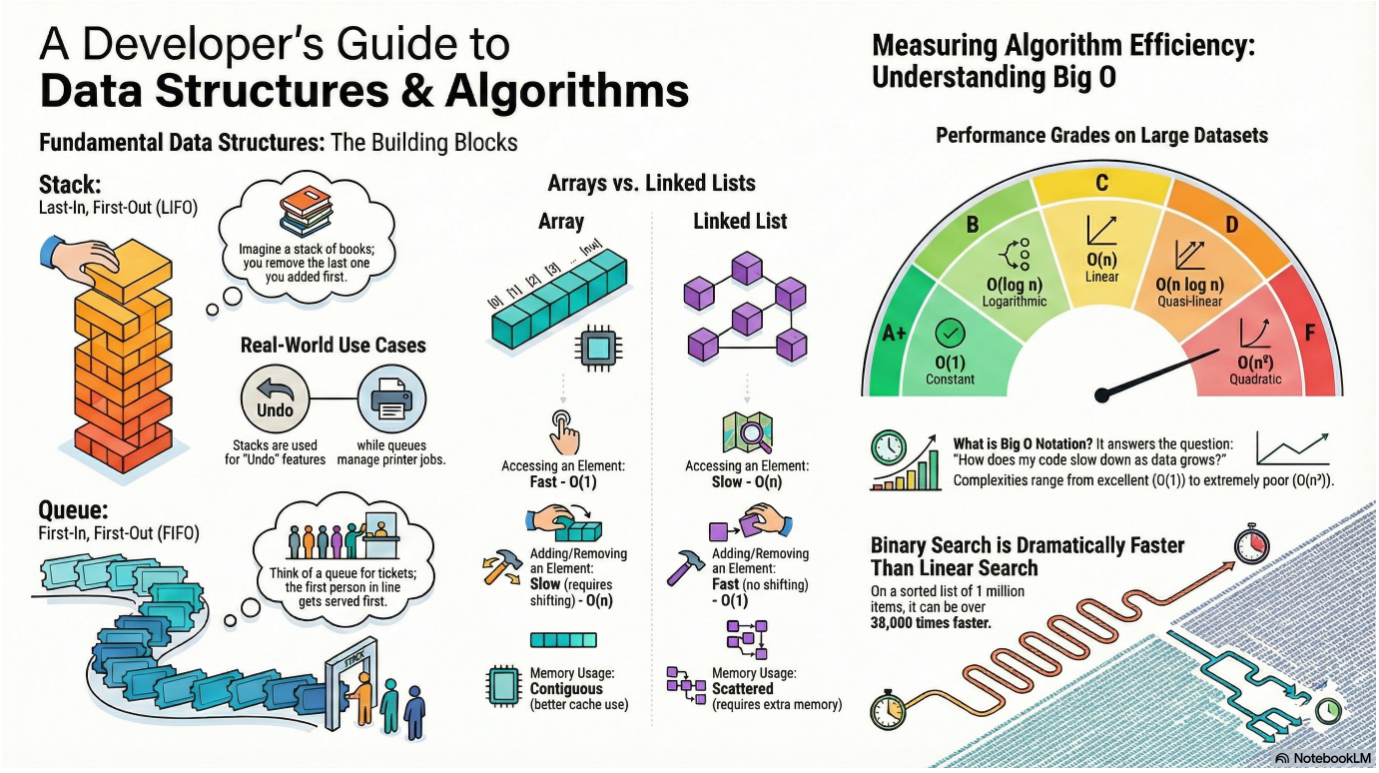
kintsugi-stack-dsa-cpp: STL | C++ STL | Standard Template Library

> "All Power is Within You; You Can Do Anything And Everything" -Swami Vivekananda

- Author: Kintsugi-Programmer



> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

C++ STL Decision Matrix & Analysis

Table 1: C++ STL Container Decision Guide

Container	Primary Use Case / Strength	Key Trade-off / When to Avoid	Underlying Data Structure
`std::vector`	Used when an array-like container is desired but needs to be **dynamic**. Provides **fast random access**. **Fast insertion/deletion at the end** (`push_back`/`pop_back`) with **Big O(1) amortized** time complexity.	**Slow insertion/deletion in the middle** (worst case **Big O(N)**). Reallocation copies (when `size` equals `capacity`) are costly. Iterators, pointers, and references may become invalid upon modification.	Dynamic Array
`std::list`	Efficient operations from **both ends** (`push_front`, `pop_front`, `push_back`, `pop_back`). Front operations are efficient.	**Random Access is NOT Possible** (e.g., `l` causes an error).	Doubly Linked List (DLL)
`std::deque`	Acts as a **Double Ended Queue**. Supports efficient operations from both ends. **Random Access is Possible** (e.g., `d`).	While implementation is different from `std::list`, it is similar in offering dual-ended access.	Dynamic Arrays

Container	Primary Use Case / Strength	Key Trade-off / When to Avoid	Underlying Data Structure
`std::stack`	Implementing **Last In First Out (LIFO)** behavior. All primary operations (`push`, `top`, `pop`) are **Big O(1)**.	Elements exit in the **reverse order** of insertion. Access is restricted to the `top`.	Container Adapter
`std::queue`	Implementing **First In First Out (FIFO)** behavior. All primary operations (`push`, `front`, `pop`) are **Big O(1)**.	Access is restricted to the `front` and `back`. Does not have a `top()` function.	Container Adapter
`std::priority_queue`	Specialized structure for **sorting elements**. Guarantees the highest priority element (largest value by default) is always at the `top` in **Big O(1)** time.	Insertion (`push`) and removal (`pop`) are logarithmic operations: **Big O(log N)**.	Max Heap or Min Heap (Complete Binary Tree)
`std::map`	Stores **Key-Value PAIRS** with **Unique Keys**. Maintains data in **default sorted order** based on keys.	Core operations (`insert`, `erase`, `count`) are slower than unordered containers at **Big O(log N)**.	Self Balancing Tree
`std::unordered_map`	Stores **Key-Value PAIRS** with **Unique Keys**. Used when **highly optimized average performance** is required. Core operations (`insert`, `erase`, `count`) are **Big O(1)** (amortized/average time).	Data is stored in **random order** (unordered, not sorted). Time complexity can reach **Big O(N)** in very rare cases (hash collisions).	(Hashing implementation)
`std::set`	Stores **Unique Values**. Maintains data in **default sorted order**. Supports bounds finding (`lower_bound`/`upper_bound`).	Insertion/deletion is logarithmic at **Big O(log N)**. Cannot access elements using array syntax.	Self Balancing Tree
`std::unordered_set`	Stores **Unique Values** in a **random order**. Core operations (`insert`, `erase`, `count`) are generally **Big O(1)** (amortized time).	**Cannot use `lower_bound()` or `upper_bound()`** because the data is not sorted. Time complexity can reach **Big O(N)** in very rare cases.	(Hashing implementation)

Table 2: C++ Function & Algorithm Analysis

Function / Algorithm	Purpose / Description	Applicable Container(s)	Time Complexity (TC)	Space Complexity (SC)
`vector::push_back`	Inserts data at the last index (end).	`std::vector`, `std::list`, `std::deque`	**Big O(1)** (Amortized constant time for `vector`)	N/A
`vector::emplace_back`	Inserts data at the last index (end), creating the object in place.	`std::vector`, `std::list`, `std::deque`	**Big O(1)** (Amortized constant time for `vector`)	N/A

Function / Algorithm	Purpose / Description	Applicable Container(s)	Time Complexity (TC)	Space Complexity (SC)
`vector::pop_back`	Deletes the data at the last index (end).	`std::vector`, `std::list`, `std::deque`	**Big O(1)**	N/A
`vector::front` / `back`	Returns the first (`front`) or last (`back`) element.	`std::vector`, `std::list`, `std::deque`	**Big O(1)**	N/A
`vector::insert` (middle)	Inserts an element at a position other than the end.	`std::vector`, `std::list`, `std::deque`	**Big O(N)** (Worst case)	N/A
`vector::erase` (middle/range)	Erases an element or a range of elements.	`std::vector`, `std::list`, `std::deque`	**Big O(N)** (Worst case)	N/A
`stack::push` / `queue::push`	Adds an element (to top/rear).	`std::stack`, `std::queue`	**Big O(1)**	N/A
`stack::pop` / `queue::pop`	Removes an element (from top/front).	`std::stack`, `std::queue`	**Big O(1)**	N/A
`priority_queue::push`	Adds an element, maintaining heap property.	`std::priority_queue`	**Big O(log N)**	N/A
`priority_queue::pop`	Removes the top (highest priority) element.	`std::priority_queue`	**Big O(log N)**	N/A
`priority_queue::top`	Checks which element is currently at the top.	`std::priority_queue`	**Big O(1)**	N/A
`map::insert` / `emplace`	Inserts a new key-value pair.	`std::map`, `std::set`	**Big O(log N)**	N/A
`map::erase`	Deletes a key-value pair.	`std::map`, `std::set`	**Big O(log N)**	N/A

Function / Algorithm	Purpose / Description	Applicable Container(s)	Time Complexity (TC)	Space Complexity (SC)
`map::count`	Returns 1 if key found, 0 otherwise.	`std::map`, `std::set`	**Big O(log N)**	N/A
`unordered_map::insert`/`erase`/`count`	Insertion, deletion, or checking key existence.	`std::unordered_map`, `std::unordered_set`	**Big O(1)** (Amortized time)	N/A
`std::sort`	Sorts elements within a range (ascending by default).	Arrays, `std::vector`, `std::vector<std::pair>`	N/A (Note: Used instead of complex algorithms like Merge/Quick/Heap Sort)	N/A
`std::reverse`	Reverses the order of elements within a range.	Containers that support iterators (e.g., `std::vector`)	N/A	N/A
`std::binary_search`	Checks if a target element exists in a **sorted range**.	Sorted Ranges (e.g., `std::vector`)	N/A	N/A

Table of Contents

Introduction to C++ STL

The C++ Standard Template Library (STL) consists of templates of Optimised Quality Codes that enhance the coding experience and improve coding time.

