```

### `std::unordered_multimap`: Allowing Duplicate Keys

Unlike `unordered_map`, `std::unordered_multimap` allows **multiple values with the same key**. This is useful when keys may correspond to several entries.

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_multimap<std::string, int> scores;

    // Insert multiple scores for the same key
    scores.insert({"Alice", 95});
    scores.insert({"Bob", 82});
    scores.insert({"Bob", 88});  // Duplicate key allowed

    // Retrieve all values for "Bob"
    auto range = scores.equal_range("Bob");
    std::cout << "Bob's scores:\n";
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->second << "\n";
    }

    return 0;
}
```

Output:

```
Bob's scores:
82
88
```

### Performance and Typical Applications

Both containers provide:

- **Fast average insertion and lookup (`O(1)`)**, outperforming ordered containers (`O(log n)`) when ordering is not required.
- Efficient memory usage with hash buckets and chaining for collision resolution.

Typical uses include:

- **Caching**: Quickly retrieving computed results by key.
- **Lookup tables**: Mapping identifiers to values in compilers, interpreters, or databases.
- **Grouping data**: Associating multiple values with keys using `unordered_multimap`.

### 4.3.1 Summary

`std::unordered_map` and `std::unordered_multimap` offer powerful, hash-based key-value storage solutions with rapid access. While `unordered_map` enforces unique keys, `unordered_multimap` allows duplicates, making them versatile tools for a wide range of programming tasks where performance and efficient key-based lookup matter most.

## 4.4 Comparing Ordered vs Unordered Containers: When to Use What

The C++ Standard Template Library offers both **ordered** and **unordered** associative containers, each optimized for different scenarios. Understanding their trade-offs helps you choose the right container based on your performance and functionality needs.

**Ordered Containers (`std::map`, `std::set`)**

- **Structure:** Implemented as balanced binary search trees (typically red-black trees).
- **Ordering:** Automatically keep elements sorted by key.
- **Performance:** Operations like insertion, lookup, and deletion have logarithmic time complexity — `O(log n)`.
- **Use Cases:**
  - When **sorted iteration or ordered traversal** is required.
  - If you need to efficiently find elements by **range queries** or ordered comparisons (e.g., all keys between X and Y).
  - Example: A leaderboard sorted by scores, or a dictionary where entries need to be printed alphabetically.

**Unordered Containers (`std::unordered_map`, `std::unordered_set`)**

- **Structure:** Use hash tables with buckets.
- **Ordering: No guaranteed order** of elements; iteration order can appear random and change as the container grows.
- **Performance:** Offer average **constant time (`O(1)`)** for insertions, lookups, and deletions, making them faster for large datasets where order is not important.
- **Use Cases:**
  - When **fast membership tests** or lookups are a priority.
  - For caching, symbol tables, or when you need to check presence quickly without caring about order.
  - Example: Checking if a user ID exists in a large set or retrieving a cached result by key.

### Practical Examples

- If you need to **print all entries sorted by key**, use `std::map` or `std::set` to benefit from automatic ordering.
- If you need to **quickly check if a value exists** without caring about order, `std::unordered_set` or `std::unordered_map` is faster and more efficient.

### Summary Table

| Feature | Ordered Containers (`map`, `set`) | Unordered Containers (`unordered_map`, `unordered_set`) |
|---|---|---|
| Internal Structure | Balanced trees | Hash tables |
| Ordering | Keys sorted | No guaranteed order |
| Lookup/Insert/Delete Complexity | O(log n) | Average O(1) |
| Use Case Example | Sorted outputs, range queries | Fast membership tests, caching |

Choosing between ordered and unordered containers depends on whether **order matters more than speed** or vice versa. Often, unordered containers are preferred for performance-critical sections, while ordered containers shine when sorted data or range operations are needed.

## 4.5 Examples: Fast Lookup and Grouping Data

To solidify your understanding of unordered containers, let's explore two practical examples that showcase their power: **fast membership tests** using `std::unordered_set` and **grouping data by category** with `std::unordered_map`.

### Example 1: Fast Membership Test with `std::unordered_set`

Suppose you want to quickly check if certain items are part of a predefined collection, like a list of banned usernames.

Full runnable code:

```
#include <iostream>
#include <unordered_set>
#include <string>

int main() {
    std::unordered_set<std::string> bannedUsers = {"alice", "bob", "charlie"};

    std::string userInput;
    std::cout << "Enter username to check: ";
```

```
    std::cin >> userInput;

    if (bannedUsers.find(userInput) != bannedUsers.end()) {
        std::cout << userInput << " is banned.\n";
    } else {
        std::cout << userInput << " is allowed.\n";
    }

    return 0;
}
```

**How it works:**

- The `unordered_set` stores unique usernames.
- `find()` uses hashing to perform a near-instant lookup, regardless of the number of banned users.
- This example illustrates how unordered containers are perfect for **fast membership testing**.

### Example 2: Grouping Data by Category with `std::unordered_map`

Now, imagine you have a list of products, each belonging to a category, and you want to group product names by category for quick access.

Full runnable code:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>

int main() {
    std::unordered_map<std::string, std::vector<std::string>> productsByCategory;

    // Adding products
    productsByCategory["Electronics"].push_back("Smartphone");
    productsByCategory["Electronics"].push_back("Laptop");
    productsByCategory["Furniture"].push_back("Chair");
    productsByCategory["Furniture"].push_back("Table");
    productsByCategory["Clothing"].push_back("T-shirt");

    // Display products grouped by category
    for (const auto& pair : productsByCategory) {
        std::cout << pair.first << ":\n";
        for (const auto& product : pair.second) {
            std::cout << "  - " << product << "\n";
        }
    }

    return 0;
}
```

**How it works:**

- `unordered_map` keys are category names (strings).

- Each key maps to a `vector` holding product names.
- The hash function quickly locates the category bucket to add or retrieve products.
- This example highlights **grouping data efficiently** with constant-time average access.

### 4.5.1 Why Choose Unordered Containers Here?

- Both examples benefit from **average constant-time lookups**.
- The order of elements isn't critical, so hash-based containers provide better performance than ordered ones.
- The bucket structure behind the scenes enables fast insertion and retrieval by hashing keys.

### 4.5.2 Summary

These examples illustrate the strength of unordered containers in real-world scenarios: quickly checking if an element exists, and grouping items by keys for efficient access. Mastering these patterns lets you write clean, fast C++ code that leverages the full power of hashing.

# Chapter 5.

# Iterators: The STLs Abstraction Layer

1. What Are Iterators? Types and Categories
2. Using Iterators with Containers
3. Iterator Operations and Validity Rules
4. Reverse Iterators and Constant Iterators
5. Iterator Examples: Traversing and Modifying Containers

# 5 Iterators: The STLs Abstraction Layer

## 5.1 What Are Iterators? Types and Categories

Iterators are one of the most fundamental concepts in the C++ Standard Template Library (STL). You can think of an **iterator** as a **generalized pointer** that provides a uniform way to access and traverse elements inside various containers, like vectors, lists, or maps. Unlike raw pointers that work only with arrays, iterators abstract the underlying container's structure, allowing generic algorithms to operate seamlessly on different container types.

### What is an Iterator?

At its core, an iterator acts like a pointer to an element inside a container. It supports operations to:

- **Access** the element it points to.
- **Move** to the next element.
- **Compare** itself with other iterators.

This abstraction lets algorithms work with any container without knowing its internal layout.

### Iterator Categories and Their Capabilities

Iterators come in different **categories**, each supporting specific operations and traversal capabilities. These categories build on each other, with each higher category supporting all operations of the previous ones plus additional features.

| Category | Description | Supported Operations |
|---|---|---|
| **Input Iterator** | Read elements in one direction, one pass only | `*it`, `++it` |
| **Output Iterator** | Write elements in one direction, one pass only | `*it = value`, `++it` |
| **Forward Iterator** | Read/write, multiple passes allowed | Input + Output operations, multi-pass |
| **Bidirectional Iterator** | Can move forward and backward | Forward + `--it` |
| **Random Access Iterator** | Supports all above + constant-time jump to any element | `it + n`, `it - n`, `it[n]`, `<`, `>`, arithmetic |

### Visualizing Iterator Traversal

Imagine a container's elements laid out in a sequence:

```
Elements:  [A] - [B] - [C] - [D] - [E]
Iterator:        ^
```

- An **input iterator** can start at the first element and move forward step-by-step (`++it`), reading values as it goes.

- A **bidirectional iterator** can move both forward (`++it`) and backward (`--it`), so it can traverse elements in either direction.
- A **random access iterator** behaves like a pointer in an array: it can jump multiple steps forward or backward (`it + 3`) and supports direct indexing (`it[2]`).

**Why Are These Categories Important?**

Knowing the category of an iterator tells you what algorithms you can use efficiently. For example:

- Sorting requires **random access iterators** because it needs fast jumps.
- Traversing a linked list only needs **bidirectional iterators** since you cannot jump arbitrarily.
- Reading from a stream supports only **input iterators** since data flows forward.

### 5.1.1 Summary

Iterators provide a powerful, unified way to access container elements without exposing their internal structure. The different iterator categories define the level of access and navigation possible, allowing STL algorithms to work efficiently with a wide variety of data structures. Understanding these categories is key to mastering STL's flexibility and power.

## 5.2 Using Iterators with Containers

Iterators are the gateway to accessing and manipulating container elements in the STL. Each container provides member functions to obtain iterators that mark the beginning and the end of its sequence, enabling traversal and element access in a consistent way.

**Obtaining Iterators**

You can get iterators pointing to the first and one-past-the-last elements using:

- `container.begin()` — returns an iterator to the first element.
- `container.end()` — returns an iterator one position beyond the last element.

These iterators define a **half-open range** `[begin(), end())` for iteration.

**Example: Traversing a Vector with Iterators**

Full runnable code:

```
#include <iostream>
#include <vector>

int main() {
```

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

    // Obtain iterators to start and end
    std::vector<int>::iterator it = numbers.begin();
    std::vector<int>::iterator end = numbers.end();

    // Traverse using iterators
    while (it != end) {
        std::cout << *it << " ";  // Dereference iterator to access element
        ++it;                     // Move to the next element
    }

    std::cout << std::endl;
    return 0;
}
```

**Output:**

```
10 20 30 40 50
```

### Iterator Dereferencing and Access

- You use `*it` to **dereference** an iterator and access the element it points to.
- You can modify elements through non-const iterators.
- Const iterators (`const_iterator`) provide read-only access.

### Iterator Invalidation

Be aware that certain container operations can **invalidate** iterators, meaning the iterator no longer points to a valid element.

- For example, **inserting or erasing** elements in `std::vector` might cause reallocation, invalidating all iterators.
- `std::list` iterators remain valid after insertions and deletions except for those pointing directly to erased elements.
- Always check container-specific rules to avoid undefined behavior.

### 5.2.1 Summary

Using `begin()` and `end()`, iterators provide a flexible, uniform way to traverse and access container elements regardless of the container type. Understanding iterator invalidation helps you write safe, bug-free code when modifying containers during iteration. Mastering iterators is key to effectively leveraging the STL.

## 5.3 Iterator Operations and Validity Rules

Iterators act like generalized pointers, so they support a range of operations that allow you to navigate and manipulate container elements efficiently. However, understanding **iterator validity** is crucial to avoid subtle bugs.

### Common Iterator Operations

- **Increment (`++it`)**: Moves the iterator forward to the next element.
- **Decrement (`--it`)**: Moves the iterator backward (only for bidirectional or random access iterators).
- **Dereferencing (`*it`)**: Accesses the element the iterator points to.
- **Comparison (`it1 == it2`, `it1 != it2`)**: Checks whether two iterators point to the same element or position.
- **Arithmetic (`it + n`, `it - n`)**: Supported by random access iterators (e.g., vectors) to jump multiple elements.
- **Subscript (`it[n]`)**: Access element `n` positions away from the iterator (random access iterators).

Example snippet:

```
auto it = vec.begin();
++it;               // Move forward
--it;               // Move backward (if supported)
if (it != vec.end()) {
    std::cout << *it;  // Access element
}
```

### Iterator Validity Rules

An iterator is **valid** as long as it points to a valid element in a container. Certain container modifications may **invalidate** iterators, meaning the iterator no longer refers to a valid location. Using invalid iterators leads to undefined behavior.

### When Are Iterators Invalidated?

| Container | Operation | Effect on Iterators |
|---|---|---|
| `std::vector` | Insertion (push_back, insert) | May invalidate all iterators if reallocation occurs |
| | Erasure | Invalidates iterators at and after erased elements |
| `std::list` | Insertion | Does **not** invalidate iterators |
| | Erasure | Invalidates iterators only to erased elements |
| `std::deque` | Insertion at ends | Invalidates some iterators |
| | Insertion in middle | Invalidates all iterators |

**Cautionary Example**

```
std::vector<int> v = {1, 2, 3};
auto it = v.begin();
v.push_back(4);    // May cause reallocation, invalidating 'it'
std::cout << *it;  // Undefined behavior if 'it' was invalidated
```

### 5.3.1 Best Practices

- Avoid storing iterators across container modifications that might invalidate them.
- When modifying containers, consider re-obtaining iterators afterward.
- Use container-specific documentation to understand iterator invalidation guarantees.

### 5.3.2 Summary

Iterator operations enable flexible container traversal, but safe use depends on understanding **when iterators remain valid**. Keeping these rules in mind prevents bugs and keeps your STL code robust and reliable.

## 5.4 Reverse Iterators and Constant Iterators

Iterators not only let you traverse containers forward but also **backward** using **reverse iterators**. Additionally, **constant iterators** provide a way to access container elements without allowing modification, enforcing safety and clarity.

**Reverse Iterators: Traversing Backward**

Most STL containers support **reverse iterators**, which move through the container from the end toward the beginning.

- Use `rbegin()` to get a reverse iterator pointing to the **last element**.
- Use `rend()` to get a reverse iterator pointing just **before the first element**.

This creates a half-open range `[rbegin(), rend())` that you can iterate over backwards.

**Example:**

Full runnable code:

```
#include <iostream>
#include <vector>
```

```cpp
int main() {
    std::vector<int> numbers = {10, 20, 30, 40, 50};

    std::cout << "Reverse traversal: ";
    for (auto rit = numbers.rbegin(); rit != numbers.rend(); ++rit) {
        std::cout << *rit << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Output:**

```
Reverse traversal: 50 40 30 20 10
```

### Constant Iterators: Read-Only Access

A **constant iterator** (`const_iterator`) guarantees **read-only access** to container elements. Attempting to modify an element through a constant iterator will cause a compile-time error, making your intentions explicit and code safer.

You can obtain constant iterators using:

- `cbegin()` and `cend()` for forward constant iteration.
- `crbegin()` and `crend()` for reverse constant iteration.

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3};

    // Using a const_iterator to traverse read-only
    for (std::vector<int>::const_iterator cit = numbers.cbegin(); cit != numbers.cend(); ++cit) {
        std::cout << *cit << " ";
        // *cit = 5;  // Error! Cannot modify through const_iterator
    }
    std::cout << std::endl;

    return 0;
}
```

### Mixed Usage Scenario

You can mix constant and non-constant iterators depending on whether you want to allow modifications.

```cpp
std::vector<int> data = {5, 10, 15, 20};

// Modify with normal iterator
for (auto it = data.begin(); it != data.end(); ++it) {
```

```
    *it += 1;
}

// Read with const_iterator
for (auto cit = data.cbegin(); cit != data.cend(); ++cit) {
    std::cout << *cit << " ";
}
```

### 5.4.1 Summary

**Reverse iterators** offer a straightforward way to traverse containers backward, while **constant iterators** enforce safe, read-only access. Mastering both allows you to write flexible and robust STL code, adapting traversal and access to your exact needs.

## 5.5 Iterator Examples: Traversing and Modifying Containers

To understand iterators better, let's explore practical examples showing how to traverse and modify containers using different iterator types.

**Example 1: Modifying Elements in a `std::vector`**

Here, we use a normal iterator to iterate over a vector and modify each element by doubling its value.

Full runnable code:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Modify each element by doubling it
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        *it = (*it) * 2;
    }

    // Print modified vector
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Output:**

```
2 4 6 8 10
```

**Explanation:** Using `begin()` and `end()`, the iterator `it` walks through the vector. Dereferencing `*it` allows us to modify each element in-place.

### Example 2: Read-Only Traversal of a `std::map` Using Constant Iterators

Constant iterators prevent modification of elements during traversal, making them ideal for read-only access.

Full runnable code:

```cpp
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> age = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35}
    };

    // Traverse with const_iterator to prevent modification
    for (std::map<std::string, int>::const_iterator cit = age.cbegin(); cit != age.cend(); ++cit) {
        std::cout << cit->first << " is " << cit->second << " years old.\n";
        // cit->second = 40; // Error: Cannot modify via const_iterator
    }

    return 0;
}
```

**Output:**

```
Alice is 30 years old.
Bob is 25 years old.
Charlie is 35 years old.
```

### Example 3: Reverse Iteration Over a `std::vector`

Using reverse iterators, you can process elements from the end to the beginning.

Full runnable code:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> data = {10, 20, 30, 40, 50};

    std::cout << "Reverse order: ";
    for (auto rit = data.rbegin(); rit != data.rend(); ++rit) {
        std::cout << *rit << " ";
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

**Output:**

```
Reverse order: 50 40 30 20 10
```

### 5.5.1 Summary

- Use **normal iterators** to traverse and **modify** container elements.
- Use **constant iterators** when you want **read-only access** to avoid accidental changes.
- Use **reverse iterators** to traverse containers **backwards** with ease.

Mastering these iterator types lets you write clear, efficient code that adapts to different data access patterns in STL containers.

# Chapter 6.

## STL Algorithms: Fundamentals

1. Overview of STL Algorithms
2. Non-modifying Algorithms: `std::for_each`, `std::find`, `std::count`
3. Modifying Algorithms: `std::copy`, `std::transform`, `std::replace`
4. Sorting and Searching: `std::sort`, `std::binary_search`
5. Using Algorithm Examples in Real Scenarios

# 6 STL Algorithms: Fundamentals

## 6.1 Overview of STL Algorithms

The C++ Standard Template Library (STL) offers a rich collection of **algorithms**—generic, reusable functions designed to operate on data stored in containers. These algorithms work uniformly with any container type by using **iterators**, which provide a common interface to access container elements. This design decouples algorithms from specific container implementations, making your code more flexible and reusable.

### What Are STL Algorithms?

STL algorithms are function templates that perform operations such as searching, sorting, counting, transforming, and more. Rather than being tied to a specific container, algorithms accept **iterator ranges**, typically specified as `[begin, end)`. This means the same algorithm can process vectors, lists, sets, or even custom container types as long as they provide compatible iterators.

For example, to find an element in a container, you pass iterators marking the start and end, and the algorithm searches the range regardless of the underlying container type.

