

C++ STL(Standard Template Library) - 01

What is STL?

What is STL ?

- C++ STL stands for the "Standard Template Library." It is a powerful and extensive collection of template classes and functions in the C++ programming language that provides general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures.
- The C++ STL is an essential part of the C++ Standard Library, and it plays a crucial role in simplifying and accelerating C++ software development. It offers a wide range of data structures (containers) and algorithms that can be easily used in C++ programs, allowing developers to focus on solving specific problems rather than reinventing the wheel for common tasks.

Why we need STL?

Why we need STL ?

The C++ Standard Template Library (STL) is a critical component of the C++ Standard Library, and it serves several important purposes in C++ programming. Here are some key reasons why we need STL:

⋮

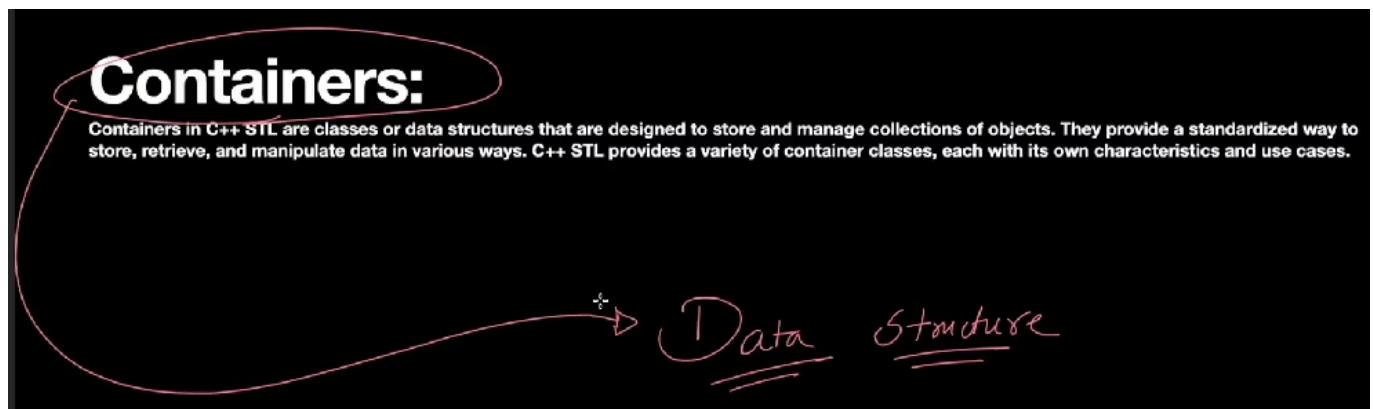
- Efficiency: STL provides highly optimized and efficient implementations of commonly used data structures and algorithms. These implementations are typically developed and tested by experts, ensuring that they perform well in various scenarios. Using STL can save you the time and effort required to implement these data structures and algorithms from scratch.
- Productivity: STL promotes code reuse and reduces the need to reinvent the wheel. It offers a wide range of container classes (like vectors, lists, sets, and maps) and algorithms (such as sorting and searching) that can be readily used in your programs. This leads to faster development and more maintainable code.
- Consistency: STL provides a standardized and consistent interface for working with data structures and algorithms. This consistency makes it easier to learn and use the library effectively. Once you understand how to use one STL container or algorithm, you can apply similar knowledge to others.
- Portability: Code written using STL is generally more portable across different compilers and platforms. The library is part of the C++ standard, ensuring that it is available and behaves consistently across compliant compilers.
- Safety: STL helps reduce common programming errors, such as buffer overflows and memory leaks, by encapsulating low-level memory management and providing safer alternatives. For example, smart pointers in STL (unique_ptr, shared_ptr) help manage memory automatically, reducing the risk of memory-related issues.
- Maintainability: Code using STL tends to be more maintainable because it leverages well-tested and standardized components. This makes it easier for developers to understand and modify code written by others or by their past selves.
- Performance Optimization: STL algorithms are designed to work efficiently with different container types, and they often outperform hand-rolled implementations. This allows you to focus on your application's logic rather than low-level performance optimizations.
- Expressiveness: STL promotes expressive code by providing a high-level interface for working with data structures and algorithms. This can lead to more readable and concise code, making your programs easier to understand and maintain.
- Community and Resources: STL has a vast user community, which means there are numerous tutorials, books, and online resources available to help you learn and use it effectively. You can tap into this wealth of knowledge to improve your C++ programming skills.

Components in STL:

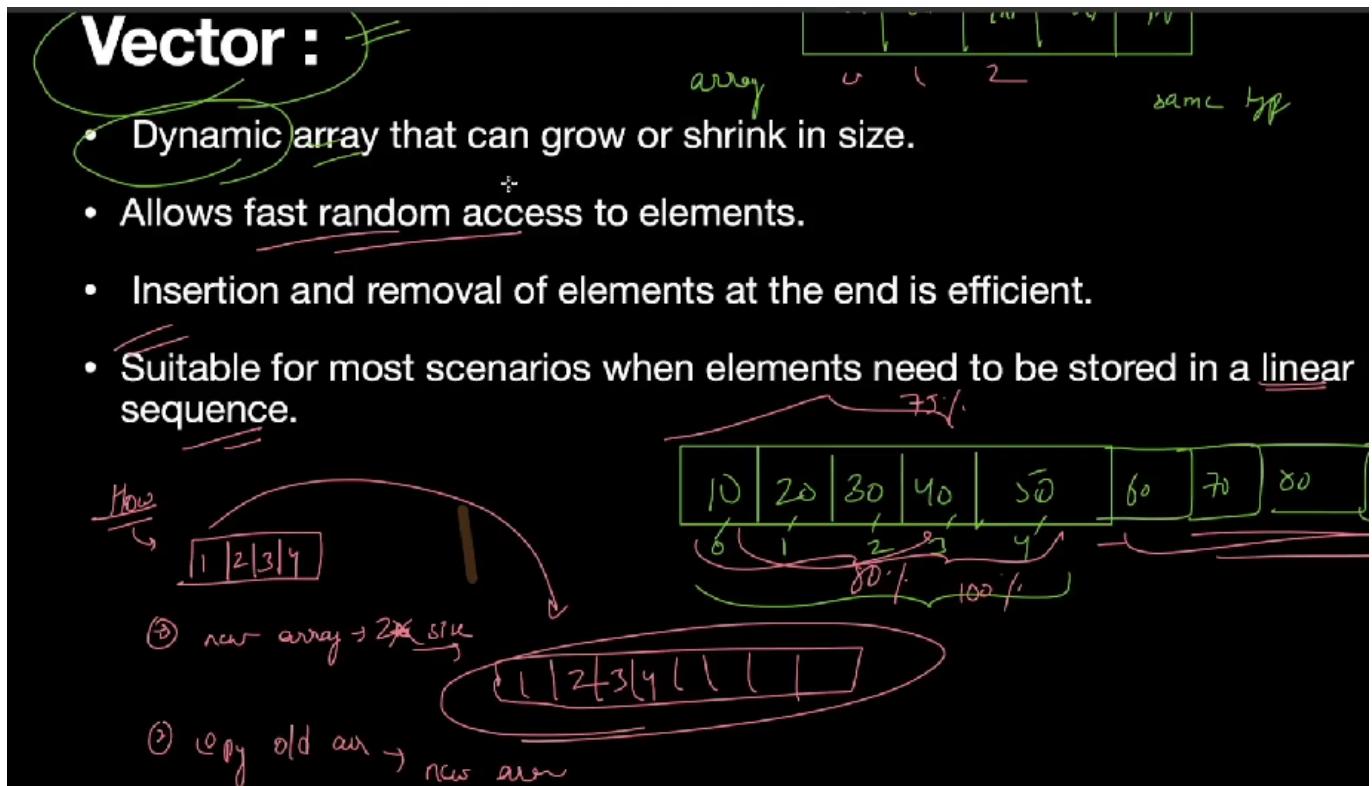
Common Components in STL

- Containers: vector, list, queue, stack, set, map etc
- Algorithms: sort(), binary_search(), reverse() etc
- Iterators
- Functors

1. Containers:



Vector:



Member Functions:

Member Functions:

- `begin()`: Returns an iterator pointing to the first element in the vector.
- `end()`: Returns an iterator pointing to the position just after the last element in the vector.
- `size()`: Returns the number of elements in the vector.
- `empty()`: Checks if the vector is empty (i.e., whether its size is 0).
- `capacity()`: Returns the number of elements that the vector can hold before needing to allocate more space.
- `reserve(size_type n)`: Requests that the vector capacity be increased to at least n elements, potentially reducing the number of reallocations.
- `max_size()`: Returns the maximum number of elements that the vector can hold due to system or library limitations.
- `front()`: Accesses the first element in the vector.
- `back()`: Accesses the last element in the vector.
- `operator[](size_type n)`: Accesses the element at the specified index without bounds checking.
- `at(size_type n)`: Accesses the element at the specified index with bounds checking.
- `push_back(const T& value)`: Adds an element to the end of the vector.
- `pop_back()`: Removes the last element from the vector.
- `insert(iterator position, const T& value)`: Inserts a new element before the specified position in the vector.
- `erase(iterator position) or erase(iterator first, iterator last)`: Removes one or more elements from the vector starting at the specified position.
- `clear()`: Removes all elements from the vector, which are destroyed, and leaves it with a size of 0.
- `swap(vector& x)`: Swaps the contents of the vector with those of another vector of the same type, including their sizes and capacities.

Code:

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

```
int main()
{
    // VECTOR:

    // creation:
    vector<int> marks;

    // creation with size 10
    vector<int> miles(10);

    // creation with size 15 and initials 0
    vector<int> distances(15, 5);

    // v.begin()
    cout << "v.start() : " << *(distances.begin()) << endl
        << endl;

    // v.end()
    cout << "v.end : " << *(distances.end()) << endl
        << endl;

    // v.push_back()
    distances.push_back(10);
    distances.push_back(20);
    distances.push_back(30);
    distances.push_back(40);

    // v.size()
    cout << "Size of vector is " << distances.size() << endl
        << endl;
    ;

    // v.pop_back()
    distances.pop_back();
    cout << "Size of vector is " << distances.size() << endl
        << endl;
    ;

    // v.front()
    cout << "Front : " << distances.front() << endl
        << endl;

    // v.back()
    cout << "Back : " << distances.back() << endl
        << endl;

    // v.empty()
    cout
        << "v.empty() : ";
    if (distances.empty())
        cout
            << "Vector is Empty" << endl
            << endl;
```

```
else
    cout << "Vector is not Empty" << endl
    << endl;

// Access using v[i]
cout << "v[i] : " << distances[15] << endl
    << endl;

// v.at(i)
cout << "v.at() : " << distances.at(15) << endl
    << endl;

// v.capacity()
vector<int> v;
cout << "Capacity : " << v.capacity() << endl;
v.push_back(10);
v.push_back(20);
v.push_back(30);
cout << "Here Size is " << v.size() << " but Capacity is " << v.capacity() <<
endl
    << endl;

// v.reserve(value) => It reserves the capacity of vector.

vector<int> v1;
cout << "Capacity: " << v1.capacity() << endl;
v1.reserve(5);
cout << "After Reserve, Now Capacity: " << v1.capacity() << endl
    << endl;

// v.max_size()
cout << "v.max_size() : " << v1.max_size() << endl
    << endl;

// v.clear()
cout << "Size : " << v.size() << endl;
cout << "v.clear() : ";
v.clear();
cout << "Now size : " << v.size() << endl
    << endl;

// v.insert(pos, value)
v.insert(v.begin(), 50);
cout << "v.begin() : " << v[0] << endl
    << endl;

// v.erase(pos) or v.erase(pos1, pos2)
v.push_back(60);
v.push_back(70);
v.push_back(80);
cout << "Size of vector is " << v.size() << endl;
v.erase(v.begin(), v.end());
cout << "Now size of vector is " << v.size() << endl
    << endl;
```

```
// v.swap(v1)
vector<int> first;
vector<int> second;

first.push_back(1);
first.push_back(2);
first.push_back(3);
second.push_back(4);
second.push_back(5);
second.push_back(6);

first.swap(second);

for (int i = 0; i < first.size(); i++)
{
    cout << "Vector first element at " << i << " is " << first[i] << endl;
}

for (int i = 0; i < first.size(); i++)
{
    cout << "Vector second element at " << i << " is " << second[i] << endl;
}
cout << endl;

return 0;
}
```

For Each Loop:

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

int main()
{
    // For Each Loop
    vector<int> first;

    first.push_back(1);
    first.push_back(2);
    first.push_back(3);

    for (int i : first)
    {
        cout << i << " ";
    }

    return 0;
}
```

Iterator:

Traverse in vector using Iterator.

Code:

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

int main()
{
    // Iterator:

    // creation:
    vector<int>::iterator it;

    // initialize and traverse:
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);

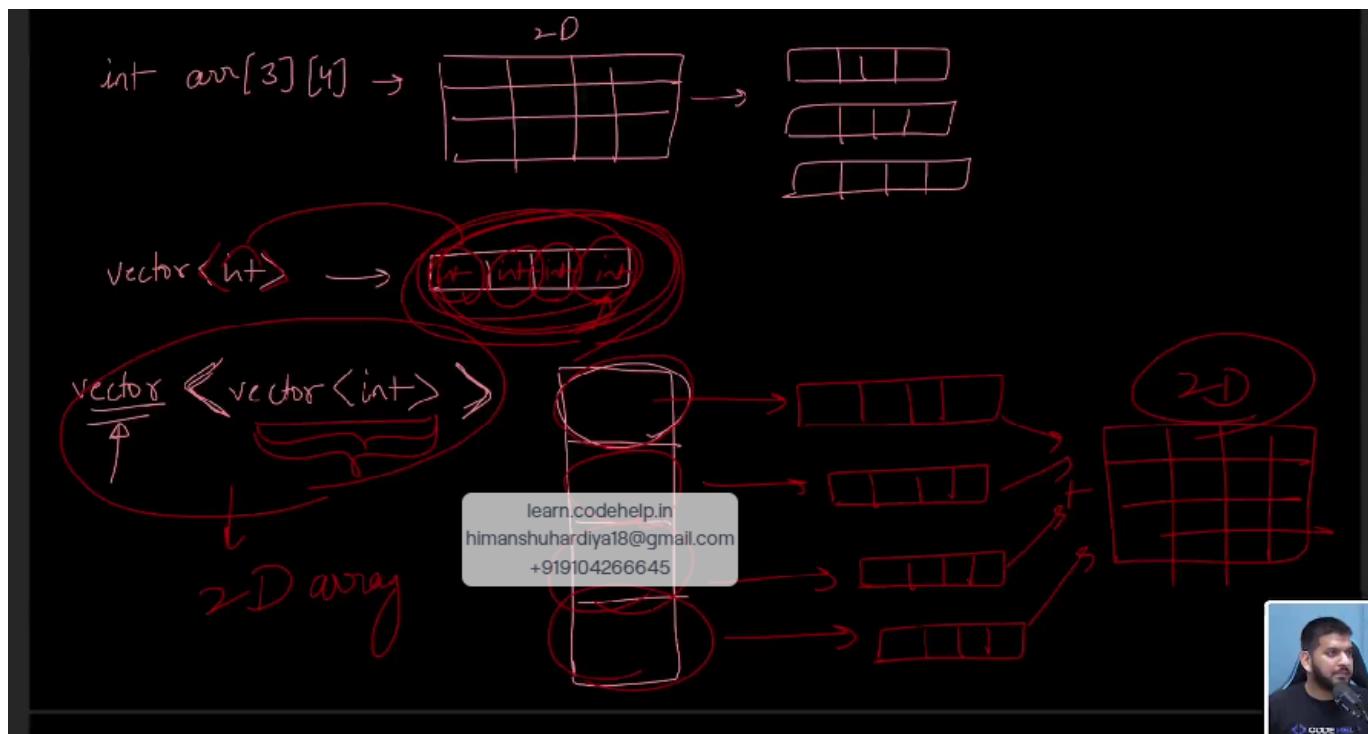
    vector<int>::iterator it2 = vec.begin();
    while (it2 != vec.end())
    {
        cout << *it2 << " ";
        it2++;
    }

    // OR

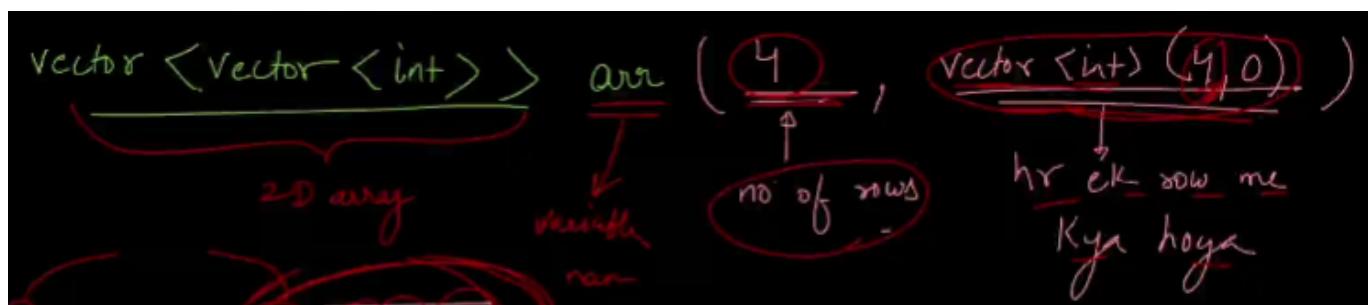
    vector<int>::iterator it3;
    for (it3 = vec.begin(); it3 != vec.end(); it3++)
    {
        cout << *it3 << " ";
    }

    return 0;
}
```

