SWEG2102 Fall 2026

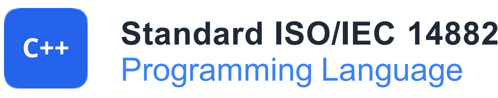
Fundamentals of Programming II

Lecture 12

Standard Template Library (STL) - Part I

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia



Standard ISO/IEC 14882
Programming Language

Outline

STL Part I

01 Introduction to STL & Templates

02 Mastering Iterators

03 Vectors in Depth

04 Lists in Depth

05 Integrated Practical Exercise

Learning Objectives

By the end of this lecture, you will be able to:

1. Explain the Standard Template Library (STL) and describe its purpose
2. Apply iterators effectively to traverse containers
3. Use vectors and lists proficiently by creating, modifying, and traversing them.
4. Analyze the performance of STL containers and operations using Big-O notation
5. Develop idiomatic C++ programs by applying STL best practices

Basics of Standard Template Library

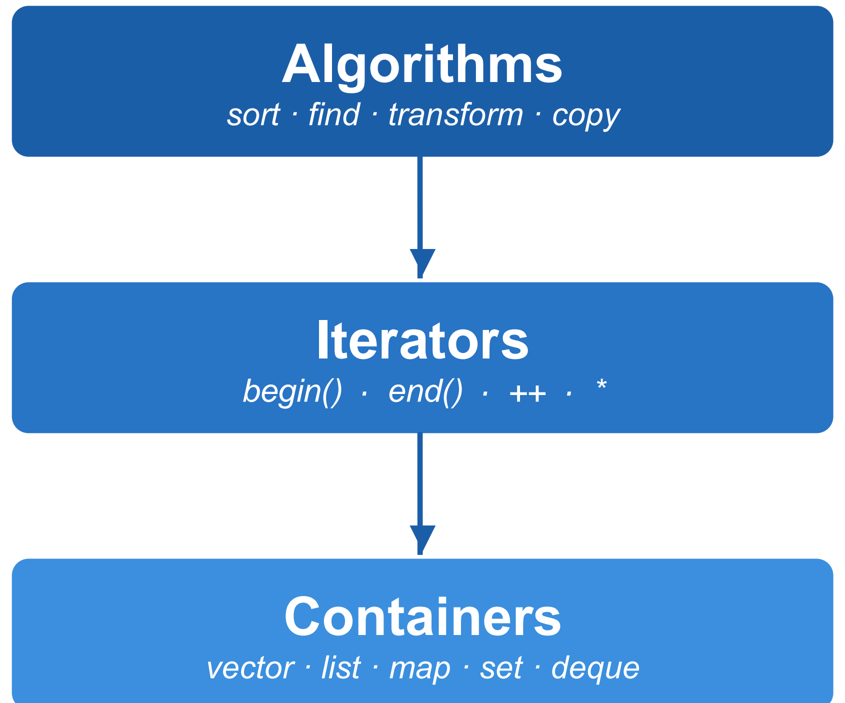
1.1 What is the STL

Core Concepts

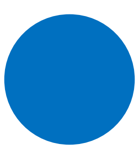
Standard Template Library (STL)

- A collection of **generic, reusable** C++ components
- Containers, algorithms, and iterators, delivered as part of the C++ Standard Library.

STL Architecture (Three Pillars)

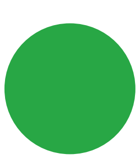


Cont'd



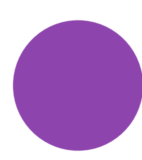
Performance

Highly optimized components.



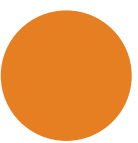
Correctness

Peer-reviewed; decades of real-world testing built in.



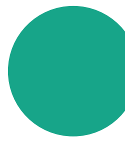
Readability

Idiomatic code other C++ programmers instantly recognize.



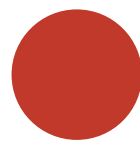
Time to Market

Use std::vector. Focus on domain logic, not data structures.