### Categories of STL Algorithms

STL algorithms are generally grouped into the following categories:

- **Non-modifying Algorithms** These algorithms inspect or query elements without changing the container. Examples include:
  - `std::for_each` — applies a function to each element.
  - `std::find` — searches for a value.
  - `std::count` — counts occurrences of a value.
- **Modifying Algorithms** These algorithms change the contents of containers, such as:
  - `std::copy` — copies elements from one range to another.
  - `std::transform` — applies a function to elements and stores results.
  - `std::replace` — replaces certain values.
- **Sorting and Searching Algorithms** These handle ordering and efficient lookup, including:
  - `std::sort` — sorts elements in ascending order.
  - `std::binary_search` — checks for presence of an element in sorted ranges.
- **Numeric Algorithms** Algorithms for numeric computations, like:
  - `std::accumulate` — sums or combines elements.
  - `std::partial_sum` — computes prefix sums.

### Why Are STL Algorithms Powerful?

By operating on iterators, STL algorithms are **container-agnostic**. This means you can write one piece of code that works seamlessly with different data structures without rewriting logic.

This separation of **algorithm logic** from **data storage**:

- Encourages code reuse.
- Enhances readability and maintainability.
- Enables composition of complex operations from simple building blocks.

### 6.1.1 Summary

STL algorithms are a cornerstone of modern C++ programming. They provide a versatile toolkit of operations that can be applied to a wide variety of container types via iterators. Understanding their categories and how they decouple algorithms from container details will empower you to write efficient, generic, and clean C++ code.

## 6.2 Non-modifying Algorithms: `std::for_each`, `std::find`, `std::count`

Non-modifying algorithms in the STL provide ways to **inspect, search, and analyze** container elements **without altering the underlying data**. These algorithms operate on iterator ranges and help you perform common tasks efficiently and cleanly.

### `std::for_each`: Applying a Function to Each Element

`std::for_each` applies a given function or callable to every element in a specified range. It's a convenient way to perform side effects like printing or logging without writing explicit loops.

**Example:**

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};

    std::for_each(nums.begin(), nums.end(), [](int n) {
        std::cout << n << " ";
```

```
    });

    std::cout << std::endl;
    return 0;
}
```

**Output:**

```
1 2 3 4 5
```

### `std::find`: Searching for a Value

`std::find` searches for the first occurrence of a specified value in the range `[begin, end)`. It returns an iterator to the found element or `end()` if the value is not present.

**Example:**

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {10, 20, 30, 40, 50};

    auto it = std::find(nums.begin(), nums.end(), 30);
    if (it != nums.end()) {
        std::cout << "Found 30 at position: " << std::distance(nums.begin(), it) << std::endl;
    } else {
        std::cout << "30 not found" << std::endl;
    }

    return 0;
}
```

### `std::count`: Counting Occurrences of a Value

`std::count` tallies how many times a particular value appears within a range.

**Example:**

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {1, 2, 2, 3, 2, 4, 5};

    int count_twos = std::count(nums.begin(), nums.end(), 2);
    std::cout << "Number of 2s: " << count_twos << std::endl;

    return 0;
}
```

### 6.2.1 Summary

These non-modifying algorithms are powerful tools to **inspect and query** container contents without changing them. Whether applying a function to each element with `for_each`, searching for values with `find`, or counting occurrences with `count`, these algorithms simplify common tasks and improve code readability by abstracting away manual loops.

## 6.3 Modifying Algorithms: `std::copy`, `std::transform`, `std::replace`

Modifying algorithms in the STL allow you to **change the contents of containers** or **create new containers** based on existing data. They operate on iterator ranges and often write results to another container or range, enabling powerful data transformations with minimal code.

### `std::copy`: Copying Elements

`std::copy` copies elements from a source range `[first, last)` to a destination beginning at a specified iterator. It requires that the destination has enough space to hold the copied elements.

**Example:**

Full runnable code:

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

int main() {
    std::vector<int> source = {1, 2, 3, 4, 5};
    std::vector<int> destination(source.size());  // Allocate space

    std::copy(source.begin(), source.end(), destination.begin());

    for (int n : destination) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Output:**

```
1 2 3 4 5
```

### `std::transform`: Applying Functions to Elements

`std::transform` applies a function or callable to each element in the input range and writes the result to an output iterator. This is ideal for element-wise transformations.

**Example:**

Full runnable code:

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

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::vector<int> squares(numbers.size());

    // Square each element
    std::transform(numbers.begin(), numbers.end(), squares.begin(),
                   [](int n) { return n * n; });

    for (int sq : squares) {
        std::cout << sq << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Output:**

```
1 4 9 16 25
```

### `std::replace`: Replacing Specific Values

`std::replace` modifies elements **in-place** by replacing all occurrences of a specified old value with a new value within a range.

**Example:**

Full runnable code:

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

int main() {
    std::vector<int> data = {5, 10, 5, 20, 5};

    // Replace all 5's with 50's
    std::replace(data.begin(), data.end(), 5, 50);

    for (int n : data) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Output:**

```
50 10 50 20 50
```

### 6.3.1 Iterator Requirements and Result Handling

- Both `std::copy` and `std::transform` require the **destination range** to be large enough to receive the output elements.
- `std::replace` works **in-place** and thus modifies the original container directly.
- All these algorithms operate on **iterator pairs**, allowing them to work with various container types.

### 6.3.2 Summary

Modifying algorithms like `std::copy`, `std::transform`, and `std::replace` offer concise, expressive ways to manipulate container data. Whether duplicating contents, applying transformations, or updating values, these algorithms handle common data modifications efficiently while working generically across container types via iterators.

## 6.4 Sorting and Searching: `std::sort`, `std::binary_search`

Sorting and searching are fundamental operations in programming, and the STL provides powerful, efficient algorithms to perform these tasks: `std::sort` for sorting and `std::binary_search` for searching sorted data.

#### `std::sort`: Efficient Sorting with Random Access Iterators

`std::sort` rearranges the elements in a range `[begin, end)` into ascending order by default. It uses an efficient sorting algorithm (typically introsort), providing an average and worst-case time complexity of **O(n log n)**, where $n$ is the number of elements.

**Key Requirements:**

- Requires **random access iterators** — meaning containers like `std::vector`, `std::deque`, or arrays, but **not** `std::list`.
- Elements must be **comparable** with the `<` operator or a custom comparator.

**Example:**

Full runnable code:

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>

int main() {
    std::vector<int> nums = {42, 10, 7, 99, 23};

    // Sort in ascending order
    std::sort(nums.begin(), nums.end());

    std::cout << "Sorted numbers: ";
    for (int n : nums) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Output:**

```
Sorted numbers: 7 10 23 42 99
```

### `std::binary_search`: Fast Search in Sorted Ranges

Once data is sorted, `std::binary_search` allows you to quickly check if a value exists in the range. It performs a binary search, which has a time complexity of **O(log n)**.

**Important:** Using `std::binary_search` on an **unsorted** range leads to undefined behavior and incorrect results.

**Example:**

Full runnable code:

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

int main() {
    std::vector<int> nums = {7, 10, 23, 42, 99};

    int target = 23;

    bool found = std::binary_search(nums.begin(), nums.end(), target);

    if (found) {
        std::cout << target << " is in the vector." << std::endl;
    } else {
        std::cout << target << " is NOT in the vector." << std::endl;
    }

    return 0;
}
```

### Ensuring Correctness

- Always **sort** the container before using `std::binary_search`.

- `std::sort` ensures the data is in the required order for binary search.
- Remember that both algorithms operate on iterator ranges, allowing flexibility in container choice (with the random access iterator constraint for `std::sort`).

### 6.4.1 Summary

`std::sort` is the go-to STL algorithm for efficient sorting of random access containers, with guaranteed O(n log n) performance. After sorting, `std::binary_search` offers a fast way to check for element presence using logarithmic time complexity. Together, these algorithms enable performant data organization and lookup in your C++ programs.

## 6.5 Using Algorithm Examples in Real Scenarios

STL algorithms shine most when combined to solve real-world problems efficiently and elegantly. Let's explore an example where we **filter**, **transform**, **sort**, and **search** a dataset using multiple STL algorithms.

**Scenario: Processing and Searching a List of Temperatures**

Imagine you have a list of daily temperature readings in Celsius and want to:

1. Extract only temperatures above a threshold (filter).
2. Convert these temperatures to Fahrenheit (transform).
3. Sort the converted temperatures.
4. Check if a particular temperature exists in the list.

**Step 1: Setup and Filtering**

We'll use `std::copy_if` to filter temperatures above 20°C into a new vector.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> tempsC = {18, 25, 22, 15, 30, 28, 10};
    std::vector<int> filteredTemps;

    // Filter temperatures > 20°C
    std::copy_if(tempsC.begin(), tempsC.end(), std::back_inserter(filteredTemps),
                 [](int temp) { return temp > 20; });

    std::cout << "Filtered temps (C): ";
    for (int t : filteredTemps) std::cout << t << " ";
    std::cout << std::endl;
```

### Step 2: Transform to Fahrenheit

Next, use `std::transform` to convert Celsius to Fahrenheit with the formula `F = C * 9/5 + 32`.

```
    std::vector<int> tempsF(filteredTemps.size());

    std::transform(filteredTemps.begin(), filteredTemps.end(), tempsF.begin(),
                   [](int c) { return c * 9 / 5 + 32; });

    std::cout << "Converted temps (F): ";
    for (int t : tempsF) std::cout << t << " ";
    std::cout << std::endl;
```

### Step 3: Sort the Temperatures

Sort the Fahrenheit temperatures using `std::sort`.

```
    std::sort(tempsF.begin(), tempsF.end());

    std::cout << "Sorted temps (F): ";
    for (int t : tempsF) std::cout << t << " ";
    std::cout << std::endl;
```

### Step 4: Search for a Specific Temperature

Use `std::binary_search` to check if 86°F is present (which corresponds to 30°C).

```
    int searchTemp = 86;
    bool found = std::binary_search(tempsF.begin(), tempsF.end(), searchTemp);

    if (found)
        std::cout << searchTemp << "°F is in the list." << std::endl;
    else
        std::cout << searchTemp << "°F is NOT in the list." << std::endl;

    return 0;
}
```

#### 6.5.1 Full Output

```
Filtered temps (C): 25 22 30 28
Converted temps (F): 77 71 86 82
Sorted temps (F): 71 77 82 86
86°F is in the list.
```

#### 6.5.2 Why Use STL Algorithms?

- **Modularity:** Each step is a simple function call, easy to understand and maintain.

- **Efficiency:** Algorithms are highly optimized and avoid manual loops.
- **Generic:** Works with any container supporting required iterator categories.
- **Readability:** Clear intent through well-named algorithms.

### 6.5.3 Summary

Combining STL algorithms such as filtering with `std::copy_if`, transforming with `std::transform`, sorting with `std::sort`, and searching with `std::binary_search` lets you write concise, efficient, and expressive code. This approach promotes productivity and helps solve complex data manipulation tasks with ease in real applications.

# Chapter 7.

## Functors, Lambdas, and Predicates in STL

1. What Are Functors and Why Use Them?
2. Writing Custom Functors
3. Using Lambdas with STL Algorithms
4. Predicate Functions: Unary and Binary Predicates
5. Practical Examples: Filtering, Custom Sorting

# 7 Functors, Lambdas, and Predicates in STL

## 7.1 What Are Functors and Why Use Them?

In C++, **functors**—also known as **function objects**—are objects that can be called like functions. This is possible because they overload the function call operator `operator()`. Instead of writing a standalone function, you create a class or struct that defines how the object behaves when used with parentheses, just like a function call.

### Basic Concept

Here's a simple example of a functor:

```
struct Adder {
    int value;
    Adder(int v) : value(v) {}

    int operator()(int x) const {
        return x + value;
    }
};

int main() {
    Adder add5(5);
    int result = add5(10);  // Calls add5.operator()(10)
    // result == 15
}
```

In this example, `Adder` holds a state (`value`) and uses it in the call operator to add to an input. You use the object like a function, but with the power of storing data inside it.

### Advantages of Functors

1. **Statefulness:** Unlike plain functions, functors can store internal data/state. This allows customized behavior based on member variables, which is useful in many algorithms that require context.
2. **Inline Performance:** Since functors are typically small objects and their call operators are often defined inline, compilers can optimize calls efficiently, sometimes outperforming regular function pointers.
3. **Flexibility:** Functors can have multiple overloaded call operators or other member functions, giving you more control than a simple function.
4. **Type Safety:** Functors are typed objects, enabling the compiler to enforce stronger type checks.

### Functors in STL Algorithms

The STL algorithms heavily rely on functors to provide **custom behavior**. For example, sorting algorithms like `std::sort` accept a functor (or any callable) as a comparison function to define how elements should be ordered.

```cpp
struct Descending {
    bool operator()(int a, int b) const {
        return a > b;  // Sort in descending order
    }
};

std::vector<int> v = {3, 1, 4, 2};
std::sort(v.begin(), v.end(), Descending());
```

Here, `Descending` is a functor that tells `std::sort` to arrange elements in descending order.

Using functors instead of raw function pointers allows the STL algorithms to be **generic and extensible**—users can provide their own callable objects to customize behavior while keeping the algorithms container-agnostic.

### 7.1.1 Summary

Functors are objects that act like functions by overloading `operator()`. They combine the power of objects—such as storing state and supporting inline calls—with the simplicity of functions. This makes them essential for customizing STL algorithms, enabling flexible, efficient, and reusable code designs. Understanding functors lays the foundation for mastering advanced STL features and writing clean, expressive C++ code.

## 7.2 Writing Custom Functors

Custom functors are user-defined classes or structs that overload the function call operator `operator()`, allowing them to be used like functions. Writing your own functors enables you to encapsulate complex behaviors, including state, which can be passed to STL algorithms for flexible, reusable operations.

### Creating a Custom Functor

Let's write a functor that compares strings by their length, useful for sorting strings by size instead of alphabetically.

Full runnable code:

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

struct CompareByLength {
    bool operator()(const std::string& a, const std::string& b) const {
        return a.size() < b.size();
```

```
    }
};

int main() {
    std::vector<std::string> words = {"apple", "banana", "kiwi", "cherry"};

    std::sort(words.begin(), words.end(), CompareByLength());

    for (const auto& word : words) {
        std::cout << word << " ";
    }
}
```

**Output:**

```
kiwi apple banana cherry
```

This functor enables `std::sort` to order words by length, showcasing how functors customize algorithm behavior.

### Functors with State

Functors can also store state through constructor parameters, which can influence their behavior during calls. Here's an example of a functor that checks if numbers are greater than a stored threshold:

Full runnable code:

```
#include <iostream>
#include <vector>

struct IsGreaterThan {
    int threshold;

    IsGreaterThan(int t) : threshold(t) {}

    bool operator()(int value) const {
        return value > threshold;
    }
};

int main() {
    std::vector<int> nums = {10, 25, 5, 30, 15};

    IsGreaterThan greaterThan20(20);

    int count = std::count_if(nums.begin(), nums.end(), greaterThan20);

    std::cout << "Numbers greater than 20: " << count << std::endl;
}
```

**Output:**

```
Numbers greater than 20: 2
```

Here, the functor stores the threshold value 20 and uses it in `operator()` to test each number.

This makes the functor reusable with different thresholds by simply changing the constructor argument.

### 7.2.1 Benefits of Custom Functors

- **Reusability:** Encapsulating logic in functors allows easy reuse across algorithms without rewriting code.
- **Statefulness:** Constructor parameters enable parameterizing behavior without global variables.
- **Clarity:** Functors clearly package logic and state, improving code readability.
- **Performance:** Defined inline, functors often allow compilers to optimize calls efficiently.

### 7.2.2 Summary

Writing custom functors involves defining classes with an overloaded `operator()`, optionally storing state through constructors. These objects provide a powerful way to inject custom behavior into STL algorithms like `std::sort` or `std::count_if`. Mastering functor creation enhances your ability to write modular, maintainable, and expressive C++ code.

## 7.3 Using Lambdas with STL Algorithms

Lambda expressions are a modern C++ feature that provides a concise way to write **inline, anonymous functors**—small blocks of code that can be passed directly to STL algorithms without needing a separate named functor class. Lambdas are perfect for short, custom operations that improve code clarity and reduce boilerplate.

### Anatomy of a Lambda

A lambda expression has this general form:

```
[capture](parameters) -> return_type {
    // function body
}
```

- **Capture list `[]`**: Specifies which variables from the surrounding scope the lambda can use. For example:
  - `[=]` captures all variables by value.
  - `[&]` captures all variables by reference.
  - `[x, &y]` captures `x` by value and `y` by reference.

- **Parameters `(…)`**: Like function parameters.
- **Return type `-> type`**: Usually inferred by the compiler, but can be explicitly specified.
- **Body `{…}`**: The code executed when the lambda is called.

### Example: Sorting with a Lambda

Suppose you want to sort a vector of integers in descending order without defining a separate functor:

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {5, 2, 9, 1, 7};

    std::sort(nums.begin(), nums.end(), [](int a, int b) {
        return a > b;  // Sort descending
    });

    for (int n : nums) std::cout << n << " ";
}
```

### Output:

```
9 7 5 2 1
```

Here, the lambda `[](int a, int b) { return a > b; }` is passed directly as the comparison function.

### Example: Finding an Element with `std::find_if`

Lambdas are handy for custom search conditions:

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

if (it != nums.end()) {
    std::cout << "First even number: " << *it << std::endl;
}
```

### Example: Applying Actions with `std::for_each`

You can also perform operations on every element inline:

```
std::for_each(nums.begin(), nums.end(), [](int& n) {
    n *= 2;  // Double each element
});
```

### 7.3.1 Why Use Lambdas?

- **Conciseness:** No need to write separate functor classes.
- **Readability:** Code stays close to where it's used, improving clarity.
- **Flexibility:** Capture local variables effortlessly for context.
- **Performance:** Compilers optimize lambdas effectively, comparable to functors.

### 7.3.2 Summary

Lambdas are lightweight, inline functors ideal for quick, custom behaviors in STL algorithms. Their capture lists allow access to surrounding variables, making them versatile. By mastering lambdas with `std::sort`, `std::find_if`, `std::for_each`, and more, you gain powerful tools for clean, expressive, and efficient C++ code.

## 7.4 Predicate Functions: Unary and Binary Predicates

In STL algorithms, **predicates** are special types of functors or functions that return a boolean (`true` or `false`) value. They serve as **conditions or rules** to control the behavior of algorithms like filtering, counting, and sorting.

### Unary Predicates

A **unary predicate** takes a single argument and returns a boolean. It's commonly used in algorithms that process elements individually, such as filtering or counting.

**Example:** Count how many numbers in a vector are greater than 10.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

bool isGreaterThan10(int x) {
    return x > 10;
}

int main() {
    std::vector<int> nums = {5, 12, 7, 20, 15};

    int count = std::count_if(nums.begin(), nums.end(), isGreaterThan10);

    std::cout << "Numbers greater than 10: " << count << std::endl;
}
```

Output:

```
Numbers greater than 10: 3
```

Here, `isGreaterThan10` is a unary predicate that controls which elements are counted.

### Binary Predicates

A **binary predicate** takes two arguments and returns a boolean. It is often used in sorting or comparing pairs of elements, where the predicate defines the order between two values.

**Example:** Sort a vector in descending order using a binary predicate.

Full runnable code:

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

bool descending(int a, int b) {
    return a > b;  // Return true if 'a' should come before 'b'
}

int main() {
    std::vector<int> nums = {5, 12, 7, 20, 15};

    std::sort(nums.begin(), nums.end(), descending);

    for (int n : nums) std::cout << n << " ";
}
```

Output:

```
20 15 12 7 5
```

The `descending` predicate tells `std::sort` how to order the elements.