Importance of STL in Practical Coding: STL is frequently used in practical coding, especially in placements, internships, online assessments, or coding contests.

- **Example 1: Sorting:** Instead of writing a sorting algorithm from scratch (like Merge Sort, Quick Sort, or Heap Sort), one can simply use the STL sort function to sort data in a single line.
- **Example 2: BFS (Breadth-First Search):** To apply BFS, a queue is needed. Instead of spending time implementing a queue from scratch, one can simply use the C++ STL queue container, saving a lot of time.

Components of STL: STL is composed of four main parts:

1. **Containers:** The most important part, heavily used in Data Structures and Algorithms (DSA). (Main focus of the lecture).
2. **Algorithms:** The second most important part.
3. **Iterators:** Covered alongside containers and algorithms.
4. **Functors:** Also covered, used occasionally with containers and algorithms.

```
// C++ STL
```

Containers: Sequential Containers

- These Containers
- Stores Data Sequentially

```
// Sequential Containers - Store Data Sequentially
```

1. Vector

A vector is very similar to a C++ array.

Vector vs. Array

Feature	C++ Array	C++ Vector
Size	Constant	Dynamic (Can resize at runtime)
Resizing	Size cannot be increased (e.g., 3 to 5) or decreased (e.g., 3 to 2) during runtime.	Can increase or decrease its size at runtime.
Usage	Used when a constant-size structure is sufficient.	Used when an array-like container is desired but needs to be dynamic.

Vector Creation and Basics

To use vectors, the vector library must be included.

To create a vector:

```
#include <vector>

// Creating a vector of integers named 'v'
vector<int> v;
```

Initially, a vector created this way has no data stored, so its size is zero.

Key Properties: Size and Capacity.

1. **Size:** The number of elements currently stored in the vector.
2. **Capacity:** The number of elements the vector can currently hold.

Vector Functions (Detailed)

Since Vector is the first container discussed, its functions are studied in detail, as the majority of C++ STL containers use the same functions or similar variations.

Function	Description	Example/Syntax	Time Complexity
`size()`	Returns the number of elements stored.	`vector.size()`	N/A (Used to check size)

Function	Description	Example/Syntax	Time Complexity
`capacity()`	Returns the maximum number of elements the vector can currently hold.	`vector.capacity()`	N/A (Used to check capacity)
`push_back()`	Inserts data at the last index (end) of the vector. Assumes the data object is already created.	`vector.push_back(1)`	Big O(1)
`emplace_back()`	Inserts data at the last index (end) of the vector. Creates the object in place at the time of insertion.	`vector.emplace_back(6)`	Big O(1)
`pop_back()`	Deletes the data at the last index of the vector.	`vector.pop_back()`	Big O(1)
`front()`	Returns the first element of the vector.	`vector.front()`	Big O(1)
`back()`	Returns the last element of the vector.	`vector.back()`	Big O(1)

Dynamic Growth and Capacity

> "When using push_back() on a std::vector in C++, does the vector always append the new element to its existing memory, or does it sometimes allocate a new block of memory, copy the old elements, and then add the new one?"

When you `push_back()`/`emplace_back()` an element to a `std::vector` in C++, its behavior depends on its **size** versus its **capacity**.

- `size()`: The number of elements currently in the vector.
- `capacity()`: The total number of elements the vector can hold in its currently allocated block of memory before it needs to get more.

Here is the process:

1. **If `size` is less than `capacity`:** The vector has pre-allocated empty space. It simply appends the new element to the end of its **existing memory block**. No new vector is created, and no copying of old elements occurs. This is a very fast, $O(1)$ operation.

```
// Vector has space
// Size: 3, Capacity: 5
[10][20][30][__][__]

vec.push_back(40);

// New element added in-place
// Size: 4, Capacity: 5
[10][20][30][40][__]
```

2. **If `size` equals `capacity`:** The vector is full and must reallocate. It:
    - Allocates a **new, larger** contiguous block of memory.
    - **Copies** (or moves) all the elements from the old memory block to the new one.
    - Deallocates the old memory block.
    - Adds the new element to the end of the new memory block.

```
// Vector is full
// Size: 3, Capacity: 3
[10][20][30]

vec.push_back(40);
```

```
// Reallocates, copies, and adds
// Size: 4, Capacity: 6 (approx)
[10][20][30][40][__][__]
(New memory location)
```

This reallocation is an $O(n)$ operation (where n is the number of elements), but it happens infrequently, giving `push_back()` an *amortized* constant time complexity, $O(1)$.

The way vectors grow in memory is important.

1. Initially, the vector size is 0.
2. When data is inserted (e.g., using `push_back(1)`), the vector size increases to 1.
3. When a second element is inserted (e.g., `push_back(2)`), the size needs to increase.
4. **Instead of just increasing the size by one box, a new vector is created in memory with double the current size (capacity)**.
5. The old data is copied into the new, larger vector.
6. The new data is then inserted.
7. If the vector size was 2, the next capacity created will be 4.

```
[] sz=0, cap=0
↓
[1] sz=1, cap=1 ,new vector arr created in memory,copy-paste old stuff with new element,
not append at last
↓
[1,2] sz=2, cap=2 ,new vector arr created in memory,copy-paste old stuff with new element,
not append at last
↓
[1,2,3] sz=3, cap=3 ,new vector arr created in memory,copy-paste old stuff with new
element, not append at last
```

Example of Size and Capacity: If a vector has three elements inserted (Size = 3), and its current capacity is 4 (because capacity doubles when filled up), then `vector.size()` returns 3 and `vector.capacity()` returns 4.

Code Example See how `capacity()` changes:

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

int main() {
    std::vector<int> vec;
    std::cout << "i | Size | Capacity\n";
    std::cout << "---------------------\n";

    for (int i = 0; i < 10; ++i) {
        vec.push_back(i);
        std::cout << i << " | " << vec.size()
                  << "    | " << vec.capacity() << "\n";
    }
    return 0;
}
```

Example Output: (Note: Exact capacity growth is compiler-dependent)

```
i | Size | Capacity
---------------------
0 | 1    | 1
```

```
1 | 2    | 2
2 | 3    | 4  <-- Reallocation!
3 | 4    | 4
4 | 5    | 8  <-- Reallocation!
5 | 6    | 8
6 | 7    | 8
7 | 8    | 8
8 | 9    | 16 <-- Reallocation!
9 | 10   | 16
```

Accessing Elements

Elements can be accessed similar to arrays, as vectors work like arrays.

1. **Square Bracket Notation (Array Syntax):** Generally preferred due to familiarity.

```
// Accessing value at index 2
vector[2] // returns the value at index 2
```

2. `.at()` **Function:** An alternative method.

```
vector.at(2) // returns the value at index 2
```

Note: Both methods return the same value.

Initialization Methods

Vectors can be initialized in several ways:

1. **Initialising with specific values (like arrays):**

```
vector<int> v = {1, 2, 3, 4, 5};
```

2. **Initialising with a defined size and a specific value:** This syntax is important when working with Dynamic Programming (DP), specifically for memoization or tabulation, where DP vectors/matrices need to be initialized with initial values.

```
// Creates a vector of size 3, where every value is 10
vector<int> v(3, 10);

// Creates a vector of size 10, where every value is -1
vector<int> v(10, -1);
```

3. **Initialising one vector using another vector's values:** Vector 1's elements are passed inside the parentheses (constructor) of Vector 2.

```
vector<int> vector1 = {1, 2, 3, 4, 5};
vector<int> vector2(vector1); // Initializes vector2 with values from vector1
```

Costly Operations: `erase()` **and** `insert()`

The `erase()` and `insert()` functions are considered costly functions.

- `push_back`, `pop_back`, `emplace_back`, `front`, and `back` work in Big O(1) time complexity (constant time).
- `erase` and `insert` work in **Big O(N)** time complexity in the worst case. This is because they perform operations in the middle of the vector.

1. **`erase()` Function:** Used to erase an element or a range of elements. `pop_back()` is used to erase the last element.

- **Erasing a single element:** Requires passing the **position** of the element, not the index. This position is passed using an **iterator**.
    - To erase the element at index 0: `vector.erase(vector.begin())`.
    - `vector.begin()` returns the position (like a pointer/memory location) of the beginning of the vector (index 0).
    - To erase the element at index 2: `vector.erase(vector.begin() + 2)`.
- **Erasing a range of elements:** Requires passing a `start` iterator and an `end` iterator.
    - The `start` position is **included**, but the `end` position is **not included** in the deletion range.
    - Example: To delete elements at indices 1 and 2, the start would be `vector.begin() + 1` and the end would be `vector.begin() + 3`.
    - `erase()` changes the **size** of the vector but leaves the **capacity** unchanged.

2. **`insert()` Function:** Used to insert an element at a position other than the end (unlike `push_back`).

- **Syntax:** Requires passing the position (iterator) where insertion should occur, followed by the value to insert.