Portability

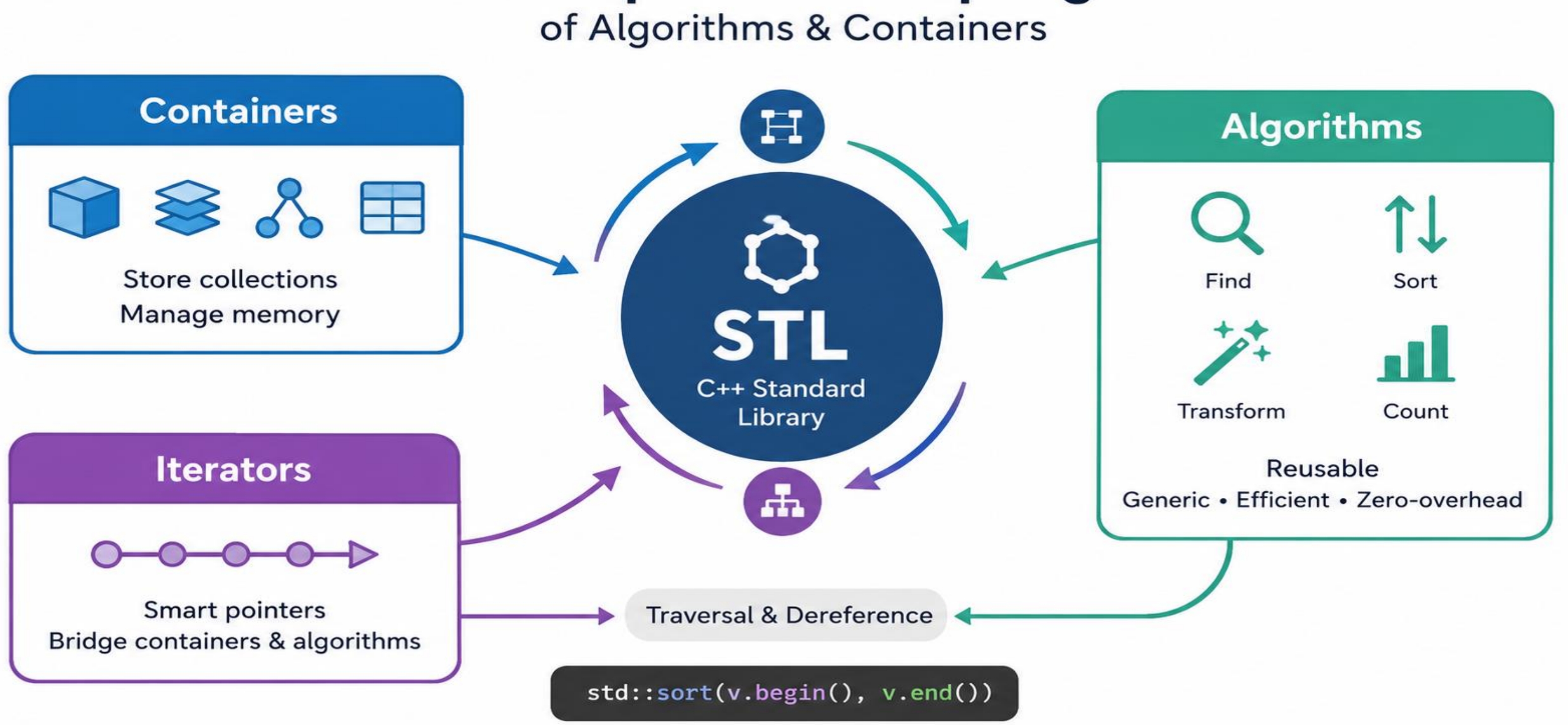
Compiles on Windows, Linux, macOS, embedded, all platforms.



Scalability

Grows from toy programs to massive systems unchanged.

Cont'd



Source: *AI-generated image (ChatGPT)*

Cont'd

CONTAINER

std::vector<int>

- Stores data
- Provides begin()/end()

ITERATORS

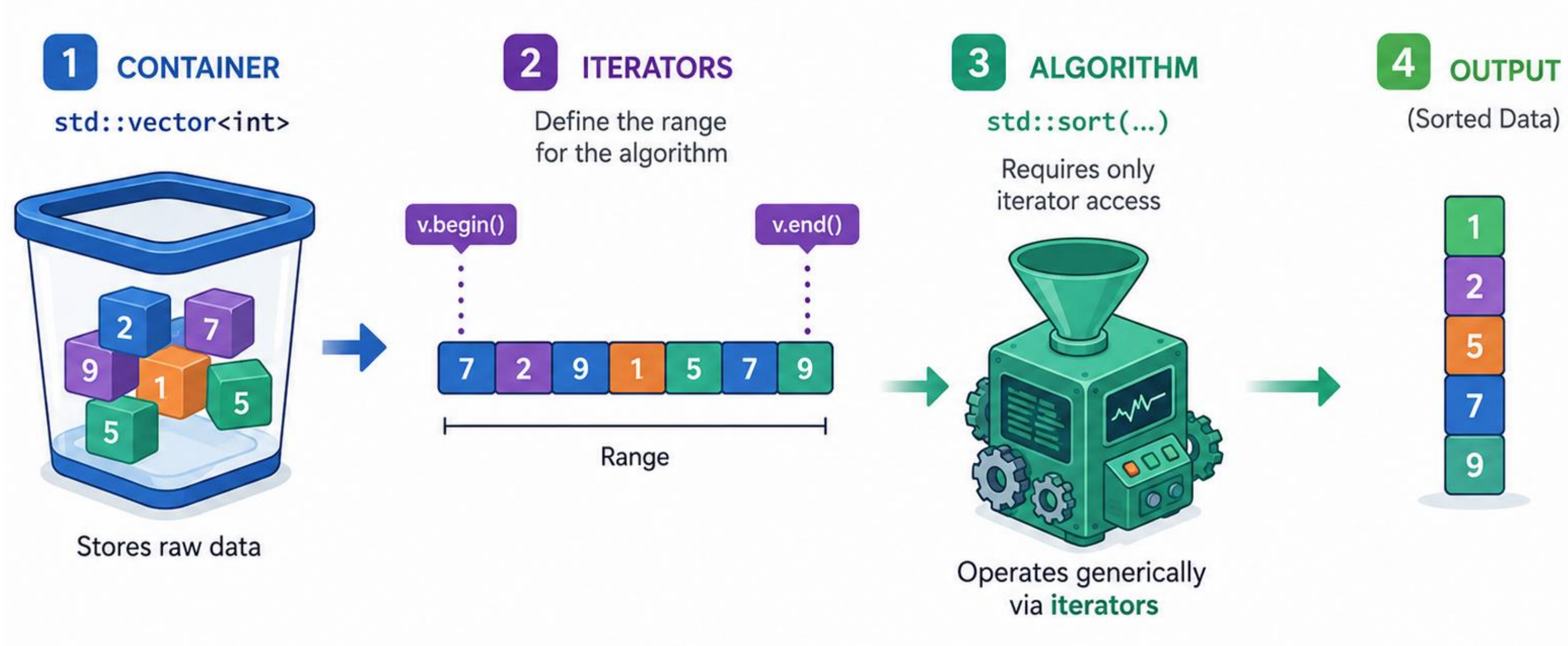
v.begin() → v.end()

