Contents

STL Introduction –

The C++ Standard Template Library (STL) is a set of template classes and functions that provides the implementation of common data structures and algorithms such as lists, stacks, arrays, sorting, searching, etc. It also provides the iterators and functors which makes it easier to work with algorithms and containers.

Components of STL

The components of STL are the features provided by Standard Template Library (STL) in C++ that can be classified into 4 types:

1. **Containers**
2. **Algorithms**
3. **Iterators**
4. **Functors**

Vectors in C++

Vectors are the same as dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Syntax to Declare Vector in C++

vector<dataType> vectorName;

1. Initialization Using List

vector<dataType> name({ *value1, value2, value3* ....});

2. Initialization With a Single Value

vector<dataType> name(size, value);

3. Initialization From Another Vector

vector<dataType> name(other_vec);

All Member Functions of std::vector

Following is the list of all member functions of std::vector class in C++:

Vector Function	Description
push_back()	Adds an element to the end of the vector.
pop_back()	Removes the last element of the vector.
size()	Returns the number of elements in the vector.
max_size()	Returns the maximum number of elements that the vector can hold.
resize()	Changes the size of the vector.
empty()	Checks if the vector is empty.
operator[]	Accesses the element at a specific position.
at()	Accesses the element at a specific position, with bounds checking.
front()	Accesses the first element of the vector.
back()	Accesses the last element of the vector.
begin()	Returns an iterator pointing to the first element of the vector.
end()	Returns an iterator pointing to the past-the-end element of the vector.
rbegin()	Returns a reverse iterator pointing to the last element of the vector.

Vector Function	Description
rend()	Returns a reverse iterator pointing to the element preceding the first element of the vector.
cbegin	Returns const_iterator to beginning
cend	Returns const_iterator to end
crbegin	Returns const_reverse_iterator to reverse beginning
crend	Returns const_reverse_iterator to reverse end
insert()	Inserts elements at a specific position in the vector.
erase()	Removes elements from a specific position or range in the vector.
swap()	Swaps the contents of the vector with those of another vector.
clear()	Removes all elements from the vector.
emplace()	Constructs and inserts an element in the vector.
emplace_back()	Constructs and inserts an element at the end of the vector.
assign()	Assigns new values to the vector elements by replacing old ones.
capacity()	Returns the size of the storage space currently allocated to the vector.

Vector Function	Description
reserve()	Requests that the vector capacity be at least enough to contain a specified number of elements.
shrink_to_fit()	Reduces memory usage by freeing unused space.
data()	Returns a direct pointer to the memory array used internally by the vector to store its owned elements.
get_allocator	Returns a copy of the allocator object associated with the vector.

Vector code Example

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

//function to print vector

void printvector(vector<int> temp , string funcname){
  cout<<"\n";
  cout<<"After "<<funcname <<" = ";
  for(auto i : temp){
    cout<< i <<" ";
  }
  cout<<endl;
}


int main()
{
    // Declaring vectors
    vector<int> vec ;
```

```
    vector<int> vec1 = {1,2,3};

    vector<int> vec2(4,3);

    vector<int> vec3(vec2);

    cout<<"Printing Vector"<<endl;

    for(auto it : vec3){
      cout<<it<<" ";
    }

    //creating vector
    vector<int> vec4 ;
    vec4.push_back(1);
    vec4.push_back(3);
    vec4.push_back(5);
    vec4.push_back(7);
    vec4.push_back(9);
    vec4.push_back(11);

    //size
    cout<<"\nSize of vector "<<vec4.size();

    vec4.pop_back();

    //print vector
    printvector(vec4 , "Pop");

    //index and front back
    cout<<"\n\nPrint Idx "<<vec4[1] <<" or print at idx 1 =
"<<vec4.at(1)<<endl;
    cout<<"\nFront and Back = "<<vec4.front() << vec4.back()<<endl;

    //erase , insert function
    vec4.erase(vec4.begin());
    printvector(vec4 , "Erase begin");
    vec4.erase(vec4.begin()+1 , vec4.begin()+3);
    printvector(vec4 , "Erase begin 3 ");
    vec4.insert(vec4.begin()+2 , 200);
    printvector(vec4 , "Insert ");
    vec4.empty();

    cout<<"print begin = "<<*(vec4.begin());