```
// Insert 100 at index 2
vector.insert(vector.begin() + 2, 100);
```

Elements after the insertion point are pushed to the right.

> Vectors are contiguous-memory arrays. This means insert and erase in the middle are slow because all subsequent elements must be shifted

> Iterator Invalidation: insert or erase invalidates all iterators, pointers, and references at or after the point of modification. If the vector has to reallocate memory (grows larger than its capacity), all iterators to it become invalid.

Other Utility Functions

- **`clear()`:** Deletes all elements in the vector, resulting in an empty vector.
    - After `clear()`, the **size** becomes zero, but the **internal capacity** remains the same.
- **`empty()`:** Checks if the vector is empty, returning a boolean value (1 for true, 0 for false).

> rbegin() (Reverse Begin): Returns a reverse iterator pointing to the last element of the container.

> rend() (Reverse End): Returns a reverse iterator pointing to the theoretical element before the first element of the container.

> This reverse_iterator is an adapter. It wraps a normal (forward) iterator, which we call its base iterator, and reverses its behavior:

> - Movement: Incrementing (++) the reverse_iterator actually decrements (--) its base iterator.

> - Dereferencing (*): This is the tricky part. A reverse_iterator does not point to the same element as its base iterator. It always points to the element one position before its base iterator.

> - Rule: *r is (conceptually) equivalent to *(base() - 1)

> rbegin() is implemented as std::reverse_iterator(container.end()).

> - Its base iterator is end().

> - When you dereference rbegin(), you get *(end() - 1), which is the last element.

> rend() is implemented as std::reverse_iterator(container.begin()).

> - Its base iterator is begin().

> - The loop stops when the reverse iterator equals rend(). This happens when its base iterator equals begin(), meaning you've processed the first element and are now "past the beginning."

Code

```
// Vector
// Dynamic array
// used when size is unknown or changes
// #include<vector>
vector<int> v;// size 0, cap 0
cout<<v.size()<<endl;// size //NA TC
//0
cout<<v.capacity()<<endl;// capacity //NA TC
//0
v.push_back(0);//push_back// O(1) TC
v.emplace_back(1);//emplace_back//O(1) TC
cout<<v.capacity()<<endl;
//2
v.pop_back();//pop_back//O(1) TC
cout<<v.capacity()<<endl;
//2
cout<<v.size()<<endl;
//1
v.push_back(1);
v.push_back(2);
cout<<v.front()<<endl;// front//O(1) TC
// 0
cout<<v.back()<<endl;// back//O(1) TC
// 2
// Dynamic Growth and Capacity
vector<int> v1 = {1,2,3};
cout<<v1.size()<<"\t"<<v1.capacity()<<endl;
// 3    3
v1.pop_back();
v1.pop_back();
cout<<v1.size()<<"\t"<<v1.capacity()<<endl;
// 1    3
// even we remove the elements, still capacity remains same
v1.push_back(1); // simple append, size<=capacity
v1.push_back(1); // simple append. size=capacity
v1.push_back(1); // size exceeds capacity,new capacity needed, new mem allocation , copy
old with new element in this new mem
cout<<v1.size()<<"\t"<<v1.capacity()<<endl;
// 4    6
// just to add one more element, new memory took of double of size, old copied and new
element got appended
v1.pop_back();
cout<<v1[2]<<"\t"<<v1.at(2)<<endl;
// 1    1
// Initialization ways
vector<int> v2 = {1,2,2};
vector<int> v3(3,-1); // {-1,-1,-1}
vector<int> v4(v3); // {-1,-1,-1}
```

```
// vector::insert()
// Big O(N) TC
// let v = {1,2,2}
v = {1,2,2};
// Insert a single value: v.insert(iterator_position, value)
// Insert multiple copies of a value: v.insert(iterator_position, count, value)
// Insert a range of values: v.insert(iterator_position, range_begin, range_end

v.insert(v.begin(),5); // at begin
// v is now {5, 1, 2, 2}
// We inserted 5 at the beginning.
v.insert(v.begin()+2,9); // at other index
// v is now {5, 1, 9, 2, 2}
// We inserted 9 at index 2 (before the 3rd element).

v.insert(v.end(),3,10); // at end + multiple copies of value
// v is now {5, 1, 9, 2, 2, 10, 10, 10}

vector<int> other = {10,20};
v.insert(v.begin()+1,other.begin(),other.end());
// v is now {5, 10, 20, 1, 9, 2, 2, 10, 10, 10}

// vector::erase()
// Big O(N) TC
// Erase a single element: v.erase(iterator_position) // Crucially, it returns an iterator
to the element after the one erased.
// Erase a range of elements: v.erase(range_begin, range_end) // The range is [begin,
end). It erases up to but not including end.

v.erase(v.begin());
// v is now { 10, 20, 1, 9, 2, 2, 10, 10, 10}

v.erase(v.begin()+1,v.end());
// v is now { 10 }

// clear() :  Deletes all elements in the vector, resulting in an empty vector. After
clear(), the size becomes zero, but the internal capacity remains the same.
// empty() : Checks if the vector is empty, returning a boolean value (1 for true, 0 for
false).
cout<<v.empty()<<endl; //0
v.clear();
cout<<v.empty()<<endl;//1

// traverse
for ( auto i : v2 ){
    cout<<i<<"\t";
}cout<<endl;
// 1       2       2

// Example: std::vector
std::vector<int> vec = {10, 20, 30};

// vec.begin() -> points to 10
// vec.end()   -> points to memory *after* 30

// it = vec.rbegin()
//   - base iterator is vec.end()
//   - *it returns *(vec.end() - 1), which is 30
// ++it
//   - base iterator becomes (vec.end() - 1), which points to 30
//   - *it returns *( (vec.end() - 1) - 1 ), which is 20
// ++it
//   - base iterator becomes (vec.end() - 2), which points to 20
```

```
//   - *it returns *( (vec.end() - 2) - 1 ), which is 10
// ++it
//   - base iterator becomes (vec.end() - 3), which is vec.begin()
//   - it now equals vec.rend(). Loop terminates.

// rbegin(), rend()
```

2. List

List is also a sequential container, like vectors.

Implementation: Lists are internally implemented as a **Doubly Linked List(DLL)**.

```
List (doubly linked list)
list<int> l = {1, 2, 3};


     PF -->   ====================   <-- push_back()
                    1 , 2 , 3
     PF -->   ====================   <-- pop_back()
```

Key Operations: Because Lists behave like a doubly linked list, operations can be performed efficiently from both ends:

- `push_back()`
- `push_front()`
- `pop_back()`
- `pop_front()`
- `emplace_back()`
- `emplace_front()`

Creation and Example: To use lists, the list library must be included.

```
#include <list>

list<int> l;
l.push_back(1);
l.push_back(2);
l.push_front(3);
l.push_front(5);
// Order becomes: 5, 3, 1, 2
```

Function Availability: Functions such as `size`, `erase`, `clear`, `begin`,`end`, `rbegin`, `rend`, `insert`, `front`,`empty` and `back` are available and work similarly to how they do in Vectors.

Major Differences between Vector and List:

Feature	Vector	List
Internal Implementation	Dynamic Array	Doubly Linked List
Front Operations	Less efficient/Costly (due to array structure)	Available and efficient
Random Access	Possible (e.g., `v`)	Not Possible (e.g., `l` will cause an error)

Code

```

```

```
// List
// Sequential container like vectors
// implemented as doubly linked lists
// Random Access is NOT Possible (e.g., d[2])
// #include<list>
list<int> l;
l = {1,2};
l.push_back(3);
l.emplace_back(4);
l.push_front(0);
l.emplace_front(-1);
l.pop_back();
l.pop_front();

// traverse
for ( auto i : l ) {
    cout<<i<<"\t";
}cout<<endl;
// 0       1       2       3

l.erase(l.begin());
cout<<l.size()<<endl; // size()
// 3
cout<<l.front()<<endl; // front()
// 1
cout<<l.back()<<endl; // back()
// 3
l.clear();// clear()
cout<<l.empty()<<endl; // empty()
// 1
l.insert(l.begin(),0); // same as push_front, emplace_front
cout<<l.front()<<endl;
// 0
l.insert(l.end(),4); // same as push_back, emplace_back
cout<<l.back()<<endl;
// 4
// Insert a single value: v.insert(iterator_position, value)
// Insert multiple copies of a value: v.insert(iterator_position, count, value)
// Insert a range of values: v.insert(iterator_position, range_begin, range_end
// rbegin(), rend()
```

3. Deque (Double Ended Queue)

- Sequential Container

Implementation: Deque are internally implemented as a **Dynamic Arrays**.

