https://www.simplifycpp.org

Modern C++ Handbooks:

Modern C++ Standard Library (STL)

Prepared by: Ayman Alheraki

Target Audience: Intermediate learners

4

Modern C++ Handbooks:
Core Modern C++ Features (C++11 to C++23)

Prepared by Ayman Alheraki

Target Audience: Beginners and intermediate learners.

simplifycpp.org

January 2025

Contents

Modern C++ Handbooks

Introduction to the Modern C++ Handbooks Series
I am pleased to present this series of booklets, compiled from various sources, including books, websites, and GenAI (Generative Artificial Intelligence) systems, to serve as a quick reference for simplifying the C++ language for both beginners and professionals. This series consists of 10 booklets, each approximately 150 pages, covering essential topics of this powerful language, ranging from beginner to advanced levels. I hope that this free series will provide significant value to the followers of https://simplifycpp.org and align with its strategy of simplifying C++. Below is the index of these booklets, which will be included at the beginning of each booklet.

Book 1: Getting Started with Modern C++

- **Target Audience:** Absolute beginners.
- **Content:**
  - **Introduction to C++:**
    - What is C++? Why use Modern C++?
    - History of C++ and the evolution of standards (C++11 to C++23).
  - **Setting Up the Environment:**
    - Installing a modern C++ compiler (GCC, Clang, MSVC).

    * Setting up an IDE (Visual Studio, CLion, VS Code).
    * Using CMake for project management.

– **Writing Your First Program:**

    * Hello World in Modern C++.
    * Understanding `main()`, `#include`, and `using namespace std`.

– **Basic Syntax and Structure:**

    * Variables and data types (`int`, `double`, `bool`, `auto`).
    * Input and output (`std::cin`, `std::cout`).
    * Operators (arithmetic, logical, relational).

– **Control Flow:**

    * `if`, `else`, `switch`.
    * Loops (`for`, `while`, `do-while`).

– **Functions:**

    * Defining and calling functions.
    * Function parameters and return values.
    * Inline functions and `constexpr`.

– **Practical Examples:**

    * Simple programs to reinforce concepts (e.g., calculator, number guessing game).

– **Debugging and Version Control:**

    * Debugging basics (using GDB or IDE debuggers).
    * Introduction to version control (Git).

Book 2: Core Modern C++ Features (C++11 to C++23)

- **Target Audience:** Beginners and intermediate learners.
- **Content:**
  - **C++11 Features:**
    - `auto` keyword for type inference.
    - Range-based `for` loops.
    - `nullptr` for null pointers.
    - Uniform initialization ({} syntax).
    - `constexpr` for compile-time evaluation.
    - Lambda expressions.
    - Move semantics and rvalue references (`std::move`, `std::forward`).
  - **C++14 Features:**
    - Generalized lambda captures.
    - Return type deduction for functions.
    - Relaxed `constexpr` restrictions.
  - **C++17 Features:**
    - Structured bindings.
    - `if` and `switch` with initializers.
    - `inline` variables.
    - Fold expressions.
  - **C++20 Features:**
    - Concepts and constraints.

    - Ranges library.
    - Coroutines.
    - Three-way comparison (<=> operator).
  - **C++23 Features:**
    - `std::expected` for error handling.
    - `std::mdspan` for multidimensional arrays.
    - `std::print` for formatted output.
  - **Practical Examples:**
    - Programs demonstrating each feature (e.g., using lambdas, ranges, and coroutines).
  - **Features and Performance :**
    - Best practices for using Modern C++ features.
    - Performance implications of Modern C++.

Book 3: Object-Oriented Programming (OOP) in Modern C++

- **Target Audience:** Intermediate learners.
- **Content:**
  - **Classes and Objects:**
    - Defining classes and creating objects.
    - Access specifiers (`public`, `private`, `protected`).
  - **Constructors and Destructors:**
    - Default, parameterized, and copy constructors.

    - Move constructors and assignment operators.
    - Destructors and RAII (Resource Acquisition Is Initialization).
  - **Inheritance and Polymorphism:**
    - Base and derived classes.
    - Virtual functions and overriding.
    - Abstract classes and interfaces.
  - **Advanced OOP Concepts:**
    - Multiple inheritance and virtual base classes.
    - `override` and `final` keywords.
    - CRTP (Curiously Recurring Template Pattern).
  - **Practical Examples:**
    - Designing a class hierarchy (e.g., shapes, vehicles).
  - **Design patterns:**
    - Design patterns in Modern C++ (e.g., Singleton, Factory, Observer).

Book 4: Modern C++ Standard Library (STL)

- **Target Audience:** Intermediate learners.
- **Content:**
  - **Containers:**
    - Sequence containers (`std::vector`, `std::list`, `std::deque`).
    - Associative containers (`std::map`, `std::set`, `std::unordered_map`).

    - Container adapters (`std::stack`, `std::queue`, `std::priority_queue`).

  - **Algorithms:**
    - Sorting, searching, and modifying algorithms.
    - Parallel algorithms (C++17).
  - **Utilities:**
    - Smart pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`).
    - `std::optional`, `std::variant`, `std::any`.
    - `std::function` and `std::bind`.
  - **Iterators and Ranges:**
    - Iterator categories.
    - Ranges library (C++20).
  - **Practical Examples:**
    - Programs using STL containers and algorithms (e.g., sorting, searching).
  - **Allocators and Benchmarks :**
    - Custom allocators.
    - Performance benchmarks.

Book 5: Advanced Modern C++ Techniques

- **Target Audience:** Advanced learners and professionals.
- **Content:**

- **Templates and Metaprogramming:**
  - Function and class templates.
  - Variadic templates.
  - Type traits and `std::enable_if`.
  - Concepts and constraints (C++20).
- **Concurrency and Parallelism:**
  - Threading (`std::thread`, `std::async`).
  - Synchronization (`std::mutex`, `std::atomic`).
  - Coroutines (C++20).
- **Error Handling:**
  - Exceptions and `noexcept`.
  - `std::optional`, `std::expected` (C++23).
- **Advanced Libraries:**
  - Filesystem library (`std::filesystem`).
  - Networking (C++20 and beyond).
- **Practical Examples:**
  - Advanced programs (e.g., multithreaded applications, template metaprogramming).
- **Lock-free and Memory Management:**
  - Lock-free programming.
  - Custom memory management.

Book 6: Modern C++ Best Practices and Principles

- **Target Audience:** Professionals.

- **Content:**
  - **Code Quality:**
    - Writing clean and maintainable code.
    - Naming conventions and coding standards.
  - **Performance Optimization:**
    - Profiling and benchmarking.
    - Avoiding common pitfalls (e.g., unnecessary copies).
  - **Design Principles:**
    - SOLID principles in Modern C++.
    - Dependency injection.
  - **Testing and Debugging:**
    - Unit testing with frameworks (e.g., Google Test).
    - Debugging techniques and tools.
  - **Security:**
    - Secure coding practices.
    - Avoiding vulnerabilities (e.g., buffer overflows).
  - **Practical Examples:**
    - Case studies of well-designed Modern C++ projects.
  - **Deployment (CI/CD):**
    - Continuous integration and deployment (CI/CD).

Book 7: Specialized Topics in Modern C++

- **Target Audience:** Professionals.
- **Content:**
  - **Scientific Computing:**
    - Numerical methods and libraries (e.g., Eigen, Armadillo).
    - Parallel computing (OpenMP, MPI).
  - **Game Development:**
    - Game engines and frameworks.
    - Graphics programming (Vulkan, OpenGL).
  - **Embedded Systems:**
    - Real-time programming.
    - Low-level hardware interaction.
  - **Practical Examples:**
    - Specialized applications (e.g., simulations, games, embedded systems).
  - **Optimizations:**
    - Domain-specific optimizations.

Book 8: The Future of C++ (Beyond C++23)

- **Target Audience:** Professionals and enthusiasts.
- **Content:**

  - Upcoming features in C++26 and beyond.
  - Reflection and metaclasses.
  - Advanced concurrency models.
  - **Experimental and Developments:**
    * Experimental features and proposals.
    * Community trends and developments.

Book 9: Advanced Topics in Modern C++

- **Target Audience:** Experts and professionals.
- **Content:**
  - **Template Metaprogramming:**
    * SFINAE and `std::enable_if`.
    * Variadic templates and parameter packs.
    * Compile-time computations with `constexpr`.
  - **Advanced Concurrency:**
    * Lock-free data structures.
    * Thread pools and executors.
    * Real-time concurrency.
  - **Memory Management:**
    * Custom allocators.
    * Memory pools and arenas.
    * Garbage collection techniques.

  - **Performance Tuning:**
    * Cache optimization.
    * SIMD (Single Instruction, Multiple Data) programming.
    * Profiling and benchmarking tools.
  - **Advanced Libraries:**
    * Boost library overview.
    * GPU programming (CUDA, SYCL).
    * Machine learning libraries (e.g., TensorFlow C++ API).
  - **Practical Examples:**
    * High-performance computing (HPC) applications.
    * Real-time systems and embedded applications.
  - **C++ projects:**
    * Case studies of cutting-edge C++ projects.

Book 10: Modern C++ in the Real World

- **Target Audience:** Professionals.
- **Content:**
  - **Case Studies:**
    * Real-world applications of Modern C++ (e.g., game engines, financial systems, scientific simulations).
  - **Industry Best Practices:**
    * How top companies use Modern C++.

    - Lessons from large-scale C++ projects.

- **Open Source Contributions:**
    - Contributing to open-source C++ projects.
    - Building your own C++ libraries.
- **Career Development:**
    - Building a portfolio with Modern C++.
    - Preparing for C++ interviews.
- **Networking and conferences :**
    - Networking with the C++ community.
    - Attending conferences and workshops.

Chapter 1

Containers

1.1 Sequence Containers (`std::vector`, `std::list`, `std::deque`)

Sequence containers are a fundamental part of the C++ Standard Library. They store elements in a linear sequence, allowing you to manage collections of objects in a specific order. The three primary sequence containers in the STL are `std::vector`, `std::list`, and `std::deque`. Each of these containers has unique characteristics, performance trade-offs, and use cases. This section will provide an in-depth exploration of these containers, their features, and their appropriate usage.

1.1.1 `std::vector`

1. **Overview:**

   `std::vector` is one of the most commonly used sequence containers in C++. It represents a dynamic array that can grow or shrink in size. Elements in a `std::vector`

are stored in contiguous memory locations, which makes it highly efficient for random access and iteration.

2. **Key Features:**

   - **Dynamic Sizing**: `std::vector` automatically manages its memory, resizing itself as elements are added or removed. This dynamic resizing is handled internally, so you don't need to worry about manual memory management.
   - **Contiguous Memory**: Elements are stored in contiguous memory, enabling fast random access using indices. This memory layout also makes `std::vector` cache-friendly, which can lead to significant performance benefits.
   - **Iterators**: Provides random-access iterators, allowing efficient traversal and manipulation of elements. Random-access iterators support operations like addition and subtraction, enabling direct access to any element in the container.
   - **Time Complexity**:
     - Access: O(1) (constant time for accessing elements by index).
     - Insertion/Deletion at the end: O(1) amortized (constant time on average).
     - Insertion/Deletion in the middle: O(n) (linear time due to element shifting).

3. **Common Operations:**

   - **Declaration**:

```
std::vector<int> vec; // Empty vector of integers
std::vector<int> vec(10, 0); // Vector with 10 elements
↪  initialized to 0
```

   - **Adding Elements**:

```
vec.push_back(42); // Add an element to the end
vec.emplace_back(42); // Construct an element in-place at the end
```

- **Accessing Elements**:

```
int x = vec[0]; // Access element by index (no bounds checking)
int y = vec.at(0); // Access element with bounds checking
```

- **Removing Elements**:

```
vec.pop_back(); // Remove the last element
vec.erase(vec.begin() + 2); // Remove element at index 2
```

- **Size and Capacity**:

```
size_t size = vec.size(); // Number of elements
size_t capacity = vec.capacity(); // Total allocated memory
```

4. **Use Cases:**

- When you need fast random access to elements.
- When most insertions and deletions occur at the end of the sequence.
- When memory locality and cache efficiency are important.

5. **Advanced Features:**

- **Reserve and Shrink**:

```
vec.reserve(100); // Reserve memory for 100 elements
vec.shrink_to_fit(); // Reduce capacity to fit the size
```

- **Range-Based For Loop**:

```
for (const auto& elem : vec) {
    std::cout << elem << std::endl;
}
```

- **Custom Allocators**:

```
std::vector<int, MyAllocator<int>> vec; // Vector with a custom
↪  allocator
```

1.1.2 `std::list`

1. **Overview:**

   `std::list` is a doubly linked list container. Unlike `std::vector`, it does not store elements in contiguous memory. Instead, each element in a `std::list` contains pointers to the previous and next elements, allowing for efficient insertions and deletions at any position.

2. **Key Features:**

   - **Non-Contiguous Memory**: Elements are stored in nodes scattered across memory, with each node pointing to its neighbors. This non-contiguous memory layout allows for efficient insertions and deletions but can lead to poorer cache performance compared to `std::vector`.

- **Bidirectional Iterators**: Provides bidirectional iterators, which allow traversal in both forward and backward directions. Bidirectional iterators support increment and decrement operations but do not support random access.
- **Time Complexity**:
  - Access: O(n) (linear time, as traversal is required).
  - Insertion/Deletion at any position: O(1) (constant time, once the position is found).

3. **Common Operations:**

- **Declaration**:

```
std::list<int> lst; // Empty list of integers
std::list<int> lst(10, 0); // List with 10 elements initialized
↪  to 0
```

- **Adding Elements**:

```
lst.push_back(42); // Add an element to the end
lst.push_front(42); // Add an element to the front
lst.emplace_back(42); // Construct an element in-place at the end
lst.emplace_front(42); // Construct an element in-place at the
↪  front
```

- **Accessing Elements**:

```
int front = lst.front(); // Access the first element
int back = lst.back(); // Access the last element
```

- **Removing Elements**:

```
lst.pop_back(); // Remove the last element
lst.pop_front(); // Remove the first element
lst.erase(lst.begin()); // Remove the first element using an
↪  iterator
```

- **Size**:

```
size_t size = lst.size(); // Number of elements
```

4. **Use Cases:**

- When frequent insertions and deletions are required at both the beginning and end of the sequence.
- When you need to insert or delete elements in the middle of the sequence without invalidating iterators.
- When random access is not a priority.

5. **Advanced Features:**

- **Splice**:

```
std::list<int> lst2;
lst.splice(lst.begin(), lst2); // Move elements from lst2 to lst
```

- **Merge**:

```
lst.merge(lst2); // Merge two sorted lists
```

- **Sort**:

```
lst.sort(); // Sort the list
```

1.1.3 `std::deque`

1. **Overview:**

   `std::deque` (short for "double-ended queue") is a sequence container that allows fast insertions and deletions at both the beginning and the end. It is implemented as a dynamic array of fixed-size arrays, providing a balance between the random access efficiency of `std::vector` and the flexibility of `std::list`.

2. **Key Features:**

   - **Dynamic Sizing**: Like `std::vector`, `std::deque` can grow or shrink dynamically. However, it does so in a way that avoids the need for reallocation of the entire container.
   - **Non-Contiguous Memory**: Elements are stored in chunks of memory, which are managed internally. This allows for efficient insertions and deletions at both ends.
   - **Random-Access Iterators**: Provides random-access iterators, enabling efficient traversal and access.
   - **Time Complexity**:
     - Access: O(1) (constant time for accessing elements by index).
     - Insertion/Deletion at the beginning or end: O(1) (constant time).

