5.4 Standard Template Library (STL) Basics

Unit 5: File Handling and Exception Handling

Object Oriented Programming with C++

Lecture 40: Standard Template Library (STL) Basics

- Course: Object Oriented Programming with C++ (DI05011021)
- Unit 5: File Handling and Exception Handling
- Topic 5.4: STL Basics - Vectors and Algorithms

Lecture 40: Standard Template Library (STL) Basics
- Course: Object Oriented Programming with C++ (DI05011021)
- Unit 5: File Handling and Exception Handling
- Topic 5.4: STL Basics - Vectors and Algorithms

Welcome everyone to Lecture 40. Today we are introducing one of the most powerful features of modern C++: The Standard Template Library, or STL. Up until now, we've built our data structures mostly from scratch or relied on raw arrays. Today, we'll see how C++ provides us with heavily optimized, built-in templates to handle collections of data and algorithms effortlessly.

What is the Standard Template Library (STL)?

- A software library for C++ that provides a collection of templates representing standard data structures and algorithms.
- Promotes code reuse, avoiding reinventing the wheel.
- Heavily optimized for performance and memory efficiency.
- Generic Programming: Works with any data type, including built-in types and custom classes.

What is the Standard Template Library (STL)?
- A software library for C++ that provides a collection of templates representing standard data structures and algorithms.
- Promotes code reuse, avoiding reinventing the wheel.
- Heavily optimized for performance and memory efficiency.
- Generic Programming: Works with any data type, including built-in types and custom classes.

The STL was designed with the philosophy of generic programming. This means algorithms are written independently of the data types they operate on, heavily relying on C++ templates. Instead of writing a custom linked list or sorting algorithm every time you start a project, you simply include the STL and use thoroughly tested, highly performant code. It saves time and prevents bugs.

The Three Pillars of STL

- Containers: Objects that store collections of data (e.g., vector, list, map).
- Algorithms: Functions that perform operations on containers (e.g., sort, find, reverse).
- Iterators: Objects that act like pointers, allowing algorithms to traverse containers uniformly.

The Three Pillars of STL
- Containers: Objects that store collections of data (e.g., vector, list, map).
- Algorithms: Functions that perform operations on containers (e.g., sort, find, reverse).
- Iterators: Objects that act like pointers, allowing algorithms to traverse containers uniformly.

To understand STL, you must understand its triad architecture. Containers hold the data. Algorithms process the data. But how do generic algorithms know how to navigate specific containers? That's where Iterators come in. Iterators bridge the gap between containers and algorithms, providing a uniform way to step through elements regardless of the underlying data structure.

Introduction to std::vector

- A dynamic array: Can grow and shrink in size automatically at runtime.
- Requires `#include <vector>`
- Elements are stored in contiguous memory locations.
- Provides constant time, O(1), random access (like standard arrays).

Introduction to std::vector
- A dynamic array: Can grow and shrink in size automatically at runtime.
- Requires #include <vector>
- Elements are stored in contiguous memory locations.
- Provides constant time, O(1), random access (like standard arrays).

Let's dive into our first and most commonly used container: std::vector. Think of a vector as a smarter, safer raw array. While standard C-style arrays have a fixed size defined at compile-time, vectors manage their own memory. As you add elements, the vector will automatically allocate more space if needed. Because memory is contiguous, it is incredibly fast to iterate over and access elements.

Creating and Initializing Vectors

- std::vector¡int¿ v1; // Empty vector of integers
- std::vector¡int¿ v2 = {1, 2, 3, 4, 5}; // Uniform initialization
- std::vector¡std::string¿ v3(10, "Hello"); // 10 elements, all initialized to 'Hello'

Creating and Initializing Vectors
- std::vector¡int¿ v1; // Empty vector of integers
- std::vector¡int¿ v2 = {1, 2, 3, 4, 5}; // Uniform initialization
- std::vector¡std::string¿ v3(10, "Hello"); // 10 elements, all initialized to 'Hello'

Creating a vector is straightforward thanks to templates. We specify the type of data we want to store inside the angle brackets. Notice the flexibility here. We can create an empty vector, use an initializer list to populate it immediately, or specify a size and a default value. This flexibility makes vector setup very clean.

