TEMPLATES

1. What Are Templates?

Definition:

Templates in C++ allow us to write generic programs. Instead of writing the same code for different data types, we write a single function/class to work with any data type**.**

Just like we pass values to functions, templates allow passing data types as parameters.

For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.

C*++ adds two new keywords to support templates: 'template' and 'typename'. The second keyword can always be replaced by the keyword 'class'. It means that we can interchangeably use class and typename keywords."*

📚 Explanation:

- template: declares a template.
- typename or class: defines the placeholder for a data type.
    - These two mean the same thing in templates.

Templates aren't actual code until you use them. When you use a template with a specific type, the compiler generates the code for that type. This is called template instantiation.

🧠 How Does a Template Detect the Data Type?

When you **call a template function or instantiate a template class**, the **compiler analyzes the arguments** you pass and **automatically deduces the type** to substitute in place of the template placeholder (usually T).

🔍 Mechanism: Type Deduction

```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(3, 5);          // Integers
    cout << add(3.5, 4.5);      // Doubles
    cout << add('A', 'B');      // Characters
}
```

What's happening under the hood?

- **When you call add(3, 5), the compiler sees both parameters are int → so it generates:**

```cpp
int add(int a, int b);
```

For `add(3.5, 4.5)` → it generates:

```cpp
double add(double a, double b);
```

For `add('A', 'B')` → it generates:

```cpp
char add(char a, char b);
```

This process is called template instantiation. One logic — many versions generated at compile-time, based on the argument types passed.

Why Do We Use Templates?

Templates solve the problem of code redundancy and type flexibility.

✅ Purpose:

- Avoid writing duplicate code for different data types.
- Enable generic programming (write once, use for all data types).
- Boost code reusability, type safety, and compile-time efficiency

Where Are Templates Used in the Industry?

Templates are core to modern C++ libraries and large-scale systems.

🏢 Real-World Applications:

- Standard Template Library (STL):
  - vector<T>, map<K, V>, stack<T>, etc.
- Generic Algorithms:
  - sort(), find(), binary_search() use templates.
- Enterprise Systems:
  - Financial systems with high-performance data structures.
- Game Engines:

    - Physics/memory optimizations with templated math libraries.
- Embedded Systems:
    - Resource-safe, type-safe implementations with zero-runtime overhead.

Function Templates

- We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

  Syntax:

  cpp Copy Edit

  ```cpp
  template <typename T>
  T functionName(T arg1, T arg2) {
      // function logic
  }
  ```

  Example:

  cpp Copy Edit

  ```cpp
  template <typename T>
  T maxValue(T a, T b) {
      return (a > b) ? a : b;
  }

  maxValue(5, 10);       // int
  maxValue(3.5, 2.1);    // double
  maxValue('a', 'b');    // char
  ```

Class Templates

Class templates like function templates, class templates are useful when a class defines something that is independent of the data type.

Class templates are **blueprints for classes** that operate with **generic data types**. This allows a class to **work with any data type** without rewriting code for each one. Similar to function templates, but applied to entire **class structures**.

When to Use Class Templates?

- When your class logic is type-independent
    - Useful for:

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees

Standard Template Library (STL) – C++

1. What is STL?

STL (Standard Template Library) is a powerful library in C++ that provides generic classes and functions for data structures and algorithms.

It is template-based, meaning it works with any data type.

2. Why STL?

✅ **Reusability**	**Avoid rewriting logic for basic structures (e.g., sorting, searching)**
✅ **Efficiency**	**STL uses well-tested, optimized implementations**
✅ **Flexibility**	**Works with any data type using templates**
✅ **Productivity**	**Reduces development time drastically**

3. Where is STL used?

- **Competitive programming**
- **System-level applications**
- **Data processing & file parsing**
- **Game engines and simulation**
- **Industry software** (e.g., CAD, ML preprocessing)

4. How is STL Structured?

STL is divided into **four main components**:

Component	Purpose
1. **Containers**	Store data (e.g., arrays, lists, stacks)
2. **Algorithms**	Operate on containers (e.g., sort, find)
3. **Iterators**	Traverse containers (like pointers)
4. **Function Objects**	Allow objects to be called like functions

1. **Containers** – *What holds the data?*

🔹 What:

Containers are **data structures** provided by STL to **store and organize** data efficiently.