    //looping and iterator
    cout<<"\nPrint vector using iterator\n";
```

```
    vector<int> :: iterator it;
    for(it = vec4.begin() ; it != vec4.end() ; it++){
      cout<<*(it)<<" ";
    }
    cout<<"\nPrint vector using reverse iterator\n";
    vector<int> :: reverse_iterator it1;
    for(it1 = vec4.rbegin() ; it1 != vec4.rend() ; it1++){
      cout<<*(it1)<<" ";
    }
    cout<<"\nPrint vector using Auto iterator\n";
    for(auto it3 : vec4){
      cout<<(it3)<<" ";
    }


}
```

Vector code Output :

Printing Vector

3 3 3 3

Size of vector 6

After Pop = 1 3 5 7 9

Print Idx 3 or print at idx 1 = 3

Front and Back = 19

After Erase begin = 3 5 7 9

After Erase begin 3 = 3 9

After Insert = 3 9 200

print begin = 3

Print vector using iterator

3 9 200

Print vector using reverse iterator

200 9 3

Print vector using Auto iterator

3 9 200

List in C++

when we say a List, we talk about a doubly linked list.

Syntax:

list <data-type> name_of_list;

Functions	Definition
front()	Returns the value of the first element in the list.
back()	Returns the value of the last element in the list.
push_front(g)	Adds a new element 'g' at the beginning of the list.
push_back(g)	Adds a new element 'g' at the end of the list.
pop_front()	Removes the first element of the list, and reduces the size of the list by 1.
pop_back()	Removes the last element of the list, and reduces the size of the list by 1.
list::begin()	begin() function returns an iterator pointing to the first element of the list.
list::end()	end() function returns an iterator pointing to the theoretical last element which follows the last element.
list rbegin() and rend()	rbegin() returns a reverse iterator which points to the last element of the list. rend() returns a reverse iterator that points to the position before the beginning of the list.

Functions	Definition
list cbegin() and cend()	cbegin() returns a constant random access iterator which points to the beginning of the list. cend() returns a constant random access iterator which points to the end of the list.
list crbegin() and crend()	crbegin() returns a constant reverse iterator which points to the last element of the list i.e reversed beginning of the container. crend() returns a constant reverse iterator which points to the theoretical element preceding the first element in the list i.e. the reverse end of the list.
empty()	Returns whether the list is empty(1) or not(0).
insert()	Inserts new elements in the list before the element at a specified position.
erase()	Removes a single element or a range of elements from the list.
assign()	Assigns new elements to the list by replacing current elements and resizing the list.
remove()	Removes all the elements from the list, which are equal to a given element.
list::remove_if()	Used to remove all the values from the list that correspond true to the predicate or condition given as a parameter to the function.
reverse()	Reverses the list.
size()	Returns the number of elements in the list.
list resize()	Used to resize a list container.
sort()	Sorts the list in increasing order.
list max_size()	Returns the maximum number of elements a list container can hold.

Functions	Definition
list unique()	Removes all duplicate consecutive elements from the list.
list::emplace_front() and list::emplace_back()	.emplace_front() function is used to insert a new element into the list container and constructs the object in-place at the beginning of the list. . emplace_back() function is used to insert a new element into the list container,and constructs the object in-place at the end of the list.
list::clear()	clear() function is used to remove all the elements of the list container, thus making it size 0.
list::operator=	This operator is used to assign new contents to the container by replacing the existing contents.
list::swap()	This function is used to swap the contents of one list with another list.
list splice()	Used to transfer elements from one list to another.
list merge()	Merges two sorted lists into one.
list emplace()	Extends the list by inserting a new element at a given position and it constructs the object in-place at the beginning of the list, potentially improving performance by avoiding a copy operation

List Code in C++

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


void print_list(list<int> l){

  for(auto ele : l){
    cout<<ele<<" ";
  }
  cout<<"\n";
}

int main()
{
    list<int> list1;
    list1.push_back(1);
    list1.push_back(2);
    list1.push_back(3);
    list1.push_front(4);

    print_list(list1);

    list1.pop_front();

    print_list(list1);

    list1.pop_back();

    print_list(list1);

    cout<<"size of list"<<list1.size()<<"\n";  }
```

Output :

4 1 2 3

1 2 3

1 2

size of list2

Deque in C++

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Syntax:

list <data-type> name_of_list;

Operations

Operation	Description
push_front()	Inserts the element at the beginning.
push_back()	Adds element at the end.
pop_front()	Removes the first element from the deque.
pop_back()	Removes the last element from the deque.
front()	Gets the front element from the deque.
back()	Gets the last element from the deque.
empty()	Checks whether the deque is empty or not.
size()	Determines the number of elements in the deque.