- Bridge: traverse and access elements

ALGORITHM

std::sort(…)

- Operates generically through iterators




```
// The STL pattern: Algorithm(Container.begin(), Container.end(), ...)
std::sort(v.begin(), v.end()); // sort any container with Random-Access iterators
std::find(v.begin(), v.end(), 42);// works on vector, list, deque — any container!
```

Cont'd

Sequence Containers *Maintain insertion order*	**vector | list | deque | array | forward_list**
Associative Containers *Ordered by key; O(log n) lookup*	**set | multiset | map | multimap**
Unordered Associative *Hash-based; O(1) average lookup*	**unordered_set | unordered_map | …**
Adapter Containers *Constrain interfaces on sequence containers*	**stack | queue | priority_queue**

Cont’d

Modification Algorithms

transform | fill | reverse | rotate | shuffle

Modify elements in-place within the container range

Non-Modifying Algorithms

find | count | search | max_element | min_element

Read-only traversal — container is unchanged

Sorting & Partitioning

sort | stable_sort | nth_element | partition | ranges::sort (C++20)

Rearrange elements; stable_sort preserves relative order

Numeric Algorithms

accumulate | inner_product | partial_sum | adjacent_difference

Mathematical operations over ranges of numbers

Quick Check

1. **What are the three core pillars of the STL?**

   ❖ Containers, Iterators, Algorithms

2. **Why is the decoupling between algorithms and containers important?**

   ❖ Algorithms work on any container that provides the right iterator type.

3. **Which standard year saw the STL officially adopted into C++?**

   A)1990 B)1998 C)2011 D)2020

   ❖ Answer: B

1.2 What is an Iterator

Core Concepts

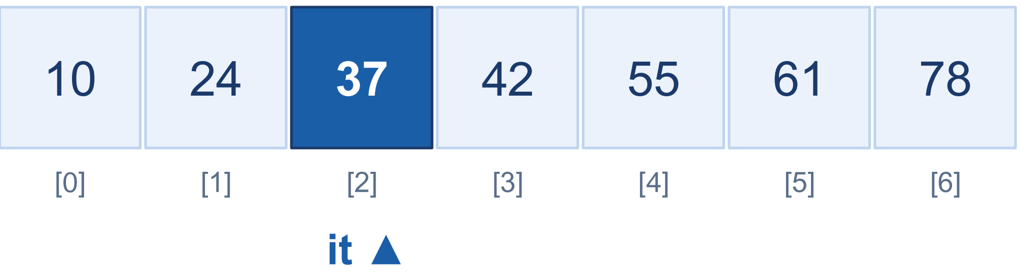
Iterator

- An object that acts as a **pointer into a container.**
- Letting you traverse, read, and modify elements without knowing the container's internal structure.

Pointer-like syntax:

- ***it** — dereference to read or write the current element.
- **++it** — advance to the next element.
- **it1 == it2** — compare positions (e.g. against end()).

Iterator as a Cursor



***it** — Read / write element

++it — Move to next slot

it == end() — Check if done

Cont'd

➢C++ defines five categories of iterators, arranged in a capability hierarchy.

❖**Input / Output:** Single-pass read (input) or write (output). Cannot revisit elements

❖**Forward:** Multi-pass read/write in one direction.

❖**Bidirectional:** Forward + backward traversal with --it. Used by list, map, set.

❖**Random-Access:** Full pointer arithmetic: it+n, it-n, it[n], comparisons. Used by vector, deque, array.

Iterator Capability Hierarchy

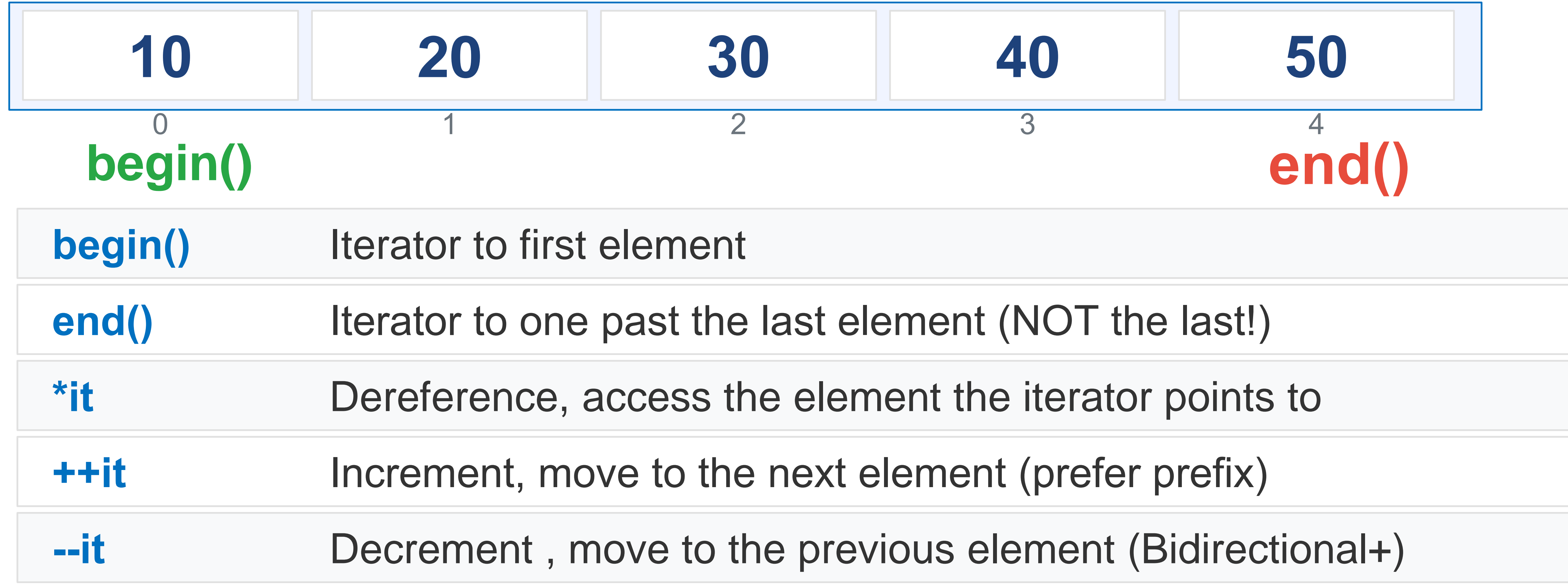
Random-Access Iterator
vector · deque · array