Adding and Removing Elements

- push_back(value): Adds an element to the end of the vector. The vector grows automatically.
- pop_back(): Removes the last element from the vector.
- clear(): Removes all elements from the vector.
- insert() and erase(): Can add/remove from specific positions (slower, O(N)).

Adding and Removing Elements
- push_back(value): Adds an element to the end of the vector. The vector grows automatically.
- pop_back(): Removes the last element from the vector.
- clear(): Removes all elements from the vector.
- insert() and erase(): Can add/remove from specific positions (slower, O(N)).

The most common way to populate a vector dynamically is using push_back(). It appends the element to the rear of the vector. If the vector runs out of allocated space, it handles the reallocation transparently. pop_back() removes the trailing element. Note that operations at the very end of a vector are fast (O(1)), but inserting or deleting from the middle requires shifting elements, making it an O(N) operation.

Accessing Vector Elements

- operator[index]: Accesses element at index without bounds checking (faster but unsafe).
- at(index): Accesses element with bounds checking; throws std::out_of_range if invalid.
- front(): Returns a reference to the first element.
- back(): Returns a reference to the last element.

Accessing Vector Elements
- operator[index]: Accesses element at index without bounds checking (faster but unsafe).
- at(index): Accesses element with bounds checking; throws std::out_of_range if invalid.
- front(): Returns a reference to the first element.
- back(): Returns a reference to the last element.

To read or modify data in a vector, we have several options. Using the square brackets acts exactly like a traditional array, which means if you go out of bounds, you get undefined behavior. For safer code, especially when dealing with user input, use the at() function. It performs bounds checking and throws an exception, fitting perfectly into our Unit 5 theme of exception handling!

Size vs. Capacity

- size(): Returns the number of elements currently stored in the vector.
- capacity(): Returns the total amount of memory currently allocated (how many elements it can hold before reallocating).
- empty(): Returns true if size is 0.

Size vs. Capacity
- size(): Returns the number of elements currently stored in the vector.
- capacity(): Returns the total amount of memory currently allocated (how many elements it can hold before reallocating).
- empty(): Returns true if size is 0.

A common point of confusion is the difference between size and capacity. Size is the logical number of items you've put into the vector. Capacity is the physical memory reserved. When size exceeds capacity, the vector usually doubles its capacity by allocating a new block of memory, copying the old elements over, and deleting the old block. This is why capacity is usually greater than or equal to size.

Iterating Over Vectors

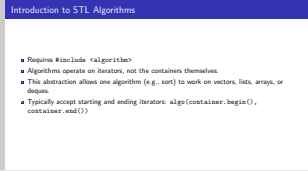
- // Range-based for loop (C++11 and later)
- for (int val : myVector) {
- cout ¡¡ val ¡¡ " ";
- }
-
- // Traditional indexed loop
- for (size_t i = 0; i ¡ myVector.size(); i++) {
- cout ¡¡ myVector[i] ¡¡ " ";
- }

Iterating Over Vectors
- // Range-based for loop (C++11 and later)
- for (int val : myVector) {
- cout ¡¡ val ¡¡ " ";
- }
-
- // Traditional indexed loop
- for (size_t i = 0; i ¡ myVector.size(); i++) {
- cout ¡¡ myVector[i] ¡¡ " ";
- }

Since C++11, the most elegant way to traverse a vector is the range-based for loop. It automatically deduces the start and end of the container. If you need to modify the elements during traversal, you would use a reference, like 'int& val'. The traditional index-based for loop is still perfectly valid and is useful if you specifically need the index number during iteration.

Introduction to STL Algorithms

- Requires `#include <algorithm>`
- Algorithms operate on iterators, not the containers themselves.
- This abstraction allows one algorithm (e.g., sort) to work on vectors, lists, arrays, or deques.
- Typically accept starting and ending iterators: `algo(container.begin(), container.end())`



Now we move to the second pillar: Algorithms. The brilliant design of STL is that algorithms don't belong to the vector class. They are free-standing functions in the ¡algorithm¿ header. By accepting iterators (specifically container.begin() and container.end()), these algorithms can manipulate almost any container in the STL. It's the ultimate expression of generic programming.