Deque Code in C++

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


void print_deque(deque<int> d){

  for(auto ele : d){
    cout<<ele<<" ";
  }
  cout<<"\n";
}

int main()
{
    deque<int> de1;
    de1.push_back(1);
    de1.push_back(2);
    de1.push_back(3);
    de1.push_front(4);

    print_deque(de1);

    de1.pop_front();

    print_deque(de1);

    de1.pop_back();

    print_deque(de1);

    cout<<"size of queue<T> ;"<<de1.size()<<"\n";   }
```

Output

4 1 2 3

1 2 3

1 2

size of queue<T> 2

Pair in C++

Pair is used to combine together two values that may be of different data types

Syntax:

pair <data_type1, data_type2> Pair_name

Pair Code in C++

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


int main()
{

  //Creating Pair
  pair<int , int>p = {1,2};

  //printing pair
  cout<<p.first <<" " <<p.second<<endl;

  //different data type
  pair<char , int>p1 = {'a',2};

  cout<<p1.first <<" " <<p1.second<<endl;

  //Pair inside Pair
  pair<int , pair<char , int> > p2 = {1, {'z' , '8'}};

  cout<<p2.first <<" " <<p2.second.first<<" "<<p2.second.second<<endl;

  //Creating vector of pair

  vector<pair<int , string>> vec = {{1 , "abc"}};
  vec.push_back({2,"def"});

  //Print vector
  for(auto v : vec){
```

```
    cout<<v.first <<" "<<v.second<<endl;
  }


}
```

Output :

1 2

a 2

1 z 56

1 abc

2 def

Stack in C++

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only

Syntax:

stack <data_type1> stack_name

The functions associated with stack are:

empty() – Returns whether the stack is empty

size() – Returns the size of the stack

top() – Returns a reference to the top most element of the stack

push(g) – Adds the element ‘g’ at the top of the stack Complexity : O(1)

pop() – Deletes the most recent entered element of the stack

Stack Code in C++

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

int main()
{

    //creating stack
    stack<int> s;
    s.push(3);
    s.push(2);
    s.push(1);
    s.push(0);

    //print size
    cout<<"size of stack"<<s.size()<<endl;

    //print stack
```

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

    //print size
    cout<<"size of stack"<<s.size()<<endl;
}
```

Output :

size of stack4

0

1

2

3

size of stack0

Queue in C++

Queues are a type of container adaptors that operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front

Syntax:

queue <data_type1> queue_name

Method	Definition
queue::empty()	Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
queue::size()	Returns the size of the queue.
queue::front()	Returns a reference to the first element of the queue.
queue::back()	Returns a reference to the last element of the queue.
queue::push(g)	Adds the element ‘g’ at the end of the queue.
queue::pop()	Deletes the first element of the queue.

Queue Code in C++

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

int main()
{

    //creating queue
    queue<int> q;
    q.push(3);
    q.push(2);
    q.push(1);
    q.push(0);

    //print queue
    cout<<"size of queue"<<q.size()<<endl;

    //print queue
    while(!q.empty()){
      cout<<q.front()<<" "<<endl;
      q.pop();
    }

    //print size
    cout<<"size of queue"<<q.size()<<endl;
}
```

Output :

size of queue4

3

2

1

0

size of queue0

Priority Queue in C++

A C++ priority queue is a type of container adapter, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue, and elements are in non-increasing or non-decreasing order

Syntax:

Priority_queue <data_type1> queue_name

Priority Queue Code in C++

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

int main()
{

    //creating queue
    priority_queue<int> q;
    q.push(3);
    q.push(5);
    q.push(1);
    q.push(10);

    //print queue
    cout<<"size of queue"<<q.size()<<endl;

    //print queue
    while(!q.empty()){
      cout<<q.top()<<" "<<endl;
      q.pop();
    }

    //print size
    cout<<"size of queue"<<q.size()<<endl;
}
```

Output :

size of queue4

10

5

3

1

size of queue0

Map in C++

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

Syntax:

map<key , value> map_name

Map code in C++:

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

int main()
{

    //creating map
    map<string , int> car;
    car["Nano"] = 100000;
    car["Creata"] = 1600000;

    car.insert({"Honda City" , 600000});

    //print map

    for(auto it : car){
      cout<<it.first <<"  "<<it.second<<"\n";
    }

    //check specific value
    cout<<"Creata price = " << car["Creata"]<<endl;

    //delete value
    car.erase("Creata");

    //find values
    if(car.find("Nano") != car.end()){
```

```
      cout<<"Nano Found";
    }else{
      cout<<"Nano not found";
    }

}
```

Output :

Creata 1600000

Honda City 600000

Nano 100000

Creata price = 1600000

Nano Found

Unorderd Map in C++

unordered_map is an associated container that stores elements formed by the combination of a key value and a mapped value. The key value is used to uniquely identify the element and the mapped value is the content associated with the key.

Syntax:

unorderd_map<key , value> map_name

Set in C++

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending.

Syntax:

Set<data_type > set_name

Set code in C++:

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

int main()
{

    //creating set
    set<string> car;
    car.insert("Nano");
    car.insert("Creata");
    car.insert("Creata");

    //print set

    for(auto it : car){
      cout<<it <<" ";
    }

    //print sizeof
    cout<<"\n"<<car.size() <<" is size of set\n";

    //understanding sorting
    set<int> num;
    num.insert({1,2,7,8,5,4,1});

    for(auto n : num){
      cout<<n <<" ";
    }
```

```
}
```

Output :

Creata Nano

2 is size of set

1 2 4 5 7 8

Unorderd set in C++

An unordered_set is an unordered associative container implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered_set take constant time O(1) on an average which can go up to linear time O(n) in the worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

Syntax:

unorderd_set<data_type> set_name

Sorting Algorithms in STL

The C++ Standard Template Library (STL) provides powerful sorting functions that work with a wide variety of data types and structures. STL sorting algorithms are efficient, easy to use, and optimized for performance.

1. Available Sorting Algorithms in STL

a) sort

- **Function**: std::sort(start, end)
- **Description**: Sorts the elements in the specified range [start, end) in ascending order by default.
- **Complexity**:
  - **Average Case**: O(n log n)
  - **Worst Case**: O(n log n)
- **Uses**: QuickSort for smaller arrays and IntroSort (combines QuickSort, HeapSort, and InsertionSort) for larger arrays to ensure efficiency.
- **Customization**: A comparator function can be passed to sort in descending order or by custom criteria.

Example:

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

using namespace std ;
int main() {
    vector<int> numbers = {3, 1, 4, 1, 5, 9};
    sort(numbers.begin(), numbers.end()); // Ascending order

    // Display sorted vector

    for (int num : numbers) {
        cout << num << " ";
    }
    return 0;
}
```

To sort array in descending order we use function

sort(start , end , greater<int>())

b) nth_element

- **Function**: std::nth_element(start, nth, end)
- **Description**: Rearranges the elements in the range [start, end) so that the element at the nth position is the same as if the entire range were sorted, with elements before it less and elements after greater.
- **Complexity**: O(n) average time complexity.
- **Use Case**: Efficient for finding the k-th smallest or largest element without fully sorting.

Example:

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

using namespace std ;

int main() {
    vector<int> numbers = {10, 5, 20, 15, 25, 0};
    nth_element(numbers.begin(), numbers.begin() + 2, numbers.end());

    // Display rearranged vector with 3rd smallest element in place
    for (int num : numbers) {
        cout << num << " ";
    }
    return 0;
}
```

Custom Sorting with Comparator Functions

- **Comparator**: A custom comparator can be passed to std::sort, std::stable_sort, etc., to sort in descending order or by custom criteria.

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

using namespace std;

struct Person {
    string name;
    int age;
};

bool compareByAge(const Person& a, const Person& b) {
    return a.age < b.age;
}

int main() {
    vector<Person> people = {{"Alice", 30}, {"Bob", 25}, {"Charlie", 35}};
    sort(people.begin(), people.end(), compareByAge);
```

```
    // Display sorted people by age
    for (const Person& p : people) {
        cout << p.name << " (" << p.age << ")\n";
    }
    return 0;
}
```

Other Algorithms

1. std::reverse

- **Description**: Reverses the order of elements in the specified range.
- **Syntax**: reverse(start, end);
- **Example**:

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

int main() {
   vector<int> numbers = {1, 2, 3, 4, 5};
   reverse(numbers.begin(), numbers.end());
}
```

2. std::rotate

- **Description**: Rotates the elements in a range, moving elements in [start, middle) to the end, and [middle, end) to the beginning.
- **Syntax**: rotate(start, middle, end);
- **Example**:

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

int main() {
   vector<int> numbers = {1, 2, 3, 4, 5};
   rotate(numbers.begin(), numbers.begin() + 2, numbers.end());
}
```

3. std::shuffle

- **Description**: Randomly shuffles elements in the specified range.
- **Syntax**: shuffle(start, end, generator);
- **Example**:

```
#include <algorithm>
#include <vector>
#include <random>
using namespace std;
```

```
int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    random_device rd;
    mt19937 g(rd());
    shuffle(numbers.begin(), numbers.end(), g);
}
```

4. find

- **Description**: Searches for the first occurrence of an element in the specified range.
- **Syntax**: find(start, end, value);
- **Example**:

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    auto it = find(numbers.begin(), numbers.end(), 3);
}
```

5. count

- **Description**: Counts the occurrences of a specific value in the range.