```
     PF -->   ======================   <-- push_back()
                    1 , 2 , 3
     PF -->   ======================   <-- pop_back()
```

Deque (d q) is implemented as a **Double Ended Queue**. It is very similar to a List.

Nomenclature Clarity: There are two common uses of "D Q" in programming:

1. **Dq (Double Ended Queue):** Spelled d e q u e. This is a sequence container.
2. **Dequeue:** Spelled d e q u e u e. This is a generic word used for popping or deleting elements from a normal queue.

> Deque != Dequeue

Key Operations: Deques also have two ends and support operations from both ends:

- `push_back` and `push_front`
- `pop_back` and `pop_front`
- `emplace_back` and `emplace_front`

Creation: To use deque, the deque library must be included.

```
#include <deque>

deque<int> d;
// Operations similar to List/Vector can be performed
```

Major Difference between List and Deque:

Feature	List	Deque
Internal Implementation	Doubly Linked List	Dynamic Arrays
Random Access	Not possible	**Possible** (e.g., `d[2]`)

```
#include <iostream>
#include <deque>
using namespace std;

int main() {
    deque<int> d = {1, 2, 3, 4, 5};

    for(int val : d) {
        cout << val << " ";
    }
    cout << endl;
    cout<<d[2]<<endl;// can't done by Lists

    return 0;
}
```

- size, erase, clear, begin, end, rbegin, rend, insert, front, back, empty

Code

```
// Deque (Double Ended Queue)
// Deque != Dequeue
// Implemented as Dynamic Arrays
// Random Access is Possible (e.g., d[2])
// #include<deque>
deque<int> d;
d = {1,2,3,4,5};
for ( auto val : d ) { cout<<val<< "\t"; }
// 1       2       3       4       5
cout<<endl;
cout<<d.front()<<endl; // front()
// 1
cout<<d.back()<<endl; // back()
// 5
d.push_back(0);
d.emplace_back(0);
```

```
d.push_front(0);
d.emplace_front(0);
d.pop_back();
d.pop_front();
d.insert(d.begin(),-1);
d.insert(d.end(), -1);
d.erase(d.begin(),d.begin()+2);
d.erase(d.end(),d.end()-2);
for ( int i=0; i<d.size(); i++) // size()
{cout<<d[i]<< "\t";} // Random Access d[i] // can't done by Lists
cout<<endl;
// Insert a single value: v.insert(iterator_position, value)
// Insert multiple copies of a value: v.insert(iterator_position, count, value)
// Insert a range of values: v.insert(iterator_position, range_begin, range_end
// Erase a single element: v.erase(iterator_position) // Crucially, it returns an iterator
to the element after the one erased.
// Erase a range of elements: v.erase(range_begin, range_end) // The range is [begin,
end). It erases up to but not including end.
d.clear();// clear()
cout<<d.empty()<<endl;// empty()
// rbegin(), rend()
```

Utility Structure: Pair

- Pairs are a special structure
- part of the C++ Utility Library.

Definition: A Pair is used to group any two values (Value 1 and Value 2) together. The data types of these two values can be different or the same.

Creation and Access: We do not need to include a specific library to create a Pair.

```
#include <utility> // <-- This is the one you need

// Creating a pair of integers
pair<int, int> p = {1, 5};

// Creating a pair of string and integer
pair<string, int> p = {"BaliBhai", 5};
```

To access the values:

- First value: `p.first`
- Second value: `p.second`

Complex Pair Structures:

1. **Pair of Pairs:** A pair where one (or both) of the components is itself a pair.

```
// First value is an integer (1). Second value is a pair (char 'a', int 3)
pair<int, pair<char, int>> p = {1, {'a', 3}};

// Accessing elements:
p.first;        // 1
p.second.first; // 'a'
// no just p.second
p.second.second;// 3
```

2. **Vector of Pairs:** A vector where each element is a pair.

```
vector<pair<int, int>> v;
```

```
vector<pair<int, int>> vec = {{1, 2}, {2, 3}, {3, 4}};

for(pair<int, int> p : vec) {// MAKE SURE TEMP PAIR VAR TOO HAVE SAMEDATATYPE AS IN
VECT
    cout << p.first << " " << p.second << endl;
}
for(auto p : vec) {
    cout << p.first << " " << p.second << endl;
}
```

Push Back vs. Emplace Back (Revisited)

The difference between `push_back` and `emplace_back` becomes clear when dealing with complex objects like Pairs.

```
// Push Back vs. Emplace Back (Revisited)
// Emplace Back : Object is created BY FUNC at the time of insertion (in-place).
// Push Back : Object is created BY US before the function call.
// difference b/w push_back and emplace_back
// push_back simpler
// emplace_back faster
vector<pair<int, int>> vec = {{1, 2}, {2, 3}, {3, 4}};
vec.push_back({4,5}); // just insert, assumes we did make relevant object(pair,etc.)
vec.emplace_back(5,6); // in-place objects create at the time of insertion
```

Function	Requirement	Action	Time of Object Creation
`push_back`	Requires the object (e.g., the Pair) to be already created.	Inserts the existing object.	Object is created *before* the function call.
`emplace_back`	Requires only the individual component values (e.g., 4 and 5).	Automatically creates the Pair in place.	Object is created *at the time of insertion* (in-place).

If only individual values (e.g., `4` and `5`) are passed to `push_back`, it will result in an error because it cannot convert two individual values into a Pair automatically.

Code

```
// Pairs
// Utility Structure
// part of the C++ Utility Library.
//  A Pair is used to group any two values (Value 1 and Value 2) together. The data types
of these two values can be different or the same.
// #include <utility> // <-- This is the one you need
pair<int,int> p = {1,5}; // NO MORE THAN 2 upper objects
pair<string,int> p1 = {"bali",100};
cout<<p1.first<<"\t";// .first
cout<<p1.second<<endl;// .second
// .first & .second is only access methods it can have'\
// bali    100
```

```cpp
// Complex Pair Structures: Pair of Pairs
pair<int, pair<string,int> > p2 = { 100, {"bali" ,100}};
cout<<p2.first<<"\t";//100
// cout<<p2.second; // NOOOOO
cout<<p2.second.first<<"\t";
cout<<p2.second.second<<endl;
// 100     bali    100

// Complex Pair Structures: Vector of Pairs
vector<pair<string,int> > p3 = {{"bali" ,100}, {"bhaskar",98}, {"bhati",99}};
for (auto i : p3) {cout<<i.first<<i.second<<endl;}
// bali100
// bhaskar98
// bhati99
for ( pair<string,int> pp : p3 ) { cout<<pp.first<<pp.second<<endl; }
// MAKE SURE TEMP PAIR VAR TOO HAVE SAMEDATATYPE AS IN VECT
// bali100
// bhaskar98
// bhati99

// Push Back vs. Emplace Back (Revisited)
// Emplace Back : Object is created BY FUNC at the time of insertion (in-place).
// Push Back : Object is created BY US before the function call.
// difference b/w push_back and emplace_back
// push_back simpler
// emplace_back faster
p3.push_back({"Rijusmit",99});
p3.emplace_back("Kintsugi-Programmer",100);
```

Containers: Non-Sequential (Container Adapters)

```cpp
// Containers: Non-Sequential (Container Adapters)
// Stack and Queues are used is short calculations of numebers orgs(order) , due to O(1)
operations of push ,pop, top/front
// Priority Queue are specialized data structure & helps in sorting elements
```

4. Stack

Stack is a **Last In First Out (LIFO)** data structure.

Visualization: A stack is visualized as a structure where elements are placed one on top of the other (like a stack of books or plates). The element added last is the first to be removed.

```
   Top
   ↓↑
|      |
+-------+
|   c   |
+-------+
|   b   |
+-------+
|   a   |
+-------+
```

Key Components: The top location is called the **Top**. Elements are added and removed *only* from the Top.

Creation and Functions: To use stacks, the stack library must be included.

```
#include <stack>

stack<int> s;
```

Function	Description	Example
`push()`	Adds an element to the top of the stack.	`s.push(1)`
`emplace()`	Similar working to `push()`.	`s.emplace(2)`
`top()`	Checks which element is currently at the top.	`s.top()`
`pop()`	Removes (deletes) the element currently at the top.	`s.pop()`
`empty()`	Checks if the stack is empty (returns boolean).	`!s.empty()` (used in loops)
`size()`	Returns the current size of the stack.	`s.size()`
`swap()`	Swaps the values of two stacks of the same type.	`s.swap(s2)`

Time Complexity (Stack Operations): `push`, `top`, and `pop` operations all have a time complexity of **Big O(1)**.

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s;

    s.push(1);
    s.push(2);
    s.push(3);

    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }

    cout << endl;
    return 0;
}
```

```
Input: 1 2 3
Output: 3 2 1 // Reversed
```

> Order of Element Entered in Stack will NOT be Same as Order of Elements Exited from Stack. It will be Reversed

Code

```
// Stack
// Time Complexity (Stack Operations): push, top, and pop operations all have a time
complexity of Big O(1).
// Containers: Non-Sequential (Container Adapters)
// LIFO DS
// #include<stack>
```

```
stack<int> s;
s.push(1); // push() // TC Big O(1)
s.push(2);
s.push(3);
s.emplace(-1); // emplace()
s.pop(); // pop() // last=top removed // TC Big O(1)
cout<<s.top()<<endl; // top() // TC Big O(1)
// 3
cout<<s.empty()<<endl; // empty()
// 0
cout<<s.size()<<endl; // size()
// 3
stack<int> s1;
s1.swap(s); // swap()
cout<<s.size()<<endl;
// 0
cout<<s1.size()<<endl;
// 3
s1.swap(s);
// dequeueing way traversal
while( ! s.empty() ) { cout<<s.top()<<" "; s.pop() ; }
cout<<endl;
// 3 2 1
// order of insert in stack != order of taking out of stack
// order of insert in stack , order of taking out of stack are reverse of each other
// Stack and Queues are used is short calculations of numebers orgs(order) , due to O(1)
operations of push ,pop, top/front
```

5. Queue

```
Dequeue <--- [  A  |  B  |  C  ] <--- Enqueue
       (Front)              (Rear)
```

Queue is a **First In First Out (FIFO)** data structure.