🔹 Why:

Instead of writing your own classes for arrays, lists, stacks, etc., C++ STL provides **generic**, reusable containers.

🔹 Types:

- **Sequence Containers**: Sequence containers store the data in the linear manner. They are also used to implement container adaptors. Eg: vector, list, deque, Arrays
- **Associative Containers**: Store elements in sorted (binary tree) order based on keys eg: set, map
- **Unordered Containers**: Unordered associative containers store the data in no particular order, but they allow the fastest insertion, deletion and search operations among all the container types in STL. Eg: unordered_map, unordered_set
- **Container Adapters**: The container adapters are the type of STL containers that adapt existing container classes to suit specific needs or requirements. Eg: stack, queue, priority_queue

Container	Best Use Case
vector	When you need dynamic array and fast random access
list	When frequent insertions/deletions are required at any position
deque	Efficient additions/removals from both ends
set	Maintain unique sorted elements
map	Key-value lookups with unique keys
unordered_map	Fast key-value access without sorting
stack	Implement undo/redo, recursive parsing, etc.
queue	Scheduling, task queues
priority_queue	Job/task scheduling where higher priority is served first

2. Algorithms – *What processes the data?*

- **What:**

STL provides **built-in algorithms** like sort, reverse, count, find, etc., to perform operations on containers.

- **Why:**

To avoid rewriting common logic and make code cleaner, reusable, and more efficient.

- **Where:**

Algorithms work on containers through **iterators**, not directly.

Types of Algorithms

Manipulative Algorithms

Manipulative algorithms perform operations that modifies the elements of the given container or rearrange their order.

Eg: copy, fill, transform, swap, replace,reverse, remove etc.

Non-Manipulative Algorithms

Non-manipulating algorithms are the type of algorithms provided by the Standard Template Library (STL) that operate on elements in a range without altering their values or the order of the elements. Eg: max_element, min_element, accumulate, count, find.

3. Iterators – *How to access elements?*

🔹 **What:**

Iterators are **pointers-like objects** used to **navigate containers**.

🔹 **Why:**

They abstract how elements are stored and provide a uniform way to traverse any container.

Container Iterator Functions

- begin(), end()
- rbegin(), rend()
- cbegin(), cend()

In C++ STL, iterators are of 5 types:

1. **Input Iterators**: Input Iterators can be used to read values from a sequence once and only move forward.

   Used In: istream_iterator, single-pass algorithms

   🏞️ *Imagine a one-way railway track where a train can only move forward and read each station's name once.*

   ```cpp
   std::istream_iterator<int> in(std::cin);  // Reads from input
   int x = *in; // Reads value from input
   ```

2. **Output Iterators**: Output Iterators can be used to write values into a sequence once and only move forward.

   Used In: ostream_iterator, outputting values to streams or containers

   🖊️ *Like writing names into a guest book where you can write only once and move ahead without going back.*

   ```cpp
   std::ostream_iterator<int> out(std::cout, " ");
   *out = 10; // Outputs 10 to console
   ```

3. **Forward Iterators**: Forward Iterators combine the features of both input and output iterators.

   Used In: forward_list

   🔁 *Like a walkie-talkie where you can both send and receive, but you can't move backwards.*

```cpp
std::forward_list<int> fl = {1, 2, 3};
std::forward_list<int>::iterator it = fl.begin();
++it; // Move forward
```

4. **Bidirectional Iterators**: Bidirectional Iterators support all operations of forward iterators and additionally can move backward.