2D Vector:

**Syntax:**

```
vector<vector<int>> arr(4, vector<int>(4,0))
```

**Code:**

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

int main()
{
    // 2D Vector

    vector<vector<int>> arr(4, vector<int>(4, 0));

    // rowcount:
    cout << "Total rows : " << arr.size() << endl;

    // columncount
```

```

cout << "Total columns : " << arr[0].size() << endl
    << endl;

// create 2D vector with different size of columns on eachrow (Jagged Array):
vector<vector<int>> arr2(4);
arr2[0] = vector<int>(4);
arr2[1] = vector<int>(2);
arr2[2] = vector<int>(5);
arr2[3] = vector<int>(3);

// Jagged Array rowcount
cout << "Total rows : " << arr2.size() << endl;

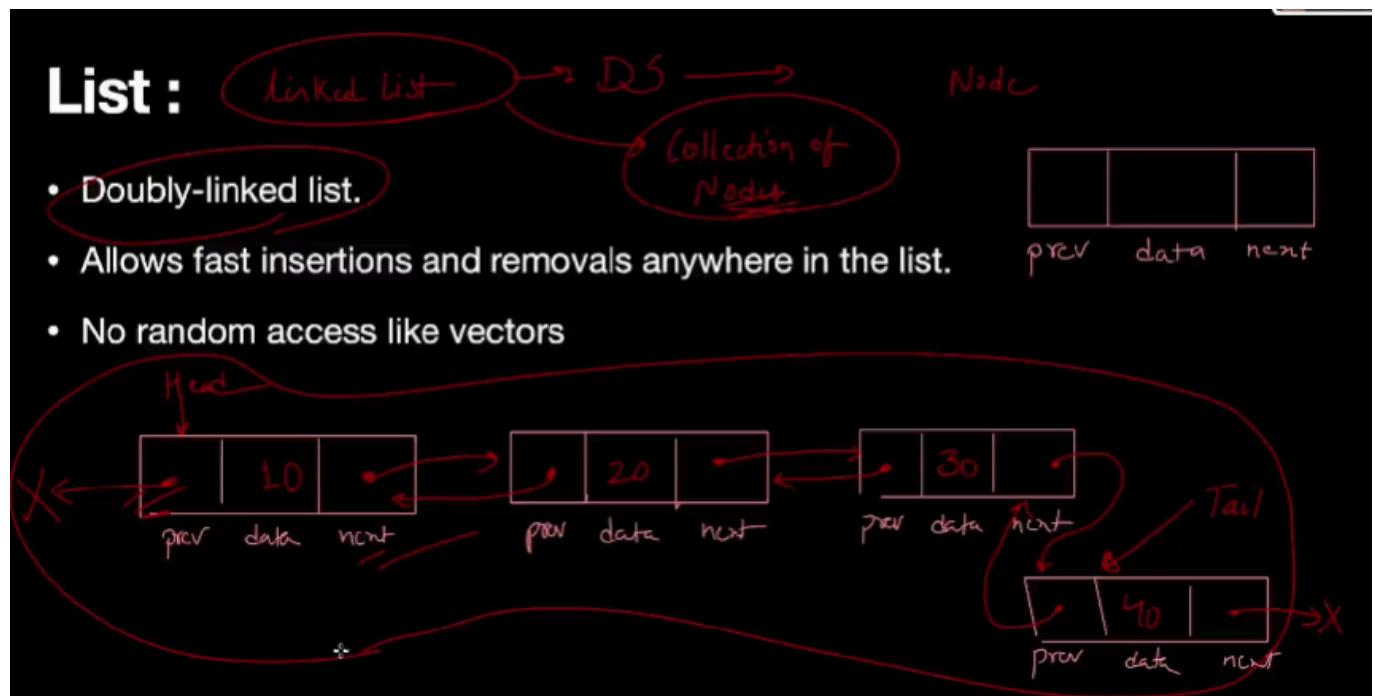
// Jagged Array columncount
cout << "Total columns : " << endl;
for (int i = 0; i < arr2.size(); i++)
{
    cout << "No. of columns in row " << i + 1 << " is " << arr2[i].size() <<
endl;
}
cout << endl;