- **Syntax**: count(start, end, value);
- **Example**:

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

int main() {
    vector<int> numbers = {1, 2, 3, 3, 4, 5};
    int cnt = count(numbers.begin(), numbers.end(), 3);
}
```

6. std::accumulate

- **Description**: Calculates the sum (or custom accumulation) of elements in a range.
- **Syntax**: accumulate(start, end, initial_value);
- **Example**:

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    int sum = accumulate(numbers.begin(), numbers.end(), 0);
```

}

7. std::min_element / std::max_element

- **Description**: Finds the minimum or maximum element in a range.
- **Syntax**: min_element(start, end); or max_element(start, end);
- **Example**:

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

int main() {
   vector<int> numbers = {1, 2, 3, 4, 5};
   auto min = min_element(numbers.begin(), numbers.end());
   auto max = max_element(numbers.begin(), numbers.end());
}
```

8. std::equal

- **Description**: Checks if two ranges are equal element-by-element.
- **Syntax**: equal(start1, end1, start2);
- **Example**:

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

```
using namespace std;

int main() {
    vector<int> numbers1 = {1, 2, 3};
    vector<int> numbers2 = {1, 2, 3};
    bool areEqual = equal(numbers1.begin(), numbers1.end(), numbers2.begin());
}
```

9. std::unique

- **Description**: Removes consecutive duplicates in a range, moving unique elements to the beginning.
- **Syntax**: unique(start, end);
- **Example**:

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

int main() {
    vector<int> numbers = {1, 1, 2, 3, 3, 4};
    auto it = unique(numbers.begin(), numbers.end());
}
```

10. std::binary_search

- **Description**: Checks if an element exists in a sorted range using binary search.
- **Syntax**: binary_search(start, end, value);
- **Example**:

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    bool found = binary_search(numbers.begin(), numbers.end(), 3);
}
```

STL Containers Complexity

Container	Operation	Time Complexity	Space Complexity
vector	Access (by index)	O(1)	O(n)
	Insert/Delete at the end	Amortized O(1)	
	Insert/Delete in the middle	O(n)	
	Insert/Delete at the beginning	O(n)	
	Search	O(n)	
deque	Access (by index)	O(1)	O(n)
	Insert/Delete at the beginning	O(1)	
	Insert/Delete at the end	O(1)	
	Insert/Delete in the middle	O(n)	
	Search	O(n)	
list	Access (by index)	O(n)	O(n)
	Insert/Delete at the beginning	O(1)	
	Insert/Delete in the middle	O(1)	
	Insert/Delete at the end	O(1)	
	Search	O(n)	
set / **multiset**	Insert/Delete/Search	O(log n)	O(n)
	Access minimum/maximum	O(1)	
unordered_set	Insert/Delete/Search	O(1) (average), O(n) (worst)	O(n)
map / **multimap**	Insert/Delete/Search	O(log n)	O(n)
unordered_map	Insert/Delete/Search	O(1) (average), O(n) (worst)	O(n)
priority_queue	Insert/Delete	O(log n)	O(n)
	Access top	O(1)	

STL Algorithms Complexity

Algorithm	Description	Time Complexity	Space Complexity
std::sort	Sorts range [start, end)	O(n log n)	O(log n)
std::stable_sort	Sorts range while preserving order of equal elements	O(n log n)	O(n)
std::partial_sort	Partially sorts top k elements	O(n log k)	O(k)
std::nth_element	Finds nth smallest element	O(n) (average), O(n^2) (worst)	O(1)
std::binary_search	Checks if an element exists	O(log n)	O(1)
std::find	Finds first occurrence of value	O(n)	O(1)
std::reverse	Reverses range	O(n)	O(1)
std::rotate	Rotates elements in range	O(n)	O(1)
std::shuffle	Randomly shuffles elements	O(n)	O(n) (depends on generator)
std::unique	Removes consecutive duplicates	O(n)	O(1)
std::equal	Checks if two ranges are equal	O(n)	O(1)
std::accumulate	Calculates sum/product/etc.	O(n)	O(1)
std::count	Counts occurrences of value	O(n)	O(1)
std::min_element	Finds minimum in range	O(n)	O(1)
std::max_element	Finds maximum in range	O(n)	O(1)
std::merge	Merges two sorted ranges	O(n)	O(n)
std::inplace_merge	Merges two sorted ranges in place	O(n log n)	O(1)