Bidirectional Iterator
list · map · set

Forward Iterator
forward_list

Input / Output Iterator
istream_iterator

Cont'd



begin()	Iterator to first element
end()	Iterator to one past the last element (NOT the last!)
*it	Dereference, access the element the iterator points to
++it	Increment, move to the next element (prefer prefix)
--it	Decrement , move to the previous element (Bidirectional+)

Range Syntax: [begin(), end()) — includes begin, EXCLUDES end. Standard STL convention.

Cont'd

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

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

  // Get iterator to firsat element
  auto it = nums.begin();
  cout << *it;    // Output: 10
  ++it;           // move forward
  cout << *it;    // Output: 20

  // Iterator-based loop
  for (auto i = nums.begin();
       i != nums.
end(); ++i)
    cout << *i <<
" ";

  // Range-based for (C++11)
  for (int n : nums)
    cout << n <<
" ";
}
```

$ Output

```
First element: 10
Second element: 20
All elements: 10 20 30 40 50
Range-based:   10 20 30 40 50
```

begin()

Returns iterator to first element (index 0)

++it

Prefix increment, more efficient than it++

Range-based for

Compiler generates iterator loop automatically

end()

Loop terminates cleanly when it reaches end()

Cont'd

	Pointers	Iterators
Type	Raw memory address	Object with operator overloads
Syntax	* for deref, ++ for increment	Same syntax — * and ++ — but customized
Speed	Always O(1) Random-Access	Varies by container (list = O(n) traversal)
Validity	Can dangle (undefined behavior)	Invalidated by specific container ops
Arithmetic	ptr + 5 always valid	Only Random-Access iterators support it

</> Cont'd

1. Dereferencing end()

*container.end() // ❌ undefined behavior — crash!

✓ Check: it != container.end() before dereferencing

2. Using Invalidated Iterators

auto it = v.begin(); v.push_back(99); // ❌ it is now invalid!

✓ After modifying a container, re-acquire fresh iterators

3. Comparing Iterator from Different Container

it1 == it2 // ❌ undefined if from different containers

✓ Only compare iterators belonging to the SAME container

4. Arithmetic on Incompatible Iterators

list_it + 5 // ❌ list has no random-access support

✓ Use std::advance(it, 5) or std::distance() for generic code

5. Off-by-One with end()

while (it <= container.end()) // ❌ processes past end!

✓ Always use != end(), never <= end(). Range is [begin, end)

Quick Check Iterators

#	Question	Answer
1	1. What does the * operator do when applied to an iterator?	Dereferences the iterator to access the element it points to.
2	2. Why does end() point one past the last element?	So loop termination is clean — when it == end(), all elements have been processed.
3	3. Which iterator type supports pointer arithmetic (+=, -=, [])?	Random-Access iterators (vector, deque, array).
4	4. Is it safe to use an iterator after push_back() on a vector?	Not reliably — push_back may trigger reallocation, invalidating iterators.
5	5. Can you compare iterators from two different containers?	No — that is undefined behavior. Iterators are bound to their container.

1.3 Introduction to vectors

What is a std::vector<T>?

➢ A smart, automatically resizable array from the C++ Standard Library (#include <vector>).

Memory Model

➢ Elements sit side-by-side in heap memory.

➢ **size()** is what you're using; **capacity()** is what's reserved.

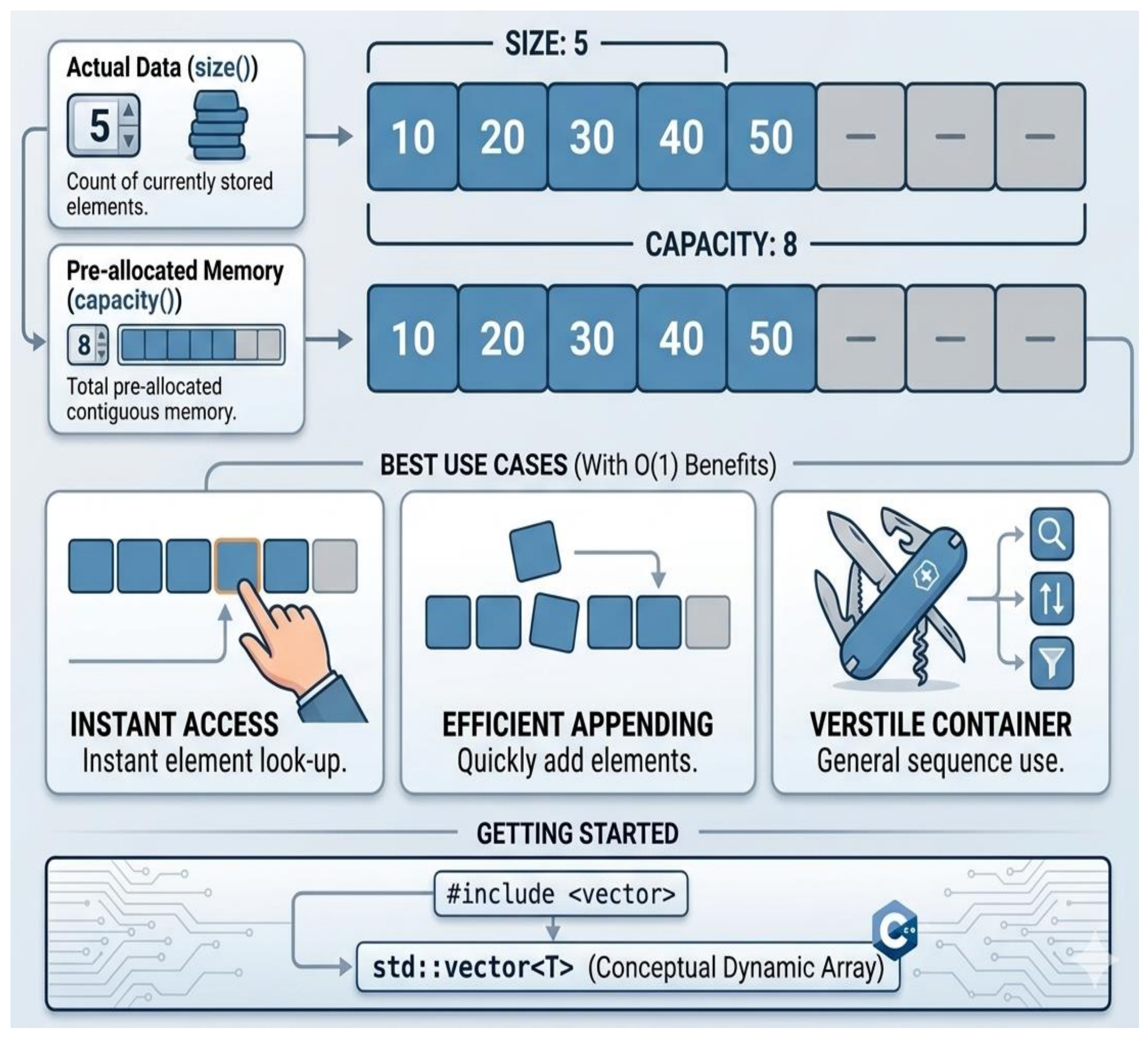
➢ Reallocates and doubles in size (~2x) when full.

•Key Characteristics

➢**Dynamic:** Grows and shrinks at runtime.

➢**Fast Access:** (1) random access via v[i] or v.at(i).

➢**Type-Safe:** Enforced by the compiler



Source: *AI-generated image (ChatGPT)*

Cont'd

Source: *AI-generated image (ChatGPT)*

Internal Pointers

_begin	_end	_capacity_end
Points to first element	One past last valid element (== size)	One past last allocated byte (== capacity)

Reallocation copies ALL elements → old iterators become invalid

Use reserve(n) if you know final size → eliminates reallocation cost entirely

Doubling strategy: amortized O(1) push_back over any sequence of operations

Cont'd

