5.4 Standard Template Library (STL) Basics

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

Lecture 44: Standard Template Library (STL) Basics

- What is the STL?
- Core Components: Containers, Algorithms, Iterators
- Deep Dive: std::vector
- Essential Algorithms: std::sort and std::find

Lecture 44: Standard Template Library (STL) Basics
- What is the STL?
- Core Components: Containers, Algorithms, Iterators
- Deep Dive: std::vector
- Essential Algorithms: std::sort and std::find

Welcome, everyone! Today we are stepping into one of the most powerful and widely used aspects of modern C++: The Standard Template Library, or STL. While our unit is titled File Handling and Exception Handling, understanding the STL is crucial because it forms the backbone of efficient data manipulation in C++, which is often read from or written to files. By the end of today, you'll see how much time and effort the STL can save you.

What is the Standard Template Library (STL)?

- A software library for the C++ programming language.
- Provides four components: Algorithms, Containers, Functions, and Iterators.
- Built entirely using C++ Templates, making it type-safe and generic.
- Part of the C++ Standard Library, meaning no extra installations are needed.

What is the Standard Template Library (STL)?
- A software library for the C++ programming language.
- Provides four components: Algorithms, Containers, Functions, and Iterators.
- Built entirely using C++ Templates, making it type-safe and generic.
- Part of the C++ Standard Library, meaning no extra installations are needed.

The STL is a collection of pre-written, highly optimized template classes and functions. Instead of writing your own linked list, dynamic array, or sorting algorithm from scratch, the STL provides these out of the box. Because it's template-based, an STL container can hold any data type—integers, strings, or even your custom user-defined objects, while maintaining strict type safety.

Why Use the STL?

- Reusability: No need to reinvent the wheel for common data structures.
- Performance: STL components are heavily optimized by compiler writers.
- Reliability: Thoroughly tested and standardized code.
- Readability: Standardized vocabulary makes code easier for other C++ developers to read.

Why Use the STL?
- Reusability: No need to reinvent the wheel for common data structures.
- Performance: STL components are heavily optimized by compiler writers.
- Reliability: Thoroughly tested and standardized code.
- Readability: Standardized vocabulary makes code easier for other C++ developers to read.

Why not just write our own arrays and sorts? First, performance. The STL algorithms are usually much faster than hand-rolled ones unless you are an expert in algorithm optimization. Second, reliability. The STL has been tested by millions of developers for decades. Finally, it provides a common language. When you see std::sort, you immediately know what is happening without having to read a custom function's implementation.

The Big Three of the STL

- 1. Containers: Objects that store data (e.g., vector, list, map).
- 2. Algorithms: Functions that perform operations on containers (e.g., sort, search, copy).
- 3. Iterators: Objects that act like pointers to traverse containers.

The Big Three of the STL
- 1. Containers: Objects that store data (e.g., vector, list, map).
- 2. Algorithms: Functions that perform operations on containers (e.g., sort, search, copy).
- 3. Iterators: Objects that act like pointers to traverse containers.

The brilliance of the STL lies in its separation of concerns. Containers just store data. Algorithms just process data. How do they communicate? Through Iterators. An algorithm like std::sort doesn't need to know if it's sorting a vector, a deque, or an array; it only needs iterators that point to the start and end of the data. This modularity is why the STL is so powerful.

Introduction to Containers: std::vector

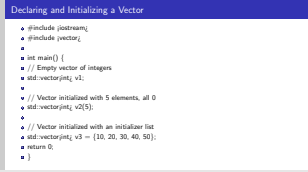
- A sequence container representing an array that can change in size.
- Header: #include ¡vector¿
- Memory is allocated contiguously, just like standard arrays.
- Fast random access (using indices), but slow insertions/deletions in the middle.

Introduction to Containers: std::vector
- A sequence container representing an array that can change in size.
- Header: #include ¡vector¿
- Memory is allocated contiguously, just like standard arrays.
- Fast random access (using indices), but slow insertions/deletions in the middle.

Let's look at our first container: the vector. Think of a vector as a smart array. Traditional C-style arrays have a fixed size. If you make an array of size 10, you can't put 11 items in it. A vector handles dynamic resizing for you automatically. Because it uses contiguous memory, reading from a vector is incredibly fast, but inserting an item in the middle requires shifting everything else, which can be slow.

Declaring and Initializing a Vector

- #include ¡iostream¿
- #include ¡vector¿
-
- int main() {
- // Empty vector of integers
- std::vector¡int¿ v1;
-
- // Vector initialized with 5 elements, all 0
- std::vector¡int¿ v2(5);
-
- // Vector initialized with an initializer list
- std::vector¡int¿ v3 = {10, 20, 30, 40, 50};
- return 0;
- }



Here we see different ways to create a vector. We must include the ¡vector¿ header. The syntax std::vector¡int¿ indicates we are using the vector template specifically for integers. v1 is empty. v2 allocates space for 5 integers and default-initializes them to 0. v3 uses modern C++ uniform initialization to fill the vector with specific values right away.