The std::sort Algorithm

- #include ¡algorithm¿
- #include ¡vector¿
- // ...
- std::vector¡int¿ v = {5, 2, 9, 1, 5, 6};
- std::sort(v.begin(), v.end());
- // Vector is now: 1, 2, 5, 5, 6, 9

The std::sort Algorithm
- #include ¡algorithm¿
- #include ¡vector¿
- // ...
- std::vector¡int¿ v = {5, 2, 9, 1, 5, 6};
- std::sort(v.begin(), v.end());
- // Vector is now: 1, 2, 5, 5, 6, 9

Sorting is arguably the most common algorithmic task. Instead of writing bubble sort or quicksort, just call std::sort. Under the hood, std::sort uses an introsort (a hybrid of quicksort, heapsort, and insertion sort), guaranteeing an excellent O(N log N) time complexity. Notice how we pass v.begin() and v.end() to specify the range we want sorted.

std::sort with Custom Comparators

- // Sort in descending order
- std::sort(v.begin(), v.end(), std::greater¡int¿());
-
- // Using a custom lambda function for objects
- std::sort(students.begin(), students.end(),
  (const Student& a, const Student& b) {
- return a.grade ¿ b.grade;
- });

std::sort with Custom Comparators
// Sort in descending order
std::sort(v.begin(), v.end(), std::greater¡int¿());
// Using a custom lambda function for objects
std::sort(students.begin(), students.end(),
(const Student& a, const Student& b) {
return a.grade ¿ b.grade;
});

By default, std::sort uses the less-than operator to sort in ascending order. But what if we want descending order, or we are sorting a vector of custom objects, like Students? We can provide a third argument to std::sort: a comparator. This can be a standard functor like std::greater, a custom function, or a modern C++ lambda expression as shown here. This allows absolute control over the sorting logic.

The std::find Algorithm

- std::vector¡int¿ v = {10, 20, 30, 40};
- auto it = std::find(v.begin(), v.end(), 30);
-
- if (it != v.end()) {
- cout ¡¡ "Found at index: " ¡¡ std::distance(v.begin(), it);
- } else {
- cout ¡¡ "Not found";
- }

The std::find Algorithm
- std::vector¡int¿ v = {10, 20, 30, 40};
- auto it = std::find(v.begin(), v.end(), 30);
-
- if (it != v.end()) {
- cout ¡¡ "Found at index: " ¡¡ std::distance(v.begin(), it);
- } else {
- cout ¡¡ "Not found";
- }

The std::find algorithm performs a linear search (O(N) time) for a specific value. It returns an iterator pointing to the first occurrence of the element. How do we know if it failed? If the element isn't found, find() returns the 'end' iterator (v.end()). We can also use std::distance to calculate the numerical index from the iterator. It is simple, elegant, and bug-free.

Why Use STL Over Raw Arrays/Pointers?

- Memory Management: No need for `new` and `delete`. Prevents memory leaks.
- Safety: Methods like `.at()` prevent buffer overflows, a massive security risk in C.
- Performance: Algorithms are rigorously tested and optimized by compiler engineers.
- Readability: `std::sort` makes intent clearer than a 20-line custom sorting function.

Why Use STL Over Raw Arrays/Pointers?
- Memory Management: No need for new and delete. Prevents memory leaks.
- Safety: Methods like .at() prevent buffer overflows, a massive security risk in C.
- Performance: Algorithms are rigorously tested and optimized by compiler engineers.
- Readability: std::sort makes intent clearer than a 20-line custom sorting function.

As we conclude our look at vectors and basic algorithms, let's reflect on why we use them. In modern C++, raw arrays and manual pointer memory management are heavily discouraged unless strictly necessary (like in embedded systems or core driver development). The STL provides safety, drastically reduces development time, prevents memory leaks through RAII, and usually performs just as fast, if not faster, than handwritten code.

Summary

- STL comprises Containers, Algorithms, and Iterators.
- `std::vector` is your default choice for a dynamic, resizable array.
- Use `<algorithm>` for operations like sorting and searching.
- Iterators provide the glue between containers and algorithms.

Summary
- STL comprises Containers, Algorithms, and Iterators.
- std::vector is your default choice for a dynamic, resizable array.
- Use <algorithm> for operations like sorting and searching.
- Iterators provide the glue between containers and algorithms.

To summarize, you now have a foundational understanding of the Standard Template Library. We explored vectors as dynamic arrays, learned how to manipulate their contents, and applied standard algorithms like sort and find using iterators. Mastering the STL is a significant milestone in transitioning from a C++ beginner to a proficient C++ developer.