Programming Techniques for Scientific Simulations I

Week 7: Algorithms, Data Structures, and Plotting

Detailed Lecture Notes

October 30, 2025

Contents

Part I
Algorithms and Data Structures in C++ (week07a)

1 Complexity Analysis: Measuring Efficiency

1.1 The Core Question

(Slide 2)

When we write a program, it's not enough for it to be *correct*. It must also be *efficient*. "Efficiency" can mean many things (memory usage, disk I/O), but most often it means **time**. We want our algorithms to be fast.

The core question of **complexity analysis** is:

> *"How does the time needed for an algorithm scale with the problem size N?"*

Here, N is the "problem size." If you're sorting a list, N is the number of items. If you're searching a database, N is the number of entries. We don't measure time in seconds, because that changes when you buy a faster computer. We measure time in the *number of operations* as a *function* of N.

Real-World Analogy: A Recipe Imagine N is the number of guests coming to dinner.

- A $O(1)$ (constant) algorithm: The recipe takes 30 minutes, whether you have 1 guest or 100. (e.g., "Pre-heat the oven").
- A $O(N)$ (linear) algorithm: The recipe takes 5 minutes per guest. (e.g., "Chop 1 potato per guest"). Double the guests, double the time.
- A $O(N^2)$ (quadratic) algorithm: You must introduce every guest to every other guest. Double the guests, and you quadruple the introduction time.

We analyze this scaling in a few different ways:

- **Worst Case Analysis:** This is the most common. It answers, "What is the *maximum* possible time your algorithm will take for a given N?" This is a guarantee. (Analogy: "How long to find a name in the phone book?" Worst case: It's the very last name.)
- **Best Case Analysis:** What is the *minimum* possible time? (Analogy: The name is the very first one.)
- **Average Case Analysis:** What is the *typical* time? This is often more useful but is much harder to calculate.
- **Amortized Analysis:** What is the average time over a *sequence* of many operations? We will see this is very important for 'std::vector'.

1.2 Asymptotic Notation: The Language of Complexity

(Slide 3)

We use a special mathematical notation to describe complexity. This notation ignores constant factors and lower-order terms, focusing only on the *dominant term* that dictates the scaling behavior as N becomes very large.

1.2.1 O / Big-Oh (Worst Case Upper Bound)

This is the most common notation. It describes the **upper bound** or the **worst-case** scenario.

- **Definition:** An algorithm is $O(f(N))$ if its runtime $t(N)$ is *always less than* some constant c times $f(N)$ (for a large enough N).
- **Analogy:** Think of it as a **speed limit for slowness**. A $O(N^2)$ algorithm is *guaranteed* to be no worse than quadratic. It might be faster (it could be $O(N)$), but it will never be $O(N^3)$.
- **Consequences:**
  1. **We ignore constants.** $O(2N)$ and $O(1000N)$ are both just $O(N)$. Why? Because as N goes to infinity, the *linear* nature is what matters, not the slope.
  2. **We keep the dominant term.** An algorithm that takes $N^2 + 50N + 1000$ steps is just $O(N^2)$.
- **Analogy for Dominant Terms:** Imagine you are building a skyscraper (N floors) and also painting the lobby. The time to build the N floors scales quadratically (N^2). The time to paint the lobby is constant. As N (the number of floors) gets huge, the lobby-painting time becomes completely irrelevant to the total project time. The N^2 term "dominates" everything else.

1.2.2 Ω / Big-Omega (Best Case Lower Bound)

This describes the **lower bound** or the **best-case** scenario.

- **Definition:** An algorithm is $\Omega(f(N))$ if its runtime $t(N)$ is *always greater than* some constant c times $f(N)$.
- **Analogy:** This is a **guarantee of work**. It says, "No matter how lucky you get, this algorithm will *never be faster than* $f(N)$."

1.2.3 Θ / Big-Theta (Tight Bound)

This is used when the worst case and best case are the same.

- **Definition:** An algorithm is $\Theta(f(N))$ if it is *both* $O(f(N))$ and $\Omega(f(N))$.
- **Analogy:** This is an **exact price**. The algorithm will *always* take this amount of time to scale, no matter what the input.

1.3 The Real-World Impact of Complexity

(Slides 4, 5, 6)

These notations aren't just academic. They have profound, practical consequences. Let's assume a computer can perform 1 billion operations per second (1 G-op/s).

Table 1: Time taken assuming 1 G-op/s (Slide 4)

Complexity	**N=10**	**N=100**	**N=1,000**	**N=1,000,000**	**N=1,000,000,000**
$O(1)$	1 ns	1 ns	1 ns	1 ns	1 ns
$O(\ln N)$	3 ns	7 ns	10 ns	20 ns	30 ns
$O(N)$	10 ns	100 ns	1 μs	1 ms	1 s
$O(N \ln N)$	30 ns	700 ns	10 μs	20 ms	30 s
$O(N^2)$	100 ns	10 μs	1 ms	**17 min**	**31.7 years**
$O(N^3)$	1 μs	1 ms	1 s	**31.7 years**	3×10^{10} years
$O(2^N)$	1 μs	10^{14} yrs	10^{285} yrs	—	—

The lesson from Table 1 is clear:

- $O(1)$, $O(\ln N)$, $O(N)$, and $O(N \ln N)$ are all exceptionally fast and scalable. We call these "efficient" algorithms.
- $O(N^2)$ (quadratic) becomes unusable very quickly. An N of one million, which is not large for modern datasets, would take 17 minutes.
- $O(N^3)$ (cubic) and $O(2^N)$ (exponential) are "intractable." They are completely unusable for anything other than trivially small N.