Adding and Accessing Elements

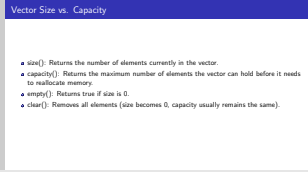
```
std::vector¡int¿ scores;

// Adding elements to the end
scores.push_back(85);
scores.push_back(92);
scores.push_back(78);

// Accessing elements
std::cout ¡¡ scores[0] ¡¡ "\n"; // Outputs 85
std::cout ¡¡ scores.at(1) ¡¡ "\n"; // Outputs 92 (bounds-checked)

// Modifying an element
scores[2] = 80;
```


Adding and Accessing Elements
std::vector¡int¿ scores;
// Adding elements to the end
scores.push_back(85);
scores.push_back(92);
scores.push_back(78);
// Accessing elements
std::cout ¡¡ scores[0] ¡¡ "\n"; // Outputs 85
std::cout ¡¡ scores.at(1) ¡¡ "\n"; // Outputs 92 (bounds-checked)
// Modifying an element
scores[2] = 80;

The push_back() method is how you add items to a vector. It places the new item at the end, expanding the vector's underlying memory if necessary. For accessing, you can use the square brackets just like a normal array. However, using the .at() method is safer because it throws an out_of_range exception if you try to access an index that doesn't exist, whereas the bracket operator results in undefined behavior.

Vector Size vs. Capacity

- size(): Returns the number of elements currently in the vector.
- capacity(): Returns the maximum number of elements the vector can hold before it needs to reallocate memory.
- empty(): Returns true if size is 0.
- clear(): Removes all elements (size becomes 0, capacity usually remains the same).



It's important to understand the difference between size and capacity. When a vector runs out of space, it doesn't just grow by one element; it usually doubles its capacity. This prevents the costly operation of memory reallocation happening on every single push_back. So, a vector might have a size of 3 (containing 3 items) but a capacity of 4 or 8.

Iterating Through a Vector

- std::vector¡std::string¿ names = {"Alice", "Bob", "Charlie"};
-
- // 1. Traditional Index-based Loop
- for (size_t i = 0; i ¡ names.size(); ++i) {
- std::cout ¡¡ names[i] ¡¡ " ";
- }
-
- // 2. Range-based For Loop (C++11 and later) - Preferred!
- for (const std::string& name : names) {
- std::cout ¡¡ name ¡¡ " ";
- }

Iterating Through a Vector
std::vector¡std::string¿ names = {"Alice", "Bob", "Charlie"};
// 1. Traditional Index-based Loop
for (size_t i = 0; i ¡ names.size(); ++i) {
std::cout ¡¡ names[i] ¡¡ " ";
}
// 2. Range-based For Loop (C++11 and later) - Preferred!
for (const std::string& name : names) {
std::cout ¡¡ name ¡¡ " ";
}

How do we loop through a vector? The old way is using a standard for loop with an index variable from 0 to size() - 1. However, modern C++ gives us the range-based for loop. It's much cleaner, less prone to off-by-one errors, and highly readable. Notice I used 'const std::string&' to avoid copying the strings while reading them.

Introduction to STL Algorithms

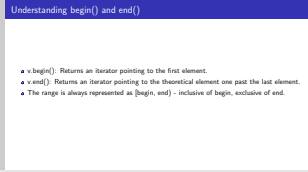
- A collection of functions designed to be used on ranges of elements.
- Header: #include ¡algorithm¿
- They do not work directly on containers, but rather on Iterators.
- Iterators point to the beginning and 'past-the-end' of a sequence.

Introduction to STL Algorithms
- A collection of functions designed to be used on ranges of elements.
- Header: #include ¡algorithm¿
- They do not work directly on containers, but rather on Iterators.
- Iterators point to the beginning and 'past-the-end' of a sequence.

Now let's move to algorithms. To use them, we must include the ¡algorithm¿ header. The genius of the STL is that algorithms don't know what a vector is. They only know about iterators. An iterator is essentially an advanced pointer. All STL algorithms take an iterator pointing to the start of the data, and an iterator pointing to one spot PAST the last element of the data.

Understanding begin() and end()

- v.begin(): Returns an iterator pointing to the first element.
- v.end(): Returns an iterator pointing to the theoretical element one past the last element.
- The range is always represented as [begin, end) - inclusive of begin, exclusive of end.



This slide visualizes the concept of begin and end. Why does end point past the last element? It acts as a sentinel value. When an algorithm is looping through elements using iterators, it knows it is finished when the current iterator is equal to the end() iterator. This [inclusive, exclusive) pattern is universal in C++.