   Used In: list, set, map

   🔁 ↔ *Like a two-way street – you can go forward and backward one step at a time.*

```cpp
std::list<int> lst = {10, 20, 30};
auto it = lst.begin();
++it;  // Move forward
--it;  // Move backward
```

5. **Random Access Iterators**: Random Access Iterators support all operations of bidirectional iterators and additionally provide efficient random access to elements.

   Used In: vector, deque, array

   🚀 *Like using an elevator with buttons – you can directly jump to any floor (element).*

```cpp
std::vector<int> v = {10, 20, 30, 40};
std::vector<int>::iterator it = v.begin();
std::cout << it[2]; // Access 3rd element (30) directly
it = it + 3; // Jump to 4th element
```

🧠 **Why Use Iterators?**

Aspect	Without Iterators (manual)	With Iterators (STL)
🔁 **Generic Traversal**	Requires knowing container type and indexing	Abstracts container type — same syntax for all
🧠 **Code Reusability**	Limited, tied to container structure	Highly reusable across all STL containers
📦 **Container Support**	Works easily only with arrays, vectors	Supports lists, sets, maps, deques, etc.
🚀 **Efficiency**	Manual errors like index overflow possible	Safer, consistent bounds and type checking
🔁 **Bi-directional/Random Access**	Complex to implement manually	Built-in iterator types support movement
📋 **STL Algorithm Compatibility**	Hard to plug into sort(), find(), etc.	STL algorithms require iterators

📌 **Real-Life Analogy**

- **Input Iterator:** Scanner that only reads forward.
- **Output Iterator:** Printer that writes line by line.
- **Forward Iterator:** A train going forward and doing tasks at each station.
- **Bidirectional Iterator:** A car on a two-way road.
- **Random Access Iterator:** An elevator where you can jump to any floor instantly.

4. Function Objects (Functors) – *Custom operations*

🔹 **What:**

A **function object** is any object that can be called like a function using () operator.

🔹 **Why:**

To define **custom logic** in STL algorithms (e.g., custom sorting).

🔹 **Header Required**

#include <functional> — This header provides predefined functors in C++ STL.

🔸 Types of Functors

1. Arithmetic Functors

Used for arithmetic operations between operands.

- plus<T>: Returns a + b
- minus<T>: Returns a - b
- multiplies<T>: Returns a * b
- divides<T>: Returns a / b
- modulus<T>: Returns a % b
- negate<T>: Returns -a

2. Relational Functors

Used for comparison between values.

- equal_to<T>: Returns a == b
- not_equal_to<T>: Returns a != b
- greater<T>: Returns a > b
- greater_equal<T>: Returns a >= b
- less<T>: Returns a < b
- less_equal<T>: Returns a <= b

3. Logical Functors

Used for logical operations.

- logical_and<T>: Returns a && b
- logical_or<T>: Returns a || b
- logical_not<T>: Returns !a

4. Bitwise Functors

Used for bit-level operations.

- bit_and<T>: Returns a & b
- bit_or<T>: Returns a | b
- bit_xor<T>: Returns a ^ b

✅ Use Cases

- Used in STL algorithms like sort(), count_if(), accumulate(), etc.
- Useful for applying complex or stateful logic as a callable object.

⚙️ **Advantages**

- Maintain state unlike plain functions.
- Inline and optimize well.
- Versatile and reusable with STL.

⚠️ **Disadvantages**

- Slightly more verbose.
- Initial learning curve due to operator overloading.

Functors provide numerous **benefits in terms of system performance**, **flexibility**, and **maintainability**. When designing large-scale systems, especially in areas like **data processing**, **real-time systems**, **financial applications**, and **game development**, functors enable developers to:

- Implement **custom algorithms** without changing the underlying framework.
- Achieve **high performance** and **low memory overhead** by avoiding function pointers and lambdas.
- Create **reusable, modular code** that can be easily maintained and extended over time.

Vectors in C++

1. Introduction to Vectors

A vector in C++ is a sequence container from the Standard Template Library (STL) that is dynamic in size and allows easy resizing. Vectors are similar to arrays, but with additional functionalities, such as resizing the container and more efficient memory handling.