```cpp
#include <vector>
// Default – empty
vector<int> v1;
// Initializer list (C++11)
vector<int> v2 = {10, 20, 30};
// n elements, all 0
vector<int> v4(5);
// n copies of value
vector<int> v5(3, 42);
// Range constructor
vector<int> v6(v2.begin(), v2.end());
// Move constructor (no copy!)
vector<int> v8 = std::move(v2);
```

Default
(empty)

Initializer
[10, 20, 30]

Size(n)
[0, 0, 0, 0, 0]

Size+Value
[42, 42, 42]

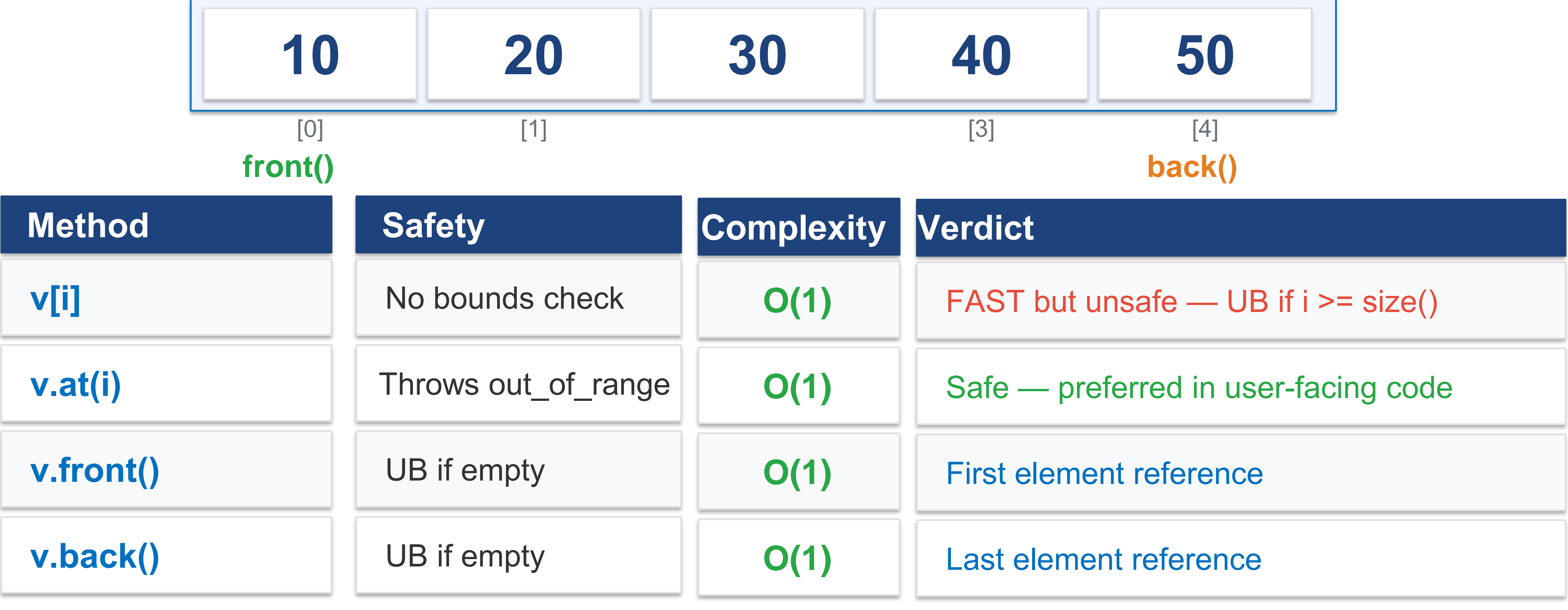
Range
Copy from another range/container

Move
Takes ownership, source becomes empty

Cont'd

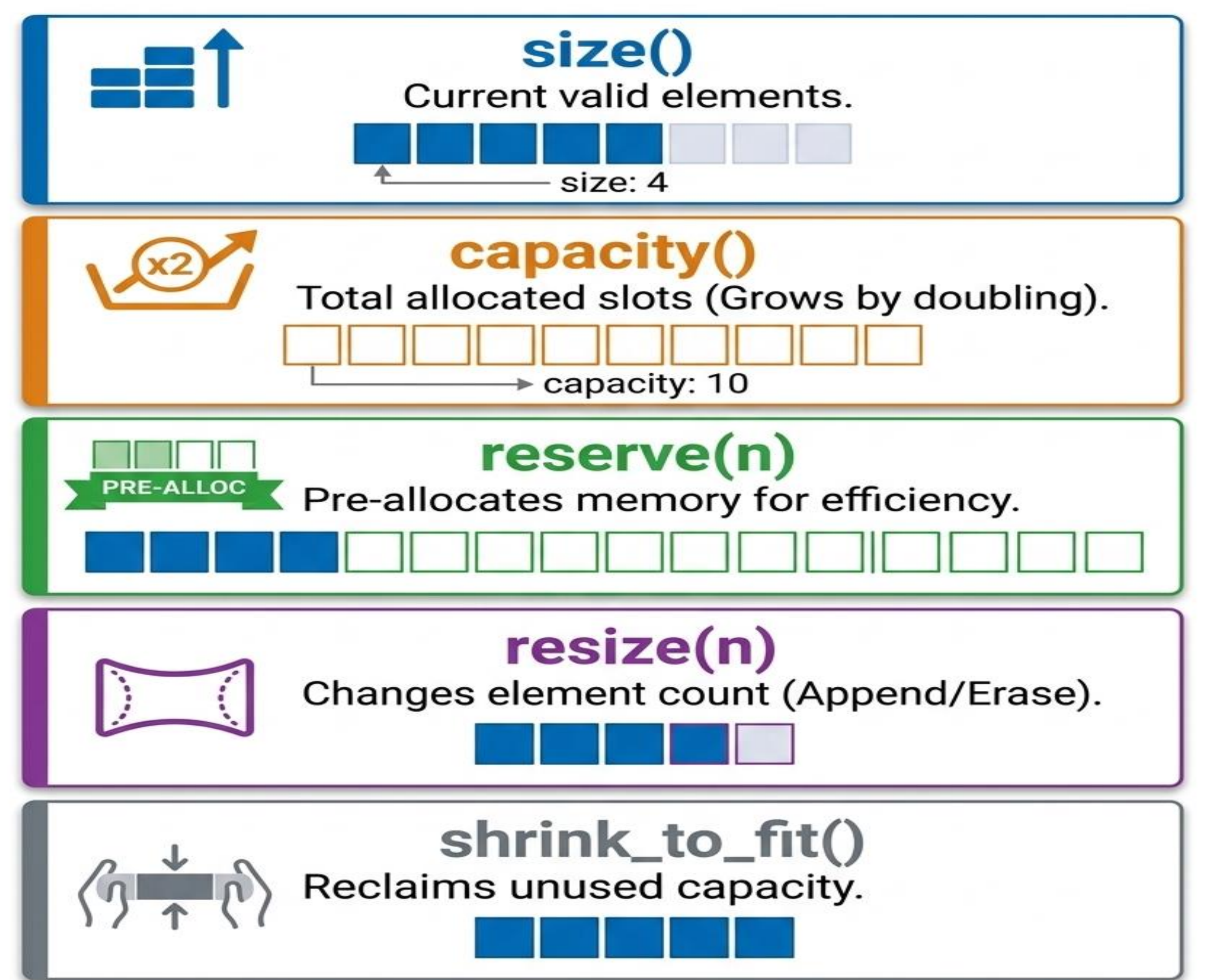
Operation	Complexity	Notes / Behaviour
push_back(v)	O(1) amortized	Append to end. May trigger reallocation (rare).
pop_back()	O(1)	Remove last element. Capacity unchanged.
insert(it, v)	O(n)	Insert before iterator. All elements after shift right.
erase(it)	O(n)	Remove at iterator. All elements after shift left.
erase(f, l)	O(n)	Remove range [first, last). O(n) shifting.
clear()	O(n)	Destroy all elements. Capacity unchanged.

Cont'd



Method	Safety	Complexity	Verdict
v[i]	No bounds check	O(1)	FAST but unsafe — UB if i >= size()
v.at(i)	Throws out_of_range	O(1)	Safe — preferred in user-facing code
v.front()	UB if empty	O(1)	First element reference
v.back()	UB if empty	O(1)	Last element reference

Cont'd



Source:
AI-generated image(Gemini 3)

Cont'd

Operation	Complexity	Notes
v[i] / v.at(i)	O(1)	Random access: vector's superpower
front() / back()	O(1)	Direct pointer deref
push_back(v)	O(1) amort.	Occasional O(n) realloc, but rare
pop_back()	O(1)	Decrement size counter
insert(it, v)	O(n)	Shifts all elements after insertion point
erase(it)	O(n)	Shifts all elements after erased position
clear()	O(n)	Destructs every element
find (linear scan)	O(n)	No built-in index; use std::find

Cont’d

Range-Based For (Recommended)