- Insertion/Deletion in the middle: O(n) (linear time due to element shifting).

3. **Common Operations:**

- **Declaration**:

```
std::deque<int> dq; // Empty deque of integers
std::deque<int> dq(10, 0); // Deque with 10 elements initialized
↪  to 0
```

- **Adding Elements**:

```
dq.push_back(42); // Add an element to the end
dq.push_front(42); // Add an element to the front
dq.emplace_back(42); // Construct an element in-place at the end
dq.emplace_front(42); // Construct an element in-place at the
↪  front
```

- **Accessing Elements**:

```
int x = dq[0]; // Access element by index (no bounds checking)
int y = dq.at(0); // Access element with bounds checking
```

- **Removing Elements**:

```
dq.pop_back(); // Remove the last element
dq.pop_front(); // Remove the first element
dq.erase(dq.begin() + 2); // Remove element at index 2
```

- **Size**:

```cpp
size_t size = dq.size(); // Number of elements
```

4. **Use Cases:**

   - When you need fast insertions and deletions at both the beginning and end of the sequence.
   - When random access to elements is required.
   - When you need a more flexible alternative to `std::vector` for certain operations.

5. **Advanced Features:**

   - **Range-Based For Loop**:

```cpp
for (const auto& elem : dq) {
    std::cout << elem << std::endl;
}
```

   - **Custom Allocators**:

```cpp
std::deque<int, MyAllocator<int>> dq; // Deque with a custom
↪  allocator
```

Comparison of Sequence Containers

Feature	`std::vector`	`std::list`	`std::deque`
Memory Layout	Contiguous	Non-contiguous	Non-contiguous chunks
Random Access	O(1)	O(n)	O(1)
Insert/Delete (End)	O(1) amortized	O(1)	O(1)
Insert/Delete (Middle)	O(n)	O(1)	O(n)
Iterators	Random-access	Bidirectional	Random-access
Use Case	Fast access, end ops	Frequent middle ops	Fast front/back ops

1.1.4 Summary

- **`std::vector`**: Ideal for scenarios requiring fast random access and memory locality. Best suited for cases where most operations occur at the end of the sequence.
- **`std::list`**: Suitable for frequent insertions and deletions at any position, especially when iterator invalidation is a concern.
- **`std::deque`**: A versatile container that combines the benefits of `std::vector` and `std::list`, offering fast operations at both ends and random access.

Understanding the strengths and weaknesses of each sequence container will help you choose the right one for your specific use case, ensuring optimal performance and efficiency in your C++ programs.

1.2 Associative Containers (`std::map`, `std::set`, `std::unordered_map`)

Associative containers are a category of containers in the C++ Standard Library that store elements in a sorted or hashed order, allowing for efficient lookup, insertion, and deletion of elements based on keys. Unlike sequence containers, which store elements in a linear sequence, associative containers organize elements in a way that prioritizes fast access by key. The primary associative containers in the STL are `std::map`, `std::set`, and `std::unordered_map`. This section will provide an in-depth exploration of these containers, their features, and their appropriate usage.

1.2.1 `std::map`

1. **Overview:**

   `std::map` is an associative container that stores key-value pairs in a sorted order based on the keys. It is implemented as a balanced binary search tree (typically a Red-Black Tree), which ensures that elements are always sorted and that operations like insertion, deletion, and lookup are efficient.

2. **Key Features:**

   - **Sorted Order**: Elements in a `std::map` are always sorted by key. By default, the sorting is done in ascending order, but you can provide a custom comparator to change the order.
   - **Unique Keys**: Each key in a `std::map` must be unique. Attempting to insert a duplicate key will not change the map.
   - **Time Complexity**:

  - Insertion: O(log n)
  - Deletion: O(log n)
  - Lookup: O(log n)

- **Iterators**: Provides bidirectional iterators, allowing traversal in both forward and backward directions.

3. **Common Operations:**

- **Declaration**:

```
std::map<int, std::string> myMap; // Empty map with int keys and
↪  string values
std::map<int, std::string, std::greater<int>> myMap; // Map
↪  sorted in descending order
```

- **Inserting Elements**:

```
myMap.insert({1, "Apple"});
myMap[2] = "Banana"; // Insert or update using the subscript
↪  operator
```

- **Accessing Elements**:

```
std::string fruit = myMap[1]; // Access value by key (creates a
↪  new element if key doesn't exist)
auto it = myMap.find(2); // Find an element by key
if (it != myMap.end()) {
    std::cout << it->second << std::endl; // Access the value
}
```

- **Removing Elements**:

```
myMap.erase(1); // Remove element by key
myMap.erase(myMap.begin()); // Remove element by iterator
```

- **Size**:

```
size_t size = myMap.size(); // Number of key-value pairs
```

4. **Use Cases:**

- When you need to store key-value pairs in a sorted order.
- When you require efficient lookup, insertion, and deletion by key.
- When keys are unique and need to be maintained in a specific order.

5. **Advanced Features:**

- **Custom Comparators**:

```
struct CaseInsensitiveCompare {
    bool operator()(const std::string& a, const std::string& b)
    ↪ const {
        return std::lexicographical_compare(a.begin(), a.end(),
        ↪ b.begin(), b.end(), [](char c1, char c2) {
            return std::tolower(c1) < std::tolower(c2);
        });
    }
};
std::map<std::string, int, CaseInsensitiveCompare> myMap;
```

- **Range-Based For Loop**:

```
for (const auto& [key, value] : myMap) {
    std::cout << key << ": " << value << std::endl;
}
```

1.2.2 `std::set`

1. **Overview:**

   `std::set` is an associative container that stores unique elements in a sorted order. Like `std::map`, it is implemented as a balanced binary search tree, ensuring that elements are always sorted and that operations like insertion, deletion, and lookup are efficient.

2. **Key Features:**

   - **Sorted Order**: Elements in a `std::set` are always sorted. By default, the sorting is done in ascending order, but you can provide a custom comparator to change the order.
   - **Unique Elements**: Each element in a `std::set` must be unique. Attempting to insert a duplicate element will not change the set.
   - **Time Complexity**:
     - Insertion: O(log n)
     - Deletion: O(log n)
     - Lookup: O(log n)
   - **Iterators**: Provides bidirectional iterators, allowing traversal in both forward and backward directions.

3. **Common Operations:**

- **Declaration**:

```
std::set<int> mySet; // Empty set of integers
std::set<int, std::greater<int>> mySet; // Set sorted in
↪  descending order
```

- **Inserting Elements**:

```
mySet.insert(42);
mySet.emplace(42); // Construct element in-place
```

- **Accessing Elements**:

```
auto it = mySet.find(42); // Find an element
if (it != mySet.end()) {
    std::cout << *it << std::endl; // Access the element
}
```

- **Removing Elements**:

```
mySet.erase(42); // Remove element by value
mySet.erase(mySet.begin()); // Remove element by iterator
```

- **Size**:

```
size_t size = mySet.size(); // Number of elements
```

4. **Use Cases:**

- When you need to store unique elements in a sorted order.

- When you require efficient lookup, insertion, and deletion of elements.
- When elements are unique and need to be maintained in a specific order.

5. **Advanced Features:**

   - **Custom Comparators**:

```
struct CaseInsensitiveCompare {
    bool operator()(const std::string& a, const std::string& b)
    ↪  const {
        return std::lexicographical_compare(a.begin(), a.end(),
        ↪  b.begin(), b.end(), [](char c1, char c2) {
            return std::tolower(c1) < std::tolower(c2);
        });
    }
};
std::set<std::string, CaseInsensitiveCompare> mySet;
```

   - **Range-Based For Loop**:

```
for (const auto& elem : mySet) {
    std::cout << elem << std::endl;
}
```

1.2.3 `std::unordered_map`

1. **Overview:**

   `std::unordered_map` is an associative container that stores key-value pairs in a hashed order. It is implemented as a hash table, which provides average constant-time complexity for insertion, deletion, and lookup operations.

2. **Key Features:**

   - **Hashed Order**: Elements in an `std::unordered_map` are stored based on the hash of their keys. This allows for fast access but does not maintain any specific order.
   - **Unique Keys**: Each key in an `std::unordered_map` must be unique. Attempting to insert a duplicate key will not change the map.
   - **Time Complexity**:
     - Insertion: O(1) average, O(n) worst-case
     - Deletion: O(1) average, O(n) worst-case
     - Lookup: O(1) average, O(n) worst-case
   - **Iterators**: Provides forward iterators, allowing traversal in a single direction.

3. **Common Operations:**

   - **Declaration**:

```
std::unordered_map<int, std::string> myMap; // Empty unordered
↪  map with int keys and string values
```

   - **Inserting Elements**:

```
myMap.insert({1, "Apple"});
myMap[2] = "Banana"; // Insert or update using the subscript
↪  operator
```

   - **Accessing Elements**:

```
std::string fruit = myMap[1]; // Access value by key (creates a
↪  new element if key doesn't exist)
auto it = myMap.find(2); // Find an element by key
if (it != myMap.end()) {
    std::cout << it->second << std::endl; // Access the value
}
```

- **Removing Elements**:

```
myMap.erase(1); // Remove element by key
myMap.erase(myMap.begin()); // Remove element by iterator
```

- **Size**:

```
size_t size = myMap.size(); // Number of key-value pairs
```

4. **Use Cases:**

- When you need to store key-value pairs with fast access by key.
- When the order of elements is not important.
- When keys are unique and need to be accessed quickly.

5. **Advanced Features:**

- **Custom Hash Functions**:

```
struct MyHash {
    size_t operator()(const std::string& key) const {
        size_t hash = 0;
```

```
        for (char c : key) {
            hash = hash * 31 + c;
        }
        return hash;
    }
};
std::unordered_map<std::string, int, MyHash> myMap;
```

- **Range-Based For Loop**:

```
for (const auto& [key, value] : myMap) {
    std::cout << key << ": " << value << std::endl;
}
```

Comparison of Associative Containers

Feature	`std::map`	`std::set`	`std::unordered_map`
Ordering	Sorted	Sorted	Hashed (unsorted)
Key Uniqueness	Unique	Unique	Unique
Time Complexity	O(log n)	O(log n)	O(1) average
Iterators	Bidirectional	Bidirectional	Forward
Use Case	Sorted key-value pairs	Sorted unique elements	Fast key-value access

1.2.4 Summary

- **`std::map`**: Ideal for storing key-value pairs in a sorted order. Best suited for scenarios where you need efficient lookup, insertion, and deletion by key while maintaining a specific order.
- **`std::set`**: Suitable for storing unique elements in a sorted order. Use it when you need efficient lookup, insertion, and deletion of elements while maintaining uniqueness and order.
- **`std::unordered_map`**: Perfect for scenarios requiring fast access to key-value pairs without the need for sorting. Use it when the order of elements is not important, and you need average constant-time complexity for operations.

Understanding the strengths and weaknesses of each associative container will help you choose the right one for your specific use case, ensuring optimal performance and efficiency in your C++ programs.

1.3 Container Adapters (`std::stack`, `std::queue`, `std::priority_queue`)

Container adapters are a special category of containers in the C++ Standard Library that provide a specific interface for accessing and manipulating elements. Unlike sequence or associative containers, container adapters do not implement their own data structures. Instead, they rely on an underlying container (such as `std::deque`, `std::list`, or `std::vector`) to store elements and provide a restricted interface for operations. The three primary container adapters in the STL are `std::stack`, `std::queue`, and `std::priority_queue`. This section will provide an in-depth exploration of these adapters, their features, and their appropriate usage.

1.3.1 `std::stack`

1. **Overview:**

   `std::stack` is a container adapter that provides a Last-In-First-Out (LIFO) data structure. It allows elements to be added and removed only from one end, known as the "top" of the stack. The underlying container used by `std::stack` is, by default, `std::deque`, but you can specify other containers like `std::vector` or `std::list`.

2. **Key Features:**

   - **LIFO Principle**: The last element added to the stack is the first one to be removed.
   - **Restricted Interface**: Provides a minimal interface for stack operations: `push`, `pop`, `top`, and `empty`.
   - **Underlying Container**: By default, uses `std::deque` as the underlying container, but you can change it to `std::vector` or `std::list`.

3. **Common Operations:**

   - **Declaration**:

```
std::stack<int> myStack; // Default underlying container is
↪  std::deque
std::stack<int, std::vector<int>> myStack; // Use std::vector as
↪  the underlying container
```

   - **Adding Elements**:

```
myStack.push(42); // Add an element to the top of the stack
```

   - **Accessing Elements**:

```
int topElement = myStack.top(); // Access the top element
```

   - **Removing Elements**:

```
myStack.pop(); // Remove the top element
```

   - **Checking if Empty**:

```
bool isEmpty = myStack.empty(); // Check if the stack is empty
```

   - **Size**:

```
size_t size = myStack.size(); // Number of elements in the stack
```

4. **Use Cases:**

- When you need to implement a LIFO data structure.
- When you require a simple interface for adding and removing elements from one end.
- When the order of element removal is important (last added is first removed).

5. **Advanced Features:**

   - **Custom Underlying Container**:

```
std::stack<int, std::list<int>> myStack; // Use std::list as the
↪  underlying container
```

   - **Swapping Stacks**:

```
std::stack<int> stack1, stack2;
stack1.push(1);
stack2.push(2);
stack1.swap(stack2); // Swap contents of stack1 and stack2
```

1.3.2 `std::queue`

1. **Overview:**

   `std::queue` is a container adapter that provides a First-In-First-Out (FIFO) data structure. It allows elements to be added at one end (the ”back”) and removed from the other end (the ”front”). The underlying container used by `std::queue` is, by default, `std::deque`, but you can specify other containers like `std::list`.

2. **Key Features:**

- **FIFO Principle**: The first element added to the queue is the first one to be removed.
- **Restricted Interface**: Provides a minimal interface for queue operations: `push`, `pop`, `front`, `back`, and `empty`.
- **Underlying Container**: By default, uses `std::deque` as the underlying container, but you can change it to `std::list`.

3. **Common Operations:**

- **Declaration**:

```
std::queue<int> myQueue; // Default underlying container is
↪  std::deque
std::queue<int, std::list<int>> myQueue; // Use std::list as the
↪  underlying container
```

- **Adding Elements**:

```
myQueue.push(42); // Add an element to the back of the queue
```

- **Accessing Elements**:

```
int frontElement = myQueue.front(); // Access the front element
int backElement = myQueue.back(); // Access the back element
```

- **Removing Elements**:

```
myQueue.pop(); // Remove the front element
```

- **Checking if Empty**:

```cpp
bool isEmpty = myQueue.empty(); // Check if the queue is empty
```

- **Size**:

```cpp
size_t size = myQueue.size(); // Number of elements in the queue
```

4. **Use Cases:**

- When you need to implement a FIFO data structure.
- When you require a simple interface for adding elements to one end and removing them from the other.
- When the order of element removal is important (first added is first removed).

5. **Advanced Features:**

- **Custom Underlying Container**:

```cpp
std::queue<int, std::list<int>> myQueue; // Use std::list as the
↪  underlying container
```

- **Swapping Queues**:

```cpp
std::queue<int> queue1, queue2;
queue1.push(1);
queue2.push(2);
queue1.swap(queue2); // Swap contents of queue1 and queue2
```

1.3.3 `std::priority_queue`

1. **Overview:**

   `std::priority_queue` is a container adapter that provides a priority-based data structure. Elements are inserted in any order but are removed based on their priority, which is determined by a comparator (by default, the largest element has the highest priority). The underlying container used by `std::priority_queue` is, by default, `std::vector`, but you can specify other containers like `std::deque`.