Visualization: A queue can be imagined like a line of people at a bank or ticket counter.

Key Components:

1. **Front:** Elements are removed (popped) from the front.
2. **Rear/Back:** Elements are inserted (pushed) from the back/rear.

Creation and Functions: To use queues, the queue library must be included.

```
#include <queue>

queue<int> q;
```

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;

    q.push(1);
    q.push(2);
```

```
    q.push(3);

    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }

    cout << endl;
    return 0;
}
```

Function	Description	Example
`push()/emplace()`	Adds an element to the rear/back.	`q.push(1)`
`front()`	Checks which element is currently at the front.	`q.front()`
`pop()`	Removes the element at the front.	`q.pop()`
`size()`, `empty()`, `swap()`	Work exactly the same way as in Stack.	N/A

Push and Pop Order: In a queue, elements are popped in the same order they were pushed (e.g., Push 1, 2, 3 results in Pop 1, 2, 3).

> Order of Element Entered in Queue will be Same as Order of Elements Exited from Queue. It will be of Same Order

Time Complexity (Queue Operations): `push`, `front`, and `pop` operations all have a time complexity of **Big O(1)**.

> NO top(), in Queue, it's front()

Code

```
// Queue
// Time Complexity (Queue Operations): push, front, and pop operations all have a time
complexity of Big O(1)
// FIFO DS
// #include<queue>
queue<int> q;
q.push(-1);
q.push(1);
q.emplace(2); // emplace()
q.push(3); // push() // pushed at rear/back // TC Big O(1)
cout<<q.front()<<endl; // front() // TC Big O(1)
q.pop(); // pop() // first removed // TC Big O(1)
cout<<q.front()<<endl;// NO top(), in Queue, it's front()
cout<<q.empty()<<endl; // empty()
// 0
cout<<q.size()<<endl; // size()
// 3
queue<int> q1;
q1.swap(q); // swap()
// dequeueing way traversal
while ( ! q1.empty() ) { cout<< q1.front() << "\t";q1.pop(); }
cout<<endl;
// 1       2       3
// order of insert in queue = order of taking out of queue
// Stack and Queues are used is short calculations of numebers orgs(order) , due to O(1)
operations of push ,pop, top/front
```

6. Priority Queue

- specialized data structure.
- helps in sorting elements

Internal Implementation: A Priority Queue internally uses a **Max Heap** or a **Min Heap** structure. A Max Heap or Min Heap is a complete binary tree.

Heap Properties:

1. **Max Heap:** The **largest element** is always at the top of the tree(default).
2. **Min Heap:** The **minimum element** is always at the top(default).

Visualization and Priority: Priority Queue is often visualized like a stack, with a **Top** where the highest priority element is stored.

By default, the highest priority element is the one with the **largest value** (i.e., it implements a Max Heap).

Default Creation (Max Heap - Largest value has highest priority): To use Priority Queue, we only need to include the `queue` header.

```
#include <queue>
priority_queue<int> pq;
```

When elements are pushed (e.g., 5, 3, 10, 4), 10 will be at the top. When elements are popped, they come out in **sorted order** (Largest first: 10, 5, 4, 3).

Reverse Order Creation (Min Heap - Smallest value has highest priority): To make the smaller element come to the top, a reverse order Priority Queue must be built. This requires passing a **Functor** (or Comparator).

```
// Syntax for Min Heap
priority_queue<int, vector<int>, greater<int>> pq;
```

The `greater<int>` is a Functor (a function object) that acts as a Comparator, defining the logic for comparison (in this case, reversing the default Max Heap order).

- push, emplace, top, pop, size, empty

Time Complexity (Priority Queue Operations): Since Priority Queue internally builds a tree structure, operations have logarithmic time complexity:

- `push` and `pop` functions: **Big O(log N)**.
- `top` function: **Big O(1)**.

ASCII art visualizations for the two main types of priority queues.

- Max-Priority Queue

The **largest** item (highest value) always has the highest priority and is at the front.

```
(Items In) --> 5, 20, 8

              [======]
              [ (20) ] <-- Top (Max value)
              [  (8) ]
              [  (5) ]
              [______]
```

- Min-Priority Queue

The **smallest** item (lowest value) always has the highest priority and is at the front.

```
(Items In) --> 42, 10, 3

              [======]
              [  (3) ] <-- Top (Min value)
              [ (42) ]
              [ (10) ]
              [______]
```

Comparison Visualization Here is a comparison showing the *same insertion* into both types of priority queues.

Let's assume both queues already contain the numbers `25` and `10`.

- Initial State

The queues are already ordered based on their rules.

```
    MAX-Priority Queue           MIN-Priority Queue
    (Top = Largest Value)        (Top = Smallest Value)
    [==================]           [==================]
    [       (25)       ] <-- Top   [       (10)       ] <-- Top
    [       (10)       ]           [       (25)       ]
    [__________________]           [__________________]
```

- The Action: Insert(15)

Now, let's insert the number **15** into both queues.

```
         (15)                         (15)
           |                            |
           v                            v
    [==================]           [==================]
    [       (25)       ]           [       (10)       ]
    [       (10)       ]           [       (25)       ]
    [__________________]           [__________________]
```

- Final State

The number `15` finds its correct logical place in each queue.

- In the **Max-Queue**, `15` is "less important" than `25`, so it goes behind it.
- In the **Min-Queue**, `15` is "less important" than `10`, so it goes behind it.

```
    MAX-Priority Queue           MIN-Priority Queue
    (15 slots in)                (15 slots in)
    [==================]           [==================]
    [       (25)       ] <-- Top   [       (10)       ] <-- Top
    [       (15)       ] <-- New   [       (15)       ] <-- New
    [       (10)       ]           [       (25)       ]
    [__________________]           [__________________]
```

As you can see, even though the same number (15) was added, its final position in the "line" is different because the *rules of priority* (largest vs. smallest) are opposites.

Code

```
// Priority Queue
// Priority Queue are specialized data structure & helps in sorting elements
// A Priority Queue internally uses a Max Heap or a Min Heap structure. A Max Heap or Min
Heap is a complete binary tree.
// Max Heap: The largest element is always at the top of the tree(default).
// Min Heap: The minimum element is always at the top(default).
// By default, the highest priority element is the one with the largest value (i.e., it
implements a Max Heap).
// Visualization and Priority: Priority Queue is often visualized like a stack, with a Top
where the highest priority element is stored.
// #include <queue>

// Max-Priority Queue
// The largest item (highest value) always has the highest priority and is at the front.
// (Items In) --> 5, 20, 8
//               [======]
//               [ (20) ] <-- Top (Max value)
//               [  (8) ]
//               [  (5) ]
//               [______]
priority_queue<int> pq; // Default Creation (Max Heap - Largest value has highest
priority)
pq.push(1); // push() // TC Big O(log N)
pq.emplace(2); // emplace()
cout<<pq.top()<<endl; // top() // TC Big O(1)
// 2
pq.pop(); // pop()
cout<<pq.size()<<endl; // size()
// 1
cout<<pq.empty()<<endl; // empty()
// 0

// Min-Priority Queue
// The smallest item (lowest value) always has the highest priority and is at the front.
// (Items In) --> 42, 10, 3
//               [======]
//               [  (3) ] <-- Top (Min value)
//               [ (42) ]
//               [ (10) ]
//               [______]
priority_queue<int , vector<int>, greater<int> > pq1; // Reverse Order Creation (Min Heap
- Smallest value has highest priority)
// The greater<int> is a Functor (a function object) that acts as a Comparator, defining
the logic for comparison (in this case, reversing the default Max Heap order).
pq1.push(1); // push() // TC Big O(log N)
pq1.emplace(2); // emplace()
cout<<pq1.top()<<endl; // top() // TC Big O(1)
// 1
pq1.pop(); // pop() // TC Big O(log N)
cout<<pq1.size()<<endl; // size()
// 1
cout<<pq1.empty()<<endl; // empty()
// 0
```

Containers: Associative Containers

- Tabular Organisation

```
// Containers: Associative Containers
// - Tabular Organisation
// map
// multimap
// set
// multiset
// unordered set
```

7. Map

Map is used to store **Key-Value PAIRS**.

Visualization: A map can be visualized as a table with two columns: Key and Value.

Key Characteristics:

1. **Unique Keys:** All keys must be unique (no duplicate keys are allowed). (Keys are like unique roll numbers or employee IDs).
2. **Duplicate Values Allowed:** Corresponding values can be duplicated (e.g., two different employee IDs can have the same name).
3. **Default Sorted Order:** Map stores data in **ascending (sorted) order** based on the keys, even Lexicographically .

Creation and Insertion:

```
#include <map>

// Key type is String, Value type is Integer
map<string, int> m;
```

Insertion/Modification (Square Bracket Notation): This is the generally used syntax for insertion or modification.