### How Predicates Control Algorithm Behavior

- **Unary predicates** decide whether an element satisfies a condition (e.g., for filtering or counting).
- **Binary predicates** decide the relative ordering of two elements (e.g., for sorting).

Both predicate types must return `bool`, guiding the STL algorithms to perform actions conditionally or reorder elements accordingly.

## 7.4.1 Summary

Unary and binary predicates are boolean functions that drive the logic of many STL algorithms. Unary predicates filter or select elements based on a condition, while binary predicates define custom orderings. Understanding and writing these predicates is key to harnessing the full power of STL's generic algorithms.

## 7.5 Practical Examples: Filtering, Custom Sorting

In this section, we'll explore practical ways to use **functors** and **lambdas** to filter elements and perform custom sorting in STL containers. Combining these with STL algorithms helps write expressive, concise, and reusable code.

### Example 1: Filtering with a Custom Functor

Suppose we want to filter and count all numbers greater than a threshold using a functor:

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

struct IsGreaterThan {
    int threshold;
    IsGreaterThan(int t) : threshold(t) {}

    bool operator()(int value) const {
        return value > threshold;
    }
};

int main() {
    std::vector<int> nums = {10, 25, 5, 30, 15};

    IsGreaterThan greaterThan20(20);

    int count = std::count_if(nums.begin(), nums.end(), greaterThan20);

    std::cout << "Numbers greater than 20: " << count << std::endl;
}
```

### Output:

```
Numbers greater than 20: 2
```

Here, the functor `IsGreaterThan` holds the threshold state and is used with `std::count_if` to filter elements.

### Example 2: Filtering with a Lambda Expression

You can achieve the same filtering inline with a lambda for quick, one-off conditions:

```
int countLambda = std::count_if(nums.begin(), nums.end(), [](int x) {
    return x > 20;
});
std::cout << "Numbers greater than 20 (lambda): " << countLambda << std::endl;
```

### Example 3: Custom Sorting with a Lambda

Custom sorting is straightforward with lambdas. Suppose you want to sort strings by their length:

Full runnable code:

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

int main() {
    std::vector<std::string> words = {"apple", "banana", "kiwi", "cherry"};

    std::sort(words.begin(), words.end(), [](const std::string& a, const std::string& b) {
        return a.size() < b.size();  // Sort ascending by length
    });

    for (const auto& word : words)
        std::cout << word << " ";
}
```

**Output:**

```
kiwi apple banana cherry
```

**How Predicates Integrate with STL Algorithms**

- **Filtering algorithms** like `std::count_if`, `std::find_if`, and `std::remove_if` take **unary predicates** to select elements meeting specific conditions.
- **Sorting algorithms** like `std::sort` accept **binary predicates** to define custom ordering rules.

Whether using a stateful functor or a compact lambda, predicates make STL algorithms adaptable to your needs. Lambdas shine when you want concise, inline logic without boilerplate, while functors excel when reusability or stored state is required.

### 7.5.1 Summary

By combining functors and lambdas with STL algorithms, you write clear, maintainable code that filters and sorts data precisely how you want. This flexibility is a core strength of the STL, enabling powerful data manipulation with minimal effort.

# Chapter 8.

## Advanced Algorithms and Utilities

1. Set Algorithms: `std::set_union`, `std::set_intersection`, `std::set_difference`
2. Numeric Algorithms: `std::accumulate`, `std::inner_product`
3. Permutations and Combinations: `std::next_permutation`, `std::prev_permutation`
4. Algorithm Adaptors and Bindings
5. Case Studies: Applying Algorithms in Complex Scenarios

# 8 Advanced Algorithms and Utilities

## 8.1 Set Algorithms: `std::set_union`, `std::set_intersection`, `std::set_difference`

The STL provides a powerful set of **set algorithms**—`std::set_union`, `std::set_intersection`, and `std::set_difference`—to perform classic set operations on sorted ranges. These algorithms allow you to combine or compare two collections efficiently, producing a new sorted result that reflects union, intersection, or difference of the inputs.

### Key Requirements

- **Sorted Input:** All set algorithms require the input ranges to be sorted according to the same strict weak ordering (usually ascending order).
- **Output Iterators:** They output results through output iterators, so you can use containers like `std::vector` or `std::set` to store results.
- **No Duplicates Assumed:** Inputs are typically sets or sorted sequences without duplicates for correct logical set operations.

#### `std::set_union`

Computes the union of two sorted ranges, producing all elements that appear in either range, without duplicates.

**Example:**

Full runnable code:

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

int main() {
    std::vector<int> set1 = {1, 3, 5, 7};
    std::vector<int> set2 = {3, 4, 5, 6};

    std::vector<int> result;
    std::set_union(set1.begin(), set1.end(),
                   set2.begin(), set2.end(),
                   std::back_inserter(result));

    for (int n : result) std::cout << n << " ";
}
```

**Output:**

```
1 3 4 5 6 7
```

#### `std::set_intersection`

Produces the intersection of two sorted ranges, outputting only elements found in both.

**Example:**

```
std::vector<int> result;
std::set_intersection(set1.begin(), set1.end(),
                      set2.begin(), set2.end(),
                      std::back_inserter(result));
```

Output:

```
3 5
```

**`std::set_difference`**

Calculates the difference between two sets—elements in the first range but not in the second.

**Example:**

```
std::vector<int> result;
std::set_difference(set1.begin(), set1.end(),
                    set2.begin(), set2.end(),
                    std::back_inserter(result));
```

Output:

```
1 7
```

### 8.1.1 Summary

Set algorithms provide an efficient and convenient way to work with sorted collections as mathematical sets. By requiring sorted input ranges and using output iterators, these algorithms integrate seamlessly with STL containers such as `std::vector` and `std::set`. Whether you need to combine, intersect, or differentiate sets, these functions are fundamental tools for advanced data manipulation in C++.

## 8.2 Numeric Algorithms: `std::accumulate`, `std::inner_product`

The STL provides numeric algorithms designed to perform common mathematical operations on sequences of data. Two of the most widely used are `std::accumulate` and `std::inner_product`, which help simplify tasks like summation and dot product calculation.

**`std::accumulate`**

`std::accumulate` computes the sum (or any other binary accumulation) of elements in a range. It takes a starting value and a binary operation (by default, addition), then applies the operation sequentially to combine all elements.

**Example: Summing Vector Elements**

Full runnable code:

```cpp
#include <iostream>
#include <vector>
#include <numeric>  // for std::accumulate

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    int sum = std::accumulate(numbers.begin(), numbers.end(), 0);

    std::cout << "Sum: " << sum << std::endl;  // Output: Sum: 15
}
```

You can also provide a custom binary operation. For instance, to multiply all elements:

```cpp
int product = std::accumulate(numbers.begin(), numbers.end(), 1, std::multiplies<int>());
```

### `std::inner_product`

`std::inner_product` calculates the dot product of two sequences. It multiplies corresponding elements pairwise and accumulates the results, optionally allowing custom binary operations for both multiplication and addition.

#### Example: Calculating Dot Product

Full runnable code:

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

int main() {
    std::vector<int> v1 = {1, 2, 3};
    std::vector<int> v2 = {4, 5, 6};

    int dot = std::inner_product(v1.begin(), v1.end(), v2.begin(), 0);

    std::cout << "Dot product: " << dot << std::endl;  // Output: 32
}
```

This computes `(1*4) + (2*5) + (3*6) = 32`.

You can customize both the multiplication and addition operations:

```cpp
int custom = std::inner_product(
    v1.begin(), v1.end(), v2.begin(), 0,
    std::plus<>(),         // addition operation
    [](int a, int b) { return a * b; }  // multiplication operation
);
```

#### Use Cases

These numeric algorithms are essential in many fields:

- **Mathematics and statistics:** Summation, averages, and weighted calculations.

- **Physics and engineering:** Dot products in vector calculations.
- **Data analysis:** Aggregating data values efficiently.

By abstracting these operations, STL lets you write concise, readable code that remains flexible through custom operations.

### 8.2.1 Summary

`std::accumulate` and `std::inner_product` simplify numerical computations on containers, allowing both default and customized operations. They are foundational tools for mathematical programming in C++, widely applicable across domains requiring efficient aggregation and product calculations.

## 8.3 Permutations and Combinations: `std::next_permutation`, `std::prev_permutation`

The STL algorithms `std::next_permutation` and `std::prev_permutation` provide an easy way to generate permutations of elements in lexicographical order. These tools are essential when you need to systematically explore all possible orderings or rearrangements of a collection, such as in combinatorial problems or backtracking algorithms.

#### How They Work

- **`std::next_permutation`** rearranges the elements into the next lexicographically greater permutation. If the current permutation is the highest possible, it returns `false` and rearranges the sequence to the lowest permutation (sorted ascending).
- **`std::prev_permutation`** does the opposite, generating the lexicographically previous permutation, or returning `false` when at the lowest permutation and resetting to the highest.

Both functions operate in-place on a container range and require **random access iterators** (e.g., `std::vector`, `std::array`).

#### Example: Generating All Permutations

The following example prints all permutations of the vector `{1, 2, 3}`:

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```cpp
int main() {
    std::vector<int> data = {1, 2, 3};

    // The vector must be initially sorted for correct usage
    std::sort(data.begin(), data.end());

    do {
        for (int x : data)
            std::cout << x << " ";
        std::cout << "\n";
    } while (std::next_permutation(data.begin(), data.end()));

    return 0;
}
```

**Output:**

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

This example illustrates how `std::next_permutation` steps through all permutations in ascending lex order.

**Applications**

- **Combinatorial problems:** Enumerate permutations when solving puzzles, scheduling, or optimization.
- **Backtracking:** Generate permutations for exhaustive search, such as solving the Traveling Salesman Problem.
- **Testing:** Generate test cases covering different orderings.

### 8.3.1 Summary

`std::next_permutation` and `std::prev_permutation` are convenient STL algorithms for navigating permutations in lexicographical order. They allow you to generate all possible arrangements efficiently, which is invaluable in many mathematical, algorithmic, and practical programming scenarios.

## 8.4 Algorithm Adaptors and Bindings

The STL offers **algorithm adaptors and bindings** to make callable objects (functions, functors, lambdas) more flexible and compatible with algorithm requirements. These utilities help transform or adapt functions to match the signature or behavior expected by algorithms.

### `std::bind`

One of the key adaptors is `std::bind`, which allows you to create new function objects by binding some arguments of a callable in advance, while leaving placeholders for others. This is useful to customize functions or member function calls without writing extra functors.

### How `std::bind` Works

- You specify the target callable (function, functor, or member function).
- Bind some arguments to fixed values.
- Use placeholders (`std::placeholders::_1`, `_2`, etc.) to represent parameters passed later during the call.

This results in a new callable object compatible with algorithms expecting a certain function signature.

### Example: Using `std::bind` with Member Functions

Consider a class with a member function, and you want to call it on objects inside a container using an STL algorithm.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

class Person {
public:
    Person(std::string n, int a) : name(n), age(a) {}
    bool isOlderThan(int x) const { return age > x; }
    std::string name;
    int age;
};

int main() {
    std::vector<Person> people = {
        {"Alice", 30}, {"Bob", 25}, {"Charlie", 35}
    };

    // Create a predicate that checks if Person is older than 28
    auto olderThan28 = std::bind(&Person::isOlderThan, std::placeholders::_1, 28);

    // Count how many people are older than 28
    int count = std::count_if(people.begin(), people.end(), olderThan28);

    std::cout << "People older than 28: " << count << std::endl;
```

```
}
```

**Output:**

```
People older than 28: 2
```

Here, `std::bind` adapts the member function `isOlderThan` to be used as a unary predicate for `std::count_if`. The first argument (`std::placeholders::_1`) corresponds to the `Person` object, while `28` is bound as the fixed age to compare.

### Other Adaptors

Besides `std::bind`, STL offers other adaptors like:

- **`std::function`**: A general-purpose polymorphic function wrapper.
- **Function pointers and functor adapters** for negation, composition, or argument swapping.

These tools enhance the expressiveness of STL algorithms, enabling more modular and readable code.

### 8.4.1 Summary

Algorithm adaptors like `std::bind` enable flexible use of callable objects by fixing some arguments or rearranging parameters. This adaptability is crucial to seamlessly integrate custom behaviors and member functions with STL algorithms, boosting code reuse and clarity.

## 8.5 Case Studies: Applying Algorithms in Complex Scenarios

In real-world programming, STL algorithms shine by combining their strengths to solve complex problems efficiently. This section presents practical case studies demonstrating how multiple algorithms and functors work together to process, filter, transform, and analyze data.

### Case Study 1: Filtering and Transforming a Dataset

Suppose you have a list of sales amounts and want to:

- Filter out sales below a certain threshold.
- Apply a 10% tax increase on the remaining sales.
- Collect the results in a new container.

Full runnable code:

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

int main() {
    std::vector<double> sales = {100.0, 250.5, 75.25, 300.0, 50.0};
    std::vector<double> filtered_sales;

    // Step 1: Filter sales >= 100
    std::copy_if(sales.begin(), sales.end(),
                 std::back_inserter(filtered_sales),
                 [](double s) { return s >= 100.0; });

    // Step 2: Apply 10% tax increase
    std::transform(filtered_sales.begin(), filtered_sales.end(),
                   filtered_sales.begin(),
                   [](double s) { return s * 1.10; });

    // Print results
    std::cout << "Updated sales after filtering and tax:\n";
    std::for_each(filtered_sales.begin(), filtered_sales.end(),
                  [](double s) { std::cout << s << " "; });
    std::cout << "\n";
}
```

**Explanation:** `std::copy_if` filters sales above the threshold using a lambda predicate. Then, `std::transform` applies the tax increase in-place. Finally, `std::for_each` prints the modified values. This pipeline uses non-modifying and modifying algorithms to handle data cleanly.

### Case Study 2: Complex Set Operations on Data

Consider two sorted datasets representing customer IDs of two different products' buyers. We want to:

- Find customers who bought either product (union).
- Find customers who bought both products (intersection).
- Find customers who bought only the first product (difference).

Full runnable code:

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

int main() {
    std::vector<int> productA = {101, 102, 105, 110};
    std::vector<int> productB = {102, 104, 110, 115};

    std::vector<int> union_result, intersection_result, difference_result;

    std::set_union(productA.begin(), productA.end(),
                   productB.begin(), productB.end(),
                   std::back_inserter(union_result));
```

```
    std::set_intersection(productA.begin(), productA.end(),
                          productB.begin(), productB.end(),
                          std::back_inserter(intersection_result));

    std::set_difference(productA.begin(), productA.end(),
                        productB.begin(), productB.end(),
                        std::back_inserter(difference_result));

    // Print results
    std::cout << "Union: ";
    for (int id : union_result) std::cout << id << " ";
    std::cout << "\nIntersection: ";
    for (int id : intersection_result) std::cout << id << " ";
    std::cout << "\nDifference (only product A): ";
    for (int id : difference_result) std::cout << id << " ";
    std::cout << "\n";
}
```

**Explanation:** These algorithms operate on sorted sequences and produce new containers via output iterators. This approach efficiently handles membership and comparison problems often encountered in database and analytics software.

### 8.5.1 Design Rationale

- **Separation of Concerns:** Use filtering, transforming, and set algorithms in modular steps for readability.
- **Efficiency:** Utilize in-place modifications and iterator-based insertion for performance.
- **Reusability:** Lambdas and functors enable customization without rewriting code.
- **Safety:** Algorithms enforce preconditions like sorted input to ensure correctness.

### 8.5.2 Summary

Combining STL algorithms and functors allows for powerful, clear solutions to complex problems. By leveraging filtering, transformation, and set operations together, developers can write concise and efficient code for everyday real-world challenges.

# Chapter 9.

# Allocators and Memory Management

1. Understanding Allocators in STL
2. Using Custom Allocators
3. Memory Considerations and Optimization
4. Examples: Custom Pool Allocator Integration

# 9 Allocators and Memory Management

## 9.1 Understanding Allocators in STL

In the C++ Standard Template Library (STL), **allocators** are objects that abstract the process of memory allocation and deallocation. While most developers use STL containers like `std::vector` or `std::map` without thinking much about memory, allocators provide a customizable way to control how containers manage their internal memory. This feature is especially valuable in performance-critical applications or embedded systems where custom memory strategies are required.

### What Is an Allocator?

An allocator is a class template that defines how memory for objects is allocated and released. STL containers use allocators to obtain raw memory when storing elements and to deallocate that memory when it is no longer needed. By default, containers use `std::allocator<T>`, which simply delegates memory management to the standard `new` and `delete` operators.

However, STL containers are designed to accept any allocator conforming to the **Allocator** concept—meaning it must implement a standard interface of types and functions, such as:

- `allocate(size_t n)` – Allocates memory for `n` objects.
- `deallocate(pointer p, size_t n)` – Frees memory previously allocated.
- `construct(pointer p, const T& val)` – Constructs an object at the given memory location (C++11 and earlier).
- `destroy(pointer p)` – Destroys an object at the given memory location (C++11 and earlier).

Note: In modern C++ (C++17 and onward), `construct` and `destroy` are deprecated, and placement `new` is used instead.

### How Allocators Affect Container Behavior

When you instantiate a container, such as:

```
std::vector<int, std::allocator<int>> v;
```

You can substitute your own allocator for specialized memory behavior, such as pooling, memory tracking, aligned storage, or placing objects in shared memory.

Allocators influence:

- **Performance**: Custom allocators may reduce fragmentation or overhead.
- **Memory layout**: Allocators can control where memory is placed.
- **Resource tracking**: Useful for debugging or profiling memory usage.

### Example: Using Default Allocator

```
std::vector<int> numbers; // uses std::allocator<int> internally
numbers.push_back(1);
```

```
numbers.push_back(2);
```

Here, the vector automatically handles memory using the default allocator, but the mechanism is extensible.

### Summary

Allocators are a foundational abstraction that separates memory management from container logic in STL. While most users rely on the default behavior, understanding and utilizing custom allocators can be a powerful tool in advanced applications where memory layout and performance are critical.

## 9.2 Using Custom Allocators

### 9.2.1 Using Custom Allocators

In the STL, containers like `std::vector` and `std::map` support a second template parameter: the allocator type. By default, they use `std::allocator<T>`, but developers can supply custom allocators to change how memory is allocated and deallocated. This is especially useful in performance-sensitive scenarios such as **memory pooling**, **tracking memory usage**, or **allocating from special regions** (e.g., shared memory, GPUs).

### Why Use a Custom Allocator?

Custom allocators allow you to:

- Reuse memory to reduce allocation overhead.
- Track memory allocations for debugging or profiling.
- Use memory from specific sources or with alignment constraints.
- Improve performance in real-time systems or game engines.