- Definition: A vector is an array that can grow or shrink in size dynamically at runtime.
- Header File: #include <vector>

2. Syntax for Declaring Vectors

Vectors are declared using the std::vector template, where the type of elements is specified inside angle brackets.

```
std::vector<int> vec;          // An empty vector of integers
std::vector<int> vec = {1, 2, 3};  // A vector with initial values
```

Type of elements: Vectors can store any type of data, like integers, floats, strings, or user-defined types.

3. Operations on Vectors

Here are some of the most common operations you can perform on vectors.

Basic Operations:

- **Push Back**: Adds an element at the end of the vector.

  vec.push_back(10); // Adds 10 to the end of vector.
- **Pop Back**: Removes the last element of the vector.

  vec.pop_back(); // Removes the last element.
- **Size**: Returns the number of elements in the vector.

  int size = vec.size(); // Returns the size of vec
- **Capacity**: Returns the total allocated memory size of the vector.

  int capacity = vec.capacity(); // Returns the capacity of vec
- **Access Elements**: Use [] or at() to access elements.

  int x = vec[0]; // Access the first element

  int y = vec.at(1); // Access the second element
- **Clear**: Removes all elements from the vector.

  vec.clear(); // Removes all elements from vec
- **Resize**: Changes the size of the vector.

  vec.resize(5); // Resizes vec to 5 elements
- **Front & Back**: Access the first and last elements of the vector.

  int front = vec.front(); // First element

  int back = vec.back(); // Last element

Memory Management in Vectors

- Vectors are dynamic and manage memory efficiently. Unlike arrays, vectors can resize as elements are added or removed.
- Vectors allocate memory in chunks to avoid frequent reallocations.
- When the vector grows beyond its capacity, it re-allocates the entire memory block and doubles its capacity to accommodate more elements.

Iterating over Vectors

You can iterate over vectors using loops and iterators.

- **Using Range-Based for Loop**:

```cpp
for (int x : vec) {
    std::cout << x << " ";
}
```

- **Using Iterators**:

```cpp
for (auto it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << " ";
}
```

- **Using Index-based for Loop**:

```cpp
for (int i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << " ";
}
```

Advanced Operations

Insert: Inserts an element at a specific position in the vector.

```cpp
vec.insert(vec.begin() + 1, 100);  // Inserts 100 at the second position
```

Erase: Removes an element from a specific position.

```cpp
vec.erase(vec.begin() + 1);  // Removes the second element
```

Swap: Swaps the contents of two vectors.

```cpp
std::vector<int> vec2 = {10, 20};
vec.swap(vec2);  // Swaps vec and vec2
```

Use Cases of Vectors

Dynamic Storage and Resizing:

- Vectors are great when the number of elements is not known in advance or when the number of elements changes dynamically.

- Example: Storing **user input**, **dynamic datasets**, or **real-time data** that may grow or shrink.

Efficient Insertions and Deletions at the End:

- Vectors allow **constant-time insertion** and **deletion** at the end of the container (push_back and pop_back).
  - Example: **Dynamic queue** where elements are added and removed frequently.

Fast Random Access:

- Vectors provide **constant-time random access** to elements using indices, just like arrays.
  - Example: **Storing large datasets** that need to be frequently accessed by index.

Space Efficiency:

- Vectors automatically handle memory management, providing a balance between performance and memory usage.
  - Example: **Database storage** where each record is stored as an object and the number of records can change over time.

Sorting and Searching:

- Vectors can be easily sorted and searched due to their ability to store elements in contiguous memory locations.
  - Example: **Sorting user records**, **searching through large data collections**.