But what if we buy a faster computer? Slide 5 shows the same table for a 10 Peta-op/s supercomputer (10,000 times faster).

- The $O(N^3)$ algorithm for $N = 1,000,000$ drops from 31.7 years to 100 seconds. This is a huge improvement!
- **But this is a trap.** A 10,000x faster computer only lets you solve a problem $\sqrt[3]{10000} \approx 21.5$ times larger in the same amount of time.
- For $N = 1,000,000,000$, the $O(N^3)$ algorithm *still* takes 3000 years.

The fundamental lesson of complexity is that a good algorithm on a slow computer will always beat a bad algorithm on a fast computer for a large enough N. You cannot fix a bad algorithm with better hardware.

When choosing (Slide 6), you *always* prefer the lower complexity class. $O(1000 \ln N)$ is still $O(\ln N)$, and it will *always* be faster than $O(N)$ once N is large enough.

1.4 Complexity Examples in C++

(Slides 7, 8, 9)

Let's analyze some simple C++ code blocks.

1.4.1 Example 1: Single Loop (Slide 7)

```cpp
int n = 1000;
for (int i = 0; i < n; ++i) {
    std::cout << i*i << std::endl;
}
```

Listing 1: A simple linear-time loop.

- The work inside the loop (multiplication, output) is constant time. We call this $O(1)$.
- The loop executes exactly n times.
- Total time = $n \times O(1) = O(n)$.
- Since the best case and worst case are identical (it *always* runs n times), we can say this is $\Theta(n)$.

1.4.2 Example 2: Nested Loop (Slide 8)

```cpp
int n = 1000;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        std::cout << i*j << std::endl;
    }
}
```

Listing 2: A nested quadratic-time loop.

- The outer loop runs n times (for $i = 0, 1, ..., n-1$).
- The inner loop runs i times.
- The total number of operations is the sum of $0 + 1 + 2 + 3 + ... + (n-1)$.
- This is a famous arithmetic series, which sums to $\frac{n(n-1)}{2} = \frac{n^2-n}{2}$.
- Using our notation rules, we drop the constants ($\frac{1}{2}$) and the lower-order term ($-n/2$).
- The complexity is $O(n^2)$.
- Since it always does this, it is $\Theta(n^2)$.

1.4.3 Example 3: Search with ‘break‘ (Slide 9)

```
// Block 1: Allocation
int n = 1000;
double* x = new double[n]; // O(n) operation
for (int i=0; i < n; ++i)  // O(n) operation
    x[i] = i;

// Block 2: Search
int pos = -1;
for (int i=0; i < n; ++i) {
    if (x[i] == y) {
        pos = i;
        break; // This is the important line!
    }
}
```

Listing 3: A linear search with different cases.

- **Block 1:** Allocating n elements is $O(n)$. The loop to fill them is $O(n)$. Total time is $O(n) + O(n)$, which is still just $O(n)$.
- **Block 2:** The ‘break‘ statement changes everything.
  - **Worst Case:** The value ‘y‘ is the last element ($x[n-1]$) or is not in the array at all. The loop runs n times. The complexity is $O(n)$.
  - **Best Case:** The value ‘y‘ is the very first element ($x[0]$). The loop runs once, hits the ‘break‘, and exits. The complexity is $O(1)$.
- **Conclusion:** Because the best and worst cases are different, we cannot use Θ. We say the algorithm has a worst-case of $O(n)$ and a best-case of $\Omega(1)$.

1.5 Amortized Analysis: The "Clever Way" to Grow Arrays

(Slides 10, 11)

The "Simple Way" (Slide 10) Imagine you have a fixed-size array of N elements, and you want to add one more. You must:

1. Allocate a *new* array of size $N + 1$.
2. *Copy* all N elements from the old array to the new one.
3. Add the new $(N + 1)^{th}$ element.

The copy step takes $O(N)$ time. If you do this for *every single element* you add, adding N elements one by one will take $O(1) + O(2) + ... + O(N) = O(N^2)$ time. This is horribly slow.

The "Clever Way" (Amortized $O(1)$) (Slide 11) This is the strategy used by C++'s 'std::vector'.

1. When you ask for an array, it secretly allocates *extra space*. This is its **capacity**. The number of elements you are using is its **size**.
2. As you add elements, it just increments the 'size'. This is a $O(1)$ operation.
3. When you try to add an element and 'size == capacity', the vector is full. It now performs an "expensive" operation:
4. It allocates a *new*, much larger array, typically **double the size** ($2N$).
5. It copies the N old elements. This one operation is $O(N)$.
6. It adds the new element.

This single $O(N)$ operation seems bad, but it just bought you N more "cheap" $O(1)$ additions.

Amortized Analogy: You have a 10-person dining table.