### A Simple Custom Allocator Example

Here's a basic custom allocator that logs allocations and deallocations:

Full runnable code:

```
#include <iostream>
#include <memory>
#include <vector>

template <typename T>
class LoggingAllocator {
public:
    using value_type = T;

    LoggingAllocator() = default;
```

```
    template <typename U>
    constexpr LoggingAllocator(const LoggingAllocator<U>&) noexcept {}

    [[nodiscard]] T* allocate(std::size_t n) {
        std::cout << "Allocating " << n << " element(s)\n";
        return static_cast<T*>(::operator new(n * sizeof(T)));
    }

    void deallocate(T* p, std::size_t n) noexcept {
        std::cout << "Deallocating " << n << " element(s)\n";
        ::operator delete(p);
    }
};

template <typename T, typename U>
bool operator==(const LoggingAllocator<T>&, const LoggingAllocator<U>&) { return true; }

template <typename T, typename U>
bool operator!=(const LoggingAllocator<T>&, const LoggingAllocator<U>&) { return false; }

int main() {
    std::vector<int, LoggingAllocator<int>> vec;
    vec.push_back(10);
    vec.push_back(20);
}
```

**Output:**

```
Allocating 1 element(s)
Allocating 2 element(s)
Deallocating 1 element(s)
Deallocating 2 element(s)
```

The custom allocator provides transparency into the container's memory behavior, which is invaluable when debugging allocation-related issues.

### Use Cases for Custom Allocators

- **Memory Pooling**: Avoids fragmentation by allocating large memory blocks and carving out small chunks.
- **Real-time Systems**: Ensures deterministic allocation time.
- **Shared Memory**: Allocates data in shared memory between processes.
- **Embedded Systems**: Allocates memory from limited or fixed memory regions.

### Summary

Custom allocators in STL offer a powerful way to control memory behavior. By plugging a custom allocator into a standard container, you can optimize performance, control allocation strategy, and gain visibility into memory usage. While implementing a fully-featured allocator can be complex, even simple versions are useful for learning and practical applications like logging and memory pooling.

## 9.3 Memory Considerations and Optimization

STL containers rely heavily on dynamic memory allocation, and the choice of allocator can significantly impact **performance**, **memory usage**, and **predictability**. Understanding how memory behaves under different allocation strategies is key to writing efficient and scalable applications.

### Key Memory Factors

1. **Fragmentation**:
    - Standard allocators (like `std::allocator`) use the system heap, which can become fragmented over time—especially with frequent allocations and deallocations of varying sizes.
    - Fragmentation leads to inefficient memory usage and potential allocation failures in long-running applications.
2. **Allocation Speed**:
    - System-level `new` and `delete` can be relatively slow due to thread safety, locking, and general-purpose behavior.
    - Custom allocators, such as **pool** or **arena** allocators, pre-allocate a large memory block and carve out chunks, which drastically reduces allocation time.
3. **Cache Locality**:
    - Cache-friendly allocation patterns, where related data is kept close in memory, improve performance due to better CPU cache utilization.
    - Containers like `std::vector` already benefit from good locality, but allocator design can further enhance this effect in custom containers or structures.

### Common Optimized Allocator Types

- **Pool Allocators**:
    - Recycle fixed-size memory blocks.
    - Best for scenarios with frequent allocation/deallocation of same-sized objects (e.g., game engines, network buffers).
- **Arena Allocators**:
    - Allocate a large memory chunk (arena) and allocate linearly within it.
    - Deallocation is typically done wholesale (free the entire arena), making them fast and suitable for short-lived objects (e.g., parsing, intermediate computation).

### Guidelines for Choosing or Designing Allocators

- **Use the default allocator** if performance is acceptable and your application is not memory-intensive.
- **Consider a pool allocator** when you create and destroy many small, fixed-size objects.

- **Use an arena allocator** for batch processing tasks where you can deallocate all memory at once.
- **Track memory usage** with logging allocators during development to diagnose issues.
- **Test allocator impact** with realistic workloads to measure gains before integrating.

**Summary**

Smart allocator choice can mitigate fragmentation, improve speed, and boost cache performance. STL's allocator interface enables seamless integration of custom strategies, making memory optimization an accessible and powerful tool for performance-critical applications.

## 9.4 Examples: Custom Pool Allocator Integration

To better understand how allocators can improve memory performance in the STL, let's implement and integrate a **custom pool allocator** with `std::vector`. Pool allocators are optimized for scenarios where many small, similarly sized objects are frequently created and destroyed—common in games, simulations, or object-heavy applications.

**What Is a Pool Allocator?**

A **pool allocator** pre-allocates a block of memory (a pool) and serves future allocation requests from that block. This approach avoids repeated calls to `new`/`delete`, significantly reducing overhead and memory fragmentation.

### 9.4.1 Step-by-Step Pool Allocator Example

Full runnable code:

```
#include <iostream>
#include <vector>
#include <memory>

template <typename T, std::size_t PoolSize = 1024>
class PoolAllocator {
public:
    using value_type = T;

    PoolAllocator() noexcept {
        pool = static_cast<T*>(::operator new(PoolSize * sizeof(T)));
        free_ptr = pool;
        allocated = 0;
    }

    ~PoolAllocator() noexcept {
        ::operator delete(pool);
    }
```

```
    template <typename U>
    PoolAllocator(const PoolAllocator<U, PoolSize>&) noexcept {}

    T* allocate(std::size_t n) {
        if (allocated + n > PoolSize) {
            throw std::bad_alloc();
        }
        T* result = free_ptr;
        free_ptr += n;
        allocated += n;
        return result;
    }

    void deallocate(T* p, std::size_t) noexcept {
        // Pool memory is deallocated at once in the destructor.
    }

    template <typename U>
    struct rebind {
        using other = PoolAllocator<U, PoolSize>;
    };

private:
    T* pool;
    T* free_ptr;
    std::size_t allocated;
};

template <typename T1, typename T2, std::size_t N>
bool operator==(const PoolAllocator<T1, N>&, const PoolAllocator<T2, N>&) { return true; }
template <typename T1, typename T2, std::size_t N>
bool operator!=(const PoolAllocator<T1, N>&, const PoolAllocator<T2, N>&) { return false; }

int main() {
    std::vector<int, PoolAllocator<int>> poolVec;
    for (int i = 0; i < 10; ++i) {
        poolVec.push_back(i);
    }

    for (int val : poolVec) {
        std::cout << val << " ";
    }
    std::cout << "\n";
    return 0;
}
```

### 9.4.2 Explanation and Benefits

- **Preallocation**: The pool allocator reserves a block of memory (`PoolSize * sizeof(T)`) on construction.
- **Fast Allocation**: `allocate()` just advances a pointer, avoiding heap allocations.
- **No Deallocation**: `deallocate()` is a no-op—memory is reclaimed only when the

allocator is destroyed.
- **Integration**: We plugged the custom allocator into `std::vector<int, PoolAllocator<int>>` seamlessly, leveraging STL's allocator-aware design.

### 9.4.3 When to Use This

- High-performance environments requiring **predictable and fast allocations**.
- Applications with **large numbers of short-lived objects** where frequent allocations would otherwise degrade performance.
- Custom memory diagnostics or **manual memory pooling strategies**.

This example shows how custom allocators like pool allocators can be used to optimize STL containers, offering fine-grained control over memory behavior and system performance.

# Chapter 10.

## String and Stream Utilities

1. `std::string` and `std::wstring`: Basic Usage and Operations
2. String Manipulation Algorithms
3. Streams and String Streams (`std::stringstream`)
4. Practical Examples: Parsing, Formatting, and Serialization

# 10 String and Stream Utilities

## 10.1 `std::string` and `std::wstring`: Basic Usage and Operations

In C++, the Standard Library provides powerful string types through `std::string` and `std::wstring`. These classes simplify working with sequences of characters and offer rich functionality for text manipulation.

**`std::string` vs. `std::wstring`**

- `std::string` is designed for narrow characters (`char`), typically used for UTF-8 or ASCII.
- `std::wstring` handles wide characters (`wchar_t`), useful for internationalization (e.g., Unicode or platform-specific wide encodings).

Both types behave similarly in functionality, differing only in the character type.

### 10.1.1 Common Operations

**1. Construction and Initialization**

```cpp
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2("World");
    std::string s3(s1 + " " + s2); // Concatenation

    std::cout << s3 << std::endl;  // Output: Hello World
    return 0;
}
```

**2. Modification and Access**

```cpp
s3[0] = 'h';              // Modify character
s3.append("!!!");         // Append
s3.insert(5, ",");        // Insert comma
s3.replace(0, 5, "Hi");   // Replace "hello" with "Hi"
```

**3. Substrings and Searching**

```cpp
std::string sub = s3.substr(3, 4);     // Substring from index 3, length 4
size_t pos = s3.find("World");         // Find substring
if (pos != std::string::npos) {
    std::cout << "Found at: " << pos << "\n";
}
```

**4. Comparison**

```cpp
std::string a = "apple", b = "banana";
if (a < b) std::cout << "apple comes before banana\n";
```

**5. Iteration**

```
for (char ch : s3) {
    std::cout << ch << ' ';
}
std::cout << '\n';
```

### 10.1.2 `std::wstring` Usage

Full runnable code:

```
#include <iostream>
#include <string>

int main() {
    std::wstring ws = L"   "; // Japanese for "Hello"
    std::wcout << ws << std::endl;
    return 0;
}
```

Note: `std::wcout` is required for wide-character output.

### 10.1.3 Conversion Between `std::string` and `std::wstring`

Conversions require utilities like `std::wstring_convert` or third-party libraries (e.g., ICU), as the process depends on character encoding.

```
#include <locale>
#include <codecvt>

std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter;
std::wstring ws = converter.from_bytes("Hello UTF-8");
std::string s = converter.to_bytes(ws);
```

### 10.1.4 Performance and Memory Considerations

- `std::string` manages memory dynamically. Operations like concatenation and insertion may trigger reallocations.
- Reserve space using `reserve()` to reduce reallocations.
- Avoid copying large strings; prefer passing by reference (`const std::string&`) when possible.

### 10.1.5 Summary

`std::string` and `std::wstring` provide a full-featured, efficient foundation for text manipulation in C++. With support for slicing, searching, comparison, and modification, they are essential tools in any C++ developer's toolkit. Understanding these types is critical before working with string algorithms, streams, or file I/O.

## 10.2 String Manipulation Algorithms

### 10.2.1 String Manipulation Algorithms

The C++ Standard Template Library provides a powerful suite of algorithms that can be applied to strings via iterators. Common tasks like transforming case, replacing characters, or searching substrings can be efficiently performed using algorithms such as `std::transform`, `std::replace`, `std::find`, and `std::search`.

#### Case Conversion with `std::transform`

`std::transform` applies a function to each character in a string. It's ideal for converting case:

Full runnable code:

```cpp
#include <iostream>
#include <algorithm>
#include <string>
#include <cctype>

int main() {
    std::string text = "Hello World!";

    // Convert to uppercase
    std::transform(text.begin(), text.end(), text.begin(), ::toupper);
    std::cout << "Uppercase: " << text << "\n";

    // Convert to lowercase
    std::transform(text.begin(), text.end(), text.begin(), ::tolower);
    std::cout << "Lowercase: " << text << "\n";

    return 0;
}
```

#### Replacing Characters with `std::replace`

Use `std::replace` to substitute all occurrences of one character with another:

```cpp
std::string sentence = "apple pie";
std::replace(sentence.begin(), sentence.end(), 'p', 'b');
std::cout << "Modified: " << sentence << "\n"; // Output: abble bie
```

### Searching Characters and Substrings

- **`std::find`** locates the first occurrence of a character:

```
auto it = std::find(sentence.begin(), sentence.end(), 'e');
if (it != sentence.end()) {
    std::cout << "Found 'e' at position: " << std::distance(sentence.begin(), it) << "\n";
}
```

- **`std::search`** locates a sequence (substring):

```
std::string haystack = "this is a test";
std::string needle = "test";

auto pos = std::search(haystack.begin(), haystack.end(), needle.begin(), needle.end());
if (pos != haystack.end()) {
    std::cout << "'test' found at index: " << std::distance(haystack.begin(), pos) << "\n";
}
```

### Trimming Whitespace (Manual Example)

While STL doesn't offer built-in trim functions, algorithms can help:

```
std::string str = "   padded text   ";
auto front = std::find_if_not(str.begin(), str.end(), ::isspace);
auto back = std::find_if_not(str.rbegin(), str.rend(), ::isspace).base();

std::string trimmed(front, back);
std::cout << "Trimmed: '" << trimmed << "'\n";
```

### 10.2.2 Summary

STL algorithms empower developers to manipulate `std::string` flexibly and efficiently. Whether you're converting case, replacing characters, or searching text, combining algorithms with string iterators leads to concise and expressive code.

## 10.3 Streams and String Streams (`std::stringstream`)

C++ provides a powerful stream-based I/O system that abstracts input and output across various data sources. One particularly useful tool in this system is `std::stringstream`, which operates on strings in memory, enabling easy parsing and formatting.

### What Is `std::stringstream`?

`std::stringstream` is part of the `<sstream>` header and behaves like an in-memory stream. It supports both input and output operations using the familiar `<<` and `>>` operators, making it ideal for tasks such as:

- Converting strings to numeric types and vice versa

- Parsing structured text like CSV
- Building complex strings through formatted output

### Basic Usage: Formatting and Parsing

Full runnable code:

```cpp
#include <iostream>
#include <sstream>
#include <string>

int main() {
    std::stringstream ss;

    // Output: convert values to string
    int id = 42;
    std::string name = "Alice";
    ss << "ID: " << id << ", Name: " << name;

    std::string output = ss.str();
    std::cout << output << "\n";

    // Clear stream before reuse
    ss.str("");
    ss.clear();

    // Input: parse string into values
    ss.str("100 John");
    int parsedId;
    std::string parsedName;
    ss >> parsedId >> parsedName;

    std::cout << "Parsed ID: " << parsedId << ", Name: " << parsedName << "\n";
    return 0;
}
```

### Converting Between Strings and Numbers

```cpp
std::string numberStr = "123.45";
double value;
std::stringstream(numberStr) >> value;

int x = 10;
std::string strVal = std::to_string(x); // or use stringstream
```

### Parsing a CSV Line

Full runnable code:

```cpp
#include <vector>
#include <sstream>
#include <iostream>

int main() {
    std::string csv = "apple,banana,orange";
    std::stringstream ss(csv);
```

```
    std::string item;
    std::vector<std::string> tokens;

    while (std::getline(ss, item, ',')) {
        tokens.push_back(item);
    }

    for (const auto& token : tokens) {
        std::cout << token << "\n";
    }

    return 0;
}
```

#### Stream State Management

When using `stringstream`, it's important to manage the stream state. Use `ss.clear()` to reset error flags and `ss.str("")` to reset the internal buffer if reusing the stream.

### 10.3.1 Summary

`std::stringstream` bridges the gap between strings and stream-based I/O, enabling structured parsing and formatting directly in memory. Its ability to handle both input and output in a unified interface makes it essential for string manipulation, conversions, and simple parsers in modern C++ applications.

## 10.4 Practical Examples: Parsing, Formatting, and Serialization

C++ provides robust string and stream utilities that can be combined to solve real-world tasks like parsing structured text, dynamically formatting output, and serializing objects. This section explores such applications using `std::string`, `std::stringstream`, and relevant STL features.

#### Example 1: Parsing Key-Value Pairs from a Configuration String

Imagine reading a configuration string like `"mode=debug;threads=4;verbose=true"`. We can parse it into key-value pairs using `std::stringstream` and string functions.

Full runnable code:

```
#include <iostream>
#include <sstream>
#include <string>
#include <unordered_map>

int main() {
```

```
    std::string config = "mode=debug;threads=4;verbose=true";
    std::unordered_map<std::string, std::string> kvMap;

    std::stringstream ss(config);
    std::string pair;

    while (std::getline(ss, pair, ';')) {
        std::stringstream pairStream(pair);
        std::string key, value;
        if (std::getline(pairStream, key, '=') && std::getline(pairStream, value)) {
            kvMap[key] = value;
        }
    }

    for (const auto& [k, v] : kvMap) {
        std::cout << k << " => " << v << "\n";
    }
}
```

**Output:**

```
mode => debug
threads => 4
verbose => true
```

### Example 2: Formatting Floating-Point Numbers with Precision

Using `std::ostringstream`, you can control how numbers are formatted, which is useful for displaying monetary values or statistics.

Full runnable code:

```
#include <iomanip> // for std::setprecision
#include <sstream>
#include <iostream>

int main() {
    double pi = 3.1415926535;
    std::ostringstream out;

    out << std::fixed << std::setprecision(3) << pi;
    std::cout << "Formatted PI: " << out.str() << "\n";
}
```

**Output:**

```
Formatted PI: 3.142
```

### Example 3: Serializing a Struct to a JSON-like String

Full runnable code:

```
#include <string>
#include <sstream>
#include <iostream>
```

```cpp
struct User {
    std::string name;
    int age;
    bool active;
};

std::string serializeToJson(const User& user) {
    std::ostringstream ss;
    ss << "{";
    ss << "\"name\": \"" << user.name << "\", ";
    ss << "\"age\": " << user.age << ", ";
    ss << "\"active\": " << (user.active ? "true" : "false");
    ss << "}";
    return ss.str();
}

int main() {
    User user = {"Alice", 30, true};
    std::string json = serializeToJson(user);
    std::cout << json << "\n";
}
```

**Output:**

```
{"name": "Alice", "age": 30, "active": true}
```

### 10.4.1 Summary

These examples demonstrate how C++ string and stream utilities can be leveraged for practical text processing tasks. By using `std::stringstream` and formatting tools, developers can parse structured input, format output with precision, and serialize data for storage or transmission—all with clear, efficient code. These techniques are essential for building robust data-driven applications.

# Chapter 11.

## Tuples and Pairs

1. Introduction to `std::pair` and `std::tuple`
2. Creating, Accessing, and Using Tuples
3. Structured Bindings with Tuples (C++17)
4. Use Cases: Returning Multiple Values, Grouping Data

# 11 Tuples and Pairs

## 11.1 Introduction to `std::pair` and `std::tuple`

The C++ Standard Library provides two powerful utility types—`std::pair` and `std::tuple`—to group multiple values into a single object. These containers are especially useful when returning multiple values from functions or combining heterogeneous types for temporary use.

### `std::pair`: Two-Element Container

`std::pair<T1, T2>` is a simple structure that holds exactly two elements, potentially of different types. These are accessed via the public members `first` and `second`.

**Example:**

Full runnable code:

```
#include <iostream>
#include <utility>

int main() {
    std::pair<std::string, int> person = {"Alice", 30};
    std::cout << "Name: " << person.first << ", Age: " << person.second << "\n";
}
```

You can create a `pair` using `std::make_pair` or aggregate initialization:

```
auto p = std::make_pair("x", 3.14);  // type: pair<const char*, double>
```

### `std::tuple`: Heterogeneous Grouping of Elements

`std::tuple<Ts...>` generalizes `std::pair` by allowing an arbitrary number of elements of varying types. It is ideal for grouping multiple return values or holding loosely related data.