8. Why Use Vectors?

Advantages:

1. **Dynamic Size**: Vectors grow dynamically, making them more flexible than arrays.
2. **Efficient Memory Management**: They allocate memory in chunks to avoid frequent reallocation.
3. **Easy Access**: Vectors provide constant-time random access to elements.
4. **STL Algorithms**: Vectors can be used seamlessly with C++ STL algorithms like sort(), find(), transform(), etc.
5. **No Need for Reallocation on Push**: Vectors automatically handle memory management, reallocating only when needed.

Disadvantages:

1. **Insertions/Deletions in the Middle**: Inserting or deleting elements in the middle of the vector can be slow (O(n)) due to the need to shift elements.

2. **Overhead for Small Containers**: Vectors may use more memory than arrays when the vector's size is smaller than its capacity.

When to Use Vectors:

- Use vectors when you need **dynamic resizing**, **efficient random access**, or when you are dealing with a collection of elements that can **grow** or **shrink** during the program's execution.
- Vectors are ideal for scenarios where elements are mostly added at the **end** and rarely removed from the middle or front.
- Avoid using vectors for cases where **frequent insertions or deletions at the middle** are required; consider using a **list** for such cases.

Stack in C++

A **stack** is a **container adapter** in C++ that operates on the **LIFO (Last In, First Out)** principle. The **last element inserted** into the stack is the **first one to be removed**. It's like a stack of plates where the last plate you put on top is the first one you take off.

Stacks are used when you need to **reverse order** operations or maintain a history of operations, such as in **undo/redo functionality**, **function calls** (call stack in recursion), or **balancing parentheses** in expressions.

Basic Operations of Stack:

1. push()

- Adds an element to the top of the stack.

2. pop()

- Removes the top element of the stack.

3. top()

- Returns the top element without removing it.

4. empty()

- Returns true if the stack is empty, otherwise returns false.

5. size()

- Returns the number of elements in the stack.

Syntax:

To use stacks in C++, you need to include the **<stack>** header file.

```cpp
#include <stack>   // Include the stack header

stack<Type> stackName;  // Declare a stack where 'Type' is the type of elements in the stack
```

Use Cases of Stack:

1. **Function Call Stack**:
    - Every time a function is called, its information (like return address and local variables) is pushed onto the **call stack**.
    - When a function finishes executing, its information is popped from the stack.
2. **Undo/Redo Operations**:
    - In text editors, stacks are used to keep track of previous operations (e.g., inserting or deleting text).
    - When a user clicks **Undo**, the last operation is popped and undone.
3. **Expression Evaluation**:
    - Stacks are used for evaluating expressions, especially in **infix-to-postfix** or **postfix expression evaluation**.
4. **Balanced Parentheses**:
    - Stacks are commonly used to check if parentheses, curly braces, and brackets are balanced in an expression.

Advantages of Using Stack:

1. **Simple**: Stacks have a very simple and straightforward LIFO structure.
2. **Efficient**: Operations such as push, pop, top, and empty are O(1), which makes stack operations very efficient.
3. **Memory**: Since stacks only need to store the data in a linear fashion, they can be more memory efficient compared to other data structures like queues or lists in specific cases.

Disadvantages of Using Stack:

1. **No Random Access**: Unlike arrays or vectors, you can only access the element at the top of the stack. No random access to other elements.
2. **Limited Operations**: You cannot directly iterate through the stack or modify elements other than the top.

Queue in C++ STL

🧠 Definition:

A Queue is a FIFO (First-In-First-Out) linear data structure. The element inserted first is the one to be removed first, like people standing in a queue at a ticket counter.

C++ provides the queue container adapter in the <queue> header.

🔧 Basic Syntax:

```cpp
#include <queue>

std::queue<datatype> queue_name;
```

🚀 **Core Operations:**

Operation	Function Signature	Description
push()	q.push(val)	Adds an element at the **back** of the queue.
pop()	q.pop()	Removes the **front** element.
front()	q.front()	Accesses the front element.
back()	q.back()	Accesses the last (rear) element.
empty()	q.empty()	Returns true if the queue is empty.
size()	q.size()	Returns number of elements in the queue.