- **Simple Way ($O(N)$):** When guest #11 arrives, you move everyone to an 11-person table. When guest #12 arrives, you move everyone to a 12-person table. This is a nightmare.
- **Clever Way (Amortized $O(1)$):** When guest #11 arrives, you move all 10 people to a 20-person table. This one move is expensive ($O(N)$). But now, guests #12 through #20 can just sit down instantly ($O(1)$).

The total cost of N additions is $O(N)$ (for the one expensive copy) plus $N \times O(1)$ (for the cheap additions). The total cost is $O(N)$. The *average* or *amortized* cost per operation is $O(N)/N = \mathbf{O(1)}$.

2 The C++ Standard Template Library (STL)

2.1 What is the STL?

(Slides 12, 13)

The **Standard Template Library (STL)** is a powerful, efficient, and well-tested library of code built into C++. Its key idea is **generic programming**: writing code that works with any data type.

The STL is built on three core pillars:

1. **Containers:** Data structures that store your data.
2. **Algorithms:** Functions that process your data.
3. **Iterators:** The "glue" that connects algorithms to containers.

Analogy:

- **Containers** are your filing cabinets, phone books, and to-do lists.

- **Algorithms** are the *actions* you take: ‘sort()‘ the files, ‘find()‘ a phone number, ‘reverse()‘ the to-do list.
- **Iterators** are a "generic hand" that knows how to point to an item and move to the next one, regardless of whether it’s in a filing cabinet or a phone book.

This design is brilliant: the ‘std::sort‘ algorithm doesn’t know what a ‘std::vector‘ is. It only knows how to use iterators. This means ‘std::sort‘ was written *once* and works on almost any container.

2.2 Simple Utilities

(Slides 14, 15)

‘std::string‘ (Slide 14) Found in ‘<string>‘, this is the C++ class for handling text. ‘std::wstring‘ is for "wide" characters (e.g., for non-English alphabets). It’s a container-like class with many useful member functions.

‘std::pair‘ (Slide 15) Found in ‘<utility>‘, ‘std::pair‘ is a simple template class that just holds two values, which can be of different types.

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

int main() {
    // Create a pair of a string and an integer
    std::pair<std::string, int> student("Alice", 20);

    // Access the elements using .first and .second
    std::cout << "Name: " << student.first << std::endl;
    std::cout << "Age: " << student.second << std::endl;

    // You can also use std::make_pair
    auto student2 = std::make_pair("Bob", 22);
}
```

Listing 4: Using std::pair

This is very useful for functions that need to return two values, or as the element type for ‘std::map‘. For more than two items, C++ offers ‘std::tuple‘.

3 STL Container Deep Dive

(Slide 16)

The STL provides many containers, which we can group into categories. We will explore the most important ones.

3.1 Common Container Operations

All STL containers share a common interface for basic operations:

```cpp
// Works for vector, list, deque, set, map, etc.
std::vector<int> container;

// Size and capacity
size_t sz = container.size();      // Number of elements
bool empty = container.empty();    // Returns true if size() == 0
size_t max_sz = container.max_size(); // Maximum possible size

// Iterators (every container provides these!)
auto it_begin = container.begin();   // Iterator to first element
auto it_end = container.end();       // Iterator past last element
auto cit_begin = container.cbegin(); // Const iterator to first
auto cit_end = container.cend();     // Const iterator past last

// Clear all elements
container.clear();  // Removes all elements, size() becomes 0

// Swap contents with another container (very fast - just swaps
    pointers)
std::vector<int> other;
container.swap(other);  // Or: std::swap(container, other);
```

Listing 5: Universal Container Operations

3.2 Sequence Containers

These containers store elements in a specific linear order that you define.

3.2.1 ‘std::vector‘ — The "Smart Array"

(Slides 17, 18, 19, 44, 47)

This is the most common and useful container. It is the C++ "smart array" and should be your default choice. Include with ‘<vector>‘.

- **Internal Structure:** A single, contiguous block of memory (Slide 17).
- **Pros:**
  - $O(1)$ **Random Access:** `v[i]` is just a pointer calculation.
  - **Excellent Cache Locality:** Because all data is side-by-side, the CPU can pre-load elements into its ultra-fast cache, making loops very fast.
  - **Amortized** $O(1)$ **push_back():** Adding to the end is (on average) $O(1)$ (Slide 19).
- **Cons:**
  - $O(N)$ **Insert/Erase:** Adding or removing an element in the *middle* is very slow (Slide 18). You have to "shift" all elements after that point, which is an $O(N)$ copy operation.

```cpp
#include <vector>

// === Construction ===
std::vector<int> v1;                      // Empty vector
std::vector<int> v2(5);                   // 5 elements, default value (0)
std::vector<int> v3(5, 42);               // 5 elements, all set to 42
std::vector<int> v4 = {10, 20, 30};      // Initialize with list
std::vector<int> v5(v4);                  // Copy constructor
std::vector<int> v6(v4.begin(), v4.begin()+2); // From range [10,20]
```

Listing 6: std::vector Construction

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

// === Adding Elements ===
v.push_back(40);         // Add to end: [10,20,30,40] - O(1) amortized
v.insert(v.begin(), 5); // Insert at beginning: [5,10,20,30,40] - O(N)
v.insert(v.begin()+2, 15); // Insert at position 2: [5,10,15,20,30,40] - O(N)
v.emplace_back(50);      // Construct in-place (more efficient) - O(1) amortized

// === Removing Elements ===
v.pop_back();            // Remove last: [5,10,15,20,30] - O(1)
v.erase(v.begin());     // Remove first: [10,15,20,30] - O(N)
v.erase(v.begin()+1);   // Remove at position 1: [10,20,30] - O(N)
v.erase(v.begin(), v.begin()+2); // Remove range: [30] - O(N)
v.clear();               // Remove all elements - O(N)
```