return 0;
}

```

List:

Array require contagious memory location to store data, while List is non-contiguous approach to store user's data.



Member Functions:

Member Functions:

- `begin()`: Returns an iterator pointing to the first element in the list.
- `end()`: Returns an iterator pointing to the past-the-end element in the list.
- `size()`: Returns the number of elements in the list.
- `empty()`: Checks if the list is empty (i.e., whether its size is 0).
- `front()`: Accesses the first element in the list.
- `back()`: Accesses the last element in the list.
- `push_back(const T& value)`: Adds an element to the end of the list.
- `pop_back()`: Removes the last element from the list.
- `insert(iterator position, const T& value)`: Inserts a new element before the specified position in the list.
- `erase(iterator position)` or `erase(iterator first, iterator last)`: Removes one or more elements from the list starting at the specified position.
- `clear()`: Removes all elements from the list, which are destroyed, and leaves it with a size of 0.
- `swap(list& x)`: Swaps the contents of the list with those of another list of the same type, including their sizes.
- `pop_front()`: Removes the first element from the list.
- `push_front(const T& value)`: Adds an element to the beginning of the list.
- `remove(const T& value)`: Removes all elements from the list that are equal to the specified value.

Code:

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

int main()
{
    // creation
    list<int> myList;

    // insertion
    myList.push_back(10);
    myList.push_back(20);
    myList.push_back(30);

    myList.push_front(6);
    myList.push_front(4);
    myList.push_front(2);

    // removal
    myList.pop_back();
    myList.pop_front();

    // size
    cout << "Size of list : " << myList.size() << endl
        << endl;

    // clear
    myList.clear();
    cout
```

```
<< "Size of list : " << myList.size() << endl
<< endl;

// is empty
if (myList.empty())
{
    cout << "List is empty!" << endl
        << endl;
}
else
{

    cout << "List is not empty!" << endl
        << endl;
}

// front
myList.push_back(2);
myList.push_front(1);
cout << "Front: " << myList.front() << endl
    << endl;

// back
cout << "Back: " << myList.back() << endl
    << endl;

// traverse'
list<int>::iterator it = myList.begin();
cout << "The list : ";
while (it != myList.end())
{
    cout << *it << " ";
    it++;
}
cout << endl
    << endl;

// remove
myList.push_back(1);
// now my list is 1=>2=>1
myList.remove(1);
cout << "The list after using remove() : ";
list<int>::iterator it2 = myList.begin();
while (it2 != myList.end())
{
    cout << *it2 << " ";
    it2++;
}
cout << endl
    << endl;

// swap
list<int> first;
list<int> second;
```

```
first.push_back(10);
first.push_back(20);
second.push_back(1);
second.push_back(2);

cout << "The list before : ";
list<int>::iterator it3 = first.begin();
while (it3 != first.end())
{
    cout << *it3 << " ";
    it3++;
}
cout << endl
    << endl;

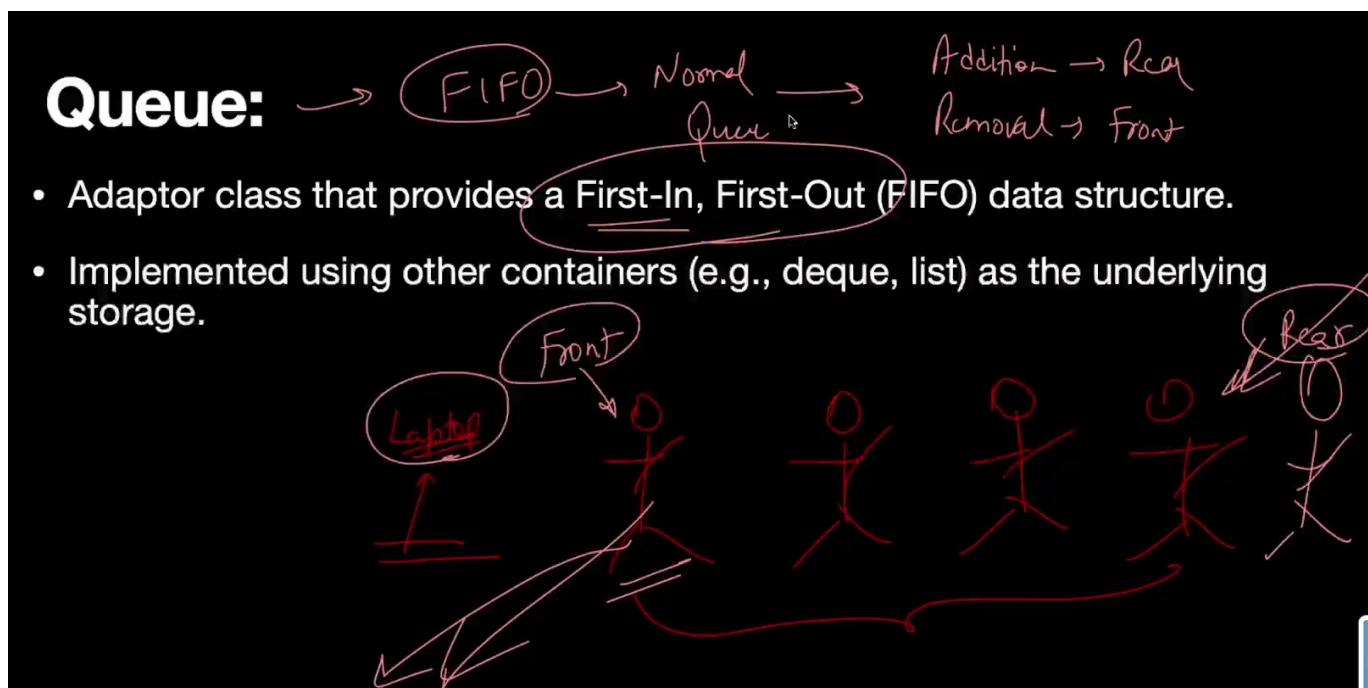
first.swap(second);

cout << "The list after : ";
list<int>::iterator it4 = first.begin();
while (it4 != first.end())
{
    cout << *it4 << " ";
    it4++;
}
cout << endl
    << endl;

// insert (insert(iterator,value))
// first : 1=>2
first.insert(first.begin(), 0);
cout << "The list after using insert : ";
list<int>::iterator it5 = first.begin();
while (it5 != first.end())
{
    cout << *it5 << " ";
    it5++;
}
cout << endl
    << endl;

// erase (erase(start_address, end_address))
first.erase(first.begin(), first.end());
cout << "The list after using erase : ";
list<int>::iterator it6 = first.begin();
while (it6 != first.end())
{
    cout << *it6 << " ";
    it6++;
}
cout << endl
    << endl;
return 0;
}
```

Queue:



Member Function:

Member Functions:

- `empty()`: Checks if the queue is empty (i.e., whether its size is 0).
- `size()`: Returns the number of elements in the queue.
- `front()`: Accesses the first element in the queue, which is the next element to be removed.
- `back()`: Accesses the last element in the queue, which is the most recently added element.
- `push(const T& value)`: Adds an element to the end of the queue.
- `pop()`: Removes the first element from the queue.
- `swap(queue& x)`: Swaps the contents of the queue with those of another queue of the same type.

Code:

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

int main()
{
    // creation
    queue<int> q;
```

```
// insertion
q.push(10);
q.push(20);
q.push(30);
// 10,20,30

// removal
q.pop();
// 20,30

// size
cout << "The Queue size is " << q.size() << endl
    << endl;

// empty
if (q.empty())
{
    cout << "Queue is Empty!" << endl
        << endl;
}
else
{
    cout << "Queue is not Empty!" << endl
        << endl;
}

// front
cout << "Front : " << q.front() << endl
    << endl;

// back
cout << "Back : " << q.back() << endl
    << endl;

// swap
queue<int> first;
queue<int> second;

first.push(1);
first.push(2);

second.push(10);
second.push(20);

cout << "Queue 1 before swap : " << first.front() << " " << first.back() <<
endl
    << endl;

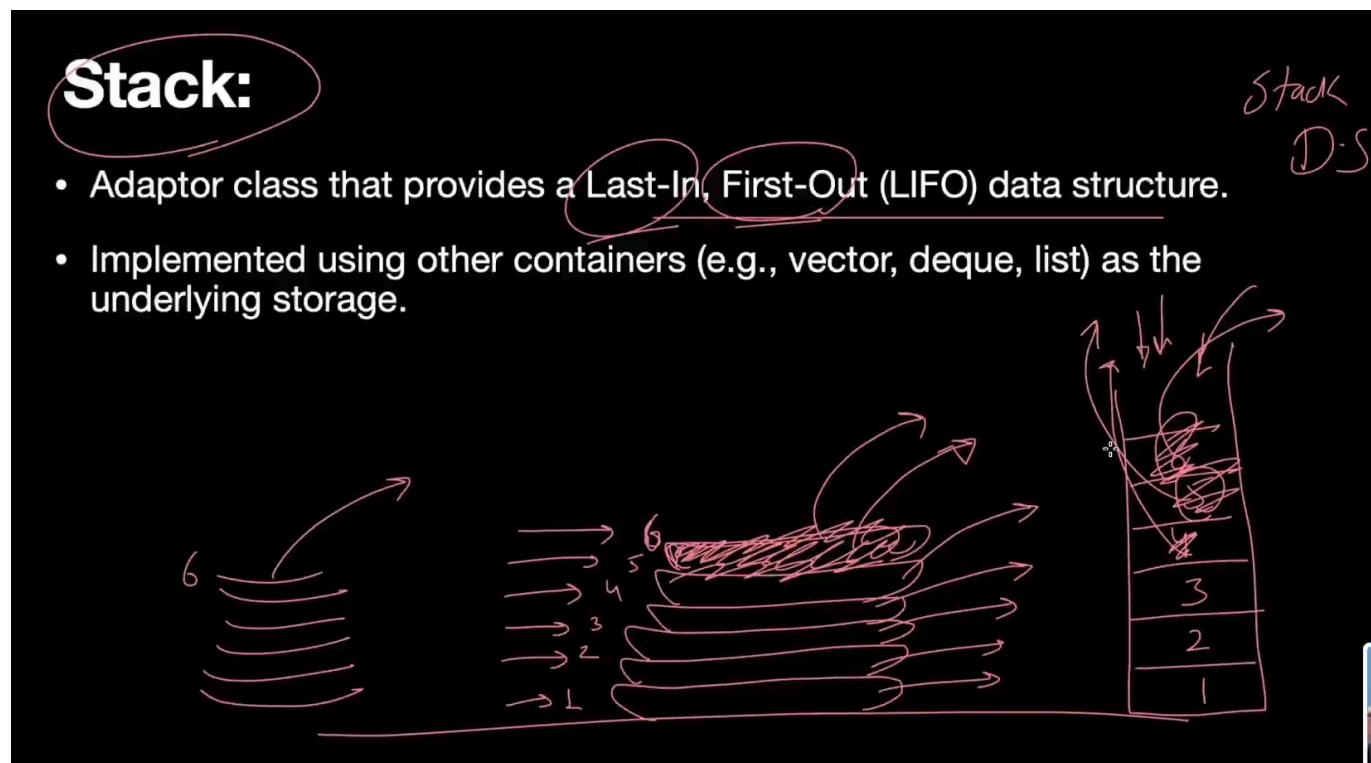
first.swap(second);

cout << "Queue 1 after swap : " << first.front() << " " << first.back() <<
endl
    << endl;
```

```
// printing
cout << "Queue : ";
while (!first.empty())
{
    cout << first.front() << " ";
    first.pop();
}
cout << endl;

return 0;
}
```

Stack:



Member Functions:

Member Functions:

- `empty()`: Checks if the stack is empty (i.e., whether its size is 0).
- `size()`: Returns the number of elements in the stack.
- `top()`: Accesses the top element of the stack, which is the most recently added element.
- `push(const T& value)`: Adds an element to the top of the stack.
- `pop()`: Removes the top element from the stack.
- `swap(stack& x)`: Swaps the contents of the stack with those of another stack of the same type.

Code:

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

int main()
{
    // creation
    stack<int> st;

    // insertion
    st.push(10);
    st.push(20);
    st.push(30);
    // st =>
    //   30
    //   20
    //   10

    // removal
    st.pop();
    // st =>
    //   20
    //   10

    // size
    cout << "Size : " << st.size() << endl
        << endl;

    // top => to observe/access topmost element in stack
    cout << "Top : " << st.top() << endl
        << endl;
```

```
// empty
if (st.empty())
{
    cout << "Stack is empty!" << endl
        << endl;
}
else
{
    cout << "Stack is not empty!" << endl
        << endl;
}

// swap
stack<int> first;
stack<int> second;

first.push(1);
first.push(2);
second.push(10);
second.push(20);

cout << "First stack top before swap: " << first.top() << endl
    << endl;

first.swap(second);

cout
    << "First stack top after swap: " << first.top() << endl
    << endl;

// printing
cout << "Stack : ";
while (!first.empty())
{
    cout << first.top() << " ";
    first.pop();
}
cout << endl;

return 0;
}
```

Deque:

Deque:

- Double-ended queue.
- Similar to vectors but allows efficient insertion and removal at both ends.
- Suitable when elements need to be inserted or removed frequently from the front or back.



Member Functions:

Member Functions:

- `begin()`: Returns an iterator pointing to the first element in the deque.
- `end()`: Returns an iterator pointing to the past-the-end element in the deque.
- `size()`: Returns the number of elements currently in the deque.
- `empty()`: Checks if the deque is empty (i.e., whether its size is 0).
- `front()`: Accesses the first element in the deque.
- `back()`: Accesses the last element in the deque.
- `operator[](size_type n)`: Accesses the element at the specified index without bounds checking.
- `at(size_type n)`: Accesses the element at the specified index with bounds checking.
- `push_back(const T& value)`: Adds an element to the end of the deque.
- `pop_back()`: Removes the last element from the deque. `pop_front()`: Removes the first element from the deque.
- `push_front(const T& value)`: Adds an element to the beginning of the deque.
- `insert(iterator position, const T& value)`: Inserts a new element before the specified position in the deque.
- `erase(iterator position)` or `erase(iterator first, iterator last)`: Removes one or more elements from the deque starting at the specified position.
- `clear()`: Removes all elements from the deque, which are destroyed, and leaves it with a size of 0.
- `swap(deque& x)`: Swaps the contents of the deque with those of another deque of the same type, including their sizes.

Code:

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

int main()
{
    // creation
    deque<int> dq;

    // insertion
    dq.push_back(10);
    dq.push_back(20);
    dq.push_back(30);
    // dq => 10,20,30
```

```
dq.push_front(7);
dq.push_front(5);
dq.push_front(3);
// dq => 3,5,7,10,20,30

// removal
dq.pop_back();
// dq => 3,5,7,10,20
dq.pop_front();
// dq => 5,7,10,20

// size
cout << "Size : " << dq.size() << endl
    << endl;

// front
cout << "Front : " << dq.front() << endl
    << endl;

// back
cout << "Back : " << dq.back() << endl
    << endl;

// empty
if (dq.empty())
{
    cout << "Deque is empty!" << endl
        << endl;
}
else
{
    cout << "Deque is not empty!" << endl
        << endl;
}

// iterator
deque<int>::iterator it = dq.begin();
cout << "Deque : ";
while (it != dq.end())
{
    cout << *it << " ";
    it++;
}
cout << endl
    << endl;

// dq[i] and dq.at(i)
// dq => 5,7,10,20
cout << "Value at 1 index : " << dq[1] << endl
    << endl;
cout << "Value at 1 index using at : " << dq.at(1) << endl
    << endl;

// Now insert(pos, value), erase(pos1, pos2), clear(), swap() are same as used
```

in vector.

```

// erase
dq.erase(dq.begin(), dq.end());
cout << "Size of dq after erase : " << dq.size() << endl
    << endl;

// insert
dq.insert(dq.begin(), 1);
cout << "Value at 0 index after insert(): " << dq[0] << endl
    << endl;

// clear
cout << "Deque Size : " << dq.size() << endl
    << endl;
dq.clear();
cout << "Now Deque Size : " << dq.size() << endl;

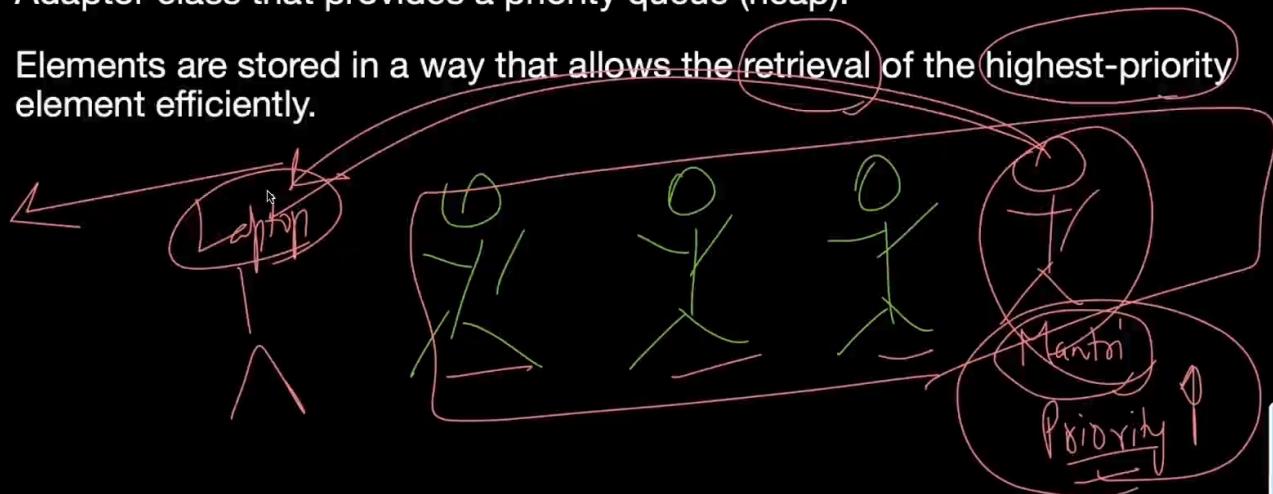
return 0;
}

```

Priority Queue:

Priority queue:

- Adaptor class that provides a priority queue (heap).
- Elements are stored in a way that allows the retrieval of the highest-priority element efficiently.



Code:

```

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

int main()
{

```

```
// min-heap => minimum value => highest priority:  
// creation  
priority_queue<int, vector<int>, greater<int>> pq1;  
  
// insertion  
pq1.push(10);  
pq1.push(30);  
pq1.push(20);  
// pq =>  
// 10  
// 20  
// 30  
  
// removal  
pq1.pop();  
// pq =>  
// 20  
// 30  
  
// top => it has highest priority  
cout << "Top element (min-heap): " << pq1.top() << endl  
     << endl;  
  
// size  
cout << "Size (min-heap) : " << pq1.size() << endl  
     << endl;  
  
// ****  
// creation  
priority_queue<int> pq;  
  
// max-heap => maximum value => highest priority  
  
// insertion  
pq.push(10);  
pq.push(30);  
pq.push(20);  
pq.push(60);  
// pq =>  
// 60  
// 30  
// 20  
// 10  
  
// removal  
pq.pop();  
// pq =>  
// 30  
// 20  
// 10  
  
// top => it has highest priority  
cout << "Top element : " << pq.top() << endl
```

```
<< endl;

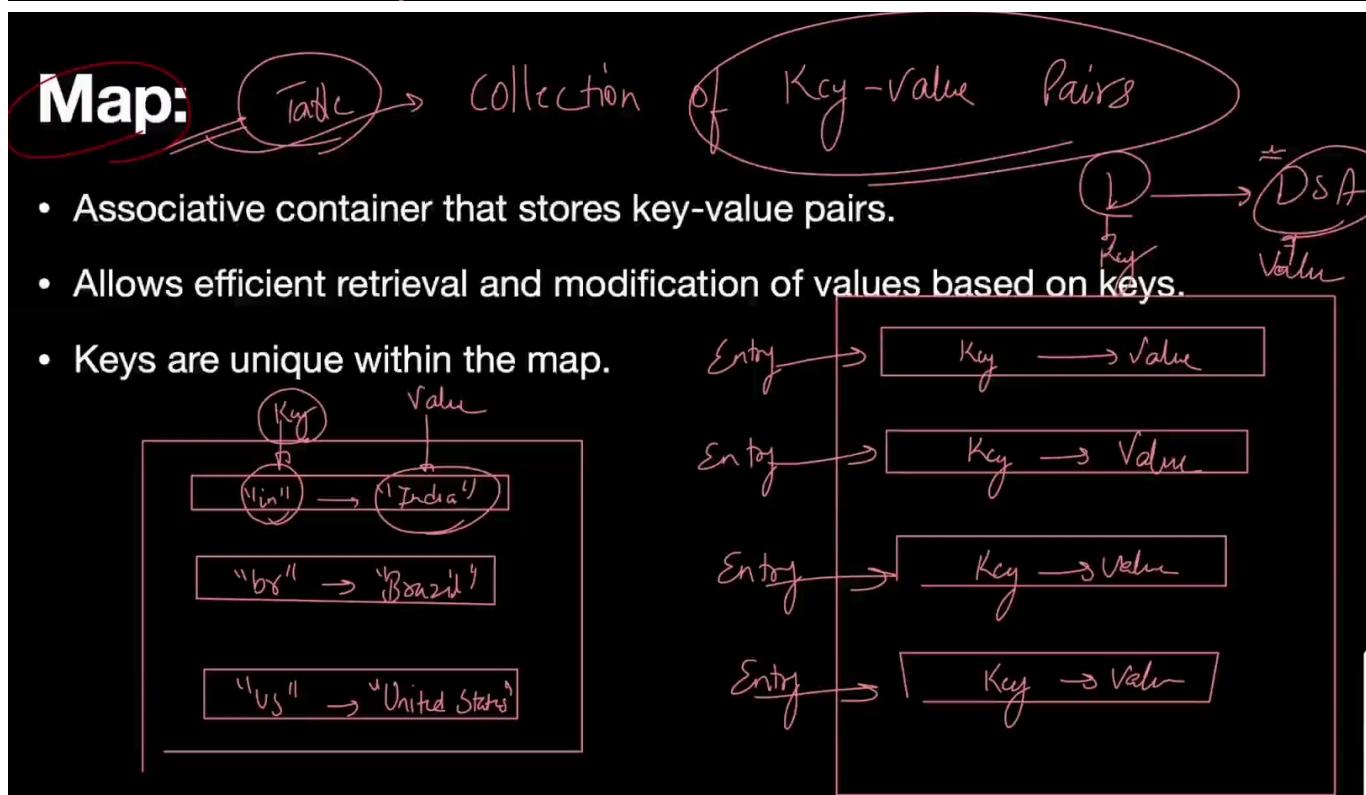
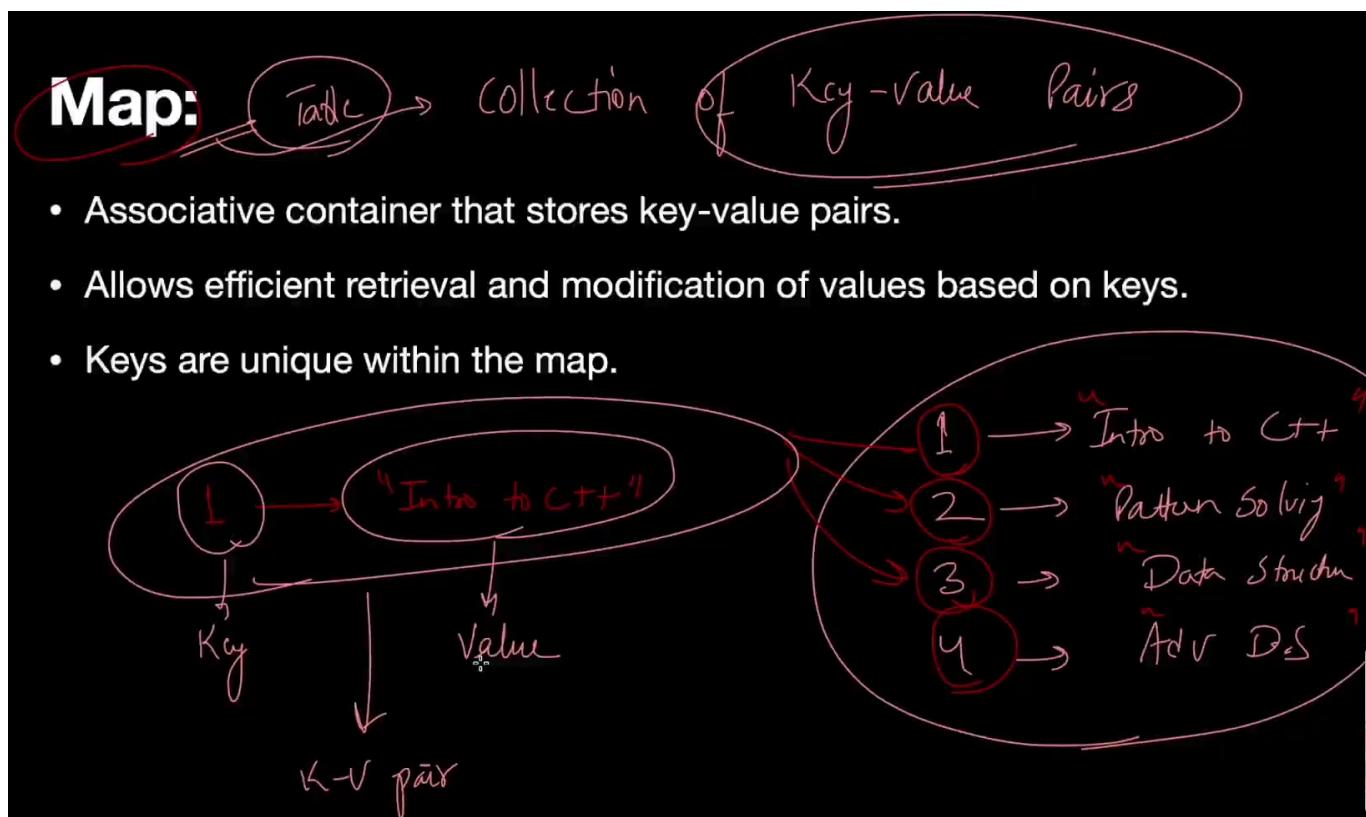
// size
cout << "Size : " << pq.size() << endl
    << endl;

// empty
if (pq.empty())
{
    cout << "Priority Queue is Empty!" << endl
        << endl;
}
else
{
    cout << "Priority Queue is not Empty!" << endl
        << endl;
}

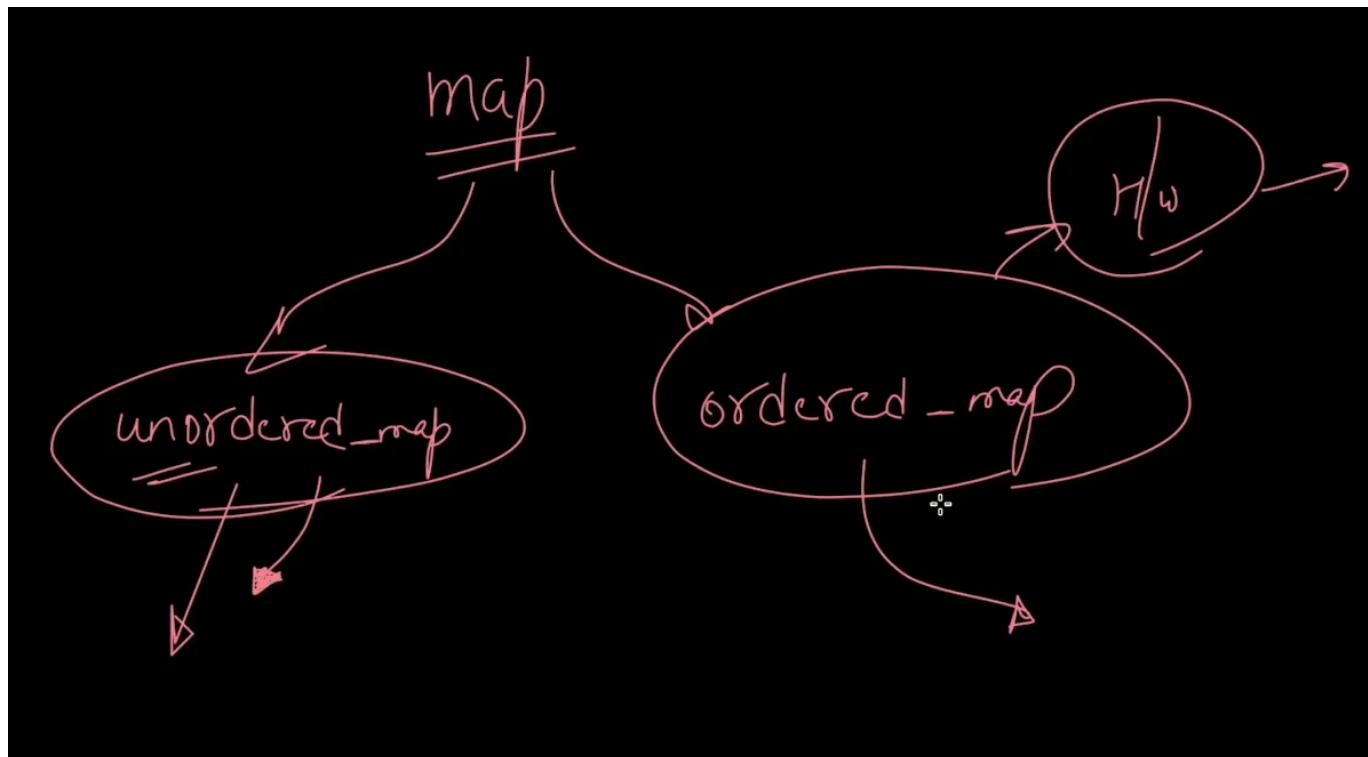
// swap() is same as others.

return 0;
}
```

Map:



Types:



Member Functions:

Member Functions:

- `begin()`: Returns an iterator pointing to the first element (key-value pair) in the map.
- `end()`: Returns an iterator pointing to the past-the-end element in the map.
- `empty()`: Checks if the map is empty (i.e., whether its size is 0).
- `size()`: Returns the number of key-value pairs currently in the map.
- `operator[](const Key& key)`: Accesses the value associated with the key, inserting a new element if the key does not exist.
- `at(const Key& key)`: Accesses the value associated with the key, throwing an exception if the key is not found.
- `insert(const std::pair<Key, Value>& value)` or `insert(iterator hint, const std::pair<Key, Value>& value)`: Inserts a new key-value pair into the map; with a hint, it can potentially improve insertion efficiency.
- `erase(const Key& key)` or `erase(iterator position)` or `erase(iterator first, iterator last)`: Removes one or more elements from the map specified by key or position.
- `clear()`: Removes all key-value pairs from the map, which are destroyed, and leaves it with a size of 0.
- `find(const Key& key)`: Returns an iterator to the element with the given key, or `end()` if the key is not found.
- `count(const Key& key)`: Returns the number of elements with the specified key (1 or 0 since `std::map` does not allow duplicate keys).

Code:

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

int main()
{
```

```
// ordered map:  
// all insertion and operations complexity are O(log n)  
  
// creation  
map<string, string> table1;  
  
// insertion  
table1["in"] = "India";  
table1.insert(make_pair("en", "England"));  
pair<string, string> p1;  
p1.first = "br";  
p1.second = "Brazil";  
table1.insert(p1);  
  
// traverse  
map<string, string>::iterator it1 = table1.begin();  
cout << "The Ordered Map : " << endl;  
while (it1 != table1.end())  
{  
    pair<string, string> p1 = *it1;  
    cout << "key : " << p1.first << " , " << "value : " << p1.second << endl;  
    it1++;  
}  
cout << endl;  
  
// ****  
// creation  
unordered_map<string, string> table;  
  
// insertion  
table["in"] = "India";  
  
table.insert(make_pair("en", "England"));  
  
pair<string, string> p;  
p.first = "br";  
p.second = "Brazil";  
table.insert(p);  
  
// size  
cout << "Size : " << table.size() << endl  
    << endl;  
  
// table[key] and table.at(key)  
cout << "Value at key=>in : " << table["in"] << endl  
    << endl;  
cout << "Value at key=>in using at() : " << table.at("in") << endl  
    << endl;  
  
// updation  
table.at("in") = "Bharat";  
cout << "Updated key=>in : " << table.at("in") << endl  
    << endl;
```

```
// iterator
unordered_map<string, string>::iterator it = table.begin();
cout << "The Map : " << endl;
while (it != table.end())
{
    pair<string, string> p = *it;
    cout << "key : " << p.first << " , " << "value : " << p.second << endl;
    it++;
}
cout << endl;

// find
if (table.find("jp") != table.end())
{
    cout << "key found!" << endl
        << endl;
}
else
{
    cout << "key not found!" << endl
        << endl;
}

// count
if (table.count("in") == 0)
{
    cout << "(count) key not found!" << endl
        << endl;
}
if (table.count("in") == 1)
{
    cout << "(count) key found!" << endl
        << endl;
}

// clear
table.clear();
cout << "Size after clear : " << table.size() << endl
    << endl;

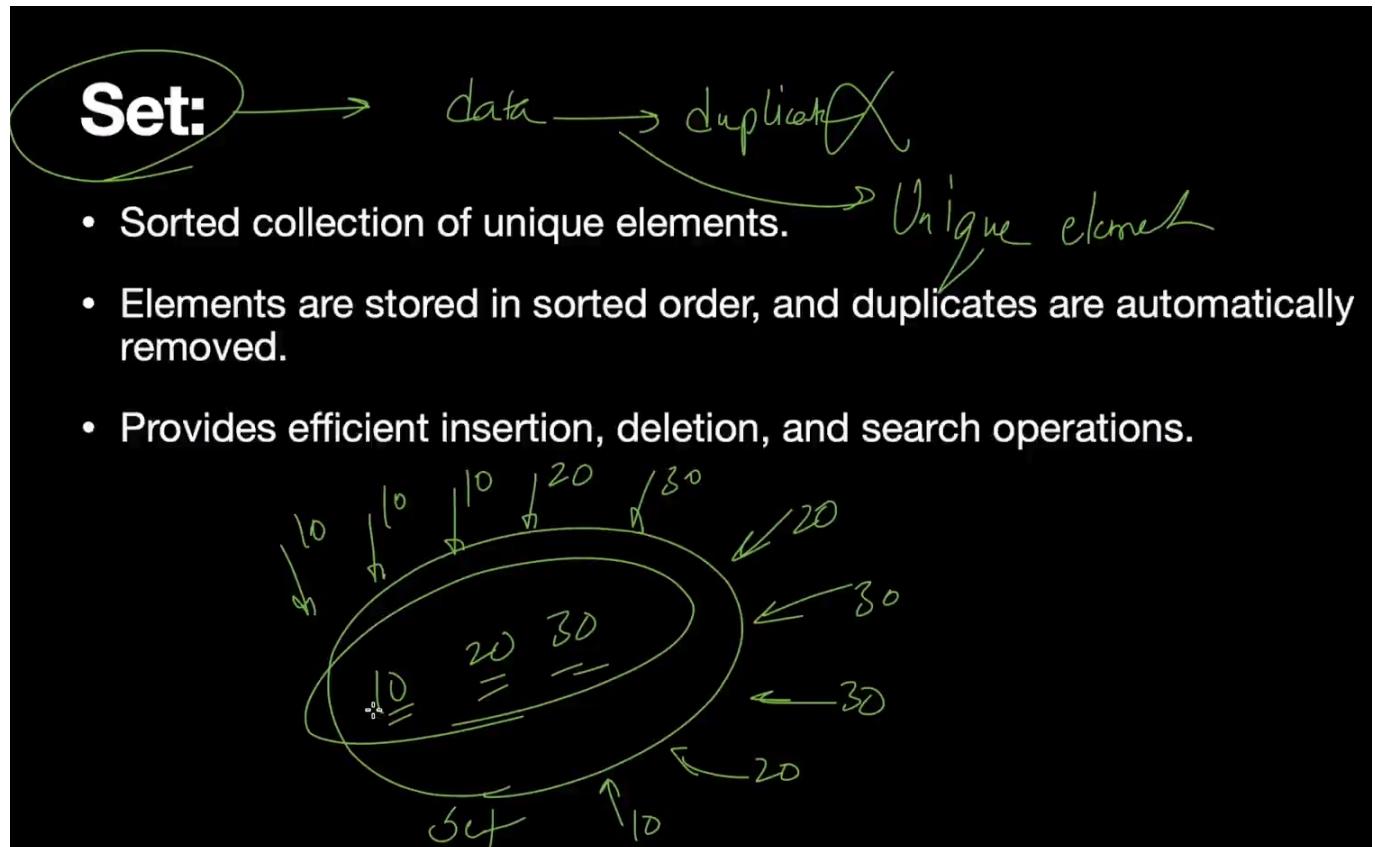
// empty
if (table.empty())
{
    cout << "Map is empty!" << endl
        << endl;
}
else
{
    cout << "Map is not empty!" << endl
        << endl;
}

// erase
```

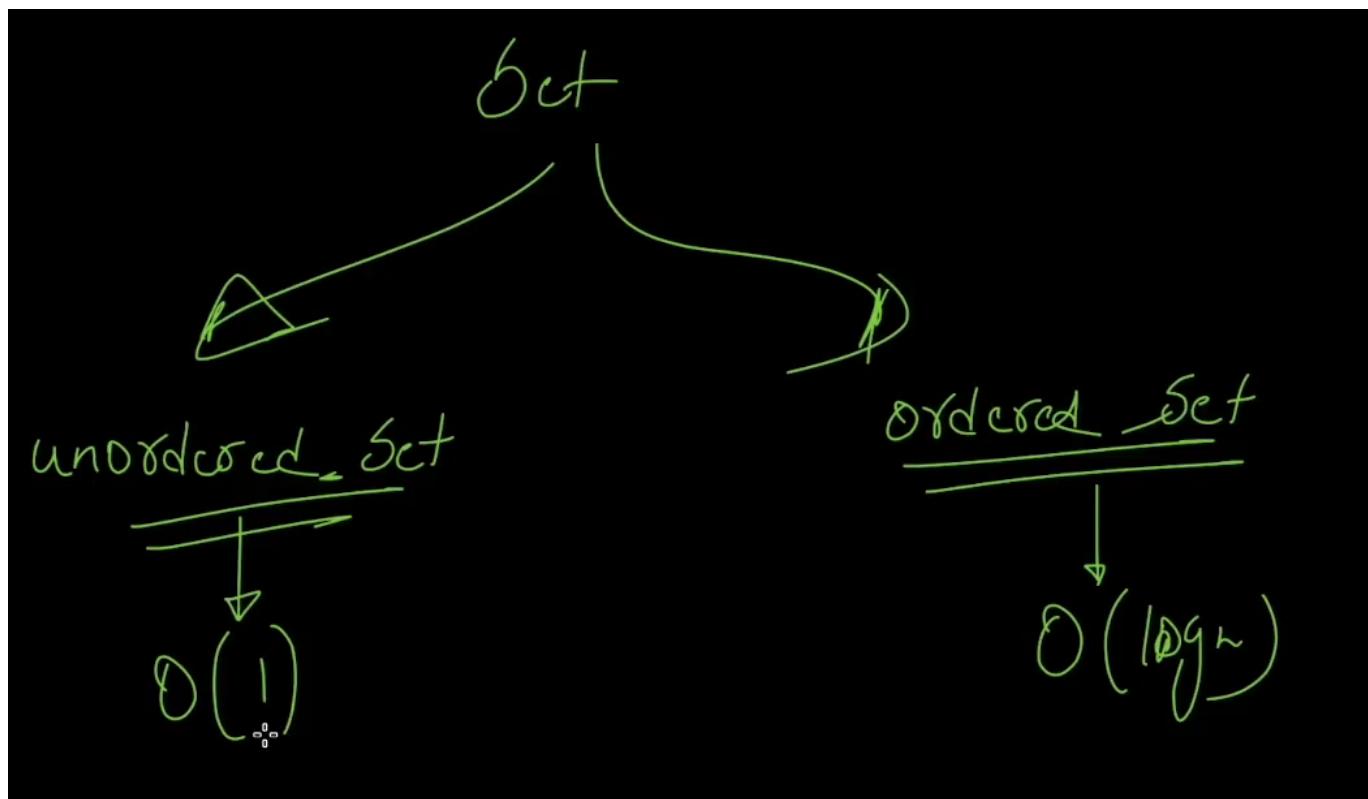
```
    table["in"] = "India";
    table["ch"] = "China";
    table["sr"] = "Sri Lanka";
    table.erase(table.begin(), table.end());
    cout << "Size of table after erase : " << table.size() << endl
        << endl;

    return 0;
}
```

Set:



Types:



Member Functions:

- ### Member Functions:
- `begin()`: Returns an iterator pointing to the first element in the set.
 - `end()`: Returns an iterator pointing to the past-the-end element in the set.
 - `empty()`: Checks if the set is empty (i.e., whether its size is 0).
 - `size()`: Returns the number of elements currently in the set.
 - `insert(const T& value)`: Inserts a new element into the set, maintaining the order and uniqueness of elements.
 - `erase(const T& key)` or `erase(iterator position)` or `erase(iterator first, iterator last)`: Removes one or more elements from the set specified by key or position.
 - `clear()`: Removes all elements from the set, which are destroyed, and leaves it with a size of 0.
 - `find(const T& key)`: Returns an iterator to the element with the given key, or `end()` if the key is not found.
 - `count(const T& key)`: Returns the number of elements with the specified key (1 or 0, since `std::set` does not allow duplicate keys).

Code:

```
#include <iostream>
#include <set>
#include <unordered_set>
using namespace std;

int main()
```

```
{  
    // creation  
    set<int> s;  
  
    // insertion  
    s.insert(10);  
    s.insert(30);  
    s.insert(20);  
  
    // traverse  
    set<int>::iterator it = s.begin();  
    cout << "Set : " << endl;  
    while (it != s.end())  
    {  
        cout << *it << " ";  
        it++;  
    }  
    cout << endl  
        << endl;  
  
    // all other methods size(), clear(), empty(), erase(), find(), count() are  
    same.  
  
    // *****  
  
    // unordered set  
    // creation  
    unordered_set<int> s1;  
  
    // insertion  
    s1.insert(10);  
    s1.insert(30);  
    s1.insert(20);  
  
    // traverse  
    unordered_set<int>::iterator it1 = s1.begin();  
    cout << "Unordered Set : " << endl;  
    while (it1 != s1.end())  
    {  
        cout << *it1 << " ";  
        it1++;  
    }  
    cout << endl;  
  
    // all other methods size(), clear(), empty(), erase(), find(), count() are  
    same.  
  
    return 0;  
}
```