2. **Key Features:**

   - **Priority-Based Order**: Elements are removed based on their priority, not the order in which they were added.
   - **Restricted Interface**: Provides a minimal interface for priority queue operations: `push`, `pop`, `top`, and `empty`.
   - **Underlying Container**: By default, uses `std::vector` as the underlying container, but you can change it to `std::deque`.
   - **Comparator**: By default, uses `std::less` to determine priority, but you can provide a custom comparator.

3. **Common Operations:**

   - **Declaration**:

```
std::priority_queue<int> myPriorityQueue; // Default underlying
↪  container is std::vector, default comparator is std::less
std::priority_queue<int, std::vector<int>, std::greater<int>>
↪  myPriorityQueue; // Use std::greater as the comparator
```

- **Adding Elements**:

```
myPriorityQueue.push(42); // Add an element to the priority queue
```

- **Accessing Elements**:

```
int topElement = myPriorityQueue.top(); // Access the top element
↪ (highest priority)
```

- **Removing Elements**:

```
myPriorityQueue.pop(); // Remove the top element
```

- **Checking if Empty**:

```
bool isEmpty = myPriorityQueue.empty(); // Check if the priority
↪ queue is empty
```

- **Size**:

```
size_t size = myPriorityQueue.size(); // Number of elements in
↪ the priority queue
```

4. **Use Cases:**

- When you need to implement a priority-based data structure.
- When you require elements to be processed based on their priority rather than their insertion order.
- When you need efficient access to the highest (or lowest) priority element.

5. **Advanced Features:**

- **Custom Comparator**:

```
struct MyComparator {
    bool operator()(int a, int b) const {
        return a > b; // Custom priority logic
    }
};
std::priority_queue<int, std::vector<int>, MyComparator>
↪  myPriorityQueue;
```

- **Swapping Priority Queues**:

```
std::priority_queue<int> pq1, pq2;
pq1.push(1);
pq2.push(2);
pq1.swap(pq2); // Swap contents of pq1 and pq2
```

Comparison of Container Adapters

Feature	`std::stack`	`std::queue`	`std::priority_queue`
Ordering	LIFO	FIFO	Priority-Based
Interface	`push`, `pop`, `top`	`push`, `pop`, `front`, `back`	`push`, `pop`, `top`
Underlying Container	`std::deque` (default)	`std::deque` (default)	`std::vector` (default)

Feature	**`std::stack`**	**`std::queue`**	**`std::priority_queue`**
Use Case	Last-In-First-Out	First-In-First-Out	Priority-Based Access

1.3.4 Summary

- **`std::stack`**: Ideal for implementing a Last-In-First-Out (LIFO) data structure. Use it when you need to add and remove elements from one end only.
- **`std::queue`**: Suitable for implementing a First-In-First-Out (FIFO) data structure. Use it when you need to add elements to one end and remove them from the other.
- **`std::priority_queue`**: Perfect for implementing a priority-based data structure. Use it when you need to process elements based on their priority rather than their insertion order.

Understanding the strengths and weaknesses of each container adapter will help you choose the right one for your specific use case, ensuring optimal performance and efficiency in your C++ programs.

Chapter 2

Algorithms

2.1 Sorting, Searching, and Modifying Algorithms

The C++ Standard Library provides a rich set of algorithms that operate on containers, enabling you to perform common tasks such as sorting, searching, and modifying elements efficiently. These algorithms are part of the `<algorithm>` header and are designed to work with iterators, making them highly flexible and applicable to a wide range of container types. This section will provide an in-depth exploration of sorting, searching, and modifying algorithms, their features, and their appropriate usage.

2.1.1 Sorting Algorithms

Sorting algorithms rearrange elements in a container so that they follow a specific order, typically ascending or descending. The C++ Standard Library provides several sorting algorithms, each with its own characteristics and performance guarantees.

1. **Key Sorting Algorithms:**

- **`std::sort`**: Sorts elements in a range into ascending order (or another order specified by a comparator).
- **`std::stable_sort`**: Sorts elements while preserving the relative order of equivalent elements.
- **`std::partial_sort`**: Partially sorts a range, ensuring that the first `n` elements are the smallest (or largest) in the range.
- **`std::nth_element`**: Rearranges elements such that the `n`-th element is in its correct sorted position, and all elements before it are less than or equal to it.
- **`std::is_sorted`**: Checks if a range is sorted.
- **`std::is_sorted_until`**: Finds the first unsorted element in a range.

2. **Common Operations:**

- **`std::sort`**:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::sort(vec.begin(), vec.end()); // Sorts in ascending order
// vec = {1, 2, 3, 4, 5}
```

- **`std::stable_sort`**:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::stable_sort(vec.begin(), vec.end()); // Sorts while
↪  preserving order of equivalents
// vec = {1, 2, 3, 4, 5}
```

- **`std::partial_sort`**:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::partial_sort(vec.begin(), vec.begin() + 3, vec.end()); //
↪ Sorts the first 3 elements
// vec = {1, 2, 3, 5, 4}
```

- **`std::nth_element`**:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::nth_element(vec.begin(), vec.begin() + 2, vec.end()); //
↪ Places the 2nd element in its sorted position
// vec = {1, 2, 3, 4, 5}
```

- **`std::is_sorted`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
bool sorted = std::is_sorted(vec.begin(), vec.end()); // Checks
↪ if the range is sorted
std::cout << (sorted ? "Sorted" : "Not sorted") << std::endl;
```

- **`std::is_sorted_until`**:

```
std::vector<int> vec = {1, 2, 3, 5, 4};
auto it = std::is_sorted_until(vec.begin(), vec.end()); // Finds
↪ the first unsorted element
std::cout << "First unsorted element: " << *it << std::endl;
```

3. **Use Cases:**

- **`std::sort`**: When you need to sort a range of elements in ascending or custom order.

- **`std::stable_sort`**: When you need to sort elements while preserving the relative order of equivalent elements.
- **`std::partial_sort`**: When you only need the top `n` elements to be sorted.
- **`std::nth_element`**: When you need to find the `n`-th smallest or largest element in a range.
- **`std::is_sorted`**: When you need to check if a range is already sorted.
- **`std::is_sorted_until`**: When you need to find the first unsorted element in a range.

2.1.2 Searching Algorithms

Searching algorithms are used to find specific elements or ranges within a container. The C++ Standard Library provides several searching algorithms, each optimized for different use cases.

1. **Key Searching Algorithms:**

   - **`std::find`**: Finds the first occurrence of a value in a range.
   - **`std::find_if`**: Finds the first element in a range that satisfies a predicate.
   - **`std::find_if_not`**: Finds the first element in a range that does not satisfy a predicate.
   - **`std::binary_search`**: Checks if a value exists in a sorted range using binary search.
   - **`std::lower_bound`**: Finds the first element in a sorted range that is not less than a given value.
   - **`std::upper_bound`**: Finds the first element in a sorted range that is greater than a given value.

- **`std::equal_range`**: Finds a subrange of elements that are equal to a given value in a sorted range.
- **`std::search`**: Finds the first occurrence of a subsequence within a range.
- **`std::find_end`**: Finds the last occurrence of a subsequence within a range.

2. **Common Operations:**

- **`std::find`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::find(vec.begin(), vec.end(), 3); // Finds the
↪  first occurrence of 3
if (it != vec.end()) {
    std::cout << "Found: " << *it << std::endl;
}
```

- **`std::find_if`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::find_if(vec.begin(), vec.end(), [](int x) { return
↪  x > 3; }); // Finds the first element greater than 3
if (it != vec.end()) {
    std::cout << "Found: " << *it << std::endl;
}
```

- **`std::find_if_not`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::find_if_not(vec.begin(), vec.end(), [](int x) {
↪  return x < 3; }); // Finds the first element not less than 3
if (it != vec.end()) {
```

```
    std::cout << "Found: " << *it << std::endl;
}
```

- **std::binary_search**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
bool found = std::binary_search(vec.begin(), vec.end(), 3); //
↪ Checks if 3 exists in the sorted range
std::cout << (found ? "Found" : "Not found") << std::endl;
```

- **std::lower_bound**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::lower_bound(vec.begin(), vec.end(), 3); // Finds
↪ the first element not less than 3
std::cout << "Lower bound: " << *it << std::endl;
```

- **std::upper_bound**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::upper_bound(vec.begin(), vec.end(), 3); // Finds
↪ the first element greater than 3
std::cout << "Upper bound: " << *it << std::endl;
```

- **std::equal_range**:

```
std::vector<int> vec = {1, 2, 3, 3, 4, 5};
auto range = std::equal_range(vec.begin(), vec.end(), 3); //
↪ Finds the range of elements equal to 3
std::cout << "Range: [" << range.first - vec.begin() << ", " <<
↪ range.second - vec.begin() << ")" << std::endl;
```

- **`std::search`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::vector<int> sub = {3, 4};
auto it = std::search(vec.begin(), vec.end(), sub.begin(),
↪  sub.end()); // Finds the first occurrence of sub in vec
if (it != vec.end()) {
    std::cout << "Found subsequence at position: " << it -
    ↪  vec.begin() << std::endl;
}
```

- **`std::find_end`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5, 3, 4};
std::vector<int> sub = {3, 4};
auto it = std::find_end(vec.begin(), vec.end(), sub.begin(),
↪  sub.end()); // Finds the last occurrence of sub in vec
if (it != vec.end()) {
    std::cout << "Found subsequence at position: " << it -
    ↪  vec.begin() << std::endl;
}
```

3. **Use Cases:**

- **`std::find`**: When you need to find the first occurrence of a specific value in a range.
- **`std::find_if`**: When you need to find the first element that satisfies a custom condition.
- **`std::find_if_not`**: When you need to find the first element that does not satisfy a custom condition.

- **`std::binary_search`**: When you need to check if a value exists in a sorted range.
- **`std::lower_bound`** and **`std::upper_bound`**: When you need to find the position where a value could be inserted in a sorted range.
- **`std::equal_range`**: When you need to find all elements equal to a given value in a sorted range.
- **`std::search`**: When you need to find the first occurrence of a subsequence within a range.
- **`std::find_end`**: When you need to find the last occurrence of a subsequence within a range.

2.1.3 Modifying Algorithms

Modifying algorithms are used to change the contents of a container, such as by copying, moving, or transforming elements. The C++ Standard Library provides several modifying algorithms, each designed for specific tasks.

1. **Key Modifying Algorithms:**

   - **`std::copy`**: Copies elements from one range to another.
   - **`std::move`**: Moves elements from one range to another.
   - **`std::transform`**: Applies a function to each element in a range and stores the result in another range.
   - **`std::replace`**: Replaces all occurrences of a value in a range with another value.
   - **`std::replace_if`**: Replaces all elements in a range that satisfy a predicate with another value.
   - **`std::fill`**: Fills a range with a specific value.

- **`std::fill_n`**: Fills the first `n` elements of a range with a specific value.
- **`std::remove`**: Removes elements equal to a specific value from a range.
- **`std::remove_if`**: Removes elements that satisfy a predicate from a range.
- **`std::unique`**: Removes consecutive duplicate elements from a range.
- **`std::reverse`**: Reverses the order of elements in a range.
- **`std::rotate`**: Rotates the elements in a range such that a specified element becomes the first element.
- **`std::shuffle`**: Randomly shuffles the elements in a range.

2. **Common Operations:**

- **`std::copy`**:

```
std::vector<int> src = {1, 2, 3, 4, 5};
std::vector<int> dst(src.size());
std::copy(src.begin(), src.end(), dst.begin()); // Copies
↪  elements from src to dst
```

- **`std::move`**:

```
std::vector<std::string> src = {"Hello", "World"};
std::vector<std::string> dst(src.size());
std::move(src.begin(), src.end(), dst.begin()); // Moves elements
↪  from src to dst
```

- **`std::transform`**:

```
std::vector<int> src = {1, 2, 3, 4, 5};
std::vector<int> dst(src.size());
std::transform(src.begin(), src.end(), dst.begin(), [](int x) {
↪  return x * 2; }); // Doubles each element
```

- **std::replace**:

```
std::vector<int> vec = {1, 2, 3, 2, 5};
std::replace(vec.begin(), vec.end(), 2, 42); // Replaces all 2s
↪  with 42
```

- **std::replace_if**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::replace_if(vec.begin(), vec.end(), [](int x) { return x % 2
↪  == 0; }, 42); // Replaces even elements with 42
```

- **std::fill**:

```
std::vector<int> vec(5);
std::fill(vec.begin(), vec.end(), 42); // Fills the vector with
↪  42
```

- **std::fill_n**:

```
std::vector<int> vec(5);
std::fill_n(vec.begin(), 3, 42); // Fills the first 3 elements
↪  with 42
```

- **std::remove**:

```
std::vector<int> vec = {1, 2, 3, 2, 5};
auto newEnd = std::remove(vec.begin(), vec.end(), 2); // Removes
↪  all 2s
vec.erase(newEnd, vec.end()); // Erases the "removed" elements
```

- **std::remove_if**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto newEnd = std::remove_if(vec.begin(), vec.end(), [](int x) {
↪  return x % 2 == 0; }); // Removes even elements
vec.erase(newEnd, vec.end()); // Erases the "removed" elements
```

- **std::unique**:

```
std::vector<int> vec = {1, 2, 2, 3, 3, 4};
auto newEnd = std::unique(vec.begin(), vec.end()); // Removes
↪  consecutive duplicates
vec.erase(newEnd, vec.end()); // Erases the "removed" elements
```

- **std::reverse**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::reverse(vec.begin(), vec.end()); // Reverses the vector
```

- **std::rotate**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::rotate(vec.begin(), vec.begin() + 2, vec.end()); // Rotates
↪  the vector so that 3 becomes the first element
```

- **`std::shuffle`**:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::random_device rd;
std::mt19937 g(rd());
std::shuffle(vec.begin(), vec.end(), g); // Shuffles the vector
```

3. **Use Cases:**

- **`std::copy`**: When you need to copy elements from one range to another.
- **`std::move`**: When you need to move elements from one range to another.
- **`std::transform`**: When you need to apply a function to each element in a range and store the result.
- **`std::replace`**: When you need to replace all occurrences of a value in a range with another value.
- **`std::replace_if`**: When you need to replace elements that satisfy a predicate with another value.
- **`std::fill`**: When you need to fill a range with a specific value.
- **`std::fill_n`**: When you need to fill the first `n` elements of a range with a specific value.
- **`std::remove`**: When you need to remove elements equal to a specific value from a range.
- **`std::remove_if`**: When you need to remove elements that satisfy a predicate from a range.
- **`std::unique`**: When you need to remove consecutive duplicate elements from a range.

- **`std::reverse`**: When you need to reverse the order of elements in a range.
- **`std::rotate`**: When you need to rotate the elements in a range such that a specified element becomes the first element.
- **`std::shuffle`**: When you need to randomly shuffle the elements in a range.

2.1.4 Summary

- **Sorting Algorithms**: Use `std::sort` for general sorting, `std::stable_sort` for stable sorting, `std::partial_sort` for partial sorting, and `std::nth_element` for finding the `n`-th element.
- **Searching Algorithms**: Use `std::find` and `std::find_if` for linear search, `std::binary_search` for binary search, and `std::lower_bound`, `std::upper_bound`, and `std::equal_range` for searching in sorted ranges.
- **Modifying Algorithms**: Use `std::copy` and `std::move` for copying and moving elements, `std::transform` for applying functions, `std::replace` for replacing values, `std::fill` for filling ranges, `std::remove` for removing values, and `std::unique` for removing consecutive duplicates.