Algorithm: std::sort

- #include ¡vector¿
- #include ¡algorithm¿
- #include ¡iostream¿
-
- int main() {
- std::vector¡int¿ nums = {50, 10, 40, 20, 30};
-
- // Sort in ascending order (default)
- std::sort(nums.begin(), nums.end());
-
- // nums is now: 10, 20, 30, 40, 50
- return 0;
- }

Algorithm: std::sort
- #include ¡vector¿
- #include ¡algorithm¿
- #include ¡iostream¿
-
- int main() {
- std::vector¡int¿ nums = {50, 10, 40, 20, 30};
-
- // Sort in ascending order (default)
- std::sort(nums.begin(), nums.end());
-
- // nums is now: 10, 20, 30, 40, 50
- return 0;
- }

Sorting an array is one of the most common tasks in programming. With the STL, it is a one-liner. std::sort takes the begin and end iterators. By default, it uses the less-than operator (¡) to sort elements in ascending order. Under the hood, this usually implements a highly optimized version of Introsort (a hybrid of Quicksort, Heapsort, and Insertion Sort) with O(N log N) complexity.

Algorithm: std::sort (Custom Ordering)

- // Sorting in descending order using greater¡int¿()
- std::sort(nums.begin(), nums.end(), std::greater¡int¿());
-
- // Or using a custom C++11 Lambda function
- std::sort(nums.begin(), nums.end(), [](int a, int b) {
- return a ¿ b;
- });

Algorithm: std::sort (Custom Ordering)
// Sorting in descending order using greater¡int¿()
std::sort(nums.begin(), nums.end(), std::greater¡int¿());
// Or using a custom C++11 Lambda function
std::sort(nums.begin(), nums.end(), [](int a, int b) {
return a ¿ b;
});

What if you want to sort descending, or sort custom objects based on a specific property? std::sort can take a third argument: a comparator. We can use built-in functors like std::greater, or we can write our own lambda function. A lambda is just an anonymous inline function. In the example, it returns true if 'a' should come before 'b'.

Algorithm: std::find

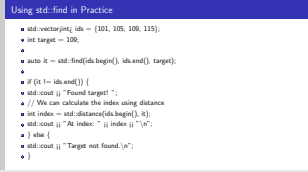
- Searches for a specific value within a given range.
- Syntax: std::find(start_iterator, end_iterator, value_to_find);
- Returns an iterator pointing to the FIRST occurrence of the value.
- Crucial: If the value is NOT found, it returns the end() iterator.

Algorithm: std::find
- Searches for a specific value within a given range.
- Syntax: std::find(start_iterator, end_iterator, value_to_find);
- Returns an iterator pointing to the FIRST occurrence of the value.
- Crucial: If the value is NOT found, it returns the end() iterator.

Next is std::find. This performs a linear search, checking each element one by one. It's incredibly useful, but the trick is understanding what it returns. It doesn't return a boolean, and it doesn't return an integer index. It returns an iterator. If the item isn't in the collection, how does it tell you? By returning the end() iterator, indicating it searched the whole range and hit the boundary.

Using std::find in Practice

- std::vector¡int¿ ids = {101, 105, 109, 115};
- int target = 109;
-
- auto it = std::find(ids.begin(), ids.end(), target);
-
- if (it != ids.end()) {
- std::cout ¡¡ "Found target! ";
- // We can calculate the index using distance
- int index = std::distance(ids.begin(), it);
- std::cout ¡¡ "At index: " ¡¡ index ¡¡ "\n";
- } else {
- std::cout ¡¡ "Target not found.\n";
- }



Here is the standard pattern for using std::find. We capture the returned iterator using the 'auto' keyword (which infers the type automatically). We then immediately check: is 'it' not equal to 'ids.end()'? If true, we found it. We can then use another STL function, std::distance, to calculate the integer index if we actually need it.

Summary of Today's Concepts

- The STL provides standard, tested, and optimized data structures and algorithms.
- std::vector is a dynamic, contiguous array and should be your default container choice.
- Algorithms operate on Iterators (begin to end), keeping them container-agnostic.
- std::sort efficiently orders elements.
- std::find searches for elements, returning the end() iterator on failure.

Summary of Today's Concepts
- The STL provides standard, tested, and optimized data structures and algorithms.
- std::vector is a dynamic, contiguous array and should be your default container choice.
- Algorithms operate on Iterators (begin to end), keeping them container-agnostic.
- std::sort efficiently orders elements.
- std::find searches for elements, returning the end() iterator on failure.

To wrap up, the STL is a massive paradigm shift in C++. Stop writing raw arrays and custom sorting loops. Rely on std::vector, std::sort, and std::find. They are safer, faster, and express your intent much more clearly to anyone reading your code.