1. Standard Template Library (STL) in C++

This is a collection of pre-written code (templates) that provides common programming data structures and functions like lists, stacks, arrays, sorting algorithms, etc. It's a powerful tool for C++ programmers to efficiently handle data and perform operations on it.

Key components of STL:

- **Containers:** Store collections of objects (vectors, lists, maps, sets, etc.)
- **Algorithms:** Perform operations on containers (sort, search, find, etc.)
- **Iterators:** Access elements in containers
- **Functors:** Objects that behave like functions

1. Containers

Containers are classes that manage collections of objects. They can be classified into several types based on how they store data:

- **Sequence Containers**: These store data in a linear sequence. Examples include:
  - vector: A dynamic array that can grow or shrink in size.
  - deque: A double-ended queue that allows fast insertions and deletions at both ends.
  - list: A doubly-linked list that allows efficient insertions and deletions at any position.
- **Associative Containers**: These store data in a way that allows for fast retrieval based on keys. Examples include:
  - set: A collection of unique elements sorted by key.
  - map: A collection of key-value pairs, where keys are unique.
  - multiset: Similar to set, but allows duplicate elements.
  - multimap: Similar to map, but allows duplicate keys.
- **Unordered Associative Containers**: These store data in a hash table for faster retrieval.
  - unordered_set: A set where elements are stored in an unordered fashion.
  - unordered_map: A map where key-value pairs are stored in an unordered fashion.
  - unordered_multiset: Similar to unordered_set, but allows duplicates.
  - unordered_multimap: Similar to unordered_map, but allows duplicate keys.
- **Container Adapters**: These provide a different interface for existing containers.
- stack: Provides a last-in, first-out (LIFO) data structure.
- queue: Provides a first-in, first-out (FIFO) data structure.
- priority_queue: Provides a priority-based queue where the highest (or lowest) priority element is always at the front.

2. Iterators

Iterators are objects that point to elements within a container and allow traversal of the container's elements. They are similar to pointers but can be more flexible. Iterators come in various types:

- **Input Iterators**: Allow reading data in a single pass.
- **Output Iterators**: Allow writing data in a single pass.
- **Forward Iterators**: Allow multiple passes in a forward direction.
- **Bidirectional Iterators**: Allow traversal in both forward and backward directions.
- **Random Access Iterators**: Allow access to any element in constant time.

3. Algorithms

STL provides a wide range of algorithms that can be applied to containers. These algorithms are generic and work with any container that provides the required iterators. Examples include:

- **Sorting**: sort(), stable_sort()
- **Searching**: find(), binary_search()
- **Modification**: fill(), copy(), remove()
- **Reordering**: reverse(), shuffle()

4. Function Objects (Functors)

Function objects are objects that can be called as if they were functions. They are used in algorithms to specify custom operations. For example, you can create a custom comparator for sorting.

5. Allocators

Allocators handle memory management for containers. They define how memory is allocated and deallocated for container elements. The default allocator is allocator, but you can create custom allocators if needed.

Tutorial: Understanding vector in C++ STL

1. What is vector?

vector is a sequence container that encapsulates dynamic size arrays. It can grow or shrink as needed and provides random access to its elements.

2. Key Characteristics

- **Dynamic Size**: Unlike arrays, vectors can automatically adjust their size.
- **Random Access**: You can access elements using an index.
- **Efficient**: Vectors are efficient for adding or removing elements at the end.
- **Contiguous Memory**: Elements are stored in a contiguous block of memory.

3. Basic Operations with vector

3.1. Creating a Vector

You can create a vector with or without initial values.

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

int main() {
   // Create an empty vector of integers
   vector<int> vec1;

   // Create a vector with 5 elements, all initialized to 0
   vector<int> vec2(5);

   // Create a vector with 5 elements, all initialized to 10
   vector<int> vec3(5, 10);

   // Create a vector with initial values
   vector<int> vec4 = {1, 2, 3, 4, 5};

   return 0;
}
```

3.2. Adding Elements

You can add elements using push_back() or emplace_back().

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

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

   // Add an element to the end
   vec.push_back(4); // vec = {1, 2, 3, 4}

   // Add an element using emplace_back (constructs in place)
   vec.emplace_back(5); // vec = {1, 2, 3, 4, 5}

   return 0;
}
```

 push_back() may involve copying or moving an object, which can be less efficient.

 emplace_back() constructs the object directly in the container, avoiding unnecessary copies or moves.

3.3. Accessing Elements

You can access elements using operator[] or at().

```
-------------------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>

int main() {
   vector<int> vec = {10, 20, 30};

   // Access elements using operator[]
   cout << "Element at index 1: " << vec[1] << endl;

   // Access elements using at()
   cout << "Element at index 2: " << vec.at(2) << endl;

   // Use front() and back() to access the first and last elements
   cout << "First element: " << vec.front() << endl;
   cout << "Last element: " << vec.back() << endl;

   return 0;
}
```

3.4. Removing Elements

You can remove elements using pop_back(), erase(), or clear().

```
-------------------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>

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

   // Remove the last element
   vec.pop_back(); // vec = {1, 2, 3, 4}

   // Remove an element at a specific position
   vec.erase(vec.begin() + 1); // vec = {1, 3, 4}

   // Remove all elements
   vec.clear(); // vec = {}

   return 0;
}
```

3.5. Iterating Through a Vector
You can use loops or iterators to traverse a vector.

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

int main() {
   vector<int> vec = {10, 20, 30, 40};

   // Using range-based for loop
   cout << "Using range-based for loop:" << endl;
   for (int value : vec) {
      cout << value << " ";
   }
   cout << endl;

   // Using iterator
   cout << "Using iterator:" << endl;
   for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
      cout << *it << " ";
   }
   cout << endl;

   return 0;
}
```

4. Important Functions and Properties

- **size()**: Returns the number of elements.
- **empty()**: Checks if the vector is empty.
- **reserve()**: Requests a change in capacity.
- **capacity()**: Returns the number of elements that can be held without reallocating.

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

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

   cout << "Size: " << vec.size() << endl;
   cout << "Capacity: " << vec.capacity() << endl;
   cout << "Is empty: " << boolalpha << vec.empty() << endl;

   // Reserve space for 10 elements
   vec.reserve(10);
   cout << "New Capacity: " << vec.capacity() << endl;

   return 0;
}
```

5. Summary

- **vector** is a dynamic array that can grow and shrink as needed.
- It supports efficient access and modification of elements.