Listing 7: std::vector Adding and Removing Elements

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

int first = v.front();      // 10 - First element
int last = v.back();        // 40 - Last element
int val = v[1];             // 20 - FAST but UNSAFE (no bounds check)
int val_safe = v.at(1);     // 20 - SAFE (throws exception if out of bounds)

// Using iterators
int first_it = *v.begin();  // 10 - First element via iterator
int last_it = *(v.end()-1); // 40 - Last element via iterator
```

Listing 8: std::vector Element Access

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

size_t s = v.size();        // 4 - Number of elements currently stored
size_t c = v.capacity();    // >= 4 - Space allocated (may be larger)
bool empty = v.empty();     // false - Equivalent to size() == 0

// Pre-allocate space (optimization!)
v.reserve(100);  // Guarantee capacity >= 100 (no reallocation until then)
v.shrink_to_fit(); // Request to reduce capacity to fit size

// Change size
```

```
v.resize(6);      // Size becomes 6, new elements are 0:
    [10,20,30,40,0,0]
v.resize(3);      // Size becomes 3: [10,20,30] (elements removed)
v.resize(5, 99);  // Size becomes 5, new elements are 99:
    [10,20,30,99,99]
```

Listing 9: std::vector Size and Capacity Management

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

// Get iterators
auto it_begin = v.begin();     // Points to first element (10)
auto it_end = v.end();         // Points PAST last element (invalid to
    dereference!)
auto rit_begin = v.rbegin();   // Reverse: points to last (50)
auto rit_end = v.rend();       // Reverse: points before first

// Iterate and modify
for (auto it = v.begin(); it != v.end(); ++it) {
    *it *= 2;  // Double each element
}
// v is now [20, 40, 60, 80, 100]

// Const iteration (read-only)
for (auto cit = v.cbegin(); cit != v.cend(); ++cit) {
    std::cout << *cit << " ";
    // *cit = 0;  // ERROR: Cannot modify through const iterator
}