Use Cases in Real Projects:

Use Case	Description
Job/Task Scheduling	**Each task is executed in the order it was received.**

Use Case	Description
CPU Scheduling (Round Robin)	Queues are used to rotate between processes in a fair order.
BFS (Breadth-First Search)	In graphs and trees for level-order traversal.
IO Buffers	Data coming in from keyboard or network gets buffered in a queue.
Message Queues in OS	Used in OS inter-process communication (IPC).
Print Spoolers	Print jobs are queued and served in FIFO order.

💡 Why Use Queues?

- Maintain processing order (FIFO)
- Decouples producer/consumer logic
- Handles streaming data effectively
- Great for scheduling and buffering systems

⚖️ Advantages:

- **FIFO Order** ensures fairness.
- **Simple API** and easy to implement.
- **Efficient**: Constant time insertions/removals.

❌ Disadvantages:

- **No random access** to elements.
- **Only sequential operations** are allowed (push at back, pop from front).

⚙️ When to Use:

When to Use Queues	Instead Use Stack / Array When…
When the **order of processing** matters (FIFO).	When **reversing** or **backtracking**.
When you're building **BFS traversal**.	For **DFS**, use a stack.
Data buffering between systems.	**Direct access** or **random access** needed.

🧠 **Specialized Queues in STL:**

Type	Header	Description
queue<T>	<queue>	Basic FIFO queue
priority_queue<T>	<queue>	Elements are served based on priority
deque<T>	<deque>	Double-ended queue (push/pop from both ends)

Priority Queue

🧠 Definition:

A Priority Queue is a type of queue in which each element is associated with a priority. The element with the highest priority is served before other elements, regardless of the order in which they appear.

C++ STL provides priority_queue in the <queue> header.

⚙️ Syntax:

```cpp
#include <queue>

std::priority_queue<int> pq; // Max-Heap by default
```

Operation	Syntax	Description
push()	pq.push(x)	Inserts element with its priority
pop()	pq.pop()	Removes element with highest priority
top()	pq.top()	Returns element with highest priority
empty()	pq.empty()	Checks if the queue is empty
size()	pq.size()	Number of elements in the queue

💡 **Use Cases:**

Application	Description
Dijkstra's Algorithm	To find shortest path efficiently

Application	Description
*A Search Algorithm**	Uses priority queues to decide next node
Task Scheduling	High priority jobs executed before others
Load Balancing	Serve most loaded or critical task first

⚖️ **Advantages:**

- O(log n) insert and remove
- Efficient for **greedy** algorithms
- Dynamic priority-based processing

❌ **Disadvantages:**

- No random access to elements
- Non-deterministic order for equal priorities

Deque (Double-Ended Queue)

🧠 Definition:

A Deque (Double-Ended Queue) is a sequence container that allows insertion and deletion of elements from both ends—front and back.

STL provides deque in the <deque> header.

⚙️ Syntax:

```cpp
#include <deque>

std::deque<int> dq;
```

✅ **Operations:**

Operation	Syntax	Description
push_back()	dq.push_back(x)	Insert at rear
push_front()	dq.push_front(x)	Insert at front
pop_back()	dq.pop_back()	Remove from rear

Operation	Syntax	Description
pop_front()	dq.pop_front()	Remove from front
front()	dq.front()	Access front element
back()	dq.back()	Access rear element
size()	dq.size()	Current number of elements
empty()	dq.empty()	Checks if deque is empty
at(index)	dq.at(2)	Random access to index

💡 **Use Cases:**

Use Case	Description
Sliding Window Problems	Efficient front and back updates
Palindrome Check	Compare elements from both ends
Undo/Redo Operations	Access elements in both directions
Job Scheduling	Add or remove jobs from both ends

⚖️ **Advantages:**

- **Flexible**: Supports both stack and queue functionalities
- **Efficient** insertions/removals at both ends
- Random access available unlike queue

❌ **Disadvantages:**

- Slightly more memory usage than vector
- Cannot directly control memory allocation strategy