- Use push_back(), pop_back(), erase(), and other functions to manipulate the vector.
- Iterate through vectors using loops or iterators.

Tutorial: Ways to Copy a Vector in C++
In C++, vectors provide several ways to copy their contents into another vector. Here are six methods to do this, each demonstrated with a simple example.

1. Iterative Method
In this method, you manually copy elements from one vector to another using a loop.

Code Example:

```
------------------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>

int main() {
   // Initializing the original vector
   vector<int> vect1{1, 2, 3, 4};

   // Creating a new vector
   vector<int> vect2;

   // Copying elements using a loop
   for (int i = 0; i < vect1.size(); i++) {
      vect2.push_back(vect1[i]);
   }

   // Displaying the elements of both vectors
   cout << "Old vector elements are: ";
   for (int i = 0; i < vect1.size(); i++)
      cout << vect1[i] << " ";
   cout << endl;

   cout << "New vector elements are: ";
   for (int i = 0; i < vect2.size(); i++)
      cout << vect2[i] << " ";
   cout << endl;

   return 0;
}
```

Output:

```
------------------------------------------------------------------------------------------------------
Old vector elements are: 1 2 3 4
New vector elements are: 1 2 3 4
```

2. Using Assignment Operator
You can use the assignment operator (=) to copy the contents of one vector to another.
Code Example:

```
------------------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>

int main() {
   // Initializing the original vector
   vector<int> vect1{1, 2, 3, 4};

   // Creating a new vector and copying using assignment
   vector<int> vect2 = vect1;

   // Displaying the elements of both vectors
   cout << "Old vector elements are: ";
   for (int i = 0; i < vect1.size(); i++)
      cout << vect1[i] << " ";
   cout << endl;

   cout << "New vector elements are: ";
   for (int i = 0; i < vect2.size(); i++)
      cout << vect2[i] << " ";
   cout << endl;


   return 0;
}
```

Output:

```
------------------------------------------------------------------------------------------------------
Old vector elements are: 1 2 3 4
New vector elements are: 1 2 3 4
```

3. Using Copy Constructor
You can copy a vector by passing it to the constructor of another vector.
Code Example:

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

int main() {
   // Initializing the original vector
   vector<int> vect1{1, 2, 3, 4};

   // Creating a new vector using the copy constructor
   vector<int> vect2(vect1);

   // Displaying the elements of both vectors
   cout << "Old vector elements are: ";
   for (int i = 0; i < vect1.size(); i++)
      cout << vect1[i] << " ";
   cout << endl;

   cout << "New vector elements are: ";
   for (int i = 0; i < vect2.size(); i++)
      cout << vect2[i] << " ";
   cout << endl;


   return 0;
}
```

Output:

```
Old vector elements are: 1 2 3 4
New vector elements are: 1 2 3 4
```

4. Using copy with back_inserter
You can use the copy function along with back_inserter to copy vector elements.
Code Example:

```
------------------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
   // Initializing the original vector
   vector<int> vect1{1, 2, 3, 4};

   // Creating a new vector
   vector<int> vect2;

   // Copying elements using copy and back_inserter
   copy(vect1.begin(), vect1.end(), back_inserter(vect2));

   // Displaying the elements of both vectors
   cout << "Old vector elements are: ";
   for (int i = 0; i < vect1.size(); i++)
      cout << vect1[i] << " ";
   cout << endl;

   cout << "New vector elements are: ";
   for (int i = 0; i < vect2.size(); i++)
      cout << vect2[i] << " ";
   cout << endl;


   return 0;
}
```

Output:

```
------------------------------------------------------------------------------------------------------
Old vector elements are: 1 2 3 4
New vector elements are: 1 2 3 4
```

Understanding list in C++

list is a container in the C++ Standard Template Library (STL) that implements a doubly linked list. Unlike vector, which stores elements contiguously in memory, list allows for non-contiguous storage. This makes insertion and deletion operations efficient, but traversal is slower compared to vectors.

Basic Syntax

To use list, include the <list> header:

```
#include <list>
```

Declaring and Initializing a List

You can declare a list like this:

```
list<int> myList;
```

You can also initialize it with values:

```
list<int> myList{10, 20, 30, 40};
```

Basic Operations

Here are some common operations you can perform on a list:

1. **Adding Elements**
    - push_front(value): Adds an element to the beginning.
    - push_back(value): Adds an element to the end.

```
myList.push_front(5);  // Adds 5 at the beginning
myList.push_back(50);  // Adds 50 at the end
```

2. **Removing Elements**
    - pop_front(): Removes the first element.
    - pop_back(): Removes the last element.

```
myList.pop_front();  // Removes the first element
myList.pop_back();   // Removes the last element
```

3. **Accessing Elements**
    - front(): Returns the first element.
    - back(): Returns the last element.

```
int firstElement = myList.front();
int lastElement = myList.back();
```

4. **Iterating Through the List**

Use iterators to loop through the list:

```
for (list<int>::iterator it = myList.begin(); it != myList.end(); ++it) {
   cout << *it << " ";
}
```

Alternatively, use a range-based for loop:

```
for (int value : myList) {
   cout << value << " ";
}
```

5. **Other Useful Functions**
   - size(): Returns the number of elements.
   - empty(): Checks if the list is empty.
   - insert(position, value): Inserts a value before the specified position.
   - erase(position): Removes the element at the specified position.
   - reverse(): Reverses the list.
   - sort(): Sorts the list in ascending order.

```
myList.insert(myList.begin(), 15);  // Insert 15 at the beginning
myList.erase(myList.begin());       // Remove the first element
myList.reverse();                   // Reverse the list
myList.sort();                      // Sort the list
```

Complete Example

Here's a complete example demonstrating various operations:

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

void printList(const list<int>& lst) {
   for (int value : lst) {
      cout << value << " ";
   }
   cout << endl;
}

int main() {
   // Initialize list with values
   list<int> myList{10, 20, 30, 40};

   // Add elements
   myList.push_front(5);
   myList.push_back(50);

   cout << "List after adding elements: ";
   printList(myList);

   // Remove elements
   myList.pop_front();
   myList.pop_back();

   cout << "List after removing elements: ";
   printList(myList);
```

```
    // Access elements
    cout << "First element: " << myList.front() << endl;
    cout << "Last element: " << myList.back() << endl;

    // Insert and erase
    myList.insert(myList.begin(), 15);
    myList.erase(myList.begin());

    cout << "List after insert and erase: ";
    printList(myList);

    // Reverse and sort
    myList.reverse();
    myList.sort();

    cout << "List after reverse and sort: ";
    printList(myList);

    return 0;
}
```

Summary

- list provides efficient insertion and deletion operations.
- It supports non-contiguous memory allocation and sequential access.
- Use iterators to traverse and manipulate elements in the list.

Tutorial: Understanding deque in C++

Introduction

A deque (double-ended queue) is a sequence container in C++ that allows fast insertion and deletion of elements at both the front and back. Unlike vector, deque does not guarantee contiguous storage, but it is efficient for operations at both ends.

Basic Syntax

To use deque, include the <deque> header and use the following syntax to declare a deque:

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
   deque<int> myDeque; // Declare an empty deque of integers
   return 0;
}
```

Creating and Initializing a Deque

You can initialize a deque with values using an initializer list:

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
   deque<int> myDeque{1, 2, 3, 4, 5}; // Initialize deque with values
   return 0;
}
```

Basic Operations

1. **Adding Elements**
   - **Push to Back**: Adds an element to the end of the deque.
   - **Push to Front**: Adds an element to the beginning of the deque.

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
   deque<int> myDeque;
   myDeque.push_back(10); // Adds 10 at the end
   myDeque.push_front(20); // Adds 20 at the beginning

   // Display the deque
   for (int n : myDeque) {
      cout << n << ' ';
```

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

2. **Accessing Elements**
    - **At**: Accesses element at a specific position.
    - **Front**: Accesses the first element.
    - **Back**: Accesses the last element.

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
    deque<int> myDeque{10, 20, 30, 40};

    cout << "Element at index 2: " << myDeque.at(2) << endl;
    cout << "First element: " << myDeque.front() << endl;
    cout << "Last element: " << myDeque.back() << endl;

    return 0;
}
```

3. **Removing Elements**
    - **Pop from Back**: Removes the last element.
    - **Pop from Front**: Removes the first element.

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
    deque<int> myDeque{10, 20, 30, 40};

    myDeque.pop_back(); // Removes 40
    myDeque.pop_front(); // Removes 10

    // Display the deque
    for (int n : myDeque) {
        cout << n << ' ';
    }
    cout << endl;

    return 0;
}
```

4. **Other Useful Functions**
    - **Size**: Returns the number of elements.
    - **Clear**: Removes all elements.

- **Resize**: Changes the number of elements.

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
   deque<int> myDeque{10, 20, 30, 40};

   cout << "Size of deque: " << myDeque.size() << endl;
   myDeque.clear(); // Clears all elements
   cout << "Size after clear: " << myDeque.size() << endl;

   myDeque.resize(5, 100); // Resizes deque to 5 elements, fills with 100
   // Display the deque
   for (int n : myDeque) {
      cout << n << ' ';
   }
   cout << endl;

   return 0;
}
```

Advanced Operations

1. **Insertion at a Specific Position**

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
   deque<int> myDeque{10, 30, 40};
   myDeque.insert(myDeque.begin() + 1, 20); // Insert 20 at index 1

   // Display the deque
   for (int n : myDeque) {
      cout << n << ' ';
   }
   cout << endl;

   return 0;
}
```

2. **Reversing and Sorting**

```
#include <deque>
#include <iostream>
#include <algorithm>
```

```
using namespace std;

int main() {
    deque<int> myDeque{40, 30, 20, 10};

    reverse(myDeque.begin(), myDeque.end()); // Reverse the deque

    // Display the reversed deque
    for (int n : myDeque) {
        cout << n << ' ';
    }
    cout << endl;

    sort(myDeque.begin(), myDeque.end()); // Sort the deque

    // Display the sorted deque
    for (int n : myDeque) {
        cout << n << ' ';
    }
    cout << endl;

    return 0;
}
```

Summary

- **deque** provides efficient insertions and deletions at both ends.
- **Basic Functions**: push_front(), push_back(), pop_front(), pop_back(), front(), back().
- **Advanced Functions**: insert(), reverse(), sort(), resize().

Tutorial: Understanding queue in C++

Introduction

A queue is a container adapter in C++ that implements a queue data structure. It follows the FIFO (First In, First Out) principle, meaning that elements are added to the back of the queue and removed from the front.

Basic Syntax

To use queue, include the <queue> header and use the following syntax:

```
#include <queue>
#include <iostream>

using namespace std;

int main() {
   queue<int> myQueue; // Declare an empty queue of integers
   return 0;
}
```

Creating and Initializing a Queue

You can create a queue and add elements to it using push():

```
#include <queue>
#include <iostream>

using namespace std;

int main() {
   queue<int> myQueue;

   myQueue.push(1); // Add 1 to the queue
   myQueue.push(2); // Add 2 to the queue
   myQueue.push(3); // Add 3 to the queue

   // Display the queue
   while (!myQueue.empty()) {
      cout << myQueue.front() << ' '; // Print the front element
      myQueue.pop(); // Remove the front element
   }
   cout << endl;

   return 0;
}
```

Basic Operations

1. **Adding Elements**

Use push() to add elements to the queue:

```
#include <queue>
#include <iostream>
```

```
using namespace std;

int main() {
   queue<int> myQueue;

   myQueue.push(10); // Add 10
   myQueue.push(20); // Add 20
   myQueue.push(30); // Add 30

   return 0;
}
```

2. **Accessing Elements**
   - **Front**: Accesses the first element.
   - **Back**: Accesses the last element.

```
-----------------------------------------------------------------------------------------------------
#include <queue>
#include <iostream>

using namespace std;

int main() {
   queue<int> myQueue;
   myQueue.push(10);
   myQueue.push(20);
   myQueue.push(30);

   cout << "Front element: " << myQueue.front() << endl;
   cout << "Back element: " << myQueue.back() << endl;

   return 0;
}
```

3. **Removing Elements**

Use pop() to remove the front element:

```
-----------------------------------------------------------------------------------------------------
#include <queue>
#include <iostream>

using namespace std;

int main() {
   queue<int> myQueue;
   myQueue.push(10);
   myQueue.push(20);
   myQueue.push(30);

   myQueue.pop(); // Remove the front element (10)

   cout << "Front element after pop: " << myQueue.front() << endl;
```

```
    return 0;
}
```

4. **Checking Size and Empty Status**
    - **Size**: Returns the number of elements.
    - **Empty**: Checks if the queue is empty.

```
#include <queue>
#include <iostream>

using namespace std;

int main() {
    queue<int> myQueue;
    myQueue.push(10);
    myQueue.push(20);

    cout << "Size of queue: " << myQueue.size() << endl;
    cout << "Is queue empty? " << (myQueue.empty() ? "Yes" : "No") << endl;

    myQueue.pop();
    myQueue.pop();

    cout << "Is queue empty after popping all elements? " << (myQueue.empty() ? "Yes" : "No") << endl;

    return 0;
}
```

Swapping Queues
You can swap the contents of two queues using the swap() function:

```
#include <queue>
#include <iostream>

using namespace std;

int main() {
    queue<int> queue1;
    queue<int> queue2;

    queue1.push(1);
    queue1.push(2);
    queue1.push(3);

    queue2.push(4);
    queue2.push(5);
    queue2.push(6);
```

```
    // Swap contents of queue1 and queue2
    queue1.swap(queue2);

    // Display the contents of queue1 after swap
    cout << "Contents of queue1 after swap: ";
    while (!queue1.empty()) {
        cout << queue1.front() << ' ';
        queue1.pop();
    }
    cout << endl;

    // Display the contents of queue2 after swap
    cout << "Contents of queue2 after swap: ";
    while (!queue2.empty()) {
        cout << queue2.front() << ' ';
        queue2.pop();
    }
    cout << endl;

    return 0;
}
```

Complexity Overview

- **Push Operation**: O(1)
- **Pop Operation**: O(1)
- **Accessing Front/Back**: O(1)
- **Swapping Queues**: O(1) for each swap operation
- **Space Complexity**: O(n), where n is the total number of elements in the queue(s).

Summary

- **queue** provides a simple interface for FIFO operations.
- **Basic Functions**: push(), pop(), front(), back(), size(), empty().
- **Advanced Functions**: swap().

Priority Queue in C++ STL

A priority queue in C++ is a type of container adapter that keeps the elements in a specific order. By default, it arranges elements in a way that the largest element is always at the top. You can also configure it to arrange elements so that the smallest is at the top. The priority queue is implemented using a heap data structure, which is a type of binary tree.

Here's a simple guide to using priority queues in C++ with examples.

Basics of priority_queue

1. **Default Max Heap**

By default, priority_queue creates a max heap, where the largest element is always at the top.

Syntax:

```
priority_queue<int> pq;
```

Example:

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

int main() {
   int arr[6] = {10, 2, 4, 8, 6, 9};
   priority_queue<int> pq;

   // Adding elements to the priority queue
   for (int i = 0; i < 6; i++) {
      pq.push(arr[i]);
   }

   // Displaying elements of the priority queue
   cout << "Priority Queue (Max Heap): ";
   while (!pq.empty()) {
      cout << pq.top() << ' ';
      pq.pop();
   }

   return 0;
}
```

Output:

Priority Queue (Max Heap): 10 9 8 6 4 2

Common Methods

1. **Inserting and Removing Elements**
   - push(): Adds an element to the queue.
   - pop(): Removes the top element of the queue.

Example:

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

void showpq(priority_queue<int> pq) {
   while (!pq.empty()) {
      cout << pq.top() << ' ';
      pq.pop();
   }
   cout << '\n';
}

int main() {
   priority_queue<int> pq;
   pq.push(10);
   pq.push(30);
   pq.push(20);
   pq.push(5);
   pq.push(1);

   cout << "Priority Queue: ";
   showpq(pq);

   cout << "\nSize of the queue: " << pq.size();
   cout << "\nTop element: " << pq.top();

   cout << "\nAfter popping the top element: ";
   pq.pop();
   showpq(pq);

   return 0;
}
```

Output:

```
-----------------------------------------------------------------------------------------------------
Priority Queue: 30 20 10 5 1
Size of the queue: 5
Top element: 30
After popping the top element: 20 10 5 1
```

2. **Accessing the Top Element**
   - top(): Returns a reference to the topmost element.

Example:

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

int main() {
   priority_queue<int> pq;
   pq.push(1);
   pq.push(20);
   pq.push(7);

   cout << "Top element: " << pq.top();

   return 0;
}
```

Output:

```css
Top element: 20
```

3. **Checking if the Queue is Empty**
    - empty(): Returns true if the queue is empty.

Example:

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

int main() {
    priority_queue<int> pq;
    pq.push(1);

    if (pq.empty()) {
        cout << "Queue is empty.";
    } else {
        cout << "Queue is not empty.";
    }

    return 0;
}
```

Output:

```
csharp
-----------------------------------------------------------------------------------------------------
Queue is not empty.
```

4. **Swapping Contents**
    - swap(): Swaps the contents of two priority queues.

Example:

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

void print(priority_queue<int> pq) {
    while (!pq.empty()) {
        cout << pq.top() << ' ';
        pq.pop();
    }
    cout << '\n';
}

int main() {
    priority_queue<int> pq1;
    priority_queue<int> pq2;

    pq1.push(1);
    pq1.push(2);
    pq1.push(3);
    pq1.push(4);
```

```
    pq2.push(3);
    pq2.push(5);
    pq2.push(7);
    pq2.push(9);

    cout << "Before swapping:\n";
    cout << "Priority Queue 1: ";
    print(pq1);
    cout << "Priority Queue 2: ";
    print(pq2);

    pq1.swap(pq2);

    cout << "\nAfter swapping:\n";
    cout << "Priority Queue 1: ";
    print(pq1);
    cout << "Priority Queue 2: ";
    print(pq2);

    return 0;
}
```

Output:

```
Before swapping:
Priority Queue 1: 4 3 2 1
Priority Queue 2: 9 7 5 3

After swapping:
Priority Queue 1: 9 7 5 3
Priority Queue 2: 4 3 2 1
```

5. **Emplacing Elements**
    - emplace(): Inserts a new element into the queue.

Example:

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

int main() {
    priority_queue<int> pq;
    pq.emplace(1);
    pq.emplace(2);
    pq.emplace(3);
    pq.emplace(4);
    pq.emplace(5);

    cout << "Priority Queue: ";
    while (!pq.empty()) {
        cout << pq.top() << ' ';
```

```
        pq.pop();
    }

    return 0;
}
```

Output:

```
Priority Queue: 5 4 3 2 1
```

6. **Getting the Type of Object**
   - value_type: Represents the type of object stored in the priority queue.

Example:

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

int main() {
    priority_queue<int>::value_type intVal = 20;
    priority_queue<string>::value_type strVal = "hello";

    cout << "Integer value: " << intVal << endl;
    cout << "String value: " << strVal << endl;

    return 0;
}
```

Output:

mathematica

```
Integer value: 20
String value: hello
```

Complexity Analysis

- **Time Complexity:**
  - push(), pop(), emplace(): O(log N)
  - top(), empty(), size(): O(1)
  - swap(): O(1)
- **Space Complexity:**
  - All operations except swap() use O(1) extra space. swap() uses O(N) space for temporary storage.

C++ STL stack Tutorial

Introduction

A stack is a container adapter that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. Imagine a stack of books: you add new books to the top and also remove books from the top.

Syntax

To use a stack in C++, you need to include the <stack> header file. Here's the syntax to create a stack:

```
#include <stack>
```

You can define a stack with this syntax:

```
template <class Type, class Container = deque<Type> > class stack;
```

- **Type**: The type of element stored in the stack (e.g., int, float, or a user-defined type).
- **Container**: The underlying container used by the stack. By default, this is deque, but you can also use vector or list.

Basic Operations

Here are some of the main functions you can use with a stack:

- empty(): Checks if the stack is empty.
- size(): Returns the number of elements in the stack.
- top(): Accesses the top element of the stack.
- push(element): Adds an element to the top of the stack.
- pop(): Removes the top element from the stack.

Example Code

Here's a simple C++ program demonstrating how to use stack:

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

int main() {
   // Create a stack of integers
   stack<int> s;

   // Push elements onto the stack
   s.push(21);
   s.push(22);
   s.push(24);
   s.push(25);

   // Pop the top element
   s.pop(); // Removes 25

   // Push an additional element
   int num = 0;
   s.push(num); // Pushes 0

   // Pop remaining elements
   s.pop(); // Removes 0
```

```
    s.pop(); // Removes 24
    s.pop(); // Removes 22

    // Print remaining elements
    cout << "Stack contents: ";
    while (!s.empty()) {
        cout << s.top() << " "; // Print top element
        s.pop(); // Remove top element
    }
    cout << endl;

    return 0;
}
```

Output

mathematica

Stack contents: 21 22

Complexity Analysis

- **Time Complexity**:
  - empty(), size(), top(), push(), and pop() operations all have a time complexity of O(1)O(1)O(1). These operations are efficient and execute in constant time.
- **Space Complexity**:
  - The space complexity is O(N)O(N)O(N), where NNN is the number of elements in the stack. The stack uses space proportional to the number of elements it contains.

Summary

- **LIFO Principle**: Stacks follow Last In, First Out order.
- **Basic Functions**: empty(), size(), top(), push(), and pop().
- **Efficiency**: Both time and space complexities are efficient for typical stack operations.

C++ set Tutorial

set is an associative container in the C++ Standard Template Library (STL) that stores unique elements in a sorted order. It's part of the <set> header file.

Key Features of set:

- **Unique Elements**: Each element in a set is unique. Duplicate elements are not allowed.
- **Sorted Order**: By default, elements are sorted in ascending order. You can change this to descending order if needed.
- **Immutable Values**: Once an element is added, its value cannot be modified. You can remove and reinsert it with a new value if necessary.
- **Underlying Data Structure**: Implements a binary search tree, typically a Red-Black tree.

Basic Syntax

To use set, include the <set> header:

```
#include <set>
```

Define a set using:

```
set<data_type> set_name;
```

Example 1: Basic Usage

Here's a simple example demonstrating basic usage of set:

```
#include <iostream>
#include <set>

int main() {
   // Create a set of characters
   set<char> mySet;

   // Insert elements into the set
   mySet.insert('G');
   mySet.insert('F');
   mySet.insert('G'); // This will be ignored because 'G' is already in the set

   // Print all elements in the set
   for (auto ch : mySet) {
      cout << ch << ' ';
   }
   cout << '\n';

   return 0;
}
```

Output:

r

```
F G
```

Explanation: The set only stores unique elements. The second insertion of 'G' is ignored.

Example 2: Set Sorted in Descending Order
You can sort the set in descending order by using greater:

```
#include <iostream>
#include <set>

int main() {
   // Create a set of integers sorted in descending order
   set<int, greater<int>> mySet;

   // Insert elements into the set
   mySet.insert(10);
   mySet.insert(5);
   mySet.insert(12);
   mySet.insert(4);

   // Print all elements in the set
   for (auto num : mySet) {
      cout << num << ' ';
   }
   cout << '\n';

   return 0;
}
```

Output:

----

12 10 5 4

Explanation: The elements are printed in descending order because of greater<int>.

Common Functions

- **begin()**: Returns an iterator to the first element.
- **end()**: Returns an iterator to one past the last element.
- **size()**: Returns the number of elements in the set.
- **empty()**: Checks if the set is empty.
- **find(value)**: Searches for an element and returns an iterator to it if found.
- **erase(value)**: Removes the element with the specified value.

Example 3: Using Common Functions

```
#include <iostream>
#include <set>

int main() {
   set<int> mySet = {10, 20, 30, 40, 50};

   // Print the size of the set
   cout << "Size: " << mySet.size() << '\n';

   // Check if the set is empty
   cout << "Is empty: " << (mySet.empty() ? "Yes" : "No") << '\n';
```

```
    // Find and print an element
    auto it = mySet.find(30);
    if (it != mySet.end()) {
        cout << "Found: " << *it << '\n';
    } else {
        cout << "Not found\n";
    }

    // Erase an element and print the updated set
    mySet.erase(20);
    cout << "Updated set: ";
    for (auto num : mySet) {
        cout << num << ' ';
    }
    cout << '\n';

    return 0;
}
```
Output:

Size: 5
Is empty: No
Found: 30
Updated set: 10 30 40 50
Explanation: Shows how to check size, emptiness, find elements, and erase elements from the set.
Summary

- set is a sorted container of unique elements.
- By default, it sorts in ascending order but can be customized.
- Common operations include insertion, deletion, and searching, all with efficient time complexities.

Tutorial: map in C++ STL

Overview

map is an associative container in C++ that stores elements in key-value pairs. Each key is unique, and values are stored in a specific sorted order based on the key.

Syntax

To use map, include the <map> header file:

```
-----------------------------------------------------------------------------------------------------
#include <map>
```

Define a map using:

```
-----------------------------------------------------------------------------------------------------
map<KeyType, ValueType> mapName;
```

Basic Functions

1. **begin()** – Returns an iterator to the first element.
2. **end()** – Returns an iterator to the position after the last element.
3. **size()** – Returns the number of elements.
4. **empty()** – Checks if the map is empty.
5. **insert(key, value)** – Adds a new key-value pair.
6. **erase(key)** – Removes an element by key.
7. **clear()** – Removes all elements.

Examples

Example 1: Basic Map Operations

This example demonstrates how to create a map, insert elements, and use iterators to print the map.

```
-----------------------------------------------------------------------------------------------------
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
   // Create a map of strings to integers
   map<string, int> mp;

   // Insert some values into the map
   mp["one"] = 1;
   mp["two"] = 2;
   mp["three"] = 3;

   // Iterate through the map and print the elements
   for (auto it = mp.begin(); it != mp.end(); ++it) {
      cout << "Key: " << it->first << ", Value: " << it->second << endl;
   }

   return 0;
}
```

Output:

```
-----------------------------------------------------------------------------------------------------
Key: one, Value: 1
Key: three, Value: 3
```

Key: two, Value: 2

Complexity:

- Time Complexity: O(n), where n is the number of elements in the map.

Example 2: Size of the Map

This example shows how to use the size() function to get the number of elements in the map.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

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

    // Insert some values into the map
    mp["one"] = 1;
    mp["two"] = 2;
    mp["three"] = 3;

    // Print the size of the map
    cout << "Size of map: " << mp.size() << endl;

    return 0;
}
```

Output:

arduino

Size of map: 3

Complexity:

- Time Complexity: O(1)

Example 3: Advanced Operations

This example demonstrates inserting elements, assigning elements to another map, and removing elements.

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

int main() {
    map<int, int> map1;

    // Insert elements
    map1.insert({1, 40});
    map1.insert({2, 30});
    map1.insert({3, 60});
    map1.insert({4, 20});
    map1.insert({5, 50});
    map1[6] = 50;  // Another way to insert
```

```
    // Print map1
    cout << "Map1:\n";
    for (const auto& pair : map1) {
        cout << "Key: " << pair.first << ", Value: " << pair.second << endl;
    }

    // Assign elements to another map
    map<int, int> map2(map1.begin(), map1.end());

    // Print map2
    cout << "Map2 after assignment:\n";
    for (const auto& pair : map2) {
        cout << "Key: " << pair.first << ", Value: " << pair.second << endl;
    }

    // Remove elements
    map2.erase(map2.begin(), map2.find(3));
    map2.erase(4);

    // Print map2 after removals
    cout << "Map2 after removals:\n";
    for (const auto& pair : map2) {
        cout << "Key: " << pair.first << ", Value: " << pair.second << endl;
    }

    return 0;
}
```

Output:

```
----------------------------------------------------------------------------------------------------
Map1:
Key: 1, Value: 40
Key: 2, Value: 30
Key: 3, Value: 60
Key: 4, Value: 20
Key: 5, Value: 50
Key: 6, Value: 50

Map2 after assignment:
Key: 1, Value: 40
Key: 2, Value: 30
Key: 3, Value: 60
Key: 4, Value: 20
Key: 5, Value: 50
Key: 6, Value: 50

Map2 after removals:
Key: 3, Value: 60
Key: 5, Value: 50
Key: 6, Value: 50
```

Complexity:

- Time Complexity: O(n log(n)), where n is the number of elements in the map.
- Auxiliary Space: O(n)

Properties

- **Ordering:** By default, keys are sorted in ascending order. You can use map<KeyType, ValueType, greater<KeyType>> for descending order.
- **Unique Keys:** Each key is unique.
- **Immutability:** Values cannot be modified once inserted, but you can remove and reinsert updated values.

Additional Functions

- **find(key)** – Finds an element by key.
- **lower_bound(key)** – Returns an iterator to the first element that is not less than the key.
- **upper_bound(key)** – Returns an iterator to the first element greater than the key.

Tutorial: Iterators in C++

Iterators are objects that point to elements within a container and allow traversal of the container's elements. They are similar to pointers but can be more flexible. Iterators come in various types based on their capabilities.

Types of Iterators:

1. **Input Iterators**: Allow reading data in a single pass.
2. **Output Iterators**: Allow writing data in a single pass.
3. **Forward Iterators**: Allow multiple passes in a forward direction.
4. **Bidirectional Iterators**: Allow traversal in both forward and backward directions.
5. **Random Access Iterators**: Allow access to any element in constant time.

1. Input Iterators

Description: Input iterators are used to read elements from a container sequentially in a single pass. They are usually used in algorithms that process a sequence of data only once.

Example: Reading data from a vector

```
#include <iostream>
#include <vector>
#include <iterator>

int main() {
   vector<int> vec = {1, 2, 3, 4, 5};
   vector<int>::iterator it;

   for (it = vec.begin(); it != vec.end(); ++it) {
      cout << *it << " ";
   }

   return 0;
}
```

Output:

```
1 2 3 4 5
```

2. Output Iterators

Description: Output iterators are used to write elements to a container sequentially in a single pass.

Example: Writing data to a vector

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

int main() {
   vector<int> vec(5);
   vector<int>::iterator it;

   int value = 1;
   for (it = vec.begin(); it != vec.end(); ++it) {
      *it = value++;
   }
```

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

    return 0;
}
```

Output:

```
1 2 3 4 5
```

3. Forward Iterators

Description: Forward iterators are like input iterators but allow multiple passes over the data.

Example: Using a forward iterator with a list

```
#include <iostream>
#include <list>
#include <iterator>

int main() {
    list<int> lst = {1, 2, 3, 4, 5};
    list<int>::iterator it;

    for (it = lst.begin(); it != lst.end(); ++it) {
        cout << *it << " ";
    }

    cout << endl;

    for (it = lst.begin(); it != lst.end(); ++it) {
        cout << *it << " ";
    }

    return 0;
}
```

Output:

```
1 2 3 4 5
1 2 3 4 5
```

4. Bidirectional Iterators

Description: Bidirectional iterators allow traversal in both forward and backward directions.

Example: Using a bidirectional iterator with a set

```
#include <iostream>
#include <set>
#include <iterator>

int main() {
```

```
    set<int> s = {1, 2, 3, 4, 5};
    set<int>::iterator it;

    cout << "Forward iteration: ";
    for (it = s.begin(); it != s.end(); ++it) {
        cout << *it << " ";
    }

    cout << "\nBackward iteration: ";
    for (it = s.end(); it != s.begin(); ) {
        cout << *(--it) << " ";
    }

    return 0;
}
```

Output:

```
mathematica

Forward iteration: 1 2 3 4 5
Backward iteration: 5 4 3 2 1
```

5. Random Access Iterators

Description: Random access iterators allow access to any element in constant time.

Example: Using a random access iterator with a vector

```
#include <iostream>
#include <vector>
#include <iterator>

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

    vector<int>::iterator it = vec.begin();

    cout << "Element at index 2: " << *(it + 2) << endl;

    cout << "All elements: ";
    for (it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }

    return 0;
}
```

Output:

```
mathematica

Element at index 2: 3
All elements: 1 2 3 4 5
```

Summary of Iterator Capabilities:

Iterator Type	Read	Write	Forward Traversal	Backward Traversal	Random Access
Input Iterator	Yes	No	Yes	No	No
Output Iterator	No	Yes	Yes	No	No
Forward Iterator	Yes	Yes	Yes	No	No
Bidirectional Iterator	Yes	Yes	Yes	Yes	No
Random Access Iterator	Yes	Yes	Yes	Yes	Yes

Iterators are a powerful feature in C++ that provide a way to traverse and manipulate the elements of a container in a flexible and efficient manner. Understanding the different types of iterators and their capabilities can help you write more efficient and readable code.

Tutorial: C++ Standard Library Algorithms

The C++ Standard Library provides a wide range of algorithms that work with containers to perform common operations. These algorithms are found in the <algorithm> header file.

1. Sorting Algorithms

sort

Sorts a range of elements.

Syntax:

```
template <class RandomIt>
void sort(RandomIt first, RandomIt last);
```

Example:

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

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

   sort(v.begin(), v.end());

   cout << "Sorted vector: ";
   for (int n : v) {
      cout << n << ' ';
   }

   return 0;
}
```

Output:

arduino

```
Sorted vector: 1 2 3 4 5
```

Complexity:

- Time Complexity: O(n log n)

---

2. Searching Algorithms

find

Finds the first occurrence of a value.

Syntax:

```
template <class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value);
```

Example:

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

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

    auto it = find(v.begin(), v.end(), 5);

    if (it != v.end()) {
        cout << "Element found: " << *it << '\n';
    } else {
        cout << "Element not found\n";
    }

    return 0;
}
```

Output:

mathematica

Element found: 5

Complexity:

- Time Complexity: O(n)

---

3. Counting Algorithms

count

Counts the number of occurrences of a value.

Syntax:

```
template <class InputIt, class T>
typename iterator_traits<InputIt>::difference_type count(InputIt first, InputIt last, const T& value);
```

Example:

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

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

    int count = count(v.begin(), v.end(), 2);

    cout << "Number of 2s: " << count << '\n';

    return 0;
}
```

Output:

javascript

Number of 2s: 3

Complexity:

- Time Complexity: O(n)

4. Modifying Algorithms

reverse

Reverses the order of elements in a range.

Syntax:

```
template <class BidirIt>
void reverse(BidirIt first, BidirIt last);
```

Example:

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

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

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

    cout << "Reversed vector: ";
    for (int n : v) {
        cout << n << ' ';
    }

    return 0;
}
```

Output:

arduino

Reversed vector: 5 4 3 2 1

Complexity:

- Time Complexity: O(n)

5. Removing Algorithms

remove

Removes all occurrences of a value (not from the container but moves them to the end).

Syntax:

```
template <class ForwardIt, class T>
ForwardIt remove(ForwardIt first, ForwardIt last, const T& value);
```

Example:

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

int main() {
```

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

    auto newEnd = remove(v.begin(), v.end(), 2);

    cout << "Vector after remove: ";
    for (auto it = v.begin(); it != newEnd; ++it) {
        cout << *it << ' ';
    }

    return 0;
}
```

Output:

arduino

```
-------------------------------------------------------------------------------------------------------
Vector after remove: 4 5 1 3
```

Complexity:

- Time Complexity: O(n)

---

6. Copying Algorithms

copy

Copies elements from one range to another.

Syntax:

```
-------------------------------------------------------------------------------------------------------
template <class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first);
```

Example:

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

int main() {
    vector<int> src = {1, 2, 3, 4, 5};
    vector<int> dest(src.size());

    copy(src.begin(), src.end(), dest.begin());

    cout << "Copied vector: ";
    for (int n : dest) {
        cout << n << ' ';
    }

    return 0;
}
```

Output:

arduino

```
-------------------------------------------------------------------------------------------------------
Copied vector: 1 2 3 4 5
```

Complexity:

- Time Complexity: O(n)

---

7. Sorting with Custom Comparator
sort with Custom Comparator
Sorts elements using a custom comparison function.
Syntax:

```
template <class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
```

Example:

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

bool compare(int a, int b) {
    return a > b; // Descending order
}

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

    sort(v.begin(), v.end(), compare);

    cout << "Sorted vector in descending order: ";
    for (int n : v) {
        cout << n << ' ';
    }

    return 0;
}
```

Output:
arduino

```
Sorted vector in descending order: 5 4 3 2 1
```

Complexity:

- Time Complexity: O(n log n)

---

8. Finding Maximum and Minimum
max_element and min_element
Finds the maximum or minimum element in a range.
Syntax:

```
template <class ForwardIt>
ForwardIt max_element(ForwardIt first, ForwardIt last);

template <class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last);
```

Example:

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

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

   auto maxIt = max_element(v.begin(), v.end());
   auto minIt = min_element(v.begin(), v.end());

   cout << "Maximum element: " << *maxIt << '\n';
   cout << "Minimum element: " << *minIt << '\n';

   return 0;
}
```

Output:

```
------------------------------------------------------------------------------------------------------
Maximum element: 5
Minimum element: 1
```

Complexity:

- Time Complexity: O(n)

---

9. Transforming Elements

transform

Applies a function to a range of elements.

Syntax:

```
------------------------------------------------------------------------------------------------------
template <class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first, InputIt last, OutputIt d_first, UnaryOperation unary_op);
```

Example:

```
------------------------------------------------------------------------------------------------------
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

int main() {
   vector<int> v = {1, 2, 3, 4, 5};
   vector<int> result(v.size());

   transform(v.begin(), v.end(), result.begin(), [](int x) { return x * x; });

   cout << "Transformed vector (squares): ";
   for (int n : result) {
      cout << n << ' ';
```

```
    }

    return 0;
}
```

Output:

java

Transformed vector (squares): 1 4 9 16 25

Complexity:

- Time Complexity: O(n)

---

10. Checking Conditions

all_of, any_of, none_of

Checks if all, any, or none of the elements satisfy a condition.

Syntax:

```
template <class InputIt, class UnaryPredicate>
bool all_of(InputIt first, InputIt last, UnaryPredicate p);

template <class InputIt, class UnaryPredicate>
bool any_of(InputIt first, InputIt last,
```

Tutorial: for_each in C++ Standard Library Algorithms
The for_each algorithm applies a function to each element in a range. This is useful for performing an operation on each element in a container without modifying the container.

```
template <class InputIt, class UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f);
```

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

void print(int n) {
   cout << n << ' ';
}

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

   for_each(v.begin(), v.end(), print);

   return 0;
}
```

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

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

   for_each(v.begin(), v.end(), [](int &n) { n *= 2; });
                                              \\[&sum](int n) { sum += n; }

   for_each(v.begin(), v.end(), [](int n) { cout << n << ' '; });

   return 0;
}
```

Lambda Functions in C++
Lambda functions, also known as lambda expressions, provide a way to define anonymous functions within your code. They are useful for short, throwaway functions that are not intended to be reused elsewhere.
Syntax:

```
[capture](parameters) -> return_type {
   body
}
```

- **capture**: Defines which variables from the surrounding scope are available to the lambda.
- **parameters**: The parameters passed to the lambda function.
- **return_type**: The return type of the lambda function. It can often be omitted as it is deduced automatically.

- **body**: The code to be executed.

Capture Clause:

- **[]**: Capture nothing.
- **[=]**: Capture all variables by value.
- **[&]**: Capture all variables by reference.
- **[variable1, &variable2]**: Capture variable1 by value and variable2 by reference.

Basic Examples of Lambda Functions

Example 1: Basic Lambda

Description: A simple lambda that prints "Hello, World!".

```
#include <iostream>

int main() {
    auto greet = []() {
        cout << "Hello, World!" << endl;
    };

    greet();
    return 0;
}
```

Example 2: Lambda with Parameters

Description: A lambda function that adds two numbers.

```
#include <iostream>

int main() {
    auto add = [](int a, int b) {
        return a + b;
    };

    cout << "Sum: " << add(3, 4) << endl;

    return 0;
}
```

Example 3: Lambda with Capture by Value

Description: Capturing variables by value.

```
#include <iostream>

int main() {
    int x = 10;
    int y = 20;

    auto add = [x, y]() {
        return x + y;
    };

    cout << "Sum: " << add() << endl;
```

```
    return 0;
}
```

Example 4: Lambda with Capture by Reference
Description: Capturing variables by reference.

```
#include <iostream>

int main() {
    int x = 10;
    int y = 20;

    auto add_and_modify = [&x, &y]() {
        x += 10;
        y += 10;
        return x + y;
    };

    cout << "Sum: " << add_and_modify() << endl;
    cout << "Modified x: " << x << endl;
    cout << "Modified y: " << y << endl;

    return 0;
}
```