```
// Default choice — clean, safe, idiomatic C++11
for (int score : scores)
  cout << score;

// Use const ref for large objects:
for (const string& s : v)
  cout << s;
```

- Best default choice for nearly all read or write loops.
- Avoids index errors and off-by-one bugs entirely.

Iterator Loop (When You Need the Iterator)

```
// Needed for erase() / insert() during loop
for (auto it = v.begin();
     it != v.end(); ++it)
  cout << *it;
```

- Required when calling erase() or insert() inside the loop.
- Dereferencing *it gives the element value.

Index Loop (Discouraged)

```
// Works but error-prone; less idiomatic
for (size_t i = 0; i < v.size(); ++i)
  cout << v[i];
```

- Use size_t (unsigned) to match v.size().
- Only choose this **when the index itself is needed**.

Cont'd

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

int main() {
  vector<int> scores;
  scores.push_back(85); push_back(92);
  push_back(78); push_back(95);

  // Find max via algorithm
  auto mx = std::max_element(
    scores.begin(), scores.end());
  cout << *mx;

  // Sort in-place
  std::sort(scores.begin(), scores.end());

  // Erase 3rd element (O(n)!)
  scores.erase(scores.begin() + 2);

  cout << "Size:" << scores.size()
       << " Cap:" << scores.capacity();
}
```

$ Output

```
Highest: 95
Sorted: 78 85 92 95
After erase: 78 85 95
Size: 3  Cap: 4
```

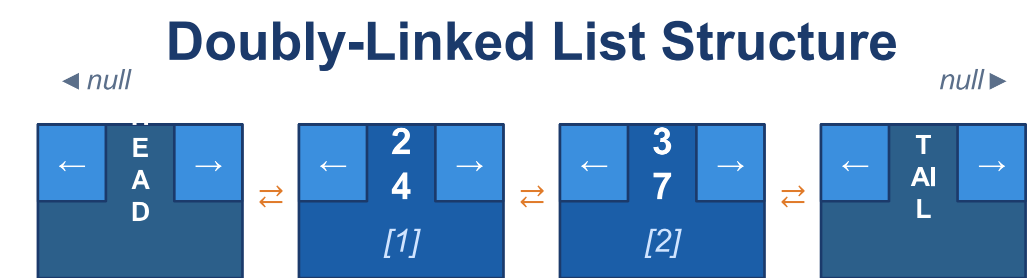
1. push_back, O(1) amortized each
2. max_element, works via iterators
3. std::sort, needs Random-Access iterators
4. erase at index 2, O(n) shift
5. size ≠ capacity after erase

Quick Check vectors

#	Question	Answer
1	1. What does push_back(v) do, and what is its time complexity?	❖ Appends v to end. O(1) amortized, occasionally O(n) for reallocation, but rare.
2	2. What is the difference between size() and capacity()?	❖ size() = number of elements; capacity() = allocated memory. capacity() ≥ size() always.
3	3. How would you efficiently build a vector of 1 million elements?	❖ Call reserve(1'000'000) first, pre-allocates memory.
4	4. Why is insertion in the middle of a vector O(n)?	❖ Every element after the insertion point must shift right, requiring O(n) copies.
5	5. Name one advantage and one disadvantage vs. a list.	❖ Advantage: O(1) random access. Disadvantage: O(n) middle insertion (list is O(1)).

1.4 Introduction to Lists

- **std::list**
- Is a **doubly-linked list** from the C++ STL.
- Every element (node) holds a value plus two pointers.
- **Key characteristics:**
  - ⚡ **O(1) insert & erase** anywhere no shifting of elements needed, just pointer re-wiring.
  - 🔄 **Bidirectional traversal** iterate forward and backward, but no random access (no [ ]).
  - 📦 **Non-contiguous memory** nodes are scattered on the heap; no cache-friendly layout like vector.



list vs vector

Operation	list	vector
Insert / erase (middle)	O(1) ✅	O(n) ❌
Random access [ ]	❌ none	O(1) ✅
Memory layout	Non-contiguous	Contiguous

std::list provides O(1) insertion and deletion at any known position, making it ideal when elements are frequently added or removed mid-sequence.