Understanding the strengths and weaknesses of each algorithm will help you choose the right one for your specific use case, ensuring optimal performance and efficiency in your C++ programs.

2.2 Parallel Algorithms (C++17)

With the introduction of C++17, the C++ Standard Library gained support for parallel algorithms, enabling you to execute standard algorithms in parallel to leverage modern multi-core processors. Parallel algorithms are designed to improve performance by dividing work across multiple threads, making them ideal for computationally intensive tasks. This section will provide an in-depth exploration of parallel algorithms, their features, and their appropriate usage.

2.2.1 Overview of Parallel Algorithms

Parallel algorithms are extensions of the existing algorithms in the C++ Standard Library that allow you to specify an execution policy. This execution policy determines whether the algorithm runs sequentially, in parallel, or with vectorized instructions. The primary execution policies are:

- **`std::execution::seq`**: Sequential execution (default).
- **`std::execution::par`**: Parallel execution.
- **`std::execution::par_unseq`**: Parallel and vectorized execution.

These execution policies are defined in the `<execution>` header.

Key Features:

- **Parallel Execution**: Algorithms can run in parallel, utilizing multiple threads to improve performance.
- **Vectorized Execution**: Algorithms can use SIMD (Single Instruction, Multiple Data) instructions for further optimization.

- **Thread Safety**: Parallel algorithms are designed to be thread-safe, but you must ensure that the operations performed by the algorithm are also thread-safe.

Common Parallel Algorithms:

- **Sorting**: `std::sort`, `std::stable_sort`
- **Searching**: `std::find`, `std::find_if`
- **Modifying**: `std::transform`, `std::for_each`, `std::replace`, `std::fill`
- **Reduction**: `std::reduce`, `std::accumulate`

2.2.2 Execution Policies

Execution policies are used to specify how an algorithm should be executed. They are passed as the first argument to parallel algorithms.

Execution Policies:

- **`std::execution::seq`**:
  - The algorithm runs sequentially, in a single thread.
  - This is the default behavior if no execution policy is specified.
  - Example:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::sort(std::execution::seq, vec.begin(), vec.end()); //
↪  Sequential sort
```

- **`std::execution::par`**:

- The algorithm runs in parallel, utilizing multiple threads.
- Suitable for algorithms that can be parallelized without vectorization.
- Example:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::sort(std::execution::par, vec.begin(), vec.end()); //
↪ Parallel sort
```

- **`std::execution::par_unseq`**:
  - The algorithm runs in parallel and may use vectorized instructions (SIMD).
  - Suitable for algorithms that can benefit from both parallel execution and vectorization.
  - Example:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::sort(std::execution::par_unseq, vec.begin(), vec.end()); //
↪ Parallel and vectorized sort
```

2.2.3 Common Parallel Algorithms

1. **Sorting Algorithms**
   - **`std::sort`**:
     - Sorts elements in a range in parallel.
     - Example:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::sort(std::execution::par, vec.begin(), vec.end()); //
↪  Parallel sort
```

- **`std::stable_sort`**:
  - Sorts elements in a range while preserving the relative order of equivalent elements.
  - Example:

```
std::vector<int> vec = {5, 3, 1, 4, 2};
std::stable_sort(std::execution::par, vec.begin(), vec.end());
↪  // Parallel stable sort
```

2. **Searching Algorithms**

- **`std::find`**:
  - Finds the first occurrence of a value in a range in parallel.
  - Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::find(std::execution::par, vec.begin(),
↪  vec.end(), 3); // Parallel find
if (it != vec.end()) {
    std::cout << "Found: " << *it << std::endl;
}
```

- **`std::find_if`**:
  - Finds the first element in a range that satisfies a predicate in parallel.

- Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::find_if(std::execution::par, vec.begin(),
↪  vec.end(), [](int x) { return x > 3; }); // Parallel
↪  find_if
if (it != vec.end()) {
    std::cout << "Found: " << *it << std::endl;
}
```

3. **Modifying Algorithms**

- **`std::transform`**:
  - Applies a function to each element in a range and stores the result in another range in parallel.
  - Example:

```
std::vector<int> src = {1, 2, 3, 4, 5};
std::vector<int> dst(src.size());
std::transform(std::execution::par, src.begin(), src.end(),
↪  dst.begin(), [](int x) { return x * 2; }); // Parallel
↪  transform
```

- **`std::for_each`**:
  - Applies a function to each element in a range in parallel.
  - Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::for_each(std::execution::par, vec.begin(), vec.end(),
↪  [](int& x) { x *= 2; }); // Parallel for_each
```

- **`std::replace`**:
  - Replaces all occurrences of a value in a range with another value in parallel.
  - Example:

```
std::vector<int> vec = {1, 2, 3, 2, 5};
std::replace(std::execution::par, vec.begin(), vec.end(), 2,
↪  42); // Parallel replace
```

- **`std::fill`**:
  - Fills a range with a specific value in parallel.
  - Example:

```
std::vector<int> vec(5);
std::fill(std::execution::par, vec.begin(), vec.end(), 42); //
↪  Parallel fill
```

4. **3.4 Reduction Algorithms**

- **`std::reduce`**:
  - Reduces a range of elements to a single value using a binary operation in parallel.
  - Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
int sum = std::reduce(std::execution::par, vec.begin(),
↪  vec.end()); // Parallel reduce
std::cout << "Sum: " << sum << std::endl;
```

- **`std::accumulate`**:

- Accumulates a range of elements to a single value using a binary operation (sequential by default, but can be parallelized with custom implementation).
- Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
int sum = std::accumulate(vec.begin(), vec.end(), 0); //
↪  Sequential accumulate
std::cout << "Sum: " << sum << std::endl;
```

2.2.4 Thread Safety and Considerations

While parallel algorithms are designed to be thread-safe, you must ensure that the operations performed by the algorithm are also thread-safe. Here are some considerations:

- **Avoid Data Races**: Ensure that the operations performed by the algorithm do not modify shared data concurrently without proper synchronization.
- **Use Thread-Safe Functions**: If the algorithm uses custom functions or lambdas, ensure that these functions are thread-safe.
- **Performance Overhead**: Parallel algorithms may introduce overhead due to thread creation and synchronization. Measure performance to ensure that parallel execution provides a benefit.

Example: Parallel Sorting and Transformation

Here's an example that demonstrates parallel sorting and transformation:

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

```cpp
#include <execution>

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

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

    // Parallel transform
    std::vector<int> result(vec.size());
    std::transform(std::execution::par, vec.begin(), vec.end(),
    ↪  result.begin(), [](int x) {
        return x * 2;
    });

    // Print results
    for (int x : result) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

2.2.5 Summary

- **Parallel Algorithms**: C++17 introduced parallel execution policies (`std::execution::par`, `std::execution::par_unseq`) to enable parallel and vectorized execution of standard algorithms.

- **Common Algorithms**: Sorting (`std::sort`, `std::stable_sort`), searching

(`std::find`, `std::find_if`), modifying (`std::transform`, `std::for_each`), and reduction (`std::reduce`) algorithms can be parallelized.

- **Thread Safety**: Ensure that operations performed by parallel algorithms are thread-safe to avoid data races.
- **Performance**: Measure performance to ensure that parallel execution provides a benefit, as it may introduce overhead.

Understanding and using parallel algorithms effectively can significantly improve the performance of computationally intensive tasks in your C++ programs.

Chapter 3

Utilities

3.1 Smart Pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`)

Smart pointers are a cornerstone of modern C++ programming, providing automatic memory management and helping to prevent common issues such as memory leaks and dangling pointers. The C++ Standard Library offers three main types of smart pointers: `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. Each type has its own use cases and characteristics, making them suitable for different scenarios. This section will provide an in-depth exploration of these smart pointers, their features, and their appropriate usage.

3.1.1 `std::unique_ptr`

1. **Overview:**

   `std::unique_ptr` is a smart pointer that owns and manages a dynamically allocated object. It ensures that the object is automatically deleted when the `std::unique_ptr`

goes out of scope. The key feature of `std::unique_ptr` is that it enforces exclusive ownership, meaning that only one `std::unique_ptr` can own a particular resource at any given time.

2. **Key Features:**

   - **Exclusive Ownership**: Only one `std::unique_ptr` can own a resource at a time.
   - **Automatic Cleanup**: The managed object is automatically deleted when the `std::unique_ptr` goes out of scope.
   - **Move Semantics**: Ownership can be transferred using move semantics, but copying is not allowed.
   - **Custom Deleters**: You can specify a custom deleter to handle resource cleanup in a specific way.

3. **Common Operations:**

   - **Declaration**:

```
std::unique_ptr<int> ptr(new int(42)); // Manages a dynamically
↪  allocated integer
```

   - **Accessing the Managed Object**:

```
int value = *ptr; // Dereference to access the managed object
int* rawPtr = ptr.get(); // Get the raw pointer (use with
↪  caution)
```

   - **Releasing Ownership**:

```
int* rawPtr = ptr.release(); // Releases ownership, returns the
↪  raw pointer
```

- **Resetting the Pointer**:

```
ptr.reset(new int(100)); // Deletes the old object and manages a
↪  new one
ptr.reset(); // Deletes the managed object and sets the pointer
↪  to nullptr
```

- **Custom Deleter**:

```
auto deleter = [](int* p) { delete p; };
std::unique_ptr<int, decltype(deleter)> ptr(new int(42),
↪  deleter);
```

4. **Use Cases:**

   - When you need exclusive ownership of a resource.
   - When you want to ensure automatic cleanup of dynamically allocated memory.
   - When you need to transfer ownership of a resource using move semantics.

5. **Example:**

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

int main() {
    std::unique_ptr<int> ptr(new int(42));
```

```
    std::cout << *ptr << std::endl; // Output: 42

    std::unique_ptr<int> ptr2 = std::move(ptr); // Transfer ownership
    if (!ptr) {
        std::cout << "ptr is now nullptr" << std::endl;
    }
    std::cout << *ptr2 << std::endl; // Output: 42

    return 0; // ptr2 automatically deletes the managed object
}
```

3.1.2 `std::shared_ptr`

1. **Overview:**

   `std::shared_ptr` is a smart pointer that allows multiple pointers to share ownership of a dynamically allocated object. The object is deleted when the last `std::shared_ptr` that owns it is destroyed or reset. This is achieved using reference counting.

2. **Key Features:**

   - **Shared Ownership**: Multiple `std::shared_ptr` instances can own the same resource.
   - **Automatic Cleanup**: The managed object is automatically deleted when the last `std::shared_ptr` owning it is destroyed or reset.
   - **Reference Counting**: Keeps track of the number of `std::shared_ptr` instances that own the resource.
   - **Custom Deleters**: You can specify a custom deleter to handle resource cleanup in a specific way.

3. **Common Operations:**

   - **Declaration**:

```
std::shared_ptr<int> ptr(new int(42)); // Manages a dynamically
↪  allocated integer
```

   - **Accessing the Managed Object**:

```
int value = *ptr; // Dereference to access the managed object
int* rawPtr = ptr.get(); // Get the raw pointer (use with
↪  caution)
```

   - **Copying and Assigning**:

```
std::shared_ptr<int> ptr2 = ptr; // Both ptr and ptr2 share
↪  ownership
```

   - **Resetting the Pointer**:

```
ptr.reset(new int(100)); // Deletes the old object and manages a
↪  new one
ptr.reset(); // Deletes the managed object and sets the pointer
↪  to nullptr
```

   - **Custom Deleter**:

```
auto deleter = [](int* p) { delete p; };
std::shared_ptr<int> ptr(new int(42), deleter);
```

4. **Use Cases:**

   - When you need shared ownership of a resource.
   - When you want to ensure automatic cleanup of dynamically allocated memory.
   - When you need to manage resources that are shared across multiple parts of your code.

5. **Example:**

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> ptr(new int(42));
    std::cout << *ptr << std::endl; // Output: 42

    std::shared_ptr<int> ptr2 = ptr; // Share ownership
    std::cout << "Use count: " << ptr.use_count() << std::endl; //
    ↪  Output: 2

    ptr.reset(); // Release ownership
    std::cout << "Use count: " << ptr2.use_count() << std::endl; //
    ↪  Output: 1

    return 0; // ptr2 automatically deletes the managed object
}
```

3.1.3 `std::weak_ptr`

1. **Overview:**

`std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object managed by a `std::shared_ptr`. It is used to break circular references that can occur with `std::shared_ptr`, which can lead to memory leaks. A `std::weak_ptr` does not affect the reference count of the managed object.

2. **Key Features:**

   - **Non-Owning Reference**: Does not own the resource and does not affect the reference count.
   - **Breaking Circular References**: Helps to break circular references that can occur with `std::shared_ptr`.
   - **Accessing the Managed Object**: You can create a `std::shared_ptr` from a `std::weak_ptr` to access the managed object, but you must check if the object still exists.

3. **Common Operations:**

   - **Declaration**:

```
std::shared_ptr<int> sharedPtr(new int(42));
std::weak_ptr<int> weakPtr(sharedPtr); // Create a weak pointer
↪  from a shared pointer
```

   - **Accessing the Managed Object**:

```
if (auto sharedPtr2 = weakPtr.lock()) { // Attempt to create a
↪  shared pointer
    std::cout << *sharedPtr2 << std::endl; // Access the managed
    ↪  object
} else {
```

```
    std::cout << "Object has been deleted" << std::endl;
}
```

- **Checking Expiration**:

```
if (weakPtr.expired()) {
    std::cout << "Object has been deleted" << std::endl;
}
```

4. **Use Cases:**

- When you need to break circular references that can occur with `std::shared_ptr`.
- When you need a non-owning reference to an object managed by a `std::shared_ptr`.
- When you want to check if an object still exists without affecting its lifetime.

5. **Example:**

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

int main() {
    std::shared_ptr<int> sharedPtr(new int(42));
    std::weak_ptr<int> weakPtr(sharedPtr);

    if (auto sharedPtr2 = weakPtr.lock()) {
        std::cout << *sharedPtr2 << std::endl; // Output: 42
    } else {
```

```
        std::cout << "Object has been deleted" << std::endl;
    }

    sharedPtr.reset(); // Delete the managed object

    if (weakPtr.expired()) {
        std::cout << "Object has been deleted" << std::endl; //
        ↪  Output: Object has been deleted
    }

    return 0;
}
```

3.1.4 Summary

- **`std::unique_ptr`**: Ideal for exclusive ownership of a resource. Use it when you need automatic cleanup and want to enforce single ownership.
- **`std::shared_ptr`**: Suitable for shared ownership of a resource. Use it when multiple parts of your code need to share access to the same resource.
- **`std::weak_ptr`**: Useful for breaking circular references and providing non-owning references. Use it when you need to check if an object still exists without affecting its lifetime.

Understanding the strengths and weaknesses of each smart pointer will help you choose the right one for your specific use case, ensuring optimal memory management and avoiding common pitfalls in your C++ programs.

3.2 `std::optional`, `std::variant`, `std::any`

The C++ Standard Library provides several utility types that help manage optional values, type-safe unions, and type-erased values. These utilities—`std::optional`, `std::variant`, and `std::any`—are part of the `<optional>`, `<variant>`, and `<any>` headers, respectively. They are designed to handle specific scenarios where traditional types and constructs may fall short. This section will provide an in-depth exploration of these utilities, their features, and their appropriate usage.

3.2.1 `std::optional`

1. **Overview:**

   `std::optional` is a utility that represents an optional value, i.e., a value that may or may not be present. It is particularly useful for functions that may or may not return a value, or for data members that may or may not be initialized.

2. **Key Features:**

   - **Optional Value**: Can either contain a value or be empty.
   - **Type Safety**: Ensures that the value, if present, is of the specified type.
   - **No Dynamic Allocation**: Does not use dynamic memory allocation, making it efficient.
   - **Monadic Operations**: Supports operations like `and_then`, `transform`, and `or_else` (C++23).

3. **Common Operations:**

   - **Declaration**:

```
std::optional<int> opt; // Empty optional
std::optional<int> opt2 = 42; // Optional with value
```

- **Checking for Value**:

```
if (opt.has_value()) {
    std::cout << "Value: " << opt.value() << std::endl;
} else {
    std::cout << "No value" << std::endl;
}
```

- **Accessing the Value**:

```
int value = opt.value(); // Throws std::bad_optional_access if
↪  empty
int valueOr = opt.value_or(100); // Returns 100 if empty
```

- **Resetting the Optional**:

```
opt.reset(); // Makes the optional empty
```

- **Monadic Operations (C++23)**:

```
std::optional<int> opt = 42;
auto result = opt.and_then([](int x) { return
↪  std::optional<int>(x * 2); }); // result = 84
```

4. **Use Cases:**

- When a function may or may not return a value.

- When a data member may or may not be initialized.
- When you need to represent a value that can be absent without using pointers or special sentinel values.

5. **Example:**

```cpp
#include <iostream>
#include <optional>

std::optional<int> divide(int a, int b) {
    if (b == 0) {
        return std::nullopt; // No value
    }
    return a / b; // Return value
}

int main() {
    auto result = divide(10, 2);
    if (result) {
        std::cout << "Result: " << *result << std::endl; // Output:
        ↪  Result: 5
    } else {
        std::cout << "Division by zero" << std::endl;
    }

    return 0;
}
```

3.2.2 `std::variant`

1. **Overview:**

`std::variant` is a type-safe union that can hold a value of one of several specified types. It is useful when you need to store a value that can be one of several types, and you want to ensure type safety.

2. **Key Features:**

   - **Type-Safe Union**: Can hold a value of one of several types.
   - **No Dynamic Allocation**: Does not use dynamic memory allocation, making it efficient.
   - **Visitation**: Supports visiting the value using `std::visit`.
   - **Index-Based Access**: Allows accessing the type of the currently stored value using `index()`.

3. **Common Operations:**

   - **Declaration**:

```
std::variant<int, double, std::string> var; // Can hold an int,
↪  double, or std::string
```

   - **Assigning a Value**:

```
var = 42; // Holds an int
var = 3.14; // Holds a double
var = "Hello"; // Holds a std::string
```

   - **Accessing the Value**:

```
if (std::holds_alternative<int>(var)) {
    int value = std::get<int>(var); // Get the int value
} else if (std::holds_alternative<double>(var)) {
    double value = std::get<double>(var); // Get the double value
} else if (std::holds_alternative<std::string>(var)) {
    std::string value = std::get<std::string>(var); // Get the
    ↪  std::string value
}
```

- **Visitation**:

```
std::visit([](auto&& arg) {
    std::cout << arg << std::endl;
}, var);
```

- **Index-Based Access**:

```
std::size_t index = var.index(); // Get the index of the
↪  currently held type
```

4. **Use Cases:**
   - When you need to store a value that can be one of several types.
   - When you want to ensure type safety in a union-like structure.
   - When you need to perform operations on a value whose type is not known at compile time.
5. **Example:**

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

int main() {
    std::variant<int, double, std::string> var = "Hello";

    std::visit([](auto&& arg) {
        std::cout << arg << std::endl; // Output: Hello
    }, var);

    var = 3.14;
    std::cout << "Index: " << var.index() << std::endl; // Output:
    ↪  Index: 1

    return 0;
}
```

3.2.3 `std::any`

1. **Overview:**

   `std::any` is a type-erased container that can hold a value of any type. It is useful when you need to store a value whose type is not known at compile time, and you want to defer type checking to runtime.

2. **Key Features:**

   - **Type Erasure**: Can hold a value of any type.
   - **Type Safety**: Ensures that the value is accessed with the correct type.
   - **Dynamic Allocation**: Uses dynamic memory allocation for type-erased storage.

- **Type Checking**: Allows checking the type of the stored value at runtime.

3. **Common Operations:**

- **Declaration**:

```
std::any anyValue; // Empty any
```

- **Assigning a Value**:

```
anyValue = 42; // Holds an int
anyValue = 3.14; // Holds a double
anyValue = std::string("Hello"); // Holds a std::string
```

- **Accessing the Value**:

```
if (anyValue.type() == typeid(int)) {
    int value = std::any_cast<int>(anyValue); // Get the int
    ↪ value
} else if (anyValue.type() == typeid(double)) {
    double value = std::any_cast<double>(anyValue); // Get the
    ↪ double value
} else if (anyValue.type() == typeid(std::string)) {
    std::string value = std::any_cast<std::string>(anyValue); //
    ↪ Get the std::string value
}
```

- **Resetting the Any**:

```
anyValue.reset(); // Makes the any empty
```

4. **Use Cases:**

   - When you need to store a value whose type is not known at compile time.
   - When you need to defer type checking to runtime.
   - When you need a flexible container that can hold any type of value.

5. **Example:**

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

int main() {
    std::any anyValue = 42;

    if (anyValue.type() == typeid(int)) {
        int value = std::any_cast<int>(anyValue);
        std::cout << "Value: " << value << std::endl; // Output:
        ↪ Value: 42
    }

    anyValue = std::string("Hello");
    if (anyValue.type() == typeid(std::string)) {
        std::string value = std::any_cast<std::string>(anyValue);
        std::cout << "Value: " << value << std::endl; // Output:
        ↪ Value: Hello
    }
```

```
    return 0;
}
```

3.2.4 Summary

- **`std::optional`**: Use it when you need to represent an optional value that may or may not be present.
- **`std::variant`**: Use it when you need to store a value that can be one of several types, ensuring type safety.
- **`std::any`**: Use it when you need to store a value whose type is not known at compile time, deferring type checking to runtime.

Understanding the strengths and weaknesses of each utility will help you choose the right one for your specific use case, ensuring type safety and flexibility in your C++ programs.

3.3 `std::function` and `std::bind`

The C++ Standard Library provides powerful utilities for working with callable objects, such as functions, lambdas, and function objects. Two of the most important utilities in this category are `std::function` and `std::bind`. These tools allow you to store, pass, and manipulate callable objects in a flexible and type-safe manner. This section will provide an in-depth exploration of `std::function` and `std::bind`, their features, and their appropriate usage.

3.3.1 `std::function`

1. **Overview:**

   `std::function` is a polymorphic function wrapper that can store, copy, and invoke any callable object—such as functions, lambdas, and function objects—that matches a specific signature. It is part of the `<functional>` header and is particularly useful for implementing callbacks, event handlers, and other scenarios where you need to store and invoke callable objects.

2. **Key Features:**

   - **Polymorphic**: Can store any callable object that matches the specified signature.
   - **Type Safety**: Ensures that the stored callable object matches the expected signature.
   - **Flexibility**: Can be used to store functions, lambdas, function objects, and more.
   - **Copyable and Movable**: Supports copy and move semantics, making it easy to pass around.

3. **Common Operations:**

   - **Declaration**:

```
std::function<int(int, int)> func; // Declares a function wrapper
↪  that takes two ints and returns an int
```

- **Assigning a Callable**:

```
func = [](int a, int b) { return a + b; }; // Assigns a lambda to
↪  the function wrapper
```

- **Invoking the Callable**:

```
int result = func(2, 3); // Calls the stored lambda, result = 5
```

- **Checking for a Stored Callable**:

```
if (func) {
    std::cout << "Function is callable" << std::endl;
} else {
    std::cout << "Function is empty" << std::endl;
}
```

- **Resetting the Function**:

```
func = nullptr; // Resets the function wrapper to an empty state
```

4. **Use Cases:**

- When you need to store a callable object for later invocation.
- When you need to pass a callable object as a parameter or return it from a function.
- When you need to implement callbacks or event handlers.

5. **Example:**

```cpp
#include <iostream>
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::function<int(int, int)> func = add; // Stores a function
    ↪ pointer
    std::cout << "Result: " << func(2, 3) << std::endl; // Output:
    ↪ Result: 5

    func = [](int a, int b) { return a * b; }; // Stores a lambda
    std::cout << "Result: " << func(2, 3) << std::endl; // Output:
    ↪ Result: 6

    return 0;
}
```

3.3.2 `std::bind`

1. **Overview:**

   `std::bind` is a utility that allows you to create a new callable object by binding arguments to a function or callable object. It is part of the `<functional>` header and is particularly useful for creating function objects with pre-bound arguments, which can then be passed around and invoked later.

2. **Key Features:**

- **Argument Binding**: Binds specific arguments to a function or callable object.
- **Placeholders**: Uses placeholders (`std::placeholders::_1`, `std::placeholders::_2`, etc.) to specify arguments that will be provided at the time of invocation.
- **Flexibility**: Can be used with functions, lambdas, function objects, and more.
- **Copyable and Movable**: Supports copy and move semantics, making it easy to pass around.

3. **Common Operations:**

- **Binding Arguments**:

```
auto boundFunc = std::bind(add, 2, 3); // Binds the arguments 2
↪  and 3 to the add function
int result = boundFunc(); // Calls add(2, 3), result = 5
```

- **Using Placeholders**:

```
auto boundFunc = std::bind(add, std::placeholders::_1, 3); //
↪  Binds the second argument to 3
int result = boundFunc(2); // Calls add(2, 3), result = 5
```

- **Binding Member Functions**:

```
struct MyClass {
    int multiply(int a, int b) { return a * b; }
};
MyClass obj;
auto boundFunc = std::bind(&MyClass::multiply, &obj,
↪  std::placeholders::_1, std::placeholders::_2);
int result = boundFunc(2, 3); // Calls obj.multiply(2, 3), result
↪  = 6
```

4. **Use Cases:**

   - When you need to create a callable object with pre-bound arguments.
   - When you need to pass a callable object with some arguments fixed and others provided later.
   - When you need to bind member functions to specific objects.

5. **Example:**

```cpp
#include <iostream>
#include <functional>

int add(int a, int b) {
    return a + b;
}

struct MyClass {
    int multiply(int a, int b) { return a * b; }
};

int main() {
    // Binding arguments to a function
    auto boundFunc = std::bind(add, 2, 3);
    std::cout << "Result: " << boundFunc() << std::endl; // Output:
    ↪ Result: 5

    // Using placeholders
    auto boundFunc2 = std::bind(add, std::placeholders::_1, 3);
    std::cout << "Result: " << boundFunc2(2) << std::endl; // Output:
    ↪ Result: 5
```

```
    // Binding member functions
    MyClass obj;
    auto boundFunc3 = std::bind(&MyClass::multiply, &obj,
    ↪  std::placeholders::_1, std::placeholders::_2);
    std::cout << "Result: " << boundFunc3(2, 3) << std::endl; //
    ↪  Output: Result: 6

    return 0;
}
```

3.3.3 Combining `std::function` and `std::bind`

`std::function` and `std::bind` can be used together to create flexible and reusable callable objects. For example, you can use `std::bind` to create a callable object with pre-bound arguments and then store it in a `std::function` for later invocation.

Example:

```
#include <iostream>
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    // Bind the first argument to 2
    auto boundFunc = std::bind(add, 2, std::placeholders::_1);

    // Store the bound function in a std::function
```

```
    std::function<int(int)> func = boundFunc;

    // Invoke the function
    std::cout << "Result: " << func(3) << std::endl; // Output: Result: 5

    return 0;
}
```

3.3.4 Summary

- **`std::function`**: Use it when you need to store, pass, or invoke callable objects in a type-safe and flexible manner.
- **`std::bind`**: Use it when you need to create callable objects with pre-bound arguments or when you need to bind member functions to specific objects.
- **Combining `std::function` and `std::bind`**: Use them together to create flexible and reusable callable objects with pre-bound arguments.

Understanding the strengths and weaknesses of `std::function` and `std::bind` will help you choose the right tool for your specific use case, ensuring flexibility and type safety in your C++ programs.

Chapter 4

Iterators and Ranges

4.1 Iterator Categories

Iterators are a fundamental concept in the C++ Standard Library, providing a way to traverse and manipulate elements in containers. Iterators abstract the details of container implementations, allowing algorithms to operate on different types of containers in a uniform manner. Iterators are categorized into several types, each with specific capabilities and constraints. Understanding these categories is crucial for writing efficient and correct code. This section will provide an in-depth exploration of iterator categories, their characteristics, and their appropriate usage.

4.1.1 Overview of Iterator Categories

Iterators are classified into five main categories, each representing a different level of functionality and access to container elements. These categories form a hierarchy, with each category inheriting the capabilities of the previous one:

1. **Input Iterators**

2. **Output Iterators**
3. **Forward Iterators**
4. **Bidirectional Iterators**
5. **Random Access Iterators**

Each category has specific requirements and guarantees, which determine how iterators can be used in algorithms and operations.

4.1.2 Input Iterators

1. **Overview:**

   Input iterators are the simplest type of iterator, providing read-only access to elements in a sequence. They are typically used for single-pass algorithms, where each element is processed exactly once.

2. **Key Features:**

   - **Read Access**: Can read elements from the sequence.
   - **Single-Pass**: Can only traverse the sequence once; cannot go back to previous elements.
   - **Incrementable**: Can be incremented to move to the next element.

3. **Common Operations:**

   - **Dereference**:

```
int value = *it; // Read the value at the current position
```

- **Increment**:

```
++it; // Move to the next element
```

- **Equality Comparison**:

```
if (it1 == it2) { /* ... */ } // Check if two iterators point to
↪  the same element
```

4. **Use Cases:**

- When you need to read elements from a sequence in a single pass.
- When working with algorithms that only require forward traversal, such as `std::find` or `std::accumulate`.

5. **Example:**

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    auto it = vec.begin(); // Input iterator

    while (it != vec.end()) {
        std::cout << *it << " "; // Output: 1 2 3 4 5
        ++it;
```

```
    }

    return 0;
}
```

4.1.3 Output Iterators

1. **Overview:**

   Output iterators provide write-only access to elements in a sequence. They are typically used for single-pass algorithms, where each element is written exactly once.

2. **Key Features:**

   - **Write Access**: Can write elements to the sequence.
   - **Single-Pass**: Can only traverse the sequence once; cannot go back to previous elements.
   - **Incrementable**: Can be incremented to move to the next element.

3. **Common Operations:**

   - **Dereference**:

     ```
     *it = value; // Write a value to the current position
     ```

   - **Increment**:

     ```
     ++it; // Move to the next element
     ```

4. **Use Cases:**

   - When you need to write elements to a sequence in a single pass.
   - When working with algorithms that only require forward traversal, such as `std::copy` or `std::fill`.

5. **Example:**

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

int main() {
    std::vector<int> vec(5);
    auto it = vec.begin(); // Output iterator

    for (int i = 0; i < 5; ++i) {
        *it = i + 1; // Write values to the sequence
        ++it;
    }

    for (int value : vec) {
        std::cout << value << " "; // Output: 1 2 3 4 5
    }

    return 0;
}
```

4.1.4 Forward Iterators

1. **Overview:**

Forward iterators provide read and write access to elements in a sequence and support multi-pass traversal. They are more powerful than input and output iterators, allowing algorithms to traverse the sequence multiple times.

2. **Key Features:**

   - **Read and Write Access**: Can read and write elements in the sequence.
   - **Multi-Pass**: Can traverse the sequence multiple times.
   - **Incrementable**: Can be incremented to move to the next element.

3. **Common Operations:**

   - **Dereference**:

   ```
   int value = *it; // Read the value at the current position
   *it = value; // Write a value to the current position
   ```

   - **Increment**:

   ```
   ++it; // Move to the next element
   ```

   - **Equality Comparison**:

   ```
   if (it1 == it2) { /* ... */ } // Check if two iterators point to
   ↪  the same element
   ```

4. **Use Cases:**

   - When you need to read and write elements in a sequence with multi-pass traversal.

- When working with algorithms that require forward traversal, such as `std::replace` or `std::unique`.

5. **Example:**

```cpp
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> list = {1, 2, 3, 4, 5};
    auto it = list.begin(); // Forward iterator

    while (it != list.end()) {
        *it *= 2; // Modify each element
        ++it;
    }

    for (int value : list) {
        std::cout << value << " "; // Output: 2 4 6 8 10
    }

    return 0;
}
```

4.1.5 Bidirectional Iterators

1. **Overview:**

   Bidirectional iterators provide all the capabilities of forward iterators, with the additional ability to traverse the sequence in both forward and backward directions.

2. **Key Features:**

- **Read and Write Access**: Can read and write elements in the sequence.
- **Multi-Pass**: Can traverse the sequence multiple times.
- **Incrementable and Decrementable**: Can be incremented to move to the next element and decremented to move to the previous element.

3. **Common Operations:**

- **Dereference**:

```
int value = *it; // Read the value at the current position
*it = value; // Write a value to the current position
```

- **Increment**:

```
++it; // Move to the next element
```

- **Decrement**:

```
--it; // Move to the previous element
```

- **Equality Comparison**:

```
if (it1 == it2) { /* ... */ } // Check if two iterators point to
↪  the same element
```

4. **Use Cases:**

- When you need to traverse a sequence in both forward and backward directions.

- When working with algorithms that require bidirectional traversal, such as `std::reverse` or `std::list::sort`.

5. **Example:**

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

int main() {
    std::list<int> list = {1, 2, 3, 4, 5};
    auto it = list.end(); // Bidirectional iterator

    while (it != list.begin()) {
        --it;
        std::cout << *it << " "; // Output: 5 4 3 2 1
    }

    return 0;
}
```

4.1.6 Random Access Iterators

1. **Overview:**

   Random access iterators provide the most functionality, allowing direct access to any element in the sequence. They support all the capabilities of bidirectional iterators, with the addition of random access and arithmetic operations.

2. **Key Features:**

   - **Read and Write Access**: Can read and write elements in the sequence.

- **Multi-Pass**: Can traverse the sequence multiple times.
- **Incrementable and Decrementable**: Can be incremented to move to the next element and decremented to move to the previous element.
- **Random Access**: Can directly access any element in the sequence using arithmetic operations.

3. **Common Operations:**

- **Dereference**:

```
int value = *it; // Read the value at the current position
*it = value; // Write a value to the current position
```

- **Increment**:

```
++it; // Move to the next element
```

- **Decrement**:

```
--it; // Move to the previous element
```

- **Arithmetic Operations**:

```
it += 3; // Move 3 elements forward
it -= 2; // Move 2 elements backward
int value = it[2]; // Access the element 2 positions ahead
```

- **Equality and Inequality Comparison**:

```
if (it1 == it2) { /* ... */ } // Check if two iterators point to
↪  the same element
if (it1 < it2) { /* ... */ } // Check if one iterator is before
↪  another
```

4. **Use Cases:**

   - When you need direct access to any element in the sequence.
   - When working with algorithms that require random access, such as `std::sort` or `std::binary_search`.

5. **Example:**

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    auto it = vec.begin(); // Random access iterator

    it += 3; // Move to the 4th element
    std::cout << *it << std::endl; // Output: 4

    it -= 2; // Move back to the 2nd element
    std::cout << *it << std::endl; // Output: 2

    std::cout << it[2] << std::endl; // Output: 4 (element 2
    ↪  positions ahead)

    return 0;
}
```

4.1.7 Summary

- **Input Iterators**: Provide read-only access and support single-pass traversal.
- **Output Iterators**: Provide write-only access and support single-pass traversal.
- **Forward Iterators**: Provide read and write access and support multi-pass traversal.
- **Bidirectional Iterators**: Provide all the capabilities of forward iterators, with the addition of backward traversal.
- **Random Access Iterators**: Provide all the capabilities of bidirectional iterators, with the addition of random access and arithmetic operations.

Understanding the strengths and weaknesses of each iterator category will help you choose the right tool for your specific use case, ensuring efficient and correct traversal and manipulation of container elements in your C++ programs.

4.2 Ranges Library (C++20)

The Ranges library, introduced in C++20, is a significant enhancement to the C++ Standard Library that simplifies working with sequences of elements, such as containers and arrays. It provides a more expressive and composable way to perform operations on ranges, making code more readable and maintainable. The Ranges library builds on the concepts of iterators and algorithms, offering a higher-level abstraction for working with sequences. This section will provide an in-depth exploration of the Ranges library, its features, and its appropriate usage.

4.2.1 Overview of the Ranges Library

The Ranges library introduces several new concepts and utilities that make it easier to work with sequences of elements. Key components of the Ranges library include:

- **Range Concepts**: Define what constitutes a range and provide constraints for working with ranges.
- **Range Adaptors**: Allow you to transform, filter, and compose ranges in a declarative manner.
- **Range Algorithms**: Provide a set of algorithms that operate directly on ranges, rather than iterators.

The Ranges library is part of the `<ranges>` header and is designed to work seamlessly with existing STL containers and algorithms.

4.2.2 Range Concepts

Range concepts define the requirements for types that represent ranges. A range is a sequence of elements that can be iterated over. The Ranges library introduces several range concepts, including:

- **`std::ranges::range`**: A type that can be iterated over using `begin()` and `end()`.
- **`std::ranges::sized_range`**: A range that knows its size, i.e., it provides a `size()` method.
- **`std::ranges::view`**: A lightweight, non-owning range that can be composed with other views.

Example:

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

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

    // Check if vec is a range
    if constexpr (std::ranges::range<decltype(vec)>) {
        std::cout << "vec is a range" << std::endl;
    }

    // Check if vec is a sized range
    if constexpr (std::ranges::sized_range<decltype(vec)>) {
        std::cout << "vec is a sized range with size " << vec.size() <<
        ↪  std::endl;
    }

    return 0;
}
```

4.2.3 Range Adaptors

Range adaptors are utilities that allow you to transform, filter, and compose ranges in a declarative manner. They provide a powerful way to create pipelines of operations on ranges. Some common range adaptors include:

- **`std::views::filter`**: Filters elements based on a predicate.
- **`std::views::transform`**: Transforms elements using a function.
- **`std::views::take`**: Takes the first `n` elements of a range.
- **`std::views::drop`**: Drops the first `n` elements of a range.
- **`std::views::reverse`**: Reverses the elements of a range.

Example:

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

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

    // Create a view that filters even numbers and squares them
    auto view = vec | std::views::filter([](int x) { return x % 2 == 0; })
                    | std::views::transform([](int x) { return x * x; });

    // Print the transformed elements
    for (int value : view) {
        std::cout << value << " "; // Output: 4 16
    }
```

```
    return 0;
}
```

4.2.4 Range Algorithms

The Ranges library provides a set of algorithms that operate directly on ranges, rather than iterators. These algorithms are more expressive and easier to use than their iterator-based counterparts. Some common range algorithms include:

- **`std::ranges::sort`**: Sorts a range.
- **`std::ranges::find`**: Finds an element in a range.
- **`std::ranges::for_each`**: Applies a function to each element in a range.
- **`std::ranges::count`**: Counts the occurrences of a value in a range.

Example:

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

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

    // Sort the range
    std::ranges::sort(vec);
```

```
    // Find an element in the range
    auto it = std::ranges::find(vec, 3);
    if (it != vec.end()) {
        std::cout << "Found: " << *it << std::endl; // Output: Found: 3
    }

    // Apply a function to each element in the range
    std::ranges::for_each(vec, [](int x) { std::cout << x << " "; }); //
    ↪  Output: 1 2 3 4 5

    return 0;
}
```

4.2.5 Composing Range Adaptors

One of the most powerful features of the Ranges library is the ability to compose range adaptors to create complex pipelines of operations. This allows you to express complex transformations and filters in a clear and concise manner.

Example:

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Create a view that filters even numbers, squares them, and takes
    ↪  the first 3 elements
    auto view = vec | std::views::filter([](int x) { return x % 2 == 0; })
```

```
                    | std::views::transform([](int x) { return x * x; })
                    | std::views::take(3);

    // Print the transformed elements
    for (int value : view) {
        std::cout << value << " "; // Output: 4 16 36
    }

    return 0;
}
```

4.2.6 Benefits of the Ranges Library

The Ranges library offers several benefits over traditional iterator-based approaches:

- **Expressiveness**: Range adaptors and algorithms allow you to express complex operations in a clear and concise manner.
- **Composability**: Range adaptors can be easily composed to create pipelines of operations.
- **Readability**: Code using the Ranges library is often more readable and easier to understand.
- **Safety**: Range concepts and constraints help catch errors at compile time, improving code safety.

4.2.7 Summary

- **Range Concepts**: Define what constitutes a range and provide constraints for working with ranges.

- **Range Adaptors**: Allow you to transform, filter, and compose ranges in a declarative manner.
- **Range Algorithms**: Provide a set of algorithms that operate directly on ranges, making code more expressive and easier to read.

Understanding and using the Ranges library effectively can significantly improve the readability, maintainability, and safety of your C++ programs. By leveraging range adaptors and algorithms, you can write more expressive and composable code that is easier to understand and maintain.

Chapter 5

Practical Examples

5.1 Programs Using STL Containers and Algorithms (e.g., Sorting, Searching)

The C++ Standard Library (STL) provides a rich set of containers and algorithms that can be used to solve a wide variety of programming problems. In this section, we will explore practical examples of programs that use STL containers (such as `std::vector`, `std::map`, and `std::set`) and algorithms (such as sorting and searching) to solve common problems. These examples will demonstrate how to effectively combine containers and algorithms to write efficient and maintainable code.

5.1.1 Sorting a Vector of Integers

Sorting is one of the most common operations in programming. The STL provides the `std::sort` algorithm, which can be used to sort elements in a container like `std::vector`.

Example: Sorting a Vector of Integers

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort

int main() {
    std::vector<int> numbers = {5, 2, 9, 1, 5, 6};

    // Sort the vector in ascending order
    std::sort(numbers.begin(), numbers.end());

    // Print the sorted vector
    for (int num : numbers) {
        std::cout << num << " "; // Output: 1 2 5 5 6 9
    }

    return 0;
}
```

Explanation:

- **`std::vector<int>`**: A dynamic array that stores integers.
- **`std::sort`**: Sorts the elements in the range `[begin, end)` in ascending order by default.
- **Range-based for loop**: Used to iterate over the sorted vector and print its elements.

5.1.2 Sorting a Vector of Strings

Sorting is not limited to integers; you can also sort other types, such as strings.

Example: Sorting a Vector of Strings

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort
#include <string>

int main() {
    std::vector<std::string> names = {"Alice", "Bob", "Charlie", "David",
    ↪  "Eve"};

    // Sort the vector in ascending order
    std::sort(names.begin(), names.end());

    // Print the sorted vector
    for (const std::string& name : names) {
        std::cout << name << " "; // Output: Alice Bob Charlie David Eve
    }

    return 0;
}
```

Explanation:

- **`std::vector<std::string>`**: A dynamic array that stores strings.
- **`std::sort`**: Sorts the strings lexicographically (alphabetically).

5.1.3 Sorting in Descending Order

You can sort elements in descending order by providing a custom comparator to `std::sort`.

Example: Sorting in Descending Order

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort

int main() {
    std::vector<int> numbers = {5, 2, 9, 1, 5, 6};

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

    // Print the sorted vector
    for (int num : numbers) {
        std::cout << num << " "; // Output: 9 6 5 5 2 1
    }

    return 0;
}
```

Explanation:

- **`std::greater<int>()`**: A comparator that sorts elements in descending order.

5.1.4 Searching in a Sorted Vector

Once a vector is sorted, you can efficiently search for elements using the `std::binary_search` algorithm, which performs a binary search in logarithmic time.

Example: Searching in a Sorted Vector

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort and std::binary_search
```

```cpp
int main() {
    std::vector<int> numbers = {5, 2, 9, 1, 5, 6};

    // Sort the vector
    std::sort(numbers.begin(), numbers.end());

    // Search for an element
    int target = 5;
    if (std::binary_search(numbers.begin(), numbers.end(), target)) {
        std::cout << target << " found in the vector." << std::endl; //
        ↪  Output: 5 found in the vector.
    } else {
        std::cout << target << " not found in the vector." << std::endl;
    }

    return 0;
}
```

Explanation:

- **`std::binary_search`**: Checks if the target element exists in the sorted range `[begin, end)`.
- **Efficiency**: Binary search works in O(log n) time, making it suitable for large datasets.

5.1.5 Finding the Position of an Element

If you need to find the position of an element in a sorted vector, you can use `std::lower_bound` or `std::upper_bound`.

Example: Finding the Position of an Element

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort, std::lower_bound, and
↪  std::upper_bound

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

    // Sort the vector (already sorted in this case)
    std::sort(numbers.begin(), numbers.end());

    // Find the first element not less than 4
    auto lower = std::lower_bound(numbers.begin(), numbers.end(), 4);
    std::cout << "Lower bound of 4 at position: " << (lower -
    ↪  numbers.begin()) << std::endl; // Output: 2

    // Find the first element greater than 4
    auto upper = std::upper_bound(numbers.begin(), numbers.end(), 4);
    std::cout << "Upper bound of 4 at position: " << (upper -
    ↪  numbers.begin()) << std::endl; // Output: 4

    return 0;
}
```

Explanation:

- **`std::lower_bound`**: Returns an iterator to the first element that is not less than the target value.
- **`std::upper_bound`**: Returns an iterator to the first element that is greater than the target value.

5.1.6 Counting Occurrences of Elements

The `std::count` algorithm can be used to count the number of occurrences of a specific element in a container.

Example: Counting Occurrences of an Element

```
#include <iostream>
#include <vector>
#include <algorithm> // for std::count

int main() {
    std::vector<int> numbers = {5, 2, 9, 1, 5, 6};

    // Count the number of occurrences of 5
    int target = 5;
    int count = std::count(numbers.begin(), numbers.end(), target);

    std::cout << target << " appears " << count << " times." << std::endl;
    ↪  // Output: 5 appears 2 times.

    return 0;
}
```

Explanation:

- **`std::count`**: Counts the number of elements in the range `[begin, end)` that are equal to the target value.
- **Linear Search**: This algorithm works in O(n) time, where `n` is the size of the container.

5.1.7 Using `std::map` for Key-Value Pairs

`std::map` is an associative container that stores key-value pairs in sorted order. It is useful for scenarios where you need to look up values by their keys efficiently.

Example: Using `std::map` to Store and Retrieve Data

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

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

    // Insert key-value pairs
    ageMap["Alice"] = 30;
    ageMap["Bob"] = 25;
    ageMap["Charlie"] = 35;

    // Access values using keys
    std::cout << "Alice's age: " << ageMap["Alice"] << std::endl; //
    ↪  Output: Alice's age: 30
    std::cout << "Bob's age: " << ageMap["Bob"] << std::endl;     //
    ↪  Output: Bob's age: 25

    // Check if a key exists
    if (ageMap.find("Charlie") != ageMap.end()) {
        std::cout << "Charlie's age: " << ageMap["Charlie"] << std::endl;
        ↪  // Output: Charlie's age: 35
    }

    return 0;
}
```

Explanation:

- **`std::map`**: Stores key-value pairs in sorted order based on the keys.
- **Efficiency**: Lookup, insertion, and deletion operations are performed in O(log n) time.

5.1.8 Using `std::set` for Unique Elements

`std::set` is a container that stores unique elements in sorted order. It is useful when you need to maintain a collection of unique items.

Example: Using `std::set` to Store Unique Elements

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

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

    // Insert elements
    uniqueNumbers.insert(5);
    uniqueNumbers.insert(2);
    uniqueNumbers.insert(9);
    uniqueNumbers.insert(2); // Duplicate, will not be inserted

    // Print the unique elements
    for (int num : uniqueNumbers) {
        std::cout << num << " "; // Output: 2 5 9
    }

    return 0;
}
```

Explanation:

- **`std::set`**: Ensures that all elements are unique and stored in sorted order.
- **Efficiency**: Insertion, deletion, and lookup operations are performed in O(log n) time.

5.1.9 Combining Containers and Algorithms: Finding Common Elements

You can combine STL containers and algorithms to solve more complex problems. For example, you can find the common elements between two vectors using `std::set_intersection`.

Example: Finding Common Elements Between Two Vectors

```
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort and std::set_intersection

int main() {
    std::vector<int> vec1 = {1, 3, 5, 7, 9};
    std::vector<int> vec2 = {2, 3, 5, 8, 9};

    // Sort both vectors
    std::sort(vec1.begin(), vec1.end());
    std::sort(vec2.begin(), vec2.end());

    // Find common elements
    std::vector<int> commonElements;
    std::set_intersection(vec1.begin(), vec1.end(),
                          vec2.begin(), vec2.end(),
                          std::back_inserter(commonElements));

    // Print the common elements
    for (int num : commonElements) {
```

```
        std::cout << num << " "; // Output: 3 5 9
    }

    return 0;
}
```

Explanation:

- **`std::set_intersection`**: Computes the intersection of two sorted ranges and stores the result in a container.
- **`std::back_inserter`**: Inserts elements at the end of the `commonElements` vector.

5.1.10 Using `std::accumulate` for Summation

The `std::accumulate` algorithm can be used to compute the sum (or other accumulations) of elements in a container.

Example: Summing Elements in a Vector

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

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

    // Compute the sum of elements
    int sum = std::accumulate(numbers.begin(), numbers.end(), 0);

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

```
    return 0;
}
```

Explanation:

- **`std::accumulate`**: Computes the sum of elements in the range `[begin, end)`, starting with an initial value (0 in this case).
- **Flexibility**: Can also be used with custom operations (e.g., multiplication) by providing a binary function.

5.1.11 Summary

- **STL Containers**: Use `std::vector` for dynamic arrays, `std::map` for key-value pairs, and `std::set` for unique elements.
- **STL Algorithms**: Use `std::sort` for sorting, `std::binary_search` for searching, `std::count` for counting, and `std::accumulate` for summation.
- **Combining Containers and Algorithms**: Solve complex problems by combining containers and algorithms, such as finding common elements with `std::set_intersection`.

These examples demonstrate how to effectively use STL containers and algorithms to write efficient and maintainable C++ programs. By mastering these tools, you can tackle a wide range of programming challenges with ease.

Chapter 6

Allocators and Benchmarks

6.1 Custom Allocators

Allocators are a fundamental part of the C++ Standard Library, responsible for managing memory allocation and deallocation for containers. By default, STL containers use the standard allocator (`std::allocator`), which relies on the global `new` and `delete` operators. However, there are scenarios where custom allocators are necessary or beneficial, such as optimizing performance, managing memory pools, or integrating with specialized memory systems. This section will provide a detailed exploration of custom allocators, their implementation, and their use cases.

6.1.1 Overview of Allocators

What Are Allocators?

Allocators are objects that encapsulate memory management strategies. They provide a way to control how memory is allocated and deallocated for STL containers. The default allocator, `std::allocator`, uses the global `new` and `delete` operators, but custom allocators can be

implemented to use different memory management strategies.

Why Use Custom Allocators?

- **Performance Optimization**: Custom allocators can be tailored to specific use cases, such as memory pools or arena allocators, to reduce fragmentation and improve performance.
- **Specialized Memory Management**: Custom allocators can be used to allocate memory from specific regions, such as shared memory or GPU memory.
- **Integration with Existing Systems**: Custom allocators can integrate with existing memory management systems, such as those used in game engines or real-time systems.

6.1.2 The Allocator Interface

Custom allocators must conform to the allocator interface defined by the C++ Standard Library. The interface includes the following key components:

- `allocate(size_t n)`: Allocates memory for `n` objects of type `T`.
- `deallocate(T* p, size_t n)`: Deallocates memory previously allocated for `n` objects of type `T`.
- `construct(T* p, Args&&... args)`: Constructs an object of type `T` at the memory location `p` using the arguments `args`.
- `destroy(T* p)`: Destroys the object of type `T` at the memory location `p`.

Additionally, custom allocators must provide typedefs for `value_type`, `pointer`, `const_pointer`, `size_type`, and `difference_type`.

6.1.3 Implementing a Custom Allocator

Let's implement a simple custom allocator that uses a fixed-size memory pool. This allocator will allocate memory from a pre-allocated buffer and manage allocations within that buffer.

Example: Fixed-Size Memory Pool Allocator

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

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

    PoolAllocator() : pool(new char[PoolSize * sizeof(T)]),
    ↪  next(pool.get()) {}

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

    T* allocate(std::size_t n) {
        if (n > PoolSize || next + n * sizeof(T) > pool.get() + PoolSize *
        ↪  sizeof(T)) {
            throw std::bad_alloc();
        }
        T* result = reinterpret_cast<T*>(next);
        next += n * sizeof(T);
        return result;
    }
```

```
    void deallocate(T* p, std::size_t n) noexcept {
        // No-op for this simple allocator
    }

    template <typename U, typename... Args>
    void construct(U* p, Args&&... args) {
        new (p) U(std::forward<Args>(args)...);
    }

    template <typename U>
    void destroy(U* p) {
        p->~U();
    }

private:
    std::unique_ptr<char[]> pool;
    char* next;
};

int main() {
    // Use the custom allocator with a vector
    std::vector<int, PoolAllocator<int, 10>> vec;

    // Add elements to the vector
    for (int i = 0; i < 10; ++i) {
        vec.push_back(i);
    }

    // Print the elements
    for (int value : vec) {
        std::cout << value << " "; // Output: 0 1 2 3 4 5 6 7 8 9
    }
```

```
    return 0;
}
```

Explanation:

- **`PoolAllocator`**: A custom allocator that allocates memory from a fixed-size pool.
- **`allocate`**: Allocates memory from the pool. Throws `std::bad_alloc` if the pool is exhausted.
- **`deallocate`**: A no-op in this simple allocator, as memory is not freed until the allocator is destroyed.
- **`construct`**: Constructs an object at the specified memory location using placement new.
- **`destroy`**: Destroys the object at the specified memory location by calling its destructor.

6.1.4 Using Custom Allocators with STL Containers

Custom allocators can be used with any STL container by specifying the allocator as a template parameter. The following example demonstrates using the custom `PoolAllocator` with a `std::vector`.

Example: Using Custom Allocator with `std::vector`

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

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

    PoolAllocator() : pool(new char[PoolSize * sizeof(T)]),
    ↪  next(pool.get()) {}

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

    T* allocate(std::size_t n) {
        if (n > PoolSize || next + n * sizeof(T) > pool.get() + PoolSize *
        ↪  sizeof(T)) {
            throw std::bad_alloc();
        }
        T* result = reinterpret_cast<T*>(next);
        next += n * sizeof(T);
        return result;
    }

    void deallocate(T* p, std::size_t n) noexcept {
        // No-op for this simple allocator
    }

    template <typename U, typename... Args>
    void construct(U* p, Args&&... args) {
        new (p) U(std::forward<Args>(args)...);
    }

    template <typename U>
```

```
    void destroy(U* p) {
        p->~U();
    }

private:
    std::unique_ptr<char[]> pool;
    char* next;
};

int main() {
    // Use the custom allocator with a vector
    std::vector<int, PoolAllocator<int, 10>> vec;

    // Add elements to the vector
    for (int i = 0; i < 10; ++i) {
        vec.push_back(i);
    }

    // Print the elements
    for (int value : vec) {
        std::cout << value << " "; // Output: 0 1 2 3 4 5 6 7 8 9
    }

    return 0;
}
```

Explanation:

- **`std::vector<int, PoolAllocator<int, 10>>`**: A `std::vector` that uses the custom `PoolAllocator` with a pool size of 10.
- **Memory Management**: The vector allocates memory from the fixed-size pool managed

by the custom allocator.

6.1.5 Use Cases for Custom Allocators

1. **Memory Pools**

   Memory pools are pre-allocated blocks of memory that can be used to reduce fragmentation and improve performance. Custom allocators can manage memory pools, ensuring that allocations are made from the pool rather than the global heap.

2. **Arena Allocators**

   Arena allocators allocate memory from a fixed-size arena and deallocate all memory at once. This is useful for scenarios where memory is allocated and deallocated in phases, such as in game engines.

3. **Shared Memory**

   Custom allocators can be used to allocate memory from shared memory regions, allowing multiple processes to share data.

4. **GPU Memory**

   Custom allocators can manage memory allocations on the GPU, enabling efficient data transfer between the CPU and GPU.

5. **Best Practices for Custom Allocators**

   - **Ensure Correctness**: Custom allocators must correctly implement the allocator interface and handle edge cases, such as out-of-memory conditions.
   - **Optimize for Performance**: Custom allocators should be designed to minimize fragmentation and maximize performance for specific use cases.
   - **Test Thoroughly**: Custom allocators should be thoroughly tested to ensure they work correctly with all STL containers and algorithms.

6.1.6 Summary

- **Custom Allocators**: Provide a way to control memory allocation and deallocation for STL containers.
- **Implementation**: Custom allocators must conform to the allocator interface and provide methods for allocation, deallocation, construction, and destruction.
- **Use Cases**: Custom allocators are useful for optimizing performance, managing memory pools, and integrating with specialized memory systems.

Understanding and using custom allocators effectively can help you optimize memory management in your C++ programs, leading to improved performance and resource utilization.

6.2 Performance Benchmarks

Performance benchmarks are essential for evaluating the efficiency of different algorithms, data structures, and memory management strategies. In this section, we will explore how to design and conduct performance benchmarks for STL containers and custom allocators. We will also discuss the tools and techniques used to measure performance, interpret results, and make informed decisions about optimizing your code.

6.2.1 Overview of Performance Benchmarks

What Are Performance Benchmarks?
Performance benchmarks are tests designed to measure the speed, memory usage, and scalability of code. They help identify bottlenecks, compare different implementations, and ensure that your code meets performance requirements.

Why Are Benchmarks Important?

- **Identify Bottlenecks**: Benchmarks help pinpoint parts of the code that are slow or inefficient.
- **Compare Implementations**: Benchmarks allow you to compare different algorithms, data structures, or memory management strategies.
- **Validate Optimizations**: Benchmarks provide empirical evidence that optimizations improve performance.

6.2.2 Designing Performance Benchmarks

Key Considerations:

- **Define Objectives**: Clearly define what you want to measure (e.g., execution time, memory usage, throughput).
- **Choose Metrics**: Select appropriate metrics, such as time complexity, space complexity, or cache performance.
- **Control Variables**: Ensure that benchmarks are run under consistent conditions (e.g., same hardware, same input data).
- **Use Realistic Data**: Use realistic input data that reflects the actual use case.

Example: Benchmarking `std::vector` vs. `std::list`
Let's design a benchmark to compare the performance of `std::vector` and `std::list` for different operations, such as insertion, deletion, and iteration.

6.2.3 Implementing Performance Benchmarks

Example: Benchmarking `std::vector` and `std::list`
We will use the `<chrono>` library to measure execution time and compare the performance of `std::vector` and `std::list` for various operations.

```
#include <iostream>
#include <vector>
#include <list>
#include <chrono>
#include <random>

// Function to generate random numbers
std::vector<int> generate_random_numbers(int count) {
    std::vector<int> numbers(count);
```

```
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 1000);

    for (int i = 0; i < count; ++i) {
        numbers[i] = dis(gen);
    }

    return numbers;
}

// Benchmark function for std::vector
void benchmark_vector(const std::vector<int>& numbers) {
    std::vector<int> vec;

    auto start = std::chrono::high_resolution_clock::now();

    // Insert elements
    for (int num : numbers) {
        vec.push_back(num);
    }

    // Iterate over elements
    for (int num : vec) {
        // Simulate some operation
    }

    // Remove elements
    while (!vec.empty()) {
        vec.pop_back();
    }
```

```
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "std::vector time: " << elapsed.count() << " seconds\n";
}

// Benchmark function for std::list
void benchmark_list(const std::vector<int>& numbers) {
    std::list<int> lst;

    auto start = std::chrono::high_resolution_clock::now();

    // Insert elements
    for (int num : numbers) {
        lst.push_back(num);
    }

    // Iterate over elements
    for (int num : lst) {
        // Simulate some operation
    }

    // Remove elements
    while (!lst.empty()) {
        lst.pop_back();
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "std::list time: " << elapsed.count() << " seconds\n";
}

int main() {
```

```
    const int count = 1000000;
    std::vector<int> numbers = generate_random_numbers(count);

    std::cout << "Benchmarking std::vector...\n";
    benchmark_vector(numbers);

    std::cout << "Benchmarking std::list...\n";
    benchmark_list(numbers);

    return 0;
}
```

Explanation:

- **`generate_random_numbers`**: Generates a vector of random integers.
- **`benchmark_vector`**: Measures the time taken to insert, iterate, and remove elements from a `std::vector`.
- **`benchmark_list`**: Measures the time taken to insert, iterate, and remove elements from a `std::list`.
- **`std::chrono::high_resolution_clock`**: Used to measure elapsed time with high precision.

6.2.4 Analyzing Benchmark Results

Interpreting Results:

- **Execution Time**: Compare the time taken by different containers or algorithms.

- **Memory Usage**: Measure the memory footprint using tools like `valgrind` or custom memory tracking.
- **Scalability**: Evaluate how performance scales with input size.

Example Output:

```
Benchmarking std::vector...
std::vector time: 0.012345 seconds
Benchmarking std::list...
std::list time: 0.045678 seconds
```

Analysis:

- **`std::vector`**: Faster for insertion and iteration due to contiguous memory layout.
- **`std::list`**: Slower for insertion and iteration due to pointer overhead, but more efficient for frequent insertions/deletions in the middle.

6.2.5 Tools for Performance Benchmarking

1. **<chrono> Library**

   The <chrono> library provides high-resolution clocks for measuring elapsed time. It is part of the C++ Standard Library and is widely used for benchmarking.

2. **Google Benchmark**

   Google Benchmark is a powerful library for writing and running benchmarks. It provides features like statistical analysis, parameterized benchmarks, and CSV output.

Example: Using Google Benchmark

```
#include <benchmark/benchmark.h>
#include <vector>
#include <list>
#include <random>

static void BM_VectorInsertion(benchmark::State& state) {
    std::vector<int> vec;
    for (auto _ : state) {
        for (int i = 0; i < state.range(0); ++i) {
            vec.push_back(i);
        }
    }
}
BENCHMARK(BM_VectorInsertion)->Range(8, 8 << 10);

static void BM_ListInsertion(benchmark::State& state) {
    std::list<int> lst;
    for (auto _ : state) {
        for (int i = 0; i < state.range(0); ++i) {
            lst.push_back(i);
        }
    }
}
BENCHMARK(BM_ListInsertion)->Range(8, 8 << 10);

BENCHMARK_MAIN();
```

Explanation:

- **`BM_VectorInsertion`**: Benchmarks insertion into a `std::vector`.
- **`BM_ListInsertion`**: Benchmarks insertion into a `std::list`.

- **`BENCHMARK`**: Registers the benchmark function and specifies the range of input sizes.

Output:

```
Run on (4 X 3500 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x2)
  L1 Instruction 32 KiB (x2)
  L2 Unified 256 KiB (x2)
  L3 Unified 8192 KiB (x1)
Load Average: 0.52, 0.58, 0.59
-------------------------------------------------------
Benchmark               Time             CPU   Iterations
-------------------------------------------------------
BM_VectorInsertion/8    18.5 ns         18.5 ns     37333333
BM_VectorInsertion/64   145 ns          145 ns      4480000
BM_VectorInsertion/512  1160 ns         1160 ns      560000
BM_VectorInsertion/4096 9280 ns         9280 ns       74667
BM_ListInsertion/8      24.7 ns         24.7 ns     28000000
BM_ListInsertion/64     197 ns          197 ns      3555556
BM_ListInsertion/512    1576 ns         1576 ns      448000
BM_ListInsertion/4096   12608 ns        12608 ns       56000
```

Analysis:

- **`std::vector`**: Faster insertion times due to contiguous memory layout.
- **`std::list`**: Slower insertion times due to pointer overhead.

6.2.6 Best Practices for Performance Benchmarks

- **Run Multiple Iterations**: Run benchmarks multiple times to account for variability and obtain reliable results.

- **Use Realistic Data**: Use input data that reflects real-world scenarios.
- **Profile Memory Usage**: Measure memory usage to identify memory leaks or excessive allocations.
- **Compare Against Baseline**: Compare performance against a baseline implementation to quantify improvements.

6.2.7 Summary

- **Performance Benchmarks**: Essential for evaluating the efficiency of code and identifying bottlenecks.
- **Tools**: Use libraries like `<chrono>` and Google Benchmark to measure and analyze performance.
- **Best Practices**: Run multiple iterations, use realistic data, and compare against a baseline.

By conducting thorough performance benchmarks, you can make informed decisions about optimizing your code and ensure that it meets performance requirements.

Appendices

Appendix A: Overview of STL Containers

This appendix provides a comprehensive summary of all the STL containers, including:

- **Sequence Containers**: `std::vector`, `std::list`, `std::deque`, `std::array`, `std::forward_list`
- **Associative Containers**: `std::set`, `std::map`, `std::multiset`, `std::multimap`
- **Unordered Associative Containers**: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset`, `std::unordered_multimap`
- **Container Adaptors**: `std::stack`, `std::queue`, `std::priority_queue`

For each container, the appendix includes:

- A brief description of its purpose and use cases.
- Time and space complexity for common operations.
- Example code snippets demonstrating basic usage.
- Key member functions and their descriptions.

Appendix B: STL Algorithms Cheat Sheet

This appendix serves as a quick reference guide to the most commonly used STL algorithms, categorized by functionality:

- **Non-modifying Algorithms**: `std::find`, `std::count`, `std::for_each`, `std::all_of`, `std::any_of`, `std::none_of`
- **Modifying Algorithms**: `std::copy`, `std::transform`, `std::replace`, `std::fill`, `std::remove`
- **Sorting and Searching**: `std::sort`, `std::binary_search`, `std::lower_bound`, `std::upper_bound`
- **Numeric Algorithms**: `std::accumulate`, `std::inner_product`, `std::partial_sum`, `std::iota`

Each algorithm is accompanied by:

- A concise explanation of its purpose.
- Syntax and parameter descriptions.
- Example usage with code snippets.

Appendix C: Iterators and Ranges

This appendix delves deeper into iterators and ranges, which are fundamental to understanding and using the STL effectively. Topics include:

- **Iterator Categories**: Input, Output, Forward, Bidirectional, Random Access.

- **Iterator Traits**: `std::iterator_traits` and custom iterators.
- **Range-Based Concepts**: Introduction to C++20 ranges and range adaptors (`std::views`).
- **Practical Examples**: Using iterators and ranges with STL algorithms and containers.

Appendix D: Smart Pointers and Memory Management

While not strictly part of the STL, smart pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`) are essential for modern C++ programming. This appendix covers:

- **Overview of Smart Pointers**: Differences, use cases, and best practices.
- **Custom Deleters**: How to define and use custom deleters with smart pointers.
- **Memory Management Patterns**: RAII (Resource Acquisition Is Initialization) and avoiding memory leaks.
- **Examples**: Practical scenarios demonstrating the use of smart pointers in STL contexts.

Appendix E: Lambda Expressions and Function Objects

This appendix explores the use of lambda expressions and function objects (functors) with STL algorithms and containers. Topics include:

- **Lambda Syntax**: Captures, parameters, and return types.
- **Function Objects**: Creating custom functors and using `std::function`.
- **Comparison with Function Pointers**: Advantages of lambdas and functors.
- **Examples**: Using lambdas with `std::sort`, `std::for_each`, and other algorithms.

Appendix F: STL Allocators

This appendix provides an advanced look at STL allocators, which control memory allocation for containers. Topics include:

- **Default Allocator**: `std::allocator` and its role in STL containers.
- **Custom Allocators**: How to create and use custom allocators.
- **Allocator-Aware Containers**: Understanding how containers interact with allocators.
- **Examples**: Implementing a custom allocator and using it with `std::vector`.

Appendix G: Error Handling and Exceptions in the STL

This appendix discusses error handling mechanisms in the STL, including:

- **Exceptions in STL**: Common exceptions thrown by STL components (e.g., `std::out_of_range`, `std::bad_alloc`).
- **Error Handling Strategies**: Using `noexcept`, `try-catch` blocks, and `std::optional`.
- **Examples**: Handling errors in STL algorithms and containers.

Appendix H: C++20 and Beyond: New STL Features

This appendix highlights the latest features and enhancements in the STL introduced in C++20 and beyond, such as:

- **Ranges Library**: `std::ranges` and range adaptors.

- **Concepts**: Using concepts with STL algorithms and containers.
- **New Algorithms**: `std::shift_left`, `std::shift_right`, `std::midpoint`.
- **Other Features**: `std::span`, `std::format`, and improvements to existing components.

Appendix I: Performance Considerations and Best Practices

This appendix provides guidance on optimizing the performance of STL-based code, including:

- **Choosing the Right Container**: Selecting the most efficient container for specific use cases.
- **Avoiding Common Pitfalls**: Issues like iterator invalidation and unnecessary copies.
- **Benchmarking and Profiling**: Tools and techniques for measuring and improving performance.
- **Examples**: Performance comparisons between different STL components.

Appendix J: Cross-Platform and Compiler-Specific Considerations

This appendix addresses cross-platform development and compiler-specific behaviors related to the STL, including:

- **STL Implementation Differences**: Variations between major standard library implementations (e.g., libstdc++, libc++, MSVC STL).
- **Compiler-Specific Extensions**: Non-standard features and extensions.

- **Portability Tips**: Writing portable STL code across different platforms and compilers.

Appendix K: Further Reading and Resources

This appendix provides a curated list of resources for intermediate learners to continue their journey with the C++ Standard Library, including:

- **Books**: Recommended books on C++ and the STL.
- **Online Resources**: Websites, blogs, and forums for learning and discussion.
- **Tools**: Useful tools for C++ development, such as IDEs, debuggers, and profilers.
- **Community**: Links to C++ communities, conferences, and user groups.

Appendix L: Exercises and Solutions

This appendix includes a set of exercises designed to reinforce the concepts covered in the book. Each exercise is accompanied by a detailed solution, explaining the reasoning and implementation. Topics covered include:

- **Container Usage**: Problems involving `std::vector`, `std::map`, and other containers.
- **Algorithm Implementation**: Writing custom algorithms using STL components.
- **Advanced Topics**: Challenges involving smart pointers, allocators, and C++20 features.

Appendix M: Glossary of STL Terms

This appendix provides a glossary of key terms and concepts related to the STL, including:

- **Definitions**: Clear explanations of terms like iterators, allocators, functors, and ranges.
- **Cross-References**: Links to relevant sections in the book for further reading.

Appendix N: Index of STL Components

This appendix serves as a quick reference index for all STL components covered in the book, including:

- **Containers**: All STL containers and their member functions.
- **Algorithms**: All STL algorithms and their usage.
- **Utilities**: Other STL utilities like `std::pair`, `std::tuple`, and `std::optional`.

References

Official C++ Standards and Documentation

These references provide the authoritative source for understanding the C++ Standard Library and its evolution:

- **ISO C++ Standard Documents**:
  - ISO/IEC 14882:2020 (C++20 Standard): The latest official standard at the time of writing.
  - ISO/IEC 14882:2017 (C++17 Standard): For understanding features introduced in C++17.
  - ISO/IEC 14882:2014 (C++14 Standard): For backward compatibility and historical context.
- **C++ Standard Library Documentation**:
  - cppreference.com: A comprehensive and up-to-date online reference for the C++ Standard Library.
  - Official documentation for major standard library implementations:
    - GNU libstdc++ (GCC): GNU C++ Library Documentation.

    * LLVM libc++ (Clang): libc++ Documentation.
    * Microsoft STL (MSVC): Microsoft STL Documentation.

Books on C++ and the STL

These books provide in-depth coverage of the C++ Standard Library and related topics:

- **The C++ Standard Library: A Tutorial and Reference** by Nicolai M. Josuttis:
  - A comprehensive guide to the STL, covering containers, algorithms, iterators, and utilities.
- **Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library** by Scott Meyers:
  - A practical guide to using the STL effectively, with tips and best practices.
- **C++ Templates: The Complete Guide** by David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor:
  - A deep dive into templates, which are foundational to understanding the STL.
- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup:
  - A beginner-to-intermediate book that introduces C++ and the STL in a practical context.
- **C++17 STL Cookbook** by Jacek Galowicz:
  - A hands-on guide to using C++17 features with the STL.

Online Tutorials and Courses

These resources provide interactive and structured learning experiences:

- **LearnCpp.com**:
  - A free online tutorial for learning C++ from the ground up, including STL topics.
- **CppCon Talks**:
  - Annual C++ conference talks available on YouTube, covering advanced STL topics and modern C++ features.
- **Pluralsight C++ Courses**:
  - Courses like ”C++ Standard Library” and ”Modern C++ Programming” for intermediate learners.
- **Udemy C++ Courses**:
  - Courses such as ”C++: From Beginner to Expert” and ”Mastering C++ Standard Library Features.”

Advanced Topics and Research Papers

For learners interested in the theoretical and advanced aspects of the STL:

- **STL Implementation Details**:
  - Research papers and articles on the design and implementation of the STL in major standard library implementations (e.g., libstdc++, libc++, MSVC STL).

- **C++ Performance Optimization**:
  - Articles and case studies on optimizing STL-based code for performance.
- **Concurrency and Parallelism**:
  - Resources on using the STL with C++11/14/17/20 concurrency features, such as `std::thread`, `std::async`, and `std::execution::par`.

Tools and Utilities

These tools help intermediate learners experiment with and debug STL-based code:

- **Compiler Explorer (godbolt.org)**:
  - An online tool for exploring how STL code is compiled and optimized by different compilers.
- **C++ Insights**:
  - A tool for visualizing how modern C++ features (e.g., lambdas, ranges) are implemented under the hood.
- **Valgrind**:
  - A memory profiling tool for detecting memory leaks and undefined behavior in STL-based programs.
- **Clang-Tidy and Static Analyzers**:
  - Tools for identifying potential issues and improving code quality in STL-heavy projects.

Community and Forums

Engaging with the C++ community is a great way to learn and stay updated:

- **Stack Overflow**:
  - A Q&A platform for asking and answering C++ and STL-related questions.
- **Reddit C++ Community**:
  - Subreddits like r/cpp and r/learnprogramming for discussions and resource sharing.
- **C++ Slack and Discord Channels**:
  - Online communities for real-time discussions and networking with other C++ developers.
- **C++ User Groups and Meetups**:
  - Local and virtual meetups for connecting with other C++ enthusiasts.

Blogs and Articles

These blogs and articles provide insights, tips, and updates on the STL and modern C++:

- **Bartek's Coding Blog**:
  - Articles on modern C++ features and STL usage.
- **Fluent C++**:
  - A blog focused on writing expressive and efficient C++ code using the STL.

- **Herb Sutter's Blog**:
  - Insights from one of the leading figures in the C++ community, covering modern C++ and STL topics.
- **Jason Turner's YouTube Channel (C++ Weekly)**:
  - Short, focused videos on C++ and STL features.

Open-Source Projects and Code Examples

Exploring real-world codebases is an excellent way to learn advanced STL usage:

- **Boost C++ Libraries**:
  - A collection of peer-reviewed, open-source libraries that extend the STL.
- **GitHub Repositories**:
  - Search for open-source projects that heavily use the STL to see practical applications.
- **STL Implementation Code**:
  - Explore the source code of major STL implementations (e.g., libstdc++, libc++, MSVC STL) to understand how they work internally.

Conferences and Workshops

Attending C++ conferences and workshops can provide valuable insights and networking opportunities:

- **CppCon**:
  - The largest annual C++ conference, featuring talks on the STL and modern C++.
- **Meeting C++**:
  - A European-based conference with a focus on C++ and the STL.
- **ACCU Conference**:
  - A conference for C++ developers, covering both beginner and advanced topics.

Historical and Foundational Resources

For learners interested in the history and evolution of the STL:

- **The Design and Evolution of C++** by Bjarne Stroustrup:
  - A book that provides context for the development of C++ and the STL.
- **Original STL Papers**:
  - Research papers by Alexander Stepanov, the creator of the STL, explaining its design principles.

Practice Platforms

These platforms offer coding challenges and exercises to practice STL usage:

- **LeetCode**:
  - Coding problems that can be solved using STL algorithms and containers.
- **HackerRank**:
  - C++ challenges that test your knowledge of the STL.
- **Codewars**:
  - Community-driven coding challenges with a focus on C++ and the STL.

Additional Reading for C++20 and Beyond

For learners looking to explore the latest features in C++20 and beyond:

- **C++20: The Complete Guide** by Nicolai M. Josuttis:
  - A detailed guide to C++20 features, including ranges, concepts, and coroutines.
- **C++ High Performance** by Björn Andrist and Viktor Sehr:
  - A book that covers modern C++ features and their impact on performance.

Style Guides and Best Practices

These resources help learners write clean, maintainable, and efficient STL-based code:

- **Google C++ Style Guide**:
  - A widely used style guide for C++ programming.
- **C++ Core Guidelines**:
  - A set of guidelines maintained by Bjarne Stroustrup and Herb Sutter for writing modern C++ code.

Miscellaneous Resources

Other useful resources for intermediate learners:

- **C++ Weekly News**:
  - A newsletter summarizing the latest developments in the C++ world.
- **C++ Podcasts**:
  - Podcasts like *CppCast* for staying updated on C++ news and trends.