```
m["TV"] = 50;
m["Laptop"] = 100;
```

- If the key exists (e.g., "TV"), the existing value is overwritten.
- If the key does not exist, a new key-value pair is inserted.

```
#include <iostream> // Added: For cout and endl
#include <map>
#include <string>   // Added: Good practice for std::string

using namespace std;

int main() {
    map<string, int> m;

    m["tv"] = 100;
    m["laptop"] = 100;
    m["headphones"] = 50;
    m["tablet"] = 120;
    m["watch"] = 50;

    m.emplace("camera", 25);
```

```
    // Fixed: The syntax is "for (auto p : m)"
    // Using "const auto& p" is more efficient as it avoids copying.
    for (const auto& p : m) {
        cout << p.first << " " << p.second << endl;
    }

    cout<<m.count("camera")<<endl;
    cout<<m["laptop"]<<endl;
    m.erase("tv");

    if (m.find("camera") != m.end()) {
        cout << "found\n";
    } else {
        cout << "not found\n";
    }

    return 0;
}
```

Map Functions

Function	Description	Time Complexity (Normal Map)
`insert()`	Inserts a new key-value pair. Requires passing a complete Pair object.	Big O(log N)
`emplace()`	Inserts a new key-value pair. Requires only passing individual key and value; creates the object in place.	Big O(log N)
`count()`	Returns the number of keys matching the input key. (Since keys are unique in a normal map, this returns 1 if found, 0 otherwise).	Big O(log N)
`erase()`	Deletes the key-value pair associated with the input key.	Big O(log N)
`find()`	Searches for a key. Returns an iterator to the element if found. If not found, returns `m.end()`.	N/A
`size()`, `empty()`	Work similar to other containers.	N/A

Map Time Complexity: Normal `map` is implemented as a **Self Balancing Tree**. Therefore, functions like `insert`, `erase`, and `count` work in **Big O(log N)** time complexity.

Code

```
// map
// Map is used to store Key-Value PAIRS.
// - Unique Keys
// - Duplicate values Allowed
// - Default Sorted Order Asc
// #include<map>
map<string,int> m;
m["apple"]=100; // insertion
m["apple"]=100; // modification
m.emplace("banana",125); // insertion // emplace() // TC Big O(log N)
m.insert({"momo",1000});// insert() // TC Big O(log N)

// traversal
// Fixed: The syntax is "for (auto p : m)"
// Using "const auto& p" is more efficient as it avoids copying.
for ( const auto& p : m) {
```

```
    cout
    << p.first
    << p.second
    << endl ;
}
// apple100
// banana125
cout<<m.count("banana")<<endl; // count() // TC Big O(log N)
// 1
cout<<m["apple"]<<endl; // random access
// 100
m.erase("apple"); // erase() // TC Big O(log N)
if ( m.find("camera") != m.end() ){ // find(), Searches for a key. Returns an iterator to
the element if found. If not found, returns m.end().
    cout<<"found\n";
} else {
    cout<<"not found\n";
}
// not found
cout<<m.size()<<endl; // size()
cout<<m.empty()<<endl; // empty()
```

8. Multi Map

```
multimap<string, int> m;
```

A Multi Map allows **multiple (duplicate) keys** to be stored. All other functionality remains the same as a normal map.

Key Difference in Usage: Because duplicate keys are allowed, we **cannot use the square bracket notation** (e.g., `m["TV"] = 100`) for insertion. We must use `insert` or `emplace`.

Erase in Multi Map: If `m.erase("TV")` is called, it removes *all* instances of the key "TV". To delete only one instance, an **iterator**(actual memory locations) must be passed to the `erase` function.

```
// Find iterator for the first instance of "TV" and erase it
m.erase(m.find("TV"));
```

Code

```
// multimap
// - Duplicate Keys Allowed, thus Random Access is not allowed
// - Duplicate values Allowed
// - Default Sorted Order Asc
// #include<map>
multimap<string,int> m1;
// m1["TV"] = 100 // NO
m1.emplace("TV",100000);
m1.erase("TV");// Erase Way 1 // it removes all instances
m1.insert({"TV",100000});
m1.erase(m1 .find("TV")); // Erase Way 2 // Find iterator for the first instance of "TV"
and erase it
cout<< m1.count("guava") << endl; // count // TC O(1)
// 0
if ( m1.find("apple") != m1.end() ){ // find(), Searches for a key. Returns an iterator to
the element if found. If not found, returns m.end().
    cout<<"found\n";
```

```
} else {
    cout<<"not found\n";
}
// not found
cout<<m1.size()<<endl; // size()
// 0
cout<<m1.empty()<<endl; // empty()
// 1
m1.insert({"TV",100000});
m1.insert({"TV",100001});
m1.insert({"TV",100002});
// traversal
// Fixed: The syntax is "for (auto p : m)"
// Using "const auto& p" is more efficient as it avoids copying.
for ( const auto& p : m1) {
    cout
    << p.first
    << p.second
    << endl ;
}
// TV100000
// TV100001
// TV100002
```

9. Unordered Map

- most imp.
- most freq.

An Unordered Map stores data in a **random order** (unordered). The keys are not sorted.

Key Characteristics:

1. **Unordered Data:** Data is stored in a random order.
2. **Unique Keys:** Keys cannot be duplicated.

Creation:

```
#include <unordered_map>
unordered_map<string, int> um;
```

Unordered Map Time Complexity: Unordered Map is implemented in a special way (using hashing, covered in later chapters). This implementation makes its functions highly optimized.

- `insert`, `erase`, and `count` functions: **Big O(1)** (Constant time, amortized time/average time).
- *Note: In very rare cases (due to hash collisions), time complexity can reach Big O(N), but for practical use and time complexity calculation, Big O(1) is assumed.*

Code

```
// unordered_map
// most imp.
// most freq.
// - Unique Keys
// - Duplicate values Allowed
// - Unordered, No Sorted Order
// - insert, erase, and count functions: Big O(1) (Constant time, amortized time/average
time).
```

```
// Note: In very rare cases (due to hash collisions), time complexity can reach Big O(N),
but for practical use and time complexity calculation, Big O(1) is assumed.
// #include<unordered_map>
unordered_map<string, int> um;
um["apple"] = 100; // insertion
um["apple"] = 100; // modification
um.emplace("banana",100); // insertion // emplace() // TC O(1)
um.insert({"guava",100}); // insertion // emplace() // TC O(1)
um.erase("guava"); // erase() // TC O(1)
cout<< um.count("guava") << endl; // count // TC O(1)
// 0
// traversal
// Fixed: The syntax is "for (auto p : m)"
// Using "const auto& p" is more efficient as it avoids copying.
for ( const auto& p : um) {
    cout
    << p.first
    << p.second
    << endl ;
}
// banana100
// apple100
if ( um.find("apple") != um.end() ){ // find(), Searches for a key. Returns an iterator to
the element if found. If not found, returns m.end().
    cout<<"found\n";
} else {
    cout<<"not found\n";
}
// not found
cout<<um.size()<<endl; // size()
// 2
cout<<um.empty()<<endl; // empty()
// 0
```

10. Set

Set is a container that stores **unique values**.

- insert, emplace, count, erase, find, size, empty, erase

Key Characteristics:

1. **Unique Values:** Duplicate values are ignored and stored only once.
2. **Default Sorted Order:** Unique values are stored in a **sorted order**.

Creation and Insertion: To use sets, the set library must be included.

```
#include <set>

set<int> s;

s.insert(1);
s.insert(2);
s.insert(3);
// If s.insert(1) is called again, the size remains 3, ignoring the duplicate. // Size
wont effect on duplicate
s.insert(3);
s.insert(4);
s.insert(5);
s.insert(6);
```

```
cout << "lower bound = " << *(s.lower_bound(4)) << endl;//4
cout << "upper bound = " << *(s.upper_bound(4)) << endl;//5


for(auto val : s) {
    cout << val << " ";
}
cout << endl;

// access ? set not supposed to access element
cout<<*(st.find(3))<<endl;// access way , it gives iterator, and dereference to value
itself
// 3
```

Set Time Complexity: Sets are also internally implemented using a **Self Balancing Tree**, similar to maps.

- `insert`, `count`, and `erase` functions: **Big O(log N)** time complexity.

Set Functions: `lower_bound()` **and** `upper_bound()`

These functions return an iterator.