**Example:**

Full runnable code:

```
#include <iostream>
#include <tuple>

int main() {
    std::tuple<std::string, int, bool> person = {"Bob", 25, true};

    std::cout << std::get<0>(person) << ", "
              << std::get<1>(person) << ", "
              << std::get<2>(person) << "\n";
}
```

Each element in a tuple is accessed via `std::get<index>(tuple)`, where `index` is a compile-time constant.

You can also create a tuple using `std::make_tuple`:

```
auto t = std::make_tuple("pi", 3.14, true);
```

### Memory and Type Considerations

Both `std::pair` and `std::tuple` are value types and store elements by value. Because elements can be of different types, these structures are templated. The memory layout is not contiguous like arrays, and the access is positional (not named), which can make code less readable without structured bindings (introduced in C++17).

### 11.1.1 Summary

- `std::pair` is a lightweight two-element container, while `std::tuple` supports multiple heterogeneous elements.
- Use `first` and `second` to access a pair; use `std::get<i>()` for tuples.
- These types are ideal for bundling multiple return values or managing grouped but unrelated data.

In upcoming sections, we'll explore more powerful features like structured bindings and practical applications.

## 11.2 Creating, Accessing, and Using Tuples

`std::tuple` is a powerful and flexible container introduced in C++11 for grouping a fixed number of elements of potentially differing types. It generalizes `std::pair` by supporting more than two elements and offers various utilities for construction, element access, and unpacking.

### Creating Tuples

You can create a tuple using the `std::tuple` constructor, initializer list, or `std::make_tuple`, which deduces the types automatically:

```
std::tuple<int, double, std::string> t1(42, 3.14, "C++");
auto t2 = std::make_tuple(100, false, "STL");
```

Tuples can contain references, pointers, or values of any type. You can also nest tuples within each other.

### Accessing Elements with `std::get`

Access to tuple elements is done via `std::get<N>(tuple)`, where `N` is a compile-time constant (zero-based index):

```cpp
std::tuple<int, std::string> person(25, "Alice");
int age = std::get<0>(person);
std::string name = std::get<1>(person);
```

Each `std::get<N>` returns a reference to the element, so you can modify the tuple directly:

```cpp
std::get<0>(person) = 30;
```

### Unpacking with `std::tie` and `std::ignore`

`std::tie` is used to unpack a tuple into individual variables:

```cpp
std::tuple<int, char, std::string> data(1, 'X', "Hello");
int id;
char label;
std::string text;

std::tie(id, label, text) = data;
```

When you want to unpack only some of the values, use `std::ignore`:

```cpp
std::tie(std::ignore, label, text) = data; // Skip the first element
```

### Returning Multiple Values from Functions

Tuples are ideal for returning multiple values of differing types from a function:

```cpp
std::tuple<int, std::string> get_user() {
    return std::make_tuple(101, "Bob");
}

auto [id, name] = get_user(); // C++17 structured binding
```

If using a pre-C++17 compiler, you can use `std::tie` instead of structured bindings.

### Practical Use Cases

Tuples are useful when grouping related data that don't warrant a full struct or class definition. Examples include:

- Returning multiple results from parsing functions.
- Associating a value with multiple metadata fields (e.g., timestamp, type, description).
- Temporary aggregation of heterogeneous data.

## 11.2.1 Summary

`std::tuple` provides a flexible way to group heterogeneous values. With access via `std::get`, unpacking using `std::tie`, and the ability to return complex results from functions, it serves as a foundational utility in modern C++ programming.

## 11.3 Structured Bindings with Tuples (C++17)

Structured bindings, introduced in C++17, provide a clean and concise syntax to unpack tuples, pairs, and other aggregate types directly into named variables. This feature greatly simplifies working with `std::tuple` and `std::pair` by eliminating the need for verbose `std::get` calls or `std::tie`.

### What Are Structured Bindings?

Structured bindings allow you to declare multiple variables at once and initialize them from the elements of a tuple, pair, or struct in a single statement. This improves readability and reduces boilerplate code.

### Basic Syntax

Here's the general form for structured bindings with tuples:

```
auto [var1, var2, var3] = tuple_object;
```

Each variable corresponds to an element of the tuple, unpacked in order.

### Example: Unpacking a Tuple

Instead of writing:

```
std::tuple<int, std::string, double> data(42, "Answer", 3.14);

int number = std::get<0>(data);
std::string text = std::get<1>(data);
double pi = std::get<2>(data);
```

You can use structured bindings:

```
auto [number, text, pi] = data;
std::cout << number << ", " << text << ", " << pi << '\n';
```

This syntax is more intuitive and concise.

### Example: Returning Multiple Values from Functions

Functions returning tuples can be unpacked easily:

```
std::tuple<int, std::string> get_user() {
    return {101, "Alice"};
}

auto [id, name] = get_user();
std::cout << "ID: " << id << ", Name: " << name << '\n';
```

### Example: Iterating Over Container of Pairs

Structured bindings are also handy when iterating over associative containers like `std::map`:

```
std::map<int, std::string> phonebook = {{1, "Alice"}, {2, "Bob"}};

for (const auto& [key, value] : phonebook) {
```

```
    std::cout << key << " => " << value << '\n';
}
```

This avoids accessing `.first` and `.second` explicitly, making code cleaner and easier to maintain.

### 11.3.1 Benefits of Structured Bindings

- **Improved readability:** Named variables directly represent tuple elements.
- **Less error-prone:** Avoids mistakes with index-based `std::get` access.
- **Simplifies complex code:** Especially when dealing with multiple return values or container iteration.

Structured bindings modernize tuple and pair usage by providing a straightforward and elegant unpacking mechanism, enhancing C++ code expressiveness and clarity.

## 11.4 Use Cases: Returning Multiple Values, Grouping Data

`std::pair` and `std::tuple` are powerful tools for grouping multiple heterogeneous values together. They enable elegant solutions to common programming problems, such as returning multiple results from a function, grouping related but different types of data, and serving as key-value elements in associative containers like maps.

#### Returning Multiple Values from a Function

Often, functions need to return more than one value. Instead of using output parameters or structs, pairs and tuples provide a clean solution:

Full runnable code:

```
#include <tuple>
#include <iostream>

// Function returning multiple values using tuple
std::tuple<int, double, std::string> get_data() {
    return {42, 3.14, "example"};
}

int main() {
    auto [id, value, name] = get_data();  // Structured binding unpacking
    std::cout << "ID: " << id << ", Value: " << value << ", Name: " << name << '\n';
}
```

This example returns an integer, a double, and a string in one call, making the code clearer and easier to maintain.

### Grouping Heterogeneous Data

Tuples allow grouping mixed types into a single object that can be stored or iterated over:

Full runnable code:

```cpp
#include <tuple>
#include <vector>
#include <iostream>

int main() {
    std::vector<std::tuple<int, std::string, double>> records = {
        {1, "Alice", 95.5},
        {2, "Bob", 88.0},
        {3, "Carol", 92.3}
    };

    for (const auto& [id, name, score] : records) {
        std::cout << id << ": " << name << " scored " << score << '\n';
    }
}
```

This example models a student record list containing mixed data types and demonstrates iteration with structured bindings for clarity.

### Using Pairs in Maps as Key-Value Elements

`std::pair` is the foundation of `std::map` and other associative containers where keys and values must be linked:

Full runnable code:

```cpp
#include <map>
#include <iostream>

int main() {
    std::map<int, std::string> phonebook;

    // Insert using pairs
    phonebook.insert(std::pair<int, std::string>(101, "Alice"));
    phonebook.insert({102, "Bob"});  // Using initializer list syntax

    for (const auto& entry : phonebook) {
        std::cout << "Number: " << entry.first << ", Name: " << entry.second << '\n';
    }
}
```

Here, each `std::pair` stores a key (phone number) and a value (name), enabling efficient lookup and storage.

### 11.4.1 Summary

- **Returning multiple values:** Use tuples to return several related results cleanly.
- **Grouping heterogeneous data:** Tuples provide a container for mixed-type collections.
- **Associative containers:** Pairs naturally express key-value pairs for maps.

By mastering pairs and tuples, you can write more expressive, compact, and maintainable C++ code that elegantly manages multiple related data items.

# Chapter 12.

# Smart Pointers and Resource Management in STL

1. Overview of `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`
2. Automatic Resource Management Using Smart Pointers
3. Custom Deleters and Use Cases
4. Examples: Managing Dynamic Memory Safely

# 12 Smart Pointers and Resource Management in STL

## 12.1 Overview of `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`

Managing dynamic memory safely and efficiently is a fundamental challenge in C++. Traditional raw pointers require explicit calls to `delete` to free resources, which can easily lead to memory leaks, dangling pointers, or double deletions. To solve these problems, the C++ Standard Library provides **smart pointers** — wrapper classes that automate resource management and help enforce ownership semantics.

### Motivation: Automatic Resource Management

Smart pointers encapsulate raw pointers and automatically release the associated memory when the smart pointer goes out of scope. This approach follows the RAII (Resource Acquisition Is Initialization) principle, reducing manual memory management errors and improving code safety and clarity.

### `std::unique_ptr`: Exclusive Ownership

`std::unique_ptr` represents **exclusive ownership** of a resource. Only one `unique_ptr` instance can own a particular resource at a time. It cannot be copied but can be moved, transferring ownership to another `unique_ptr`.

Full runnable code:

```cpp
#include <memory>
#include <iostream>

int main() {
    std::unique_ptr<int> ptr1 = std::make_unique<int>(42);
    std::cout << *ptr1 << '\n';

    // std::unique_ptr<int> ptr2 = ptr1; // Error: copy not allowed
    std::unique_ptr<int> ptr2 = std::move(ptr1);  // Transfer ownership

    if (!ptr1) std::cout << "ptr1 is empty after move\n";
    std::cout << *ptr2 << '\n';
}
```

`unique_ptr` is lightweight and ideal when a single owner manages the lifetime of a resource.

### `std::shared_ptr`: Shared Ownership with Reference Counting

`std::shared_ptr` enables **shared ownership** of a resource. Multiple `shared_ptr`s can point to the same object, and the resource is destroyed only when the last `shared_ptr` owning it is destroyed or reset. This is implemented through **reference counting** internally.

Full runnable code:

```cpp
#include <memory>
#include <iostream>

int main() {
    std::shared_ptr<int> sp1 = std::make_shared<int>(100);
    std::shared_ptr<int> sp2 = sp1;  // Shared ownership

    std::cout << "Count: " << sp1.use_count() << '\n';  // 2

    sp1.reset();  // Releases one owner
    std::cout << "Count after reset: " << sp2.use_count() << '\n';  // 1
    std::cout << *sp2 << '\n';
}
```

Because `shared_ptr` manages reference counting, it is slightly heavier than `unique_ptr` but allows multiple owners safely, including across threads. The reference counting is **thread-safe** in standard implementations.

### `std::weak_ptr`: Non-owning Observer

`std::weak_ptr` is a **non-owning** smart pointer that observes an object managed by `shared_ptr` without affecting its lifetime or reference count. It is useful to break reference cycles or check if a resource is still alive without extending its lifetime.

Full runnable code:

```cpp
#include <memory>
#include <iostream>

int main() {
    std::shared_ptr<int> sp = std::make_shared<int>(77);
    std::weak_ptr<int> wp = sp;  // Observes but doesn't own

    if (auto locked = wp.lock()) {  // Attempts to get shared_ptr
        std::cout << *locked << '\n';
    }

    sp.reset();  // Resource destroyed here

    if (wp.expired()) {
        std::cout << "Resource no longer exists\n";
    }
}
```

`weak_ptr` is essential in complex ownership graphs to avoid memory leaks caused by cyclic references.

## 12.1.1 Summary

| Smart Pointer | Ownership Type | Copy-able | Thread-safe Reference Counting | Use Case |
|---|---|---|---|---|
| `std::unique_ptr` | Exclusive | No | N/A | Single owner, lightweight resource management |
| `std::shared_ptr` | Shared | Yes | Yes | Shared ownership, multiple owners, thread-safe |
| `std::weak_ptr` | Non-owning observer | Yes | Yes | Observing shared resource without ownership, breaking cycles |

Smart pointers greatly simplify resource management in modern C++, making code safer and less error-prone while clearly expressing ownership semantics.

## 12.2 Automatic Resource Management Using Smart Pointers

Managing dynamic memory manually in C++ is error-prone. Forgetting to delete allocated memory leads to **memory leaks**, while deleting memory prematurely or multiple times causes **dangling pointers** and undefined behavior. Smart pointers solve these issues by **automating object lifetime management** through RAII (Resource Acquisition Is Initialization), ensuring resources are properly released.

### RAII and Smart Pointers

RAII is a C++ design idiom where resource acquisition (like dynamic memory allocation) is tied to the lifetime of an object. When the object goes out of scope, its destructor releases the resource automatically. Smart pointers apply RAII to dynamic memory by owning pointers to allocated objects and deleting them automatically when they go out of scope.

```
void foo() {
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    // No need to manually delete, memory freed automatically when ptr goes out of scope
}
```

Here, `ptr` manages a dynamically allocated integer. Once `foo()` ends, `ptr` is destroyed and the memory freed, eliminating the risk of leaks.

### Ownership Transfer and `std::move`

Smart pointers like `std::unique_ptr` enforce exclusive ownership. To transfer ownership, you must use `std::move` to explicitly indicate the transfer:

```
void process(std::unique_ptr<int> p) {
    std::cout << "Value: " << *p << '\n';
}
```

```cpp
int main() {
    auto ptr = std::make_unique<int>(10);
    process(std::move(ptr));  // Ownership moves to process()

    if (!ptr) {
        std::cout << "ptr is now empty after move\n";
    }
}
```

`std::move` converts `ptr` into an rvalue, allowing `process` to take ownership. Afterward, `ptr` no longer owns the memory and cannot be used to access it safely, preventing dangling pointers.

### Smart Pointers in Classes

Smart pointers can be members of classes to manage dynamically allocated resources safely:

```cpp
class Widget {
    std::unique_ptr<int> data;

public:
    Widget(int value) : data(std::make_unique<int>(value)) {}

    void print() const {
        std::cout << "Value: " << *data << '\n';
    }
};

int main() {
    Widget w(123);
    w.print();
} // Automatically deletes the owned int when 'w' is destroyed
```

No manual cleanup is needed in the destructor, and resource leaks are avoided even if exceptions occur.

### 12.2.1 Summary

By encapsulating raw pointers, smart pointers automate memory management and ensure:

- Resources are released when no longer needed.
- Memory leaks and dangling pointers are prevented.
- Ownership transfer is explicit and safe using `std::move`.
- RAII principles simplify resource handling, including in complex scopes or classes.

This automatic resource management improves code safety, clarity, and maintainability in modern C++.

## 12.3 Custom Deleters and Use Cases

Smart pointers in C++ not only manage dynamic memory but can also handle other resources like file handles, sockets, or custom objects that require special cleanup. This flexibility is made possible by **custom deleters**—user-defined functions or function objects that specify how a resource should be released when the smart pointer is destroyed.

### What Are Custom Deleters?

By default, `std::unique_ptr` and `std::shared_ptr` call `delete` to free memory. However, some resources need different cleanup routines. A **custom deleter** tells the smart pointer what function to invoke to properly release the resource.

You can specify a custom deleter as a template parameter or as a constructor argument.

### Using Custom Deleters with `std::unique_ptr`

Full runnable code:

```
#include <iostream>
#include <cstdio>
#include <memory>

// Custom deleter for FILE*
struct FileCloser {
    void operator()(FILE* file) const {
        if (file) {
            std::fclose(file);
            std::cout << "File closed.\n";
        }
    }
};

int main() {
    // Open a file using fopen (C-style)
    std::unique_ptr<FILE, FileCloser> filePtr(std::fopen("example.txt", "w"));

    if (filePtr) {
        std::fprintf(filePtr.get(), "Hello, world!\n");
    } // filePtr destructor calls FileCloser::operator(), closing the file automatically
}
```

Here, `FileCloser` is a functor acting as a custom deleter that closes the file using `fclose`. When `filePtr` goes out of scope, the file is automatically closed, preventing resource leaks.

### Custom Deleters with `std::shared_ptr`

`std::shared_ptr` also supports custom deleters, which is especially useful for managing resources shared across multiple owners:

Full runnable code:

```
#include <iostream>
#include <memory>
```

```cpp
void socket_close(int* socket) {
    if (socket) {
        std::cout << "Closing socket " << *socket << std::endl;
        delete socket; // Simulate cleanup
    }
}

int main() {
    std::shared_ptr<int> socketPtr(new int(42), socket_close);

    // Multiple shared_ptr instances can share ownership safely
    std::shared_ptr<int> anotherPtr = socketPtr;

} // When last shared_ptr is destroyed, socket_close is called
```

The custom deleter here simulates closing a socket. Once all `shared_ptr` instances owning the socket are destroyed, the deleter is invoked exactly once.

### 12.3.1 Practical Use Cases

- Managing **file handles** opened with C APIs (`fopen`).
- Handling **network sockets** or OS-level resources.
- Cleaning up **custom objects** that require specific shutdown procedures.
- Replacing raw pointers in **legacy code** that uses special resource management.

### 12.3.2 Summary

Custom deleters extend the power of smart pointers beyond memory management, enabling **safe, automatic cleanup of any resource**. This ensures cleaner, safer code and reduces the risk of resource leaks in complex applications.

## 12.4 Examples: Managing Dynamic Memory Safely

Smart pointers are essential tools in modern C++ for managing dynamic memory safely and efficiently. They help avoid common pitfalls like memory leaks, dangling pointers, and manual `delete` mistakes. Below are comprehensive examples demonstrating how to use smart pointers with dynamic arrays, linked data structures, and polymorphic objects.

### Example 1: Managing Dynamic Arrays with `std::unique_ptr`

Using `std::unique_ptr` with arrays ensures automatic cleanup without manual `delete[]` calls.

Full runnable code:

```cpp
#include <iostream>
#include <memory>

int main() {
    // Allocate dynamic array of 5 integers
    std::unique_ptr<int[]> arr(new int[5]{1, 2, 3, 4, 5});

    // Access and modify elements using array syntax
    for (int i = 0; i < 5; ++i) {
        arr[i] *= 2;
    }

    // Print elements
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n'; // Output: 2 4 6 8 10

    // No need to manually delete[] arr; automatically freed
}
```

**Key point:** Use `std::unique_ptr<T[]>` for arrays, and avoid raw pointers entirely.

### Example 2: Building a Simple Linked List with `std::shared_ptr`

Smart pointers make managing linked data structures safer by automating node lifetime.

Full runnable code:

```cpp
#include <iostream>
#include <memory>

struct Node {
    int value;
    std::shared_ptr<Node> next;

    Node(int val) : value(val), next(nullptr) {}
};

int main() {
    auto head = std::make_shared<Node>(10);
    head->next = std::make_shared<Node>(20);
    head->next->next = std::make_shared<Node>(30);

    // Traverse list and print values
    for (auto current = head; current != nullptr; current = current->next) {
        std::cout << current->value << " -> ";
    }
    std::cout << "nullptr\n"; // Output: 10 -> 20 -> 30 -> nullptr
```

```
    // No manual deletes; nodes freed when no longer referenced
}
```

**Key point:** Using `std::shared_ptr` allows multiple parts of your program to share ownership safely.

**Example 3: Polymorphic Objects with `std::unique_ptr` and Virtual Destructors**

Smart pointers handle polymorphic objects correctly when destructors are virtual.

Full runnable code:

```
#include <iostream>
#include <memory>

class Base {
public:
    virtual void speak() { std::cout << "Base speaking\n"; }
    virtual ~Base() = default; // Virtual destructor
};

class Derived : public Base {
public:
    void speak() override { std::cout << "Derived speaking\n"; }
};

int main() {
    std::unique_ptr<Base> ptr = std::make_unique<Derived>();

    ptr->speak(); // Output: Derived speaking

    // Automatic cleanup calls Derived destructor correctly
}
```

**Key point:** Always ensure base classes have virtual destructors when using smart pointers with inheritance.

### 12.4.1 Why Use Smart Pointers?

- **Automatic cleanup:** No need for explicit `delete`, reducing leaks.
- **Exception safety:** Resources released even if exceptions occur.
- **Clear ownership semantics:** `unique_ptr` for exclusive ownership, `shared_ptr` for shared.
- **Simplified code:** Easier to write and maintain than raw pointer management.

### 12.4.2 Summary

Smart pointers transform manual memory management into safe, automated operations. By adopting them in dynamic arrays, linked structures, and polymorphic hierarchies, C++ programmers create robust, maintainable, and efficient codebases without sacrificing control or performance.

# Chapter 13.

## Concurrency Utilities in STL

1. Thread Support Library Basics
2. `std::thread`: Creating and Managing Threads
3. Mutexes, Locks, and Condition Variables
4. Atomic Operations
5. Practical Examples: Parallel STL Algorithms and Synchronization

# 13 Concurrency Utilities in STL

## 13.1 Thread Support Library Basics

With the rise of multicore processors, writing programs that perform multiple tasks concurrently has become essential for maximizing performance. C++11 introduced a standardized **Thread Support Library** as part of the STL to provide portable, efficient tools for multithreading.

### Motivation for Multithreading

Multithreading allows a program to execute multiple sequences of instructions (threads) **in parallel**, improving CPU utilization and responsiveness. Common uses include:

- Performing background computations without blocking user interfaces.
- Parallelizing tasks to reduce total execution time.
- Handling asynchronous I/O and event-driven programming.

### Basic Terminology and Thread Lifecycle

- **Thread**: The smallest unit of execution, having its own call stack but sharing process memory.
- **Main thread**: The thread where `main()` runs.
- **Spawned thread**: Any additional thread created during execution.
- **Thread lifecycle states**:
  - **Not started**: Thread object created but not running.
  - **Running**: Thread is executing its task.
  - **Blocked**: Thread is waiting (e.g., on I/O or synchronization).
  - **Joined or Detached**: Thread has finished execution or runs independently.

### Creating and Managing Threads in STL

C++ provides the `std::thread` class to manage threads. You can create a thread by passing a callable (function, lambda, or functor):

Full runnable code:

```cpp
#include <iostream>
#include <thread>

void task() {
    std::cout << "Thread running\n";
}

int main() {
    std::thread t(task);  // Spawn new thread executing task()

    t.join();             // Wait for thread to finish before continuing
```

```
    std::cout << "Thread joined\n";
}
```

- `std::thread t(task);` starts a new thread.
- `t.join();` blocks the main thread until `t` completes.
- Alternatively, you can call `t.detach()` to let the thread run independently.

#### Concurrency Challenges

Writing multithreaded programs introduces complexities such as:

- **Data races**: Concurrent access to shared data without synchronization leads to undefined behavior.
- **Deadlocks**: Two or more threads wait indefinitely for resources held by each other.
- **Race conditions**: The program outcome depends on unpredictable thread execution order.

#### How STL Helps Address These Issues

C++ STL concurrency utilities provide tools to safely manage threads and shared data:

- **Mutexes (`std::mutex`)** for exclusive access.
- **Locks (`std::lock_guard`, `std::unique_lock`)** for safer mutex management.
- **Condition variables (`std::condition_variable`)** for thread synchronization.
- **Atomic operations (`std::atomic`)** for lock-free thread-safe operations.

By combining these with `std::thread`, C++ programmers can build efficient, safe multi-threaded applications without platform-specific code.

### 13.1.1 Summary

C++11's Thread Support Library offers a powerful, standardized way to create and manage threads, helping developers harness parallelism. Understanding thread lifecycles and common concurrency pitfalls sets the foundation for mastering advanced synchronization techniques covered in later sections.

## 13.2 `std::thread`: Creating and Managing Threads

The `std::thread` class in C++11 provides a simple and portable way to create and manage threads. Each `std::thread` object represents a single thread of execution. This section explains how to launch threads, synchronize them with `join()`, or detach them to run independently, with practical examples.

### Creating and Launching Threads

You create a thread by instantiating `std::thread` with a callable — this can be a function, lambda expression, or callable object (functor):

Full runnable code:

```cpp
#include <iostream>
#include <thread>

void printMessage(int id) {
    std::cout << "Hello from thread " << id << "\n";
}

int main() {
    std::thread t1(printMessage, 1);  // Launch thread running printMessage(1)
    std::thread t2(printMessage, 2);  // Launch another thread

    // Wait for threads to complete
    t1.join();
    t2.join();

    return 0;
}
```

- The `std::thread` constructor starts the thread immediately.
- Arguments following the callable are passed to the thread function.
- Threads run concurrently with the main thread.

### Joining Threads: Waiting for Completion

Calling `join()` on a thread blocks the calling thread until the spawned thread finishes execution. It is essential to call `join()` (or `detach()`) on every `std::thread` object before it is destroyed to avoid program termination.

```cpp
t1.join(); // Waits for thread t1 to finish
```

If you forget to join or detach, the destructor of `std::thread` will call `std::terminate()`, ending the program abruptly.

### Detaching Threads: Running Independently

Alternatively, you can call `detach()` to allow a thread to run independently in the background:

```cpp
t1.detach(); // Thread continues running after main thread exits
```

- Detached threads run freely and cannot be joined later.
- Use detach cautiously: since you lose control, the program may exit before detached threads finish.

### Launching Multiple Threads

You can spawn many threads to parallelize tasks:

Full runnable code:

```cpp
#include <vector>
#include <iostream>
#include <thread>
void worker(int id) {
    std::cout << "Worker " << id << " starting\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::cout << "Worker " << id << " finished\n";
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 5; ++i) {
        threads.emplace_back(worker, i);
    }

    // Join all threads to ensure they finish before main exits
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "All workers done\n";
}
```

This example creates five worker threads that simulate work with sleep, then joins all to safely wait for their completion.

**Summary**

- `std::thread` launches threads immediately upon construction.
- Always call `join()` to wait for thread completion or `detach()` to run threads independently.
- Joining prevents premature termination and ensures safe synchronization.
- Managing multiple threads with containers like `std::vector<std::thread>` enables scalable parallelism.

By understanding and applying these fundamentals, you can safely create multithreaded programs leveraging C++ STL concurrency utilities.

## 13.3 Mutexes, Locks, and Condition Variables

In multithreaded programming, when multiple threads access shared data concurrently, race conditions can occur, causing unpredictable behavior and data corruption. To prevent this, **synchronization primitives** are used to control thread access to shared resources. The C++ STL provides several such primitives: `std::mutex`, `std::lock_guard`, `std::unique_lock`, and `std::condition_variable`. This section introduces these tools and demonstrates their usage.

### Thread Safety and Critical Sections

A **critical section** is a part of code that accesses shared resources and must not be executed by more than one thread at a time. To enforce this, a **mutex** (mutual exclusion) is used.

- **`std::mutex`** is a simple synchronization primitive used to protect critical sections.
- Threads lock the mutex before accessing shared data and unlock it after finishing.
- If another thread tries to lock the mutex while it's already locked, it blocks until the mutex is released.

#### `std::mutex` and `std::lock_guard`

Manual locking and unlocking of a mutex can lead to errors such as forgetting to unlock or exceptions causing premature exit. To manage this safely, **RAII wrappers** like `std::lock_guard` exist:

Full runnable code:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;
int shared_counter = 0;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // Lock mutex on scope entry
        ++shared_counter;                        // Critical section
    }                                            // Mutex automatically released here
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << shared_counter << "\n";
}
```

Here, `std::lock_guard` ensures that the mutex is released even if an exception occurs, preventing deadlocks.

#### `std::unique_lock`

`std::unique_lock` is a more flexible locking mechanism than `lock_guard`. It allows deferred locking, unlocking, and re-locking within its lifetime, making it useful for advanced scenarios, including waiting on condition variables.

Example:

```
std::unique_lock<std::mutex> lock(mtx, std::defer_lock); // Mutex not locked yet
lock.lock();    // Lock explicitly
```

```
// Critical section here
lock.unlock();  // Unlock explicitly
```

### std::condition_variable

Sometimes, threads need to wait for certain conditions before continuing execution, e.g., waiting for data availability or a flag. `std::condition_variable` facilitates this by allowing threads to wait and be notified when a condition changes.

- Threads wait on a condition variable, releasing the mutex while waiting.
- When notified (`notify_one` or `notify_all`), waiting threads wake up and re-acquire the mutex to check the condition.

Example of a producer-consumer pattern using condition variables:

Full runnable code:

```
#include <queue>
#include <iostream>
#include <condition_variable>

std::queue<int> dataQueue;
std::mutex mtx;
std::condition_variable cv;
bool done = false;

void producer() {
    for (int i = 1; i <= 5; ++i) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            dataQueue.push(i);
            std::cout << "Produced: " << i << "\n";
        }
        cv.notify_one();  // Notify consumer
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    {
        std::lock_guard<std::mutex> lock(mtx);
        done = true;
    }
    cv.notify_all();  // Notify consumer to stop waiting
}

void consumer() {
    std::unique_lock<std::mutex> lock(mtx);
    while (!done || !dataQueue.empty()) {
        cv.wait(lock, [] { return done || !dataQueue.empty(); });
        while (!dataQueue.empty()) {
            int value = dataQueue.front();
            dataQueue.pop();
            std::cout << "Consumed: " << value << "\n";
        }
    }
}

int main() {
```

```
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();
}
```

### Summary

- Use `std::mutex` to protect critical sections and prevent data races.
- Prefer `std::lock_guard` for simple, exception-safe locking.
- Use `std::unique_lock` for advanced locking scenarios, especially with condition variables.
- Use `std::condition_variable` to block threads until notified of a change, enabling thread communication and synchronization.

Together, these primitives help write safe and efficient multithreaded programs by controlling access and coordinating thread execution.

## 13.4 Atomic Operations

In multithreaded programming, **atomic operations** provide a way to perform certain operations on shared data without the overhead of locks or mutexes. The C++ Standard Library's `<atomic>` header offers atomic types and operations that guarantee **lock-free**, thread-safe manipulation of variables, helping to avoid race conditions while maintaining high performance.

### What Are Atomic Types?

An **atomic type** ensures that operations on its instances occur indivisibly, meaning no thread can observe the variable in a partially modified state. These operations are **atomic** with respect to other threads, so concurrent access is safe without explicit synchronization.

The STL provides `std::atomic<T>` for many fundamental types like `int`, `bool`, and pointers. For example:

```
#include <atomic>

std::atomic<int> counter(0);
```

Operations such as incrementing or comparing and exchanging values happen atomically, avoiding data races without locking.

### Use Cases for Atomic Operations

Atomic operations are especially useful when:

- You need **simple shared state updates** (like counters, flags).

- Lock contention must be minimized for performance reasons.
- You want to avoid the complexity and overhead of mutexes.
- You are implementing **lock-free** or **wait-free** algorithms.

However, atomic operations work best for simple cases. Complex data structures or critical sections often still require mutexes.

### Basic Example: Atomic Integer

Here's a simple example of incrementing a shared atomic counter from multiple threads:

Full runnable code:

```cpp
#include <iostream>
#include <thread>
#include <atomic>
#include <vector>

std::atomic<int> atomic_counter(0);

void increment() {
    for (int i = 0; i < 1000; ++i) {
        atomic_counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 4; ++i)
        threads.emplace_back(increment);

    for (auto& t : threads)
        t.join();

    std::cout << "Final counter: " << atomic_counter.load() << "\n"; // Expected 4000
}
```

In this example, `fetch_add` atomically increments `atomic_counter` without race conditions or locks. The `memory_order_relaxed` argument specifies minimal synchronization to improve performance where ordering constraints are not critical.

### Atomic Flags: Lightweight Synchronization

Another useful atomic type is `std::atomic_flag`, designed for simple boolean flags:

Full runnable code:

```cpp
#include <atomic>
#include <iostream>
#include <thread>

std::atomic_flag lock_flag = ATOMIC_FLAG_INIT;

void try_lock(int thread_id) {
```

```
    while (lock_flag.test_and_set(std::memory_order_acquire)) {
        // Spin-wait (busy-wait) until flag is cleared
    }
    std::cout << "Thread " << thread_id << " acquired lock\n";

    // Critical section simulation
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    lock_flag.clear(std::memory_order_release);
    std::cout << "Thread " << thread_id << " released lock\n";
}

int main() {
    std::thread t1(try_lock, 1);
    std::thread t2(try_lock, 2);

    t1.join();
    t2.join();
}
```

Here, `atomic_flag` acts as a **spinlock** where threads repeatedly attempt to acquire the lock by setting the flag atomically.

### Summary

- **`std::atomic` types** enable lock-free, thread-safe operations on fundamental data.
- Atomic operations minimize synchronization overhead in simple shared state scenarios.
- Use **`std::atomic<int>`** for counters and shared integers.
- Use **`std::atomic_flag`** for lightweight, low-level synchronization (spinlocks).
- For complex data or logic, prefer mutexes as atomics have limited use cases.

Atomic operations are powerful tools in concurrent programming, enabling safe, efficient communication between threads while reducing the cost of locking mechanisms.

## 13.5 Practical Examples: Parallel STL Algorithms and Synchronization

C++17 introduced **parallel algorithms** to the STL, allowing many standard algorithms to run concurrently on multiple threads, improving performance on large data sets with minimal programmer effort. This section demonstrates how to combine these parallel algorithms with thread synchronization to efficiently process data and manage output safely.

### Parallel STL Algorithms: Overview

Parallel STL algorithms are variants of existing algorithms that accept an execution policy parameter to specify parallel execution:

- `std::execution::seq` — sequential execution (default)
- `std::execution::par` — parallel execution using multiple threads

- `std::execution::par_unseq` — parallel and vectorized execution

For example, you can parallelize `std::for_each`, `std::sort`, and `std::transform`.

### Example 1: Parallel Data Processing

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> data(1'000'000);
    std::iota(data.begin(), data.end(), 1);

    // Parallel transform: square each element
    std::transform(std::execution::par, data.begin(), data.end(), data.begin(),
                   [](int x) { return x * x; });

    // Print first 5 squared values
    for (int i = 0; i < 5; ++i)
        std::cout << data[i] << ' ';
    std::cout << '\n';
}
```

Here, `std::transform` runs concurrently across the large vector to square each element, significantly speeding up processing on multi-core CPUs.

### Synchronizing Output with Mutexes

When multiple threads write to shared resources like `std::cout`, race conditions can corrupt output. Use `std::mutex` or `std::lock_guard` to synchronize:

```
#include <mutex>
std::mutex cout_mutex;

void print_element(int x) {
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << x << ' ';
}
```

Combining this with parallel algorithms requires care, as parallel algorithms do not guarantee order and invoke the callable concurrently.

### Example 2: Parallel `for_each` with Synchronized Output

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>
#include <mutex>
```

```
std::mutex cout_mutex;

int main() {
    std::vector<int> numbers = {10, 20, 30, 40, 50};

    std::for_each(std::execution::par, numbers.begin(), numbers.end(),
                  [](int n) {
                      std::lock_guard<std::mutex> lock(cout_mutex);
                      std::cout << "Number: " << n << '\n';
                  });

    return 0;
}
```

This ensures output from different threads does not interleave, maintaining readability.

### Pitfalls and Best Practices

- **Avoid data races:** Use synchronization primitives when accessing shared state.
- **Order not guaranteed:** Parallel algorithms do not preserve the order of operations unless documented.
- **Performance trade-offs:** For small data, overhead of parallelization may outweigh benefits.
- **Exceptions:** Exception handling in parallel algorithms can be complex; prefer careful error management.

### Summary

- Parallel STL algorithms simplify concurrent data processing.
- Synchronization tools like mutexes are essential to protect shared resources during parallel execution.
- Using parallel algorithms can dramatically improve performance on large datasets.
- Careful design and testing are needed to avoid concurrency pitfalls.

By combining parallel STL algorithms with synchronization primitives, modern C++ lets you write efficient, safe concurrent programs with clear and maintainable code.

# Chapter 14.

# STL in Modern C (C11 and Beyond)

1. Range-based For Loops with STL
2. Move Semantics and Emplace Functions (`emplace_back`, `emplace`)
3. `std::optional`, `std::variant`, and `std::any` Overview
4. Using STL with Modern Language Features for Cleaner Code

# 14 STL in Modern C (C11 and Beyond)

## 14.1 Range-based For Loops with STL

Range-based for loops, introduced in C++11, provide a modern and concise syntax to iterate over STL containers and other iterable objects. They greatly simplify code that traditionally relied on explicit iterators or index-based loops, making it easier to read and write.

### Traditional Iteration vs. Range-based For

Before range-based for, iterating a container like `std::vector` typically required explicit use of iterators:

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

Or using an index-based loop for random-access containers:

```
for (size_t i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << ' ';
}
```

Both approaches involve boilerplate code for managing iterators or indices.

### Range-based For Loop Syntax

The range-based for loop eliminates this boilerplate by abstracting the iteration mechanism:

```
for (const auto& element : vec) {
    std::cout << element << ' ';
}
```

Here, `element` takes each value from `vec` in sequence. The compiler generates the necessary iterator code behind the scenes.

### Benefits of Range-based For

- **Simplicity:** Less verbose, easier to write.
- **Safety:** Avoids off-by-one errors common in index loops.
- **Readability:** Focuses on the operation rather than iteration details.
- **Flexibility:** Works with any container supporting `begin()` and `end()`.

### Examples with Different STL Containers

**Vector:**

```
std::vector<std::string> names = {"Alice", "Bob", "Charlie"};
for (const auto& name : names) {
    std::cout << name << '\n';
}
```

**Map:**

```cpp
std::map<int, std::string> idToName = {{1, "Alice"}, {2, "Bob"}};
for (const auto& [id, name] : idToName) {
    std::cout << id << ": " << name << '\n';
}
```

*Note:* The above example uses structured bindings (C++17) for clarity.

**Set:**

```cpp
std::set<int> uniqueNumbers = {3, 1, 4, 1, 5};
for (int num : uniqueNumbers) {
    std::cout << num << ' ';
}
```

### Summary

Range-based for loops offer a clean and efficient way to traverse STL containers. Compared to traditional iterator or index loops, they reduce verbosity, minimize errors, and improve code clarity. By leveraging this feature, C++ programmers write more expressive and maintainable code, especially when working with diverse STL containers like vectors, maps, and sets.

## 14.2 Move Semantics and Emplace Functions (`emplace_back`, `emplace`)

C++11 introduced **move semantics** and **perfect forwarding** to improve performance by eliminating unnecessary copying of objects. These features are particularly impactful when working with STL containers, optimizing how objects are inserted or constructed inside containers.

### Move Semantics: A Quick Overview

Traditionally, inserting an object into a container like `std::vector` involved copying the object. Copying can be expensive, especially for complex or large objects. Move semantics enable *transferring* resources from a source object (an rvalue) to a new target object without costly deep copies.

For example:

```cpp
std::string s = "Hello";
std::vector<std::string> vec;

vec.push_back(s);          // Copies s
vec.push_back(std::move(s)); // Moves s, leaving s in a valid but unspecified state
```

The move constructor steals the internal data (like heap buffers) instead of copying, improving efficiency.

### Perfect Forwarding and `emplace` Functions

C++11 also introduced **perfect forwarding**, which allows forwarding arguments exactly as received (whether lvalues or rvalues) to constructors. This enables the STL containers to *construct elements in place* within their storage, avoiding temporary objects and extra moves or copies.

This is where `emplace_back` and `emplace` shine:

- `push_back` and `insert` take an object and copy or move it into the container.
- `emplace_back` and `emplace` construct the object *directly* in the container's memory, forwarding the constructor arguments.

### Practical Differences and Benefits

Consider a class `Widget` with a costly copy constructor:

```
struct Widget {
    Widget(int x, std::string name) : x(x), name(std::move(name)) {}
    int x;
    std::string name;
};
```

#### Using `push_back`:

```
std::vector<Widget> widgets;
Widget w(42, "example");
widgets.push_back(w);   // Copies w
widgets.push_back(std::move(w)); // Moves w
```

#### Using `emplace_back`:

```
widgets.emplace_back(42, "example"); // Constructs Widget directly inside the vector
```

With `emplace_back`, the `Widget` is constructed *in place* using the provided arguments, avoiding copy or move operations altogether.

### Code Example: Emplace vs Push Back

Full runnable code:

```
#include <iostream>
#include <vector>
#include <string>

struct Widget {
    Widget(int id, std::string n) : id(id), name(std::move(n)) {
        std::cout << "Constructed Widget " << id << '\n';
    }
    Widget(const Widget& other) : id(other.id), name(other.name) {
        std::cout << "Copied Widget " << id << '\n';
    }
    Widget(Widget&& other) noexcept : id(other.id), name(std::move(other.name)) {
        std::cout << "Moved Widget " << id << '\n';
    }
    int id;
```

```
    std::string name;
};

int main() {
    std::vector<Widget> v;

    std::cout << "Using push_back:\n";
    Widget w(1, "Alpha");
    v.push_back(w);            // copy
    v.push_back(std::move(w)); // move

    std::cout << "\nUsing emplace_back:\n";
    v.emplace_back(2, "Beta"); // construct in place

    return 0;
}
```

**Output:**

```
Using push_back:
Constructed Widget 1
Copied Widget 1
Moved Widget 1

Using emplace_back:
Constructed Widget 2
```

**Summary**

- Move semantics optimize container insertions by enabling resource transfers rather than copies.
- `emplace_back` and `emplace` leverage perfect forwarding to construct elements *in place*, avoiding temporary objects and extra moves/copies.
- Using `emplace` functions often improves performance and reduces overhead, especially for complex objects.
- Prefer `emplace_back` or `emplace` when you want to construct objects inside containers with direct constructor arguments.

These modern features make STL containers more efficient and expressive, enhancing the power of C++11 and beyond.

## 14.3 `std::optional`, `std::variant`, and `std::any` Overview

C++17 introduced powerful utilities—`std::optional`, `std::variant`, and `std::any`—that help write safer, more expressive, and flexible code. These types improve how you handle nullable values, type-safe unions, and type-erased storage in STL-based applications.

### `std::optional`: Handling Nullable Values Safely

`std::optional<T>` represents an object that **may or may not contain a value** of type `T`. It's a modern alternative to pointers or sentinel values (`nullptr`, `-1`) for indicating the absence of a value.

- **Purpose:** Express "nullable" or optional values without unsafe pointers.
- **Syntax:** `std::optional<int> opt;` declares an optional integer.
- **Usage:** You can check whether it contains a value with `opt.has_value()` or convert to `bool`.

### Example: Optional return from a function

Full runnable code:

```cpp
#include <iostream>
#include <optional>
#include <vector>

std::optional<int> find_even(const std::vector<int>& numbers) {
    for (int n : numbers) {
        if (n % 2 == 0) return n;  // Return first even number
    }
    return std::nullopt;  // No even number found
}

int main() {
    std::vector<int> nums = {1, 3, 5, 6, 7};
    auto result = find_even(nums);

    if (result)
        std::cout << "Found even number: " << *result << "\n";
    else
        std::cout << "No even number found.\n";
}
```

### `std::variant`: Type-Safe Unions for Multiple Possible Types

`std::variant<Ts...>` is a **type-safe union** that holds exactly one of the specified types at a time. It replaces unsafe unions and `void*` pointers, providing compile-time type safety.

- **Purpose:** Store different types in a single variable with safe access.
- **Syntax:** `std::variant<int, std::string> v;`
- **Access:** Use `std::get<T>(v)` or `std::visit` for safe visitation.

### Example: Variant-based polymorphism

Full runnable code:

```cpp
#include <iostream>
#include <variant>
#include <string>

using VarType = std::variant<int, std::string>;
```

```cpp
void print_variant(const VarType& v) {
    std::visit([](auto&& arg) {
        std::cout << "Value: " << arg << "\n";
    }, v);
}

int main() {
    VarType v = 10;
    print_variant(v);

    v = std::string("Hello Variant");
    print_variant(v);
}
```

### `std::any`: Type-Erased Storage for Arbitrary Types

`std::any` can **hold any type of object** by erasing the specific type information, allowing storage of heterogeneous values without templates.

- **Purpose:** Store any type dynamically with runtime type checking.
- **Syntax:** `std::any a = 5;`
- **Access:** Use `std::any_cast<T>(a)` to retrieve the stored value safely.

### Example: Using `std::any`

Full runnable code:

```cpp
#include <iostream>
#include <any>
#include <string>

int main() {
    std::any a = 42;
    std::cout << "Int stored in any: " << std::any_cast<int>(a) << "\n";

    a = std::string("Now a string");
    std::cout << "String stored in any: " << std::any_cast<std::string>(a) << "\n";

    try {
        // This cast will throw because 'a' currently holds a string
        std::cout << std::any_cast<int>(a) << "\n";
    } catch (const std::bad_any_cast& e) {
        std::cout << "Bad any_cast: " << e.what() << "\n";
    }
}
```

#### 14.3.1 Summary

| Utility | Purpose | Type Safety | Example Use Case |
|---|---|---|---|
| `std::optional` | Optional/nullable values | Compile-time safe | Function return indicating "no result" |
| `std::variant` | Holds one value from multiple possible types | Compile-time safe | Type-safe polymorphism without inheritance |
| `std::any` | Type-erased storage for arbitrary types | Runtime-checked | Heterogeneous containers, plugin systems |

Together, these utilities greatly enhance the flexibility and safety of modern C++ code, allowing you to express intent clearly and handle varied data types elegantly within STL-based programs.

## 14.4 Using STL with Modern Language Features for Cleaner Code

Modern C++ (C++11 and beyond) introduces features that significantly enhance the expressiveness, readability, and maintainability of STL-based code. By combining these features with STL containers and algorithms, developers can write safer, cleaner, and more concise code.

### `auto`: Type Inference for Readability

Using `auto` eliminates verbose type declarations, especially when working with complex iterator or lambda return types.

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

STL iterators and return types from algorithms often have long, unreadable types. `auto` simplifies this without losing type safety.

### `decltype`: Precise Type Extraction

`decltype` helps deduce the exact type of an expression, which is especially useful when writing generic code or interacting with STL algorithms.

```
auto x = 10;
decltype(x) y = 20; // y is also an int
```

You can also use `decltype(auto)` to perfectly forward types in generic functions.

### Lambdas: In-place, Readable Function Objects

Lambdas allow inline, anonymous functions—ideal for use with STL algorithms like `std::for_each`, `std::sort`, and `std::transform`.

```
std::vector<int> nums = {5, 3, 1, 4, 2};
std::sort(nums.begin(), nums.end(), [](int a, int b) {
    return a < b;
});
```

Lambdas are often clearer than writing separate function objects or functor classes, and they support captures for local context.

### `constexpr`: Compile-Time Evaluation

With `constexpr`, functions and values can be evaluated at compile time, improving performance and safety when applicable.

```
constexpr int square(int x) { return x * x; }

constexpr int size = square(5); // evaluated at compile time
std::array<int, size> arr;
```

When used with STL containers (especially fixed-size ones like `std::array`), `constexpr` ensures static safety.

### Structured Bindings: Clean Tuple/Pairs Unpacking

C++17's structured bindings make it easy to unpack `std::pair` and `std::tuple` without verbose `std::get`.

```
std::map<std::string, int> ages = {{"Alice", 30}, {"Bob", 25}};
for (const auto& [name, age] : ages) {
    std::cout << name << ": " << age << "\n";
}
```

This drastically improves readability when iterating over associative containers.

## 14.4.1 Combined Example

```
std::vector<std::string> words = {"apple", "banana", "pear", "kiwi"};

std::sort(words.begin(), words.end(), [](const auto& a, const auto& b) {
    return a.length() < b.length();
});

for (const auto& word : words) {
    std::cout << word << "\n";
}
```

This snippet uses `auto`, lambdas, and range-based for loops to produce clean, expressive STL code.

### 14.4.2 Summary

Modern C++ features, when combined with STL, offer:

- **Cleaner syntax** (`auto`, lambdas, structured bindings)
- **Compile-time safety** (`constexpr`, `decltype`)
- **Enhanced flexibility** (generic programming, concise iteration)

These tools encourage writing idiomatic, elegant C++ code while leveraging the full power of the STL.

# Chapter 15.

# Performance Considerations and Best Practices

1. Choosing the Right Container and Algorithm
2. Avoiding Common Pitfalls and Inefficient Patterns
3. Measuring and Improving STL Code Performance
4. Case Study: Optimizing a Real-World Application with STL

# 15 Performance Considerations and Best Practices

## 15.1 Choosing the Right Container and Algorithm

Selecting the appropriate STL container and algorithm is critical to writing efficient and maintainable C++ code. Each container offers distinct performance characteristics, and the right choice depends on the specific access patterns, mutation needs, and memory constraints of your application. Similarly, using the most suitable algorithm can drastically reduce runtime overhead.

### Understanding Container Trade-offs

STL containers differ in their internal structures, affecting performance:

- **`std::vector`**: Best for dynamic arrays with fast random access (`O(1)`), but inserting/removing elements in the middle is costly (`O(n)`). Use when elements are frequently accessed by index and appends dominate.
- **`std::list` / `std::forward_list`**: Implement linked lists with fast insertions/removals at any position (`O(1)` if iterator is known), but lack random access. Use when frequent middle insertions/removals are needed, and traversal performance is less critical.
- **`std::deque`**: Efficient insertion/removal at both ends (`O(1)`), while still supporting random access. Prefer it over `vector` if front operations are frequent.
- **`std::set` / `std::map`**: Ordered containers using balanced binary trees (`O(log n)` insert, delete, search). Ideal when ordering and uniqueness are required.
- **`std::unordered_set` / `std::unordered_map`**: Hash-based containers offering average-case `O(1)` access but no order. Great for fast lookups where ordering is not important.

### Choosing the Right Algorithm

STL algorithms are container-agnostic and operate on iterator ranges. The algorithm choice should complement the container and the problem:

- Use **`std::sort`** with `vector` or `deque`, which have random-access iterators. For non-random-access containers like `list`, use `list::sort()`.
- Use **`std::find`**, **`std::count`**, and other linear search algorithms with sequences when no faster index structure exists.
- When frequent key lookups are needed, prefer associative containers like `map` or `unordered_map` instead of `vector` with `std::find`.

### Comparative Scenario

Suppose you're building a lookup table for configuration values:

```
std::map<std::string, int> config = {
    {"timeout", 30}, {"retries", 5}, {"delay", 10}
};
```

If you need frequent ordered iteration, `std::map` is appropriate. But if order isn't needed and you care about speed:

```
std::unordered_map<std::string, int> config = {
    {"timeout", 30}, {"retries", 5}, {"delay", 10}
};
```

This provides faster average-case access.

### General Guidelines

- Prefer `vector` unless another container is clearly more suitable—it's cache-friendly and performs well for most use cases.
- Use `reserve()` on vectors to avoid repeated reallocations.
- For constant time removals, `unordered_set` or `unordered_map` may be ideal.
- Always prefer algorithms over manual loops—they're optimized and more expressive.

By understanding container properties and algorithm requirements, developers can make informed choices that yield scalable, efficient STL-based applications.

## 15.2 Avoiding Common Pitfalls and Inefficient Patterns

Even though the STL offers powerful and efficient tools, misuse can easily lead to subtle bugs and performance degradation. Recognizing common pitfalls and applying best practices ensures that your STL code remains both correct and performant.

### Unnecessary Copying

Many STL algorithms and container operations involve value semantics, which can result in unexpected copying if not carefully handled.

**Pitfall**: Returning large containers by value without move semantics.

```
std::vector<int> getLargeVector() {
    std::vector<int> v(1000000, 42);
    return v; // Risk: costly copy if RVO/move isn't used
}
```

**Fix**: Use `std::move` or rely on Return Value Optimization (RVO) in modern compilers.

```
std::vector<int> getLargeVector() {
    std::vector<int> v(1000000, 42);
    return v; // Efficient with RVO or move constructor
}
```

Also, prefer `emplace_back` over `push_back` when constructing elements in-place to avoid temporary objects.

### Iterator Invalidation

Many STL containers invalidate iterators when modified, especially during insertions or deletions.

**Pitfall**: Using iterators after erasing elements from a vector.

```
std::vector<int> v = {1, 2, 3, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == 3) {
        v.erase(it); // Undefined behavior: 'it' is invalid after erase
    }
}
```

**Fix**: Use the return value of `erase`, which points to the next valid element.

```
for (auto it = v.begin(); it != v.end(); ) {
    if (*it == 3) {
        it = v.erase(it); // Safe: 'it' is updated correctly
    } else {
        ++it;
    }
}
```

### Misusing Algorithms with Wrong Iterators

Using algorithms that require specific iterator types with unsupported containers can cause inefficiencies or compilation errors.

**Pitfall**: Applying `std::sort` on a `std::list`.

```
std::list<int> lst = {3, 1, 4};
std::sort(lst.begin(), lst.end()); // Error: list iterators are not random access
```

**Fix**: Use `list::sort()` instead.

```
lst.sort(); // Correct and efficient
```

### Improper Synchronization in Multithreaded Code

Accessing STL containers from multiple threads without synchronization leads to race conditions.

**Pitfall**:

```
std::vector<int> data;
std::thread t1([&](){ data.push_back(1); });
std::thread t2([&](){ data.push_back(2); });
```

**Fix**: Use a mutex to protect shared access.

```
std::mutex mtx;
std::thread t1([&](){ std::lock_guard<std::mutex> lock(mtx); data.push_back(1); });
```

### 15.2.1 Summary

Avoiding common STL pitfalls requires awareness of copy semantics, iterator validity, algorithm constraints, and thread safety. Favor modern C++ practices like move semantics, range-based loops, and `auto` to write clean, efficient, and robust code.

## 15.3 Measuring and Improving STL Code Performance

Optimizing C++ STL-based code requires both accurate measurement and targeted improvements. Without reliable metrics, optimization can lead to premature or misguided changes. This section focuses on tools and techniques to profile STL code and enhance its efficiency with data-driven decisions.

### Profiling and Benchmarking Tools

Several tools are available to measure performance characteristics of STL code:

- **`std::chrono`** (C++11): Measure execution time precisely.
- **Valgrind (Linux)** or **Instruments (macOS)**: Analyze memory usage and leaks.
- **perf** (Linux), **Visual Studio Profiler** (Windows): Capture CPU usage, hotspots.
- **Google Benchmark**: A dedicated microbenchmarking library for reliable timing and comparison.

For example, using `std::chrono` to benchmark a sorting operation:

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

int main() {
    std::vector<int> data(1000000);
    std::generate(data.begin(), data.end(), rand);

    auto start = std::chrono::high_resolution_clock::now();
    std::sort(data.begin(), data.end());
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = end - start;
    std::cout << "Sort time: " << diff.count() << " seconds\n";
}
```

### Performance Tuning Strategies

Once bottlenecks are identified, the next step is optimization. Some common STL performance improvements include:

- **Choosing better containers**: `std::unordered_map` can outperform `std::map` for large datasets when order is unimportant.
- **Reducing reallocations**: Use `reserve()` for vectors when size is predictable.
- **Using `emplace` instead of `insert` or `push_back`** to avoid unnecessary copying.

**Before Optimization (copy-heavy loop)**:

```
std::vector<std::string> names;
for (int i = 0; i < 10000; ++i) {
    std::string s = "User" + std::to_string(i);
    names.push_back(s);
}
```

**After Optimization (emplace to avoid copy)**:

```
std::vector<std::string> names;
names.reserve(10000); // Prevents repeated reallocations
for (int i = 0; i < 10000; ++i) {
    names.emplace_back("User" + std::to_string(i));
}
```

This reduces the number of string copies and memory allocations significantly.

### Memory Usage and Cache Optimization

STL containers allocate memory dynamically, so reducing fragmentation is important. Favor contiguous containers like `std::vector` for better cache locality.

**Example**: Switching from `std::list` to `std::vector` can lead to drastic performance improvements in iteration-heavy workloads due to better cache friendliness.

### Summary

Improving STL performance is a process: measure, analyze, and then optimize. Use profiling tools to locate inefficiencies, prefer cache-friendly containers like `vector`, reduce dynamic allocations with `reserve`, and avoid unnecessary copying with `emplace` and move semantics. Always validate improvements with benchmarks to ensure real gains.

## 15.4 Case Study: Optimizing a Real-World Application with STL

In this section, we walk through a real-world example where STL-based code is analyzed, optimized, and benchmarked to improve performance. The application in question is a simple log processor that ingests lines of data, extracts keywords, and counts frequencies.