// Modern range-based for loop (preferred when you don't need iterator
     itself)
for (auto& elem : v) {       // Reference to modify
    elem += 10;
}
for (const auto& elem : v) { // Const reference for read-only
    std::cout << elem << " ";
}
```

Listing 10: std::vector Iterator Operations

3.2.2 `std::deque` — The "Double-Ended Queue"

(Slide 20)

`deque` (pronounced "deck") is very similar to `vector` but with one extra power: it supports fast insertion/removal at *both the front and the back*. Include with `<deque>`.

- **Key Feature:** $O(1)$ `push_front()`, `pop_front()`, `push_back()`, `pop_back()`.
- **Analogy:** A line of people where you can add/remove from both the front and the back of the line instantly.

- **Internals:** More complex than `vector` (often a list of small arrays). It still provides $O(1)$ random access `d[i]`, but it's slightly slower than `vector`'s access due to bad cache locality.
- **Use Case:** Use `deque` if you *know* you need to add/remove from the front. Otherwise, `vector` is usually faster.

3.2.3 `std::list` — The "Linked List"

(Slides 24, 49)

`list` is a **doubly-linked list**. It is fundamentally different from `vector`. Include with `<list>`.

- **Internal Structure:** A chain of "nodes." Each node contains a value and two pointers: one to the 'next' node and one to the 'previous' node.
- **Pros:**
  - $O(1)$ **Insert/Erase:** If you have an iterator pointing to a location, you can insert or remove an element in $O(1)$ time. You just re-wire the pointers of the neighbors. No element shifting is needed.
- **Cons:**
  - $O(N)$ **Access:** You *cannot* do `l[i]`. To get the 100th element, you *must* start at the beginning and follow the `next` pointer 100 times.
  - **Bad Cache Locality:** The nodes can be scattered all over memory, which is slow for the CPU's cache.

Analogy:

- `vector`: A row of mailboxes. Fast to jump to mailbox #100 ($O(1)$), but slow to add a new mailbox in the middle ($O(N)$).
- `list`: A treasure hunt. Fast to add a new clue in the middle ($O(1)$), but to find clue #100, you must follow the first 99 clues ($O(N)$).

Because `list` is so different, it has its own special member functions (Slide 49) that are much faster than the generic algorithms:

- `l.sort()`: A special $O(N \ln N)$ sort that just re-wires pointers, never copying values.
- `l.splice(it, other_list)`: Moves all nodes from `other_list` into `l` at position `it`. This is a $O(1)$ operation!
- `l.remove(value)`: Removes all elements equal to `value`.
- `l.remove_if(predicate)`: Removes all elements for which the predicate function returns `true`. (This is key for the Penna model!)

3.3 Container Adapters

(Slides 21, 22, 23, 48)

Adapters are not new containers. They are "wrappers" that provide a simpler, more restrictive interface on top of an existing container (usually ‘std::deque‘ by default).

Analogy: A Pez dispenser. It’s just a wrapper around a stack of candy, but it *restricts* you. You can only ‘push()‘ (load) from one end and ‘pop()‘ (eat) from the other. This restriction is a *feature*, as it enforces a specific access pattern.

3.3.1 `std::stack`

(Slide 21) Implements a **LIFO (Last-In, First-Out)** structure. Include with `<stack>`.

- **Analogy:** A stack of plates. You put a plate on top, you take a plate from the top.
- **Operations ($O(1)$):**
  - `s.push(value)`: Add an element to the top.
  - `s.pop()`: Remove the top element.
  - `s.top()`: Get a reference to the top element.

3.3.2 `std::queue`

(Slide 22) Implements a **FIFO (First-In, First-Out)** structure. Include with `<queue>`.

- **Analogy:** A checkout line (or "queue") at a store.
- **Operations ($O(1)$):**
  - `q.push(value)`: Add an element to the *back*.
  - `q.pop()`: Remove the element from the *front*.
  - `q.front()`: Get a reference to the front element.
  - `q.back()`: Get a reference to the back element.

3.3.3 `std::priority_queue`

(Slide 23) A special queue where elements are removed based on priority, not arrival time. Include with `<queue>`.

- **Analogy:** An Emergency Room waiting line. Patients are seen by severity (priority), not by who arrived first.
- By default, "priority" means "largest value."
- **Operations:**
  - `pq.push(value)`: Adds an element, sorting it into the queue. ($O(\log N)$)
  - `pq.pop()`: Removes the *highest priority* element. ($O(\log N)$)
  - `pq.top()`: Get a reference to the highest priority element. ($O(1)$)

3.4 Associative Containers (Trees)

(Slides 25, 26, 27, 28, 50)

We have a problem:

- `vector`: Fast access ($O(1)$), but slow search ($O(N)$ in an unsorted vector).
- `list`: Fast insert/erase ($O(1)$), but slow access and search ($O(N)$).

What if we need **fast search, fast insert, AND fast erase**?

Solution: A **Balanced Binary Search Tree (BST)**.

- **Structure (Slide 26, 27):** A tree made of nodes. Each node has a value, a pointer to a 'left' child (with a smaller value) and a 'right' child (with a larger value).
- **Performance:** The tree is automatically "balanced" to keep it bushy, not stringy. This guarantees that the height of the tree is $\log N$.
- This means **search, insert, and erase are all** $O(\log N)$.
- $O(\log N)$ (logarithmic) is *extremely* fast (see Table 1).

Analogy: The game of "20 Questions." You start at the root and ask a "smaller or larger?" question at each node, dividing the remaining search space in half each time.

The STL provides two main tree-based containers:

3.4.1 `std::set`

Include with `<set>`.

- **What it is:** Stores a collection of **unique, sorted** keys.
- **Analogy:** A VIP guest list. It's sorted alphabetically, and you can't be on the list twice.

- **Use Case:** When you just need to know if an item *exists* in a set, and you need to do it quickly.
- **Syntax:**

```
#include <set>
std::set<std::string> banned_users;
banned_users.insert("Alice");
banned_users.insert("Bob");
banned_users.insert("Alice"); // This does nothing, "Alice" is already in.

// Fast O(log N) lookup
if (banned_users.count("Bob") > 0) {
    // ...
}

```

Listing 11: Using std::set

3.4.2 `std::map`

Include with `<map>`.

- **What it is:** Stores a collection of **unique, sorted key-value pairs**.
- **Analogy:** A dictionary or a phone book. The "key" is the word (e.g., "algorithm"), and the "value" is the definition. The keys are sorted.
- **Use Case:** Associating one piece of data with another.
- **Syntax:**

```
#include <map>
#include <string>

std::map<std::string, int> student_ages;

// Insert using [] operator (O(log N))
student_ages["Charlie"] = 21;
student_ages["David"] = 19;

// Insert using .insert()
student_ages.insert(std::make_pair("Eve", 23));

// Fast O(log N) lookup
std::cout << "David's age: " << student_ages["David"] << std::endl;

```

Listing 12: Using std::map

Note: `multiset` and `multimap` also exist, which allow duplicate keys.

4 The Magic Glue: Iterators

4.1 The $N \times M$ Problem

(Slides 30-34)

We have a problem. We have M containers (`vector`, `list`, `deque`...) and N algorithms (`find`, `copy`, `sort`...).

- To loop through a `vector`, you use a pointer: `for (T* p = ...)`
- To loop through a `list`, you use a node: `for (node* p = ...)`

The code is different! (Slide 30). Does this mean we have to write $N \times M$ different functions (e.g., `find_in_vector`, `find_in_list`, `sort_vector`, `sort_list`)? This would be a nightmare (Slides 31-34).

4.2 The Solution: Generic Traversal

(Slides 35, 36)

The answer is **NO**. The STL solves this with **Iterators**. An iterator is an object that acts like a "generic pointer." It abstracts away the details of the container.

Every container provides two functions:

- `container.begin()`: Returns an iterator to the *first* element.
- `container.end()`: Returns an iterator *past the last* element.

All iterators, no matter what container they come from, support common operations:

- `++it`: Move to the next element.
- `*it`: Get the value of the element (`dereference`).
- `it1 == it2`: Compare two iterators.

Now, we can write **one** generic loop that works on **any** container:

```
// This code works if 'c' is a vector, a list, a deque, or a set!
for (auto it = c.begin(); it != c.end(); ++it) {
    auto value = *it;
    // ... do something with value ...
}
```

Listing 13: The generic iterator loop (Slide 35)

This is so common that C++11 introduced a "range-based for loop" that is just syntactic sugar for the code above:

```
// This is the preferred, modern way to loop
for (auto const& element : c) {
    // ... do something with element ...
}
```

Listing 14: The modern C++11 loop (Slide 36)

4.3 How Iterators are Implemented

(Slides 37, 38)

This "generic pointer" is just an abstraction.

- **For `std::vector` (Slide 37):** An iterator *is* just a raw pointer. `begin()` returns a `T*` to the first element. `++it` is just pointer arithmetic.
- **For `std::list` (Slide 38):** An iterator is a small *class* that holds a pointer to a `node`. This class **overloads the operators** to *pretend* to be a pointer:
  - Its `operator++()` function is defined to mean `p = p->next`.
  - Its `operator*()` function is defined to mean `return p->value`.

The algorithm doesn't know or care about this difference. It just calls `++it` and `*it`, and the magic works.

4.4 Complete Iterator Operations Reference

4.4.1 Essential Iterator Operators

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

// Get iterator to first element
auto it = v.begin();

// Get iterator past the last element (DO NOT dereference!)
auto it_end = v.end();

// Const iterators (read-only)
auto cit = v.cbegin();  // Points to first element (const)
auto cit_end = v.cend(); // Points past last element (const)

// Reverse iterators (iterate backwards)
auto rit = v.rbegin();  // Points to last element
auto rit_end = v.rend(); // Points before first element
```

Listing 15: Core Iterator Operations: Obtaining Iterators

```
std::vector<int> v = {10, 20, 30};
auto it = v.begin();

// Dereference to get value
int value = *it;  // value = 10

// Access member of pointed object (for objects/structs)
std::vector<std::string> names = {"Alice", "Bob"};
auto name_it = names.begin();
int length = name_it->length();  // Calls string::length()
// Equivalent to: (*name_it).length()
```

Listing 16: Dereferencing - Accessing Values

```cpp
std::vector<int> v = {10, 20, 30, 40};
auto it = v.begin();

// Pre-increment (preferred for iterators)
++it;  // Now points to 20
int val1 = *it;  // val1 = 20

// Post-increment (creates a copy, slightly less efficient)
auto old_it = it++;  // it moves to 30, old_it still at 20
int val2 = *old_it;  // val2 = 20
int val3 = *it;      // val3 = 30

// Pre-decrement (for bidirectional iterators)
--it;  // Back to 20

// Post-decrement
it--;  // Back to 10
```

Listing 17: Incrementing and Decrementing Iterators

```cpp
std::vector<int> v = {10, 20, 30};
auto it1 = v.begin();
auto it2 = v.begin();
auto it_end = v.end();

// Equality comparison
if (it1 == it2) {
    // True: both point to same position
}

// Inequality comparison (most common for loops!)
if (it1 != it_end) {
    // True: it1 is not past-the-end
    int value = *it1;  // Safe to dereference
}

// IMPORTANT: Always check iterator != end() before dereferencing!
while (it1 != v.end()) {
    std::cout << *it1 << " ";
    ++it1;
}
```

Listing 18: Comparison Operators - Essential for Loops!

```cpp
std::vector<int> v = {10, 20, 30, 40, 50};
auto it = v.begin();

// Addition (jump forward)
auto it_plus_3 = it + 3;  // Points to 40
int val = *it_plus_3;     // val = 40

// Subtraction (jump backward)
auto it_minus_1 = it_plus_3 - 1;  // Points to 30

// Compound assignment
it += 2;  // Move forward 2 positions (now at 30)
it -= 1;  // Move backward 1 position (now at 20)
```

```cpp

// Array-like indexing
int value = it[2];  // Access element 2 positions ahead
                    // Equivalent to *(it + 2)

// Distance between iterators
auto dist = v.end() - v.begin();  // dist = 5 (size of vector)

// Relational comparisons
auto it1 = v.begin();
auto it2 = v.begin() + 2;
if (it1 < it2) {  // True: it1 comes before it2
    // ...
}
```

Listing 19: Random Access Iterator Operations (vector, deque only)

4.4.2 Complete Example: Using All Iterator Operations

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

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

    // === Basic Iteration ===
    std::cout << "Forward iteration:\n";
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";  // Dereference to get value
    }
    std::cout << "\n";

    // === Reverse Iteration ===
    std::cout << "Reverse iteration:\n";
    for (auto rit = numbers.rbegin(); rit != numbers.rend(); ++rit) {
        std::cout << *rit << " ";
    }
    std::cout << "\n";

    // === Random Access ===
    auto it = numbers.begin();
    std::cout << "Element at begin: " << *it << "\n";           // 10
    std::cout << "Element at begin+2: " << *(it + 2) << "\n";   // 30
    std::cout << "Element using []: " << it[3] << "\n";         // 40