```
MultiSet Example to Show `lower_bound()` and `upper_bound()` :
---

            LB          UB
           /  \        /  \
          ( b  )      ( c  )
    a   a   b   b   b   c   c   d
    0   1   2   3   4   5   6   7
          ^           ^
          |           |
       (idx 2)     (idx 5)


lower_bound('b') ----> should NOT be less than key
                       (points to first 'b' at index 2)

upper_bound('b') ----> greater than key
                       (points to 'c' at index 5)
```

1. `lower_bound()`**:** Sets a condition that the returned value's Iterator must be *at least* the key passed.

- If the key exists, it returns the key's value.
- If the key does not exist, it returns the value that is **immediately greater** than the key.
- If no value greater than or equal to the key exists, it returns `set.end()` (which dereferences to 0).

Condition: Value should **NOT be less than** the key.

2. `upper_bound()`**:** Sets a condition that the returned value's Iterator must be **greater than** the key passed.

- It returns the first value that is strictly greater than the key.
- If the key is `b`, the upper bound cannot be `b` itself; it must be the first value after all instances of `b`.

Condition: Value should be **Greater than** the key.

> upper_bound is NOT vice-versa of lower_bound, it's completely different stuff

Multi Set and Unordered Set

1. **Multi Set:** Stores multiple (duplicate) elements. The elements are stored in ascending order.

```
multiset<int> s;
```

2. **Unordered Set:** Stores unique elements but in a **random order(Not Sorted)**.

```
unordered_set<int> s;
```

Unordered Set Time Complexity: Similar to Unordered Map, functions like `insert`, `count`, and `erase` generally work in **Big O(1)** (amortized time).(O(n) at v.rare cases)

Constraint on Bounds: The concepts of `lower_bound()` and `upper_bound()` cannot be used with **Unordered Set** because they require the data to be arranged in a sorted order.

> Unordered Map and Unordered Set are Very Optimised DS and USed Frequently in DSA Prob. Solving

Code

```
// Set
// Set is a container that stores unique values.
// - Unique Values
// - Default Sorted Order
// internally implemented using a Self Balancing Tree
// #include <set>
set<int> st;
st.insert(1); // insert() // Big O(log N) time complexity
st.insert(2);
st.insert(3);
st.insert(4);
    st.insert(4); // Duplicates IGNORED
// upper_bound is NOT vice-versa of lower_bound, it's completely different stuff
cout << "lower bound = " << *(st.lower_bound(1)) << endl; // lower_bound() // should NOT
be less than key
// lower bound = 1
cout << "upper bound = " << *(st.upper_bound(1)) << endl; // upper_bound() // greater than
key
// upper bound = 2
// traverse
for(auto val : st) {
    cout << val << " ";
}cout << endl;
// access ? set not supposed to access element
cout<<*(st.find(3))<<endl;// access way , it gives iterator, and dereference to value
itself
// 3
cout<<st.count(4)<<endl;// count() // Big O(log N) time complexity // 1 obv
st.erase(4);// erase() // Big O(log N) time complexity

// multiset
// - Duplicate Values Allows
// - Default Sorted Order ASC
multiset<int> ms;

// unordered_set
// - Unique Values
// - Random Order(Not Sorted)
// - insert, count, and erase generally Big O(1) (amortized time).( O(n) at v.rare cases)
```

```
unordered_set<int> us;
// Unordered Map and Unordered Set are Very Optimised DS and USed Frequently in DSA Prob. Solving
```

Iterators -- actual memory locations

Iterators are frequently used to loop over containers. They can be imagined like pointers.

Vector Iterators

Vectors provide two main functions that return iterators: `begin` and `end`.

1. **`vector.begin()`:** Returns an iterator pointing to the beginning of the vector (index 0).
    - If dereferenced (`*vector.begin()`), it returns the value stored at index 0.
2. **`vector.end()`:** This is a common misconception point.
    - `vector.end()` **DOES NOT** point to the last index (index n-1).
    - It points to the memory location **immediately following** the last index, which often contains garbage data.
    - Dereferencing `vector.end()` will likely return a random value or 0.

```
[   array   ][other memory...]
+---+---+---+---+
| 0 | 1 | 2 |   |...
+---+---+---+---+
  ^           ^
  |           |
begin()     end()
```

Looping Using Iterators

Iterators are used to write loops over vectors.

Creating an Iterator: An iterator is created by defining its special type.

```
vector<int>::iterator it;
```

The scope is specified using `::` (colon colon) to indicate that the iterator is associated with the `vector<int>` type.

Forward Loop Logic:

1. Initialize the iterator: Start at `vector.begin()`.
2. Condition: Loop continues as long as the iterator is **NOT equal to** `vector.end()`.
3. Increment: Use `it++` to move to the next position.
4. Access Value: Dereference the iterator (`*it`) to print the value.

```
// Example of a Forward Loop
for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    cout << (*it) << endl;
}
```

Reverse Iterators

Reverse iterators allow traversing the container in a backward fashion.

1. `vector.rbegin()` **(Reverse Begin):** Points to the **last index** of the vector.
2. `vector.rend()` **(Reverse End):** Points to the memory location **just before** the first index (index 0).

```
+---+---+---+---+
|   | 0 | 1 | 2 |
+---+---+---+---+
  ^           ^
  |           |
rend()      rbegin()
```

Reverse Loop Logic: The loop starts at `rbegin()` and ends when the iterator equals `rend()`.

```
// Example of a Reverse Loop
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); it++) {
    // Print logic
}
```

The `auto` Keyword

In modern C++, defining the long type signature for iterators (like `vector<int>::iterator`) is not necessary.

The `auto` keyword automatically deduces the type of the variable.

```
// Using auto for a forward iterator
for (auto it = v.begin(); it != v.end(); it++) { ... }

// Using auto for a reverse iterator
for (auto it = v.rbegin(); it != v.rend(); it++) { ... }
```

Code

```
// Iterators -- actual memory locations

// Vector Iterators
vector<int> v5 = {11111,2,3};
cout<< *(v5.begin()) << endl;  // begin() // (*vector.begin()) dereferenced // begin() ->
index=0 element
// 11111
cout<< *(v5.end()-1 ) << endl;  // end() // (*vector.end()) dereferenced // end() ->
NOTindex=n-1lastelement, end()-1 does give index=n-1lastelement
// 3

// Looping Using Iterators
vector<int>::iterator it; // ::iterator
// The scope is specified using `::` (colon colon) to indicate that the iterator is
associated with the `vector<int>` type.
for ( vector<int>::iterator it = v5.begin(); it!=v5.end(); it++) {cout<<(*it)<<"\t";}
cout<<endl;
// 11111      2       3

// Reverse Iterators
// Reverse iterators allow traversing the container in a backward fashion
// -- in reverse iterators goes left to right, ++ goes right to left ( REVERSE OF NORMAL
ITERATORS )
cout<< *(v5.rbegin() ) << endl;// rbegin() -> last element, index = n-1
```

```
// 3
cout<< *(v5.rend()-1 ) << endl;// rend() -> mem. loc. just before index = 0 / first
element, rend()-1 gives First Element / index = 0
// 11111

// auto
// The `auto` keyword automatically deduces the type of the variable.
for ( auto it = v5.begin(); it!=v5.end(); it++ ){cout<<(*it)<<"\t";}
cout<<endl; // 11111      2       3
for ( auto it = v5.rbegin() ; it!=v5.rend(); it++ ){cout<<(*it)<<"\t";}
cout<<endl; // 3       2       11111 // Reverse Order
```

Algorithms

The STL provides numerous algorithms; the most frequently used ones in competitive programming questions are discussed. To use algorithms, the `<algorithm>` header must be included (Implicitly from the usage context).

1. `sort()`

Used for sorting arrays or vectors.

Syntax: Requires passing the starting position and the ending position.

```
// Sorting an array 'arr' of size 'n'
sort(arr, arr + n); // even thou arr (array when called) is itself pointer to arr[0]

// Sorting a vector 'v'
sort(v.begin(), v.end());
```

Note: The ending position passed must be the location just after the last element, as the end position is not included in the sort range.

Default Sorting Order: By default, `sort()` arranges data in **ascending order**.

Sorting in Descending Order (Using a Comparator): To sort in descending (reverse) order, a third optional parameter, a **Comparator** (or Functor), must be passed.

```
// Using the built-in Greater Int comparator
sort(v.begin(), v.end(), greater<int>());
```

Crucially, the parentheses `()` must be used after `greater<int>` or the code will not work.