### Initial Implementation

```
#include <iostream>
#include <map>
#include <string>
```

```cpp
#include <sstream>
#include <vector>

void processLogs(const std::vector<std::string>& logs) {
    std::map<std::string, int> keywordCount;

    for (const auto& line : logs) {
        std::istringstream iss(line);
        std::string word;
        while (iss >> word) {
            ++keywordCount[word];
        }
    }

    for (const auto& [word, count] : keywordCount) {
        std::cout << word << ": " << count << "\n";
    }
}
```

**Performance Characteristics:**

- Processes 100,000 log lines in ~**2.3 seconds**.
- High memory usage due to `std::map` (tree-based structure).
- Frequent string copies and dynamic memory allocation.

### Step 1: Profiling and Bottleneck Detection

Using tools like `gprof` or `perf`, we identify bottlenecks:

- `std::map` insertions are expensive (`O(log n)`).
- Copying strings during parsing increases overhead.
- `std::ostringstream` is slower compared to tokenizing without streams.

### Step 2: Optimizing Containers and Parsing

We replace `std::map` with `std::unordered_map` for average `O(1)` inserts and use `string_view` to avoid copying substrings.

**Optimized Version:**

```cpp
#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>
#include <string_view>

void processLogsOptimized(const std::vector<std::string>& logs) {
    std::unordered_map<std::string, int> keywordCount;

    for (const auto& line : logs) {
        size_t start = 0, end;
        while ((end = line.find(' ', start)) != std::string::npos) {
            std::string_view word(line.data() + start, end - start);
            ++keywordCount[std::string(word)]; // Convert view to string for map key
            start = end + 1;
        }
```

```
        std::string_view lastWord(line.data() + start);
        ++keywordCount[std::string(lastWord)];
    }

    for (const auto& [word, count] : keywordCount) {
        std::cout << word << ": " << count << "\n";
    }
}
```

### Step 3: Benchmarking Improvements

- **Processing time reduced** from ~2.3 seconds to ~**1.1 seconds**.
- **Memory usage lowered** by avoiding intermediate string objects.
- **Insertion speed improved** due to `unordered_map`'s hash table design.

### Key Takeaways

| Optimization | Effect |
|---|---|
| `unordered_map` | Faster insertion and lookup |
| `string_view` | Reduced allocations and copies |
| Manual parsing | More control and performance vs streams |

### Conclusion

This case study shows how careful profiling and STL expertise can lead to significant performance gains. By choosing the right container (`unordered_map`), minimizing unnecessary object creation (`string_view`), and replacing generic parsing with targeted logic, we doubled the performance of a common data processing task. These techniques generalize well across many STL-driven applications.

# Chapter 16.

## Real-World Projects and Examples

1. Building a Contact Manager Using STL Containers and Algorithms
2. Implementing a Simple Cache with `std::unordered_map`
3. Text Processing and Analysis Using STL Algorithms
4. Interactive CLI To-Do List Application Using STL
5. Advanced: Custom Container Implementation

# 16 Real-World Projects and Examples

## 16.1 Building a Contact Manager Using STL Containers and Algorithms

In this section, we'll build a simple contact manager using STL containers and algorithms. This project demonstrates practical use of `std::vector`, `std::map`, and STL algorithms for storing, managing, and querying contact data.

### Overview

The application maintains a list of contacts, each with a name, phone number, and email. It supports CRUD (Create, Read, Update, Delete) operations, along with search and sort functionality. The primary data structure will be a `std::vector<Contact>`, which allows efficient iteration and manipulation.

### Contact Structure

We define a simple `Contact` struct:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

struct Contact {
    std::string name;
    std::string phone;
    std::string email;
};

void printContact(const Contact& c) {
    std::cout << "Name: " << c.name
              << ", Phone: " << c.phone
              << ", Email: " << c.email << "\n";
}
```

### CRUD and Utility Functions

```
void addContact(std::vector<Contact>& contacts, const Contact& newContact) {
    contacts.push_back(newContact);
}

void listContacts(const std::vector<Contact>& contacts) {
    std::cout << "--- Contact List ---\n";
    for (const auto& c : contacts)
        printContact(c);
}

void deleteContact(std::vector<Contact>& contacts, const std::string& name) {
    contacts.erase(std::remove_if(contacts.begin(), contacts.end(),
        [&name](const Contact& c) {
            return c.name == name;
        }), contacts.end());
```

```
}

void searchContacts(const std::vector<Contact>& contacts, const std::string& query) {
    std::cout << "--- Search Results ---\n";
    for (const auto& c : contacts) {
        if (c.name.find(query) != std::string::npos ||
            c.email.find(query) != std::string::npos) {
            printContact(c);
        }
    }
}

void sortContactsByName(std::vector<Contact>& contacts) {
    std::sort(contacts.begin(), contacts.end(),
        [](const Contact& a, const Contact& b) {
            return a.name < b.name;
        });
}
```

### Interactive CLI Driver

```
int main() {
    std::vector<Contact> contacts;
    int choice;

    while (true) {
        std::cout << "\nContact Manager\n"
                  << "1. Add Contact\n"
                  << "2. List Contacts\n"
                  << "3. Delete Contact\n"
                  << "4. Search Contacts\n"
                  << "5. Sort Contacts by Name\n"
                  << "0. Exit\n"
                  << "Enter choice: ";
        std::cin >> choice;
        std::cin.ignore();

        if (choice == 0) break;

        if (choice == 1) {
            Contact c;
            std::cout << "Name: "; std::getline(std::cin, c.name);
            std::cout << "Phone: "; std::getline(std::cin, c.phone);
            std::cout << "Email: "; std::getline(std::cin, c.email);
            addContact(contacts, c);
        } else if (choice == 2) {
            listContacts(contacts);
        } else if (choice == 3) {
            std::string name;
            std::cout << "Enter name to delete: ";
            std::getline(std::cin, name);
            deleteContact(contacts, name);
        } else if (choice == 4) {
            std::string query;
            std::cout << "Search by name or email: ";
            std::getline(std::cin, query);
            searchContacts(contacts, query);
```

```
        } else if (choice == 5) {
            sortContactsByName(contacts);
            std::cout << "Contacts sorted by name.\n";
        } else {
            std::cout << "Invalid choice.\n";
        }
    }

    return 0;
}
```

### Summary

This example illustrates how STL containers (`std::vector`) and algorithms (`std::sort`, `std::remove_if`, and lambda expressions) can be effectively combined to build a functional contact manager. It demonstrates core STL principles—efficiency, clarity, and expressive code—while remaining extensible for future enhancements such as file I/O, duplicate checking, or GUI integration.

## 16.2 Implementing a Simple Cache with `std::unordered_map`

A cache is a data structure that temporarily stores frequently accessed data to speed up retrieval. The Standard Template Library (STL) provides `std::unordered_map`, a hash table that offers average constant-time complexity for insertions and lookups—ideal for implementing a simple cache.

In this section, we'll design a lightweight key-value cache supporting fast access, insertion, and a basic Least Recently Used (LRU) eviction policy.

### Design Overview

We use `std::unordered_map` to map keys to values for fast access. To implement an eviction strategy such as LRU, we also use a `std::list` to track access order and a second map from keys to list iterators for constant-time updates.

### Cache Implementation

Full runnable code:

```
#include <iostream>
#include <unordered_map>
#include <list>
#include <string>

template<typename Key, typename Value>
class LRUCache {
    using ListIt = typename std::list<Key>::iterator;

    size_t capacity;
```

```
    std::unordered_map<Key, std::pair<Value, ListIt>> cache;
    std::list<Key> usage;

public:
    LRUCache(size_t cap) : capacity(cap) {}

    void put(const Key& key, const Value& value) {
        auto it = cache.find(key);
        if (it != cache.end()) {
            // Key exists: update value and usage
            it->second.first = value;
            usage.erase(it->second.second);
            usage.push_front(key);
            it->second.second = usage.begin();
        } else {
            // Evict if full
            if (cache.size() >= capacity) {
                Key lru = usage.back();
                usage.pop_back();
                cache.erase(lru);
            }
            usage.push_front(key);
            cache[key] = { value, usage.begin() };
        }
    }

    bool get(const Key& key, Value& value) {
        auto it = cache.find(key);
        if (it == cache.end()) return false;

        // Move accessed key to front
        usage.erase(it->second.second);
        usage.push_front(key);
        it->second.second = usage.begin();
        value = it->second.first;
        return true;
    }

    void display() const {
        std::cout << "Cache content (most recent first): ";
        for (const auto& k : usage)
            std::cout << k << " ";
        std::cout << "\n";
    }
};

int main() {
    LRUCache<std::string, std::string> cache(3);

    cache.put("a", "apple");
    cache.put("b", "banana");
    cache.put("c", "cherry");
    cache.display();

    std::string result;
    if (cache.get("a", result)) {
        std::cout << "Got: " << result << "\n";
    }
```

```
    cache.put("d", "date");  // Evicts "b"
    cache.display();

    return 0;
}
```

### Conclusion

Using `std::unordered_map` enables efficient key-value access in constant average time. Combined with a `std::list`, we implement an LRU eviction policy while maintaining performance. This cache structure is useful in applications such as memoization, image processing, and database query optimization. It also demonstrates how STL containers can be composed for practical, real-world systems.

## 16.3 Text Processing and Analysis Using STL Algorithms

Text processing is a common task in many real-world applications such as search engines, log analyzers, and natural language processing tools. The C++ Standard Template Library (STL) offers powerful containers and algorithms that make parsing, tokenizing, and analyzing text efficient and expressive.

This section walks through a practical example that reads a block of text, splits it into words (tokenization), and performs frequency analysis using STL tools such as `std::map`, `std::istringstream`, `std::for_each`, and `std::accumulate`.

### Tokenization and Word Counting

We'll write a program that processes an input text and counts the frequency of each word.

Full runnable code:

```
#include <iostream>
#include <sstream>
#include <map>
#include <string>
#include <algorithm>
#include <cctype>

// Helper to convert string to lowercase
std::string to_lower(const std::string& word) {
    std::string result = word;
    std::transform(result.begin(), result.end(), result.begin(),
                   [](unsigned char c) { return std::tolower(c); });
    return result;
}

// Helper to remove punctuation from a word
std::string strip_punctuation(const std::string& word) {
    std::string result;
```

```
    std::copy_if(word.begin(), word.end(), std::back_inserter(result),
                 [](unsigned char c) { return std::isalnum(c); });
    return result;
}

int main() {
    std::string text =
        "The quick brown fox jumps over the lazy dog. "
        "The dog, surprised, barked loudly!";

    std::istringstream iss(text);
    std::map<std::string, int> word_count;
    std::string word;

    while (iss >> word) {
        word = strip_punctuation(to_lower(word));
        ++word_count[word];
    }

    std::cout << "Word Frequencies:\n";
    for (const auto& [key, count] : word_count) {
        std::cout << key << ": " << count << "\n";
    }

    return 0;
}
```

### Explanation

- **`std::istringstream`** is used for splitting the string into whitespace-separated words.
- **`std::map`** is used to associate each unique word (key) with its count (value).
- **`std::transform`** and **`std::copy_if`** help normalize words (convert to lowercase and remove punctuation).
- **Structured bindings** (C++17) are used in the `for` loop for concise iteration over map entries.

### Enhancing with `std::accumulate` (Optional Insight)

If the word counts were stored in a `std::vector<std::pair<std::string, int>>`, we could use `std::accumulate` to sum total word counts:

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

int total_words = std::accumulate(
    word_count.begin(), word_count.end(), 0,
    [](int sum, const auto& pair) { return sum + pair.second; });

std::cout << "Total words: " << total_words << "\n";
```

This demonstrates how STL algorithms allow elegant expression of complex logic.

### Conclusion

Text analysis with STL is efficient and modular. By combining `std::map`, `std::transform`, and `std::accumulate`, developers can perform non-trivial tasks like word frequency analysis with concise and maintainable code. These patterns are widely applicable in building tools for data analysis, indexing, and report generation.

## 16.4 Interactive CLI To-Do List Application Using STL

In this section, we'll build a simple command-line To-Do List application using standard STL components like `std::vector`, `std::string`, and `std::algorithm`. The app will allow users to add tasks, list them, mark them as done, remove them, and sort them. This example demonstrates how STL containers and algorithms can be combined to manage state and user interaction in a lightweight yet powerful way.

### Features and Design

The application supports:

- Adding a task
- Listing all tasks
- Marking tasks as complete
- Removing tasks
- Sorting tasks alphabetically

We'll define a `Task` struct and store the task list in a `std::vector`. We'll use `std::getline` for input parsing and `std::sort`, `std::remove_if`, and `std::for_each` to manipulate the task list.

### Complete Code Example

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <iomanip>

struct Task {
    std::string description;
    bool completed = false;
};

void add_task(std::vector<Task>& tasks, const std::string& desc) {
    tasks.push_back({desc, false});
}

void list_tasks(const std::vector<Task>& tasks) {
```

```
    std::cout << "\nTo-Do List:\n";
    for (size_t i = 0; i < tasks.size(); ++i) {
        std::cout << std::setw(2) << i + 1 << ". ["
                  << (tasks[i].completed ? 'x' : ' ') << "] "
                  << tasks[i].description << "\n";
    }
}

void mark_done(std::vector<Task>& tasks, size_t index) {
    if (index < 1 || index > tasks.size()) {
        std::cout << "Invalid task number.\n";
        return;
    }
    tasks[index - 1].completed = true;
}

void remove_task(std::vector<Task>& tasks, size_t index) {
    if (index < 1 || index > tasks.size()) {
        std::cout << "Invalid task number.\n";
        return;
    }
    tasks.erase(tasks.begin() + index - 1);
}

void sort_tasks(std::vector<Task>& tasks) {
    std::sort(tasks.begin(), tasks.end(), [](const Task& a, const Task& b) {
        return a.description < b.description;
    });
}

int main() {
    std::vector<Task> tasks;
    std::string command;

    std::cout << "Simple CLI To-Do List\nType 'help' for commands.\n";

    while (true) {
        std::cout << "\n> ";
        std::getline(std::cin, command);

        if (command == "exit") break;
        else if (command == "help") {
            std::cout << "Commands:\n"
                      << "  add <task description>\n"
                      << "  list\n"
                      << "  done <task number>\n"
                      << "  remove <task number>\n"
                      << "  sort\n"
                      << "  exit\n";
        } else if (command.rfind("add ", 0) == 0) {
            add_task(tasks, command.substr(4));
        } else if (command == "list") {
            list_tasks(tasks);
        } else if (command.rfind("done ", 0) == 0) {
            mark_done(tasks, std::stoi(command.substr(5)));
        } else if (command.rfind("remove ", 0) == 0) {
            remove_task(tasks, std::stoi(command.substr(7)));
        } else if (command == "sort") {
```

```
            sort_tasks(tasks);
            std::cout << "Tasks sorted.\n";
        } else {
            std::cout << "Unknown command.\n";
        }
    }

    std::cout << "Goodbye!\n";
    return 0;
}
```

**Explanation**

- **`std::vector<Task>`** is used for dynamic task storage.
- **`std::sort`** helps alphabetically organize tasks.
- **`std::getline`** and `std::stoi` handle basic command parsing.
- Input is matched against command strings to dispatch corresponding functions.
- Index validation ensures safety when modifying the task list.

**Conclusion**

This to-do list project illustrates how STL containers and algorithms can be effectively used to build a real, interactive application. With minimal code and maximum clarity, STL enables developers to manage state, process input, and manipulate data efficiently — essential skills in modern C++ programming.

## 16.5 Advanced: Custom Container Implementation

Creating a custom container compatible with the C++ Standard Template Library (STL) is a powerful exercise in understanding how STL algorithms and iterators work. In this section, we'll build a minimal custom container that supports `insert`, `erase`, and iteration, and is compatible with standard STL algorithms like `std::for_each` and `std::find`.

To be STL-compatible, a custom container should:

- Provide a nested `iterator` type or an external iterator class.
- Implement `begin()` and `end()` member functions.
- Follow the iterator traits and category conventions if interoperability with STL algorithms is desired.

### 16.5.1 Key Components of an STL-Compatible Container

1. **Container Interface**: Functions like `insert()`, `erase()`, `size()`, `operator[]`, etc.
2. **Iterator Implementation**: Typically a nested class that defines standard iterator

operations.

3. **Traits and Type Aliases**: Such as `value_type`, `iterator`, and `const_iterator`.

### 16.5.2 Example: CustomVector A Minimal Dynamic Array

Full runnable code:

```
#include <iostream>
#include <algorithm>
#include <iterator>

template<typename T>
class CustomVector {
private:
    T* data;
    size_t cap;
    size_t len;

public:
    CustomVector() : data(nullptr), cap(0), len(0) {}

    ~CustomVector() { delete[] data; }

    void insert(const T& value) {
        if (len >= cap) {
            size_t new_cap = cap == 0 ? 1 : cap * 2;
            T* new_data = new T[new_cap];
            std::copy(data, data + len, new_data);
            delete[] data;
            data = new_data;
            cap = new_cap;
        }
        data[len++] = value;
    }

    void erase(size_t index) {
        if (index >= len) return;
        for (size_t i = index; i + 1 < len; ++i)
            data[i] = data[i + 1];
        --len;
    }

    T& operator[](size_t index) { return data[index]; }
    const T& operator[](size_t index) const { return data[index]; }

    size_t size() const { return len; }

    // Iterator definition
    class iterator {
    private:
        T* ptr;
    public:
        using iterator_category = std::forward_iterator_tag;
```

```
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = std::ptrdiff_t;

        iterator(T* p) : ptr(p) {}

        T& operator*() const { return *ptr; }
        T* operator->() const { return ptr; }

        iterator& operator++() { ++ptr; return *this; }
        iterator operator++(int) { iterator tmp = *this; ++ptr; return tmp; }

        bool operator==(const iterator& other) const { return ptr == other.ptr; }
        bool operator!=(const iterator& other) const { return ptr != other.ptr; }
    };

    iterator begin() { return iterator(data); }
    iterator end() { return iterator(data + len); }
};

int main() {
    CustomVector<int> vec;
    vec.insert(5);
    vec.insert(10);
    vec.insert(15);

    std::cout << "Values:";
    std::for_each(vec.begin(), vec.end(), [](int x) {
        std::cout << " " << x;
    });
    std::cout << "\n";

    auto it = std::find(vec.begin(), vec.end(), 10);
    if (it != vec.end())
        std::cout << "Found value: " << *it << "\n";

    vec.erase(1);
    std::cout << "After erase:";
    for (auto v : vec) {
        std::cout << " " << v;
    }
}
```

### 16.5.3 Explanation

- **Dynamic Buffer Management**: `insert()` reallocates memory when needed, mimicking `std::vector` growth.
- **Custom Iterator**: Implements basic forward iterator operations and traits, enabling use with STL algorithms.
- **Iterator Category**: Declared as `std::forward_iterator_tag`, so it's compatible with a wide range of STL functions.

- **STL Compatibility**: `std::for_each` and `std::find` work seamlessly with our container.

### 16.5.4 Conclusion

This example demonstrates how to implement a custom container that integrates smoothly with the STL ecosystem. While production-grade containers require more features (exception safety, const-iterators, move semantics), this minimal example highlights the core components required for compatibility. Building such containers deepens understanding of how STL components interact and offers powerful customization opportunities for advanced use cases.