    // === Modifying Through Iterator ===
    auto modify_it = numbers.begin();
    *modify_it = 15;  // Changes first element from 10 to 15

    // === Iterator Arithmetic ===
    auto start = numbers.begin();
    auto end = numbers.end();
    auto distance = end - start;  // Number of elements: 5

```

```
    auto middle = start + (distance / 2);  // Points to middle element
    std::cout << "Middle element: " << *middle << "\n";  // 30

    // === Const Iterators (Read-Only) ===
    for (auto cit = numbers.cbegin(); cit != numbers.cend(); ++cit) {
        std::cout << *cit << " ";  // Can read
        // *cit = 100;  // ERROR: Cannot modify through const iterator
    }
    std::cout << "\n";

    return 0;
}
```

Listing 20: Comprehensive iterator usage example

4.5 Iterator Categories

(Slide 39)

Not all iterators are created equal. They are categorized by their "power."

4.5.1 Iterator Category Hierarchy

1. **Input/Output Iterator:** Weakest. Can only move forward and be read-/written once.
   - **Supported:** `++it`, `it++`, `*it`, `==`, `!=`
   - **Example:** `std::istream_iterator`
2. **Forward Iterator:** Can move forward (++) many times.
   - **Supported:** All Input operations, plus multi-pass guarantee
   - **Example:** `std::forward_list::iterator`
   - **Use case:** `std::find()`, `std::replace()`
3. **Bidirectional Iterator:** Can move forward (++) and backward (–).
   - **Supported:** All Forward operations, plus `–it`, `it–`
   - **Examples:** `std::list::iterator`, `std::set::iterator`, `std::map::iterator`
   - **Use case:** `std::reverse()`, `std::find_end()`
4. **Random Access Iterator:** Most powerful. Can jump to any position in $O(1)$ time.
   - **Supported:** All Bidirectional operations, plus:
     - `it + n`, `it - n`, `it += n`, `it -= n`
     - `it[n]` (array subscript)

  - `it2 - it1` (distance)
  - <, >, <=, >= (comparison)
- **Examples:** `std::vector::iterator`, `std::deque::iterator`, raw pointers
- **Use case:** `std::sort()`, `std::binary_search()`

```cpp
// Forward Iterator (std::forward_list)
std::forward_list<int> fl = {1, 2, 3, 4, 5};
auto flit = fl.begin();
++flit;          // OK: Can move forward
// --flit;       // ERROR: Cannot move backward
// flit + 2;     // ERROR: Cannot jump

// Bidirectional Iterator (std::list)
std::list<int> lst = {1, 2, 3, 4, 5};
auto lit = lst.begin();
++lit;           // OK: Can move forward
--lit;           // OK: Can move backward
// lit + 2;      // ERROR: Cannot jump

// Random Access Iterator (std::vector)
std::vector<int> vec = {1, 2, 3, 4, 5};
auto vit = vec.begin();
++vit;           // OK: Can move forward
--vit;           // OK: Can move backward
vit = vit + 3;   // OK: Can jump
int val = vit[1]; // OK: Can use subscript
```

Listing 21: Iterator Category Example

Important: Algorithms specify the *minimum* category they need. `std::find` only needs a Forward iterator. `std::sort` *requires* a Random Access iterator (which is why you can't call `std::sort` on a `std::list`).

5 The Generic Algorithms

(Slides 53-63)

Now we get the payoff. The `<algorithm>` header contains dozens of pre-built, highly-optimized functions that operate on iterators. **You should always prefer these to writing your own loops.**

5.1 Example: `std::find`

(Slide 54) `std::find` searches a range for a value.

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

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

```
int value_to_find = 30;

// find returns an iterator
auto it = std::find(v.begin(), v.end(), value_to_find);

// --- CRITICAL CHECK ---
// If not found, find returns the .end() iterator!
if (it != v.end()) {
    std::cout << "Found it! Value is " << *it << std::endl;
} else {
    std::cout << "Value not found." << std::endl;
}
```

Listing 22: Using std::find

The implementation of 'find' is just the simple generic loop from Slide 35.

5.2 Example: `std::find_if`

(Slide 55) `find_if` is more powerful. Instead of a value, it takes a **predicate**: a function (or function-like object) that returns `bool`.

```
// A predicate function
bool isEven(int x) {
    return x % 2 == 0;
}

std::vector<int> v = {1, 3, 5, 6, 7, 9};

// Find the first element for which isEven() returns true
auto it = std::find_if(v.begin(), v.end(), isEven);

if (it != v.end()) {
    std::cout << "First even number is " << *it << std::endl; // Prints 6
}
```

Listing 23: Using std::find$_{i}f$

5.3 Tricky Case 1: Member Functions as Predicates

(Slide 56)

What if your predicate is a *member function* of a class?

```
class Animal {
public:
    bool is_pregnant() const;
    // ...
};

std::list<Animal> flock;
// ...
// This will NOT compile!
```

```cpp
auto it = std::find_if(flock.begin(), flock.end(), &Animal::
    is_pregnant);
```

The `find_if` algorithm doesn't know how to call a member function. It expects a global function `bool(Animal)`.

Solution: Use an adapter from `<functional>` called `std::mem_fn`.

```cpp
#include <functional> // Need this!

// This works!
auto it = std::find_if(flock.begin(), flock.end(),
                       std::mem_fn(&Animal::is_pregnant));
```

5.4 Tricky Case 2: Copying into Empty Containers

(Slide 57)

`std::copy` copies elements from one range to a destination iterator.

```cpp
std::vector<int> v = {1, 2, 3};
std::vector<int> w; // w is EMPTY!

// This will CRASH!
// w.begin() points to nothing, so copy tries to write to
// invalid memory.
std::copy(v.begin(), v.end(), w.begin());
```

Solution 1 (Clumsy): `w.resize(v.size());` first.

Solution 2 (Elegant): Use a `std::back_inserter` from `<iterator>`.

```cpp
#include <iterator> // Need this!

// This works!
// std::back_inserter(w) is an iterator adapter that
// "pretends" to be a normal iterator, but its "write"
// operation (operator=) actually calls w.push_back().
std::copy(v.begin(), v.end(), std::back_inserter(w));

// w is now {1, 2, 3}
```

5.5 Algorithm Naming Conventions

(Slide 62)

The algorithm names are very consistent.

- Suffix **_if**: Takes a predicate instead of a value.
  - `find(..., val)`
  - `find_if(..., pred)`
- Suffix **_copy**: Does not modify the original range. Writes a *copy* of the result to a destination.

  - `reverse(beg, end)`: Reverses the range in-place.
  - `reverse_copy(beg, end, dest)`: Writes a reversed copy to `dest`, leaving `[beg, end)` unchanged.

- Suffix **_copy_if**: Combines both.

  - `remove_copy_if(beg, end, dest, pred)`: Copies all elements *except* those for which `pred` is true.

For a full list of all algorithms, see a C++ reference like 'cppreference.com'.

6 Application and Summary

6.1 Application: The Penna Model

(Slides 58, 64)

The exercise for this week is to code the `Population` class for the Penna model. You can (and should) use the STL to make this trivial.

- The `Population` can just be a `std::list<Animal>`. We use `list` because we expect to remove many animals from the middle (when they die), and `list` is $O(1)$ for this.

- To remove all dead animals, you don't need to write your own loop. You can use the `list`'s special `remove_if` function combined with the `mem_fn` adapter.

```
#include <list>
#include <functional>
#include "animal.hpp"

class Population {
public:
    void remove_dead() {
        // This one line replaces an entire, complex,
        // error-prone for-loop.
        pop_.remove_if(std::mem_fn(&Animal::is_dead));
    }
    // ...
private:
    std::list<Animal> pop_;
};
```

The challenge is to write the entire `Population` class **without any raw loops** (`for`, `while`), using only STL algorithms. This greatly increases reliability.

6.2 Summary

(Slide 65)

- **Rule 1:** Before you write any code, **check the C++ standard library**. `find`, `sort`, `vector`, `map`, etc., are already written, heavily optimized, and bug-free.
- **Rule 2:** When you design your own classes, try to emulate the STL. Provide `.begin()` and `.end()` iterators so your classes can be used with generic algorithms.
- **Rule 3:** Don't be scared by the long error messages. Template metaprogramming (the `<...>` syntax) can produce huge, unreadable errors. This is normal. Look at the *first* line of the error; that's usually where the real problem is.

Part II
Plotting / Scientific Visualization (week07b)

7 Plotting Our Data

(Slide 2)

Our scientific simulations will produce large amounts of data (e.g., population size over time). A text file full of numbers is useless for understanding; we need to *visualize* it. This section introduces the tools we can use.

7.1 Gnuplot

- **Website:** `http://www.gnuplot.info/`
- **What it is:** A very old, powerful, and stable command-line plotting program.
- **Pros:** It's fast, universal (Linux, macOS, Windows), and excellent for generating 2D and 3D plots quickly, often from within a script.

7.2 Python + Matplotlib

- **Website:** `https://www.python.org/` and `https://matplotlib.org/`
- **What it is:** The combination of the Python programming language and its most popular plotting library, Matplotlib.
- **Pros:** This is the *de facto* standard in many scientific fields. It is extremely flexible, powerful, and can create publication-quality graphs.

7.3 Best Practice: Python Virtual Environments

When using Python, it is highly recommended to use **virtual environments** (e.g., via Python's built-in 'venv' module).

- **What it is:** A tool that creates an isolated, self-contained "bubble" for each of your projects.
- **Analogy:** A virtual environment is like a separate, clean workshop for each project. For "Project A," you can install a special "bandsaw" (e.g., 'matplotlib version 3.0'). For "Project B," you can install a different "bandsaw" (e.g., 'matplotlib version 3.5'). These don't interfere. This prevents a library update for one project from breaking all your other projects.
- **How to use (basic):**

```
# 1. Create a virtual environment named "my-project-env"
python3 -m venv my-project-env

# 2. Activate it (on Linux/macOS)
source my-project-env/bin/activate

# 3. Install libraries. They will only be installed inside this "bubble
pip install matplotlib
pip install numpy

# 4. Run your script
python my_simulation.py

# 5. Deactivate when you are done
deactivate
```

The lecture will cover Python in more detail later. For now, demo code can be found in the ‘week07/plotting‘ directory. For 3D visualization of complex simulation data, we will later look at powerful tools like **ParaView** and **VisIt**.