```
#include <iostream>   // For cout and endl
#include <vector>     // For std::vector
#include <algorithm>  // For std::sort
#include <utility>    // For std::pair
#include <functional> // For std::greater

int main() {
    // --- Example 1: Sorting a C-style array (ascending) ---
    std::cout << "1. Sorting a C-style array (ascending):\n";
    int arr[5] = {3, 5, 1, 8, 2};

    // Sorts the array from arr[0] up to (but not including) arr[5]
    std::sort(arr, arr + 5);
```

```
    for (int val : arr) {
        std::cout << val << " ";
    }
    std::cout << "\n\n"; // Add newlines for separation

    // --- Example 2: Sorting a vector (descending) ---
    std::cout << "2. Sorting a vector (descending):\n";
    std::vector<int> vec = {3, 5, 1, 8, 2};

    // Sorts the vector using a comparator (std::greater) for descending order
    std::sort(vec.begin(), vec.end(), std::greater<int>());

    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << "\n\n";

    // --- Example 3: Sorting a vector of pairs (default ascending) ---
    std::cout << "3. Sorting a vector of pairs (default ascending):\n";
    std::vector<std::pair<int, int>> vec_pair = {{3, 1}, {2, 1}, {7, 1}, {5, 2}};

    // Default sort for pairs:
    // 1. Sorts by the .first element
    // 2. If .first elements are equal, sorts by the .second element
    std::sort(vec_pair.begin(), vec_pair.end());

    for (auto p : vec_pair) {
        std::cout << "(" << p.first << ", " << p.second << ")\n";
    }
    std::cout << std::endl;

    return 0;
}
```

- Output of the combined code:

```
1. Sorting a C-style array (ascending):
1 2 3 5 8

2. Sorting a vector (descending):
8 5 3 2 1

3. Sorting a vector of pairs (default ascending):
(2, 1)
(3, 1)
(5, 2)
(7, 1)
```

Custom Comparators

Custom logic can be defined to sort complex data types, such as Pairs, based on specific criteria (e.g., sorting by the second value).

> Play of just indv. case true/false

Implementation: A custom comparator is a **Boolean function** that returns `true` or `false` based on comparison.

1. The function accepts two instances of the data type being sorted (e.g., `p1` and `p2` for Pairs).

2. It defines the logic for deciding the relative order of `p1` and `p2`.
    - If `p1` should come before `p2` in the desired sorted order, return `true`.
    - Otherwise, return `false`.

Example: Sorting Pairs based on the Second Value (Ascending): The function checks if the second value of `p1` is less than the second value of `p2`. If true, they are already in the correct sorted order, so return `true`.

```
bool compare_second(pair<int, int> p1, pair<int, int> p2) {
    if (p1.second < p2.second) {
        return true;
    }
    // If p1.second is >= p2.second
    return false;
}
// Usage: sort(v.begin(), v.end(), compare_second);
```

Example: Sorting Pairs with Tie-Breaking Logic: If the second values are equal (`p1.second == p2.second`), the sorting should fall back to the first value.

```
bool compare_custom(pair<int, int> p1, pair<int, int> p2) {
    // 1. Sort based on the second parameter (ascending)
    if (p1.second < p2.second) return true;
    if (p1.second > p2.second) return false;

    // 2. Tie-breaking (if seconds are equal): Sort based on the first parameter
(ascending)
    if (p1.first < p2.first) return true;

    // Default case (if p1.first >= p2.first)
    return false;
}
```

```
bool comparator(pair<int, int> p1, pair<int, int> p2) {
    if(p1.second < p2.second) return true;
    if(p1.second > p2.second) return false;

    if(p1.first < p2.first) return true;
    else return false;
}


int main() {
    vector<pair<int, int>> vec = {{3, 1}, {2, 1}, {7, 1}, {5, 2}};

    sort(vec.begin(), vec.end(), comparator);

    for(auto p : vec) {
        cout << p.first << " " << p.second << endl;
    }
}
```

2. `reverse()`

Used to reverse the order of elements within a range.

Syntax: Requires passing the starting and ending iterators (or pointers).

```
reverse(v.begin(), v.end());
```

The reversal can be applied to a specific range (e.g., `vector.begin() + 1` to `vector.begin() + 3`).

3. `next_permutation()` and `prev_permutation()`

These functions rely on the elements being arranged in **lexicographical order** (sorted order, like a dictionary).

1. **`next_permutation()`:** Calculates and modifies the container to the next lexicographically greater permutation.
2. **`prev_permutation()`:** Calculates and modifies the container to the previous lexicographically smaller permutation.

Syntax: Requires passing the beginning and ending iterators (works for strings, vectors, etc.).

```
string s = "ABC";
next_permutation(s.begin(), s.end()); // Changes s to "ACB"
```

```
string :
"abc"

permutations ,lexographical order :
abc
acb
bac
bca
cab
cba

total
= 6
= 3!
```

4. Utility Functions

Function	Description	Example
`max()`	Returns the maximum of two or more values.	`max(6, 5)`
`min()`	Returns the minimum of two or more values.	`min(6, 10)`
`swap()`	Swaps the values of two variables.	`swap(a, b)`
`max_element()`	Finds the maximum element in a range. Returns an **iterator** to that element.	`*max_element(v.begin(), v.end())`
`min_element()`	Finds the minimum element in a range. Returns an **iterator** to that element.	`*min_element(v.begin(), v.end())`

5. `binary_search()`

Used to perform binary search on a **sorted range**.

Functionality: Returns a **boolean value** (`true` or `false`) indicating whether the target element exists.

Syntax:

```

```

```
// v must be sorted
binary_search(v.begin(), v.end(), target_value);
```

6. Built-in Pop Count

These functions are used to **count set bits** (bits whose value is equal to 1) in a number.

- This functionality is typically built into the GCC compiler.
- It is often used in competitive programming but is **not compatible with all C++ standards**, making it less useful in industry practice.

Data Type	Function Name	Usage Example
`int` (Integer)	`__builtin_popcount`	`__builtin_popcount(15)` // returns 4
`long int`	`__builtin_popcountl`	`__builtin_popcountl(n2)` // trailing 'l' stands for long
`long long int`	`__builtin_popcountll`	`__builtin_popcountll(n3)` // trailing 'll' stands for long long

```
eg :
int n = 15;

now int => 4B => 32b
15 => 0000000000.....1111
                    [1111] these bits are set bits => 4 setbits
```

```
int n = 15;
long int n2 = 15;
long long int n3 = 15;


cout << __builtin_popcount(n) << endl;
cout << __builtin_popcountl(n2) << endl;
cout << __builtin_popcountll(n3) << endl;
```

Code

```
// Custom Logic for // Custom Comparators
bool custom_comp(pair<int,int> p1, pair<int,int> p2 ){
    if (p1.second > p2.second) return true;
    if (p1.second < p2.second) return false;
    if (p1.second == p2.second) { if (p1.first > p2.first ) return true;
    else return false; } // be clear of all cases else compliler will give warning/ error
// 1.cpp:8:1: warning: control reaches end of non-void function [-Wreturn-type]
    else return false;
};
// Usage: sort( v.begin(), v.end() , custom_comp ) ;
```

```
// Algorithms
// #include<algorithm>

// sort()
// By default, sort() arranges data in ascending order.
```

```
int arr1[3] = {3,1,2 };
sort(arr1, arr1+3); // sort() array
vector<int> varr1 = {3,1,2};
sort(varr1.begin(), varr1.end()); // sort() vector
sort(varr1.begin(), varr1.end(), greater<int>() ); // sort() DESC order // greater<int>(),
don't forget to put () at end of greater<int>
vector<pair<int,int> > vp = {{1,2},{0,1},{-1,3}};
sort(vp.begin(),vp.end()); // sort() vector of pairs // default ASC
    // Default sort for pairs:
    // 1. Sorts by the .first element
    // 2. If .first elements are equal, sorts by the .second element

// Custom Comparators
// Custom logic can be defined to sort complex data types, such as Pairs, based on
specific criteria (e.g., sorting by the second value).
// Play of just indv. case true/false
sort(vp.begin(),vp.end(), custom_comp );
for(auto p : vp) {
    cout << p.first << " " << p.second << "\t";
}
cout<<endl;
// -1 3    1 2     0 1
// reverse()
reverse(vp.begin(),vp.end()); // reverse the order of elements within a range.
reverse(vp.begin()+1,vp.begin()+2); // specific range

// next_permutation() & prev_permutation()
string ss = "ABC";
int fact = 1;
for ( int i = 1; i<=ss.size(); i++ ){ fact*=i; }
for ( int i =0; i<fact; i++)
{
cout<<ss<<"\t";
next_permutation(ss.begin(),ss.end()); // next_permutation()
}cout<<endl;
// ABC     ACB     BAC     BCA     CAB     CBA
for ( int i =0; i<fact; i++)
{
prev_permutation(ss.begin(),ss.end()); // prev_permutation()
cout<<ss<<"\t";
}cout<<endl;
// CBA     CAB     BCA     BAC     ACB     ABC

// Utility Functions
cout<<max(5,660)<<endl; // max()
// 5
cout<<min(5,-600)<<endl; // min()
// -600
int a=10,b=20;
swap(a,b); // swap()
cout<<a<<b<<endl;
// 2010
vector<int> v6 = {11111,-2,3};
cout<< *(max_element(v6.begin(),v6.end()))<<"\t"; // max_element()
cout<< *(min_element(v6.begin(),v6.end()))<<endl; // min_element()
// 11111   -2

// binary_search()
// Used to perform binary search on a sorted range.
// Functionality: Returns a boolean value (true or false) indicating whether the target
element exists.
cout<<binary_search( v6.begin(), v6.end(), 3)<<endl; //
//1
```

```
cout<<binary_search( v6.begin(), v6.end(), -3)<<endl;
//0

// __builtin_popcount()
// tells how much '1s' required at right hand side when number is represented in binary
// This functionality is typically built into the GCC compiler.
// It is often used in competitive programming but is not compatible with all C++
standards, making it less useful in industry practice.
int nn = 7;
long int nn1 = 15;
long long int nn2 = 15;
cout<< __builtin_popcount(nn) << endl; //3 i.e. 7 woule be  000....111 in binary
cout<< __builtin_popcount(nn1) << endl; //4 i.e. 15 woule be  000....1111 in binary
cout<< __builtin_popcount(nn2) << endl; //4 , in another datatype
```

End-of-File

The kintsugi-stack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with 💚 Kintsugi-Programmer