Cont'd

Node Anatomy

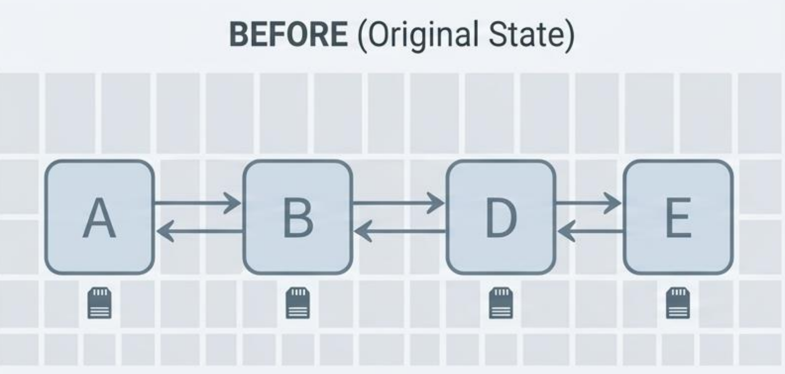
prev
pointer to previous node

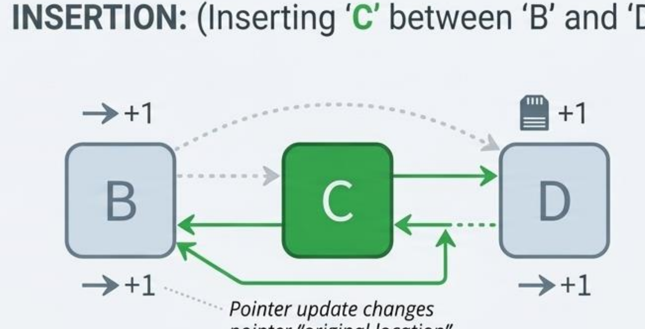
data
Element value stored

next
pointer to next node

Sentinel Nodes: Head (before first) and Tail (after last) simplify boundary logic, no null-pointer special cases needed.

O(1) Insert — No Element Shifting Required





Source:
AI-generated image(Gemini 3)

Cont'd

```cpp
#include <list>

// Default – empty
list<int> l1;

// Initializer list
list<int> l2 = {10, 20, 30};

// n copies of value
list<int> l5(3, 42);

// Range from another container!
vector<int> v = {100, 200, 300};
list<int> l9(v.begin(), v.end());

// Move constructor (C++11)
list<int> l8 = std::move(l2);
```

STL Interface Consistency

Same constructors as vector. Range constructor works across containers, iterator abstraction!

List vs Vector — Same API

- No reserve()
- No capacity()
- No operator[]
- push_front() added
- remove(val) added

Cont'd

Operation	Complexity	Notes
push_front(v)	O(1)	Prepend allocate node, update head pointers
push_back(v)	O(1)	Append allocate node, update tail pointers
pop_front()	O(1)	Remove head node, update pointers
pop_back()	O(1)	Remove tail node, update pointers
insert(it, v)	O(1)	Insert before iterator 2 pointer updates, no shift!
erase(it)	O(1)	Unlink node, update neighbours no shift!
remove(val)	O(n)	Traverse and erase all nodes equal to val
operator[]	N/A	NOT supported use iterators + std::advance

</> Cont'd

Operation	std::vector	std::list
Random access v[i]	O(1)	✗ N/A
push_back	O(1)✓	O(1)✓
pop_back	O(1)✓	O(1)✓
push_front	O(n)✗	O(1)✓
pop_front	O(n)✗	O(1)✓
insert / erase middle	O(n)✗	O(1)✓
find (linear)	O(n)	O(n)
Iterator invalidation	On reallocate	Only erased
Cache efficiency	Excellent	Poor (scattered)
Memory overhead/elem	Low (compact)	High (+2 ptrs)

Cont'd

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

int main() {
  list<int> tasks;
  tasks.push_back(3);
  tasks.push_back(1);
  tasks.push_back(2);

  // O(1) prepend!
  tasks.push_front(5);

  // Find then O(1) erase
  auto it = std::find(
    tasks.begin(), tasks.end(), 1);
  if (it != tasks.end())
    tasks.erase(it);

  // Remove ALL matching values
  tasks.push_back(3); push_back(3);
  tasks.remove(3);

  // Bidirectional iteration
  for (auto it = tasks.rbegin();
       it != tasks.rend(); ++it)
    cout << *it;
}
```

```
Initial:            3 1 2
After push_front(5): 5 3 1 2
After erase(1):   5 3 2
After remove(3): 5 2
Backward:         2 5
```

1. push_front is O(1) vector would be O(n)
2. std::find works on any container via iterators
3. erase(it) is O(1) no element shifting!
4. remove(val) traverses entire list O(n)
5. rbegin()/rend() bidirectional iteration

Quick Check Lists

1. **What is the time complexity of erase(iterator) on a list?**
   - ❖ O(1) just deallocate the node and update two neighbour pointers.
2. **Why can't you subscript a list like a vector (list[i])?**
   - ❖ Random access requires traversing from head O(n).
3. **Name two operations where lists are faster than vectors.**
   - ❖ push_front / pop_front (O(1) vs O(n)) and insert/erase anywhere (O(1) vs O(n)).
4. **What is the main disadvantage of lists vs vectors?**
   - ❖ No random access (O(n) to find element N); pointer overhead per node; poor cache locality.
5. **Would you use a vector or list for a task queue? Why?**
   - ❖ List, push_front and pop_back are both O(1), making it ideal for FIFO queue semantics.

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip**: Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia