**STL (Standard Template Library)**

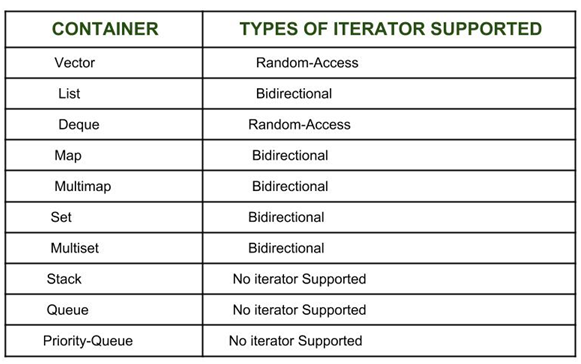
1. **Iterators:** An iterator is an object that points to an element inside the container. We can use iterators to move through the contents of the container.

Depending upon the functionality of iterators they can be divided into five categories.

Graphical user interface, text, application, chat or text message

Description automatically generated

Now each of these iterators is not supported by all the containers in STL, different containers support different iterators, like vectors support Random access Iterators, while list supports bidirectional Iterators.



Types of Iterators:

* Input Iterators: Input Iterators can only be used in single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.

Example: **Practical implementation (std::find, std::equal, std::equal\_range , and std::count**. )

* output Iterators: output Iterators can only be used in the single-pass algorithm for being assigned elements. (Example : **std::move**)
* Forward Iterator: These Iterators can only be in the forwarding direction and that too one step at a time.

Example: (**std::replace, std::reverse\_copy**)

* Bidirectional Iterators: These Iterators can move in both directions, which is why their name is directional.

Example: (**std::random\_shuffle)**

* Random-Access Iterators: They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers. (**std::random\_shuffle**)

Table

Description automatically generated

Benefits of Iterators:

Convenience in programming: It is better to use iterators to iterate through the contents of containers as if we will not use an iterator and access elements using the [] operator. Then we need to be always worried about the size of the container, whereas with an iterator we can simply use member function end() and iterate through the contents without having to keep anything in mind.

1. Operations of Iterators:

Begin() – This function is used to return the beginning position of the container

End(): This function is used to return the after-end position of the container

Advance(): This function is used to increment the iterator position till the specified number is mentioned in its arguments.

Next(): This function returns the new iterator that the iterator would point to after advancing the positions mentioned in its arguments.

Prev(): This function returns the new iterator that the iterator would point to after decrementing the positions mentioned in its arguments.

**Example**:

|  |  |
| --- | --- |
| **#include<iostream>**  **#include<iterator> // for iterators**  **#include<vector> // for vectors**  **using namespace std;**  **int main()**  **{**  **vector<int> ar = { 1, 2, 3, 4, 5 };**    **// Declaring iterators to a vector**  **vector<int>::iterator ptr = ar.begin();**  **vector<int>::iterator ftr = ar.end();**    **// Using next() to return new iterator**  **// points to 4**  **auto it = next(ptr, 3);**    **// Using prev() to return new iterator**  **// points to 3**  **auto it1 = prev(ftr, 3);** | **// Using advance() to increment iterator position**  **// points to 4**  **advance(ptr, 3);**    **// Displaying iterator position**  **cout << "The position of new iterator using next() is : ";**  **cout << \*it << " ";**  **cout << endl;**    **// Displaying iterator position**  **cout << "The position of new iterator using prev() is : ";**  **cout << \*it1 << " ";**  **cout << endl;**  **return 0;**  **}** |

Containers: Container classes store objects and data.

1. Sequence containers: Implement data structures that can be accessed in a data sequential manner.

Vector: 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. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

1. **Begin()** – Returns an iterator pointing to the first element in the vector.

**Syntax :**

**vectorname.begin()**

**Parameters**: No parameters are passed.

**Return Type:** This function returns a bidirectionaliterator pointing to the first element.

**Errors and Exceptions:**

* It has a no exception throw guarantee.
* Shows error when a parameter is passed.

**Time Complexity:** O(1)

1. **End**(): function is used to return an iterator pointing next to the last element of the vector container. end() function returns a bidirectional iterator.

**Syntax** :  vectorname.end()

**Parameters** : No parameters are passed.

**Return Type:** This function returns a bidirectionaliterator pointing to the next to the last element.

**Errors and Exceptions**

* It has a no exception throw guarantee.
* Shows error when a parameter is passed.

1. **vector::cbegin():** The function returns an iterator which is used to iterate the container.

The iterator points to the beginning of the vector and the Iterator cannot modify the contents of the vector.

**Syntax**: vectorname.cbegin()

**Parameters**: There is no parameter

**Return value**: Constant random access iterator points to the beginning of the vector.

**Exception**: No exception

1. **vector::empty()** : The empty() function is used to check if the vector container is empty or not.

**Syntax**: vectorname.empty()

**Parameters :** No parameters are passed.

**Returns** : True, if vector is empty False, Otherwise

**Errors and Exceptions:**  
It has a no exception throw guarantee.   
Shows error when a parameter is passed.

1. **vector::size():** The size function is used to return the size of the vector container or the number of elements in the vector container.

**Syntax** :  vectorname.size()

**Parameters**: No parameters are passed.

**Returns**: Number of elements in the container.

Example:

Text

Description automatically generated

Output: ***Segmentation Fault SIGEGV***

we get Segmentation Fault (SIGSEGV) because the return type of size() is size\_t which is an alias for unsigned long int.-> unsigned long int var = 0;-> cout << var – 1;  // This will be equal to 18446744073709551615-> vector<data\_type> vec;-> cout << vec.size() – 1;  // This will also be equal to 18446744073709551615

**How to fix:**

**It is advisable to typecast container.size() to integer type in order to avoid Segmentation Fault (SIGSEGV).**

**Change above line: for(int i= 0 ; I <= (int)vec.size()-1; i++).**

1. **vector::max\_size():** The **vector::max\_size()** is a built-in function in C++ STL which returns the maximum number of elements that can be held by the vector container.

**Parameters:** The function does not accept any parameters. **Return value:** The function returns the maximum numbers that can fit into the vector container. **Time Complexity –**Constant O(1)

1. **vector::capacity() :** The **vector::capacity()** function is a built-in function that returns the size of the storage space currently allocated for the vector, expressed in terms of elements.

**Syntax**:  vector\_name.capacity()

**Parameters:** The function does not accept any parameters. Return Value: The function returns the

size of the storage space currently allocated for the vector, expressed in terms of elements.

**Time Complexity** – Constant O(1)

1. **vector::resize()** : The function alters the container’s content in actual by inserting or deleting the elements from it. It happens so,

* If the given value of n is less than the size at present then extra elements are demolished.
* If n is more than the current size of the container then upcoming elements are appended at the end of the vector.

Syntax: vectorname.resize(int n, int val)

* n – it is new container size, expressed in a number of elements.
* val – if this parameter is specified then new elements are initialized with this value.
  + Return value:
* This function does not return anything.
  + Exception:
* The only exception if it so happens is [Bad\_alloc](https://www.geeksforgeeks.org/bad_alloc-in-cpp/" \t "_blank) thrown, if reallocation fails.

1. The vector::shrink\_to\_fit(): is a built-in function in C++ STL which reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.

Syntax: vector\_name.shrink\_to\_fit()

Parameters: The function does not accept any parameters.

Return value: The function does not return anything.

Time Complexity – Linear O(N)

1. vector::operator= : This operator is used to assign new contents to the container by replacing

the existing contents. It also modifies the size according to the new contents.

**Syntax** :  vectorname1 = (vectorname2)

**Parameters** : Another container of the same type.

**Result** : Assign the contents of the container passed as parameter to the container written on left side of the operator.

* **Errors and Exceptions**  
  If the containers are of different types, an error is thrown.   
  It has a basic no exception throw guarantee otherwise.

**Time Complexity – Linear O(N)**

1. **vector::operator[]:** This operator is used to reference the element present at the position given inside the operator. It is similar to the at() function, the only difference is that the at() function throws an out-of-range exception when the position is not in the bounds of the size of the vector, while this operator causes **undefined behavior**.

* **Errors and Exceptions**  
  If the position is not present in the vector, it shows undefined behavior.   
  It has a no exception throw guarantee otherwise.
* Time Complexity – Constant O(1)

1. **vector:: assign()** is an STL in C++ that assigns new values to the vector elements by replacing old ones. It can also modify the size of the vector if necessary.

**The syntax for assigning constant values:**vectorname.assign(int size, int value)

**Parameters**:

size - number of values to be assigned

value - value to be assigned to the vector name

**The syntax for assigning values from an array or list:**

vectorname.assign(arr, arr + size)

**Parameters**:

arr - the array which is to be assigned to a vector

size - number of elements from the beginning which has to be assign

**The syntax for modifying values from a vector :**

vectorname.assign(InputIterator first, InputIterator last)

**Parameters**:

**first** - Input iterator to the initial position range.

**last** - Input iterator to the final position range.

**Syntax for assigning values with initializer list**

vectorname.assign((initializer\_list)

Parameter: initializer\_list

1. **vector::push\_back():** push\_back() function is used to push elements into a vector from the back. The new value is inserted into the vector at the end, after the current last element and the container size is increased by 1.

**Syntax**: vectorname.push\_back(value)

**Parameters** : The value to be added in the back is passed as the parameter

**Result** : Adds the value mentioned as the parameter to the back of the vector named vectorname

**Errors and Exceptions**  
1. **Strong exception guarantee** – if an exception is thrown, there are no changes in the container.   
2. If the value passed as an argument is not supported by the vector, it shows undefined behavior.

1. vector::pop\_back(): The pop\_back() function is used to pop or remove elements from a vector from the back. The value is removed from the vector from the end, and the container size is decreased by 1.

**Syntax** :  vectorname.pop\_back()

**Parameters** : No parameters are passed

**Result** : Removes the value present at the end or back of the given vector named as vectorname

**Errors and Exceptions**

* 1. **No-Throw-Guarantee –** If the container is not empty, the function never throws exceptions.   
     2. If the vector is empty, it shows undefined behavior.

1. std::vector::insert() is a built-in function in C++ STL that inserts new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.

**Time Complexity – Linear O(N)**

**Syntax**: vector\_name.insert (position, val)

**Parameter**: The function accepts two parameters specified as below:

* + position – It specifies the iterator which points to the position where the insertion is to be done.
  + val – It specifies the value to be inserted.

**Syntax:** vector\_name.insert(position, size, val)

**Parameter:**The function accepts three parameters specified as below**:**

**position –**It specifies the iterator which points to the position where the insertion is to be done**.**

size – It specifies the number of times a val is to be inserted at the specified position.

val – It specifies the value to be inserted.

**Syntax:** vector\_name.insert(position, iterator1, iterator2)

**Parameter:**The function accepts three parameters specified as below:

**position –**It specifies the position at which insertion is to be done in the vector.

**iterator1 –**It specifies the starting position from which the elements are to be inserted

**iterator2 –**It specifies the ending position till which elements are to be inserted

1. vector::clear() : clear() function is used to remove all the elements of the vector container, thus making it size 0.

**Syntax** : vectorname.clear()

**Parameters** : No parameters are passed.

**Result** : All the elements of the vector are removed ( or destroyed )

**Errors and Exceptions**  
1. It has a no exception throw guarantee.   
2. Shows error when a parameter is passed.

1. vector::erase(): erase() function is used to remove elements from a container from the specified position or range.

**Syntax** :

1. vectorname.erase(position)

2. vectorname.erase(startingposition, endingposition)

**Parameters** :

Position of the element to be removed in the form of iterator.

or the range specified using start and end iterator.

**Result** :

Elements are removed from the specified

position of the container.

**Examples**:

Input : myvector= {1, 2, 3, 4, 5}, iterator= myvector.begin()+2

myvector.erase(iterator);

**Output** : 1, 2, 4, 5

**Input** : myvector= {1, 2, 3, 4, 5, 6, 7, 8}, iterator1= myvector.begin()+3, iterator2= myvector.begin()+6

myvector.erase(iterator1, iterator2);

**Output** : 1, 2, 3, 7, 8

**Errors and Exceptions**  
1. It has a no-exception throw guarantee if the position is valid.   
2. Shows undefined behavior otherwise.  
Removing an element from a particular position:

1. **vector::swap():** This function is used to swap the contents of one vector with another vector of the same type and the sizes of vectors may differ.

**Syntax:** ***vectorname1*.swap(*vectorname2*)**

**Parameters:** The name of the vector with which the contents have to be swapped.

**Result:** All the elements of the 2 vectors are swapped.

**Examples:**

**Input:** myvector1 = {1, 2, 3, 4}

myvector2 = {3, 5, 7, 9}

myvector1.swap(myvector2);

**Output:** myvector1 = {3, 5, 7, 9}

myvector2 = {1, 2, 3, 4}

**Input:** myvector1 = {1, 3, 5, 7}

myvector2 = {2, 4, 6, 8}

myvector1.swap(myvector2);

**Output:** myvector1 = {2, 4, 6, 8}

myvector2 = {1, 3, 5, 7}

**Errors and Exceptions**

It throws an error if the vector is not of the same type.

It has a basic no-exception throw guarantee otherwise.

1. The **vector::emplace()** is an STL in C++ that extends the container by inserting a new element at the position. Reallocation happens only if there is a need for more space. Here the container size increases by one.

**Syntax:**  Template iterator vector\_name.emplace (const\_iterator position, element);

**Parameter:**  The function accepts two mandatory parameters which are specified as below:

* *position* – It specifies the iterator pointing to the position in the container where the new element is to be inserted.
* element- It specifies the element to be inserted in the vector container.

**Return value:** The function returns an iterator that points to the newly inserted element.

**Complexity:**Linear

1. **std::emplace\_back():** This function is used to insert a new element into the vector container, the new element is added to the end of the vector.

**Syntax : *vectorname*.emplace\_back(*value*)**

**Parameters :** The element to be inserted into the vector is passed as the parameter.

**Result :** The parameter is added to the vector at the end position.

**Examples:**

**Input:** myvector{1, 2, 3, 4, 5};

myvector.emplace\_back(6);

**Output:** myvector = 1, 2, 3, 4, 5, 6

**Input:** myvector{};

myvector.emplace\_back(4);

**Output:** myvector = 4

**Errors and Exceptions:**

It has a strong exception guarantee, therefore, no changes are made if an exception is thrown.

The parameter should be of the same type as that of the container, otherwise, an error is thrown.

**emplace\_back() vs push\_back()**

* 1. **push\_back()** copies a string into a vector. First, a new string object will be implicitly created initialized with provided char\*. Then push\_back will be called which will copy this string into the vector using the move constructor because the original string is a temporary object. Then the temporary object will be destroyed.
  2. **emplace\_back()** constructs a string in-place, so no temporary string will be created but rather emplace\_back() will be called directly with char\* argument. It will then create a string to be stored in the vector initialized with this char\*. So, in this case, we avoid constructing and destroying an unnecessary temporary string object.



**STL (Standard Template Library) :** STL is a set of c++ template classes to provide common programming data structures and functions such as lists, stacks , arrays e.t.c. It is a library of container classes, algorithms and Iterators.

**STL components :**

* Containers: Containers are used to store the data
* Algorithms: Algorithm library provides some sorting, searching algorithms
* Iterators: Iterators are used to traverse the data
* Functions.

**1) Container:** A container is an object that stores a collection of objects in a specific way. It provides four different types of containers

* sequential containers
* Associative containers
* unordered associative containers.
* adapter container

1. **Sequential containers**: Sequential containers allow us to store elements that can be accessed in sequential order.

* Array
* vector
* list
* deque
* forward list

1. **Associative containers:** Associative containers allow us to store elements in sorted order

* set
* map
* multimap
* multiset

1. **unordered Associative containers:** Unordered Associative containers provide an unsorted version of associative containers.

* unordered set
* unordered map
* unordered multiset
* unordered multimap

1. **container Adapter:** Provide different interfaces for sequential containers.

* queue
* priority queue
* stack

2) **algorithms**: The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements.

STL provides different types of algorithms that can be implemented upon any container with the help of iterators. Thus now we don’t have to define complex algorithms instead we just use the built-in functions provided by the algorithm library in STL.

**i) Sorting algorithm:** We will be studying about three methods under sorting algorithms, namely:

sort, is\_sorted , partial\_sort

* **Sort :** sort(start\_iterator, end\_iterator ) : sorts the range defined by iterators start\_iterator and end\_iterator in ascending order.

**sort**(start\_iterator, end\_iterator, compare\_function) : this also sorts the given range but you can

define how the sorting should be done by compare\_function.

* **partial\_sort()**: sorts the first half of elements in the given range, the other half of elements remain as they were initially. partial\_sort() also has two variations:

**partial\_sort(start, middle, end )** : sorts the range from start to end in such a way that the elements before middle are in ascending order and are the smallest elements in the range.

**partial\_sort(start, middle, end, compare\_function)**: sorts the range from start to end in such a way that the elements before middle are sorted with the help of compare\_function and are the smallest elements in the range.

* **is\_sorted method:** The function of the STL return true if the given range is sorted. There are two versions of is\_sorted():

**is\_sorted(start\_iterator, end\_iterator)**: checks the range defined by iterators start\_iterator and end\_iterator is ascending order.

**is\_sorted(start\_iterator, end\_iterator, comapre\_function)**: It also checks the given range but you can define how the sorting must be done.

1. **std::array:** The array is a collection of homogeneous objects and this array container is defined for constant size arrays or (static size). This container wraps around fixed-size arrays and the information of its size is not lost when declared to a pointer.

header : #include <array>

syntax : std::array<T, N> arr;

Declaration : std::array<int,5> array;

Initialization : std::array<int,5> array = {1,2,3,4} before c++ 17

std::Array arr = {1,2,3,5};after c++ 17

std::array<int,5> array {1,2,3,4,5}

Assign using initializer list : std::array<int,5> array; array = {1,2,3,4}

std::array is a sequential container that encapsulates fixed-size arrays

array size is needed at compile time.

Assign by value is actually by value

Access Elements: i) at() ii) [] iii) front() iv) back() v) data()

However, since C++17, it is allowed to omit the type and size. They can only be omitted together, but not one or the other, and only if the array is explicitly initialized.

**std::array myArray { 9, 7, 5, 3, 1 }; // The type is deduced to std::array<int, 5>**

**std::array myArray { 9.7, 7.31 }; // The type is deduced to std::array<double, 2>**

Since C++20, it is possible to specify the element type but omit the array length. This makes the creation of std::array a little more like the creation of C-style arrays. To create an array with a specific type and deduced size, we use the std::to\_array function:

**auto myArray1 { std::to\_array<int, 5>({ 9, 7, 5, 3, 1 }) }; // Specify type and size**

**auto myArray2 { std::to\_array<int>({ 9, 7, 5, 3, 1 }) }; // Specify type only, deduce size**

**auto myArray3 { std::to\_array({ 9, 7, 5, 3, 1 }) }; // Deduce type and size**

**Note**: Unfortunately, std::to\_array is more expensive than creating a std::array directly, because it actually copies all elements from a C-style array to a std::array. For this reason, std::to\_array should be avoided when the array is created many times (e.g. in a loop).

**Size and sorting:**

The size() function can be used to retrieve the length of the std::array

**std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };**

**std::cout << "length: " << myArray.size() << '\n';**

Because std::array doesn’t decay to a pointer when passed to a function, the size() function will work even if you call it from within a function:

**Example** :

#include <array>

#include <iostream>

void printLength(const std::array<double, 5>& myArray)

{

std::cout << "length: " << myArray.size() << '\n';

}

int main()

{

std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };

printLength(myArray);

return 0;

}

**Summary:**

std::array is a great replacement for built-in fixed arrays. It’s efficient because it doesn’t use any more memory than built-in fixed arrays. The only real downside of a std::array over a built-in fixed array is a slightly more awkward syntax, that you have to explicitly specify the array length (the compiler won’t calculate it for you from the initializer unless you also omit the type, which isn’t always possible), and the signed/unsigned issues with size and indexing. But those are comparatively minor quibbles — we recommend using std::array over built-in fixed arrays for any non-trivial array use.

1. **std::vector:** It is a sequential container and is also known as a dynamic array or array list. Its size grows and shrinks dynamically and no need to provide size at compile time.

**Element access:** at() , [], front() , back(), data()

**Modifiers:** insert(), emplace(), push\_back(), emplace\_back(), pop\_back(), resize(), swap() , erase(), clear()

**Header:** #include <vector>

// no need to specify length at the declaration

**std::vector<int> array;**

**std::vector<int> array2 = { 9, 7, 5, 3, 1 }; // use initializer list to initialize array (before C++11)**

**std::vector<int> array3 { 9, 7, 5, 3, 1 }; // use uniform initialization to initialize array**

// as with std::array, the type can be omitted since C++17

**std::vector array4 { 9, 7, 5, 3, 1 }; // deduced to std::vector<int>**

**emplace**: The vector::emplace() is an STL in C++ which extends the container by inserting a new element at the position. Reallocation happens only if there is a need for more space. Here the container size increases by one.

#include <iostream>

#include <vector>

using namespace std;

int main()

{

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

**// insert element by emplace function at front**

auto it = vec.emplace(vec.begin(), 15);

// insert element by emplace function at the end

auto it1 = vec.emplace(vec.end(), 35);

// insert element by emplace function in the middle

auto it2 = vec.emplace(vec.begin() + 2, 16);

// print the elements of the vector

cout << "The vector elements are: ";

for (auto it = vec.begin(); it != vec.end(); ++it)

cout << \*it << " ";

return 0;

}

emplace\_back() vs push\_back()

**push\_back()** copies a string into a vector. First, a new string object will be implicitly created and initialized with provided char\*. Then push\_back will be called which will copy this string into the vector using the move constructor because the original string is a temporary object. Then the temporary object will be destroyed.

**emplace\_back()** constructs a string in place, so no temporary string will be created but rather emplace\_back() will be called directly with char\* argument. It will then create a string to be stored in the vector initialized with this char\*. So, we avoid constructing and destroying an unnecessary temporary string object in this case.

**Important Points about std::vector**:

* **Ordered Collection**: In std::vector all elements will remain in the same order in which they are inserted.
* **Provides random access**: Indexing is very fast in std::vector using operator [], just like arrays.
* **Performance**: It Performs better if insertion and deletion are in end only and gives the worst performance if insertion/deletion is at the middle or at starting of the vector.
* **Contains Copy**: It always stores copies of the object, not the same reference. So, if you are adding objects of user-defined classes you should define the copy constructor and assignment operator in your class.

**Why Should I use std::vector?**

std::vector give the same kind of fast performance in indexing and iterations as arrays. But it won’t have a Fixed Size limitation like Arrays. You don’t need to provide the fixed size for std::vector in advance. Just start inserting elements in std::vector and it will automatically expand its size.

**what are 5 different ways to initialize the vector?**

#include <iostream>

#include <vector>

#include <iterator>

#include <list>

#include <string>

void example1() {

// Initialize vector with 5 integers

// Default value of all 5 ints will be 0.

std::vector<int> vecOfInts(5);

for (int x : vecOfInts)

std::cout << x << std::endl;

}

void example2() {

// Initialize vector to 5 string objects with value "Hi"

std::vector<std::string> vecOfStr(5, "Hi");

for (std::string str : vecOfStr)

std::cout << str << std::endl;

}

void example3() {

// Create an array of string objects

std::string arr[] = { "first", "sec", "third", "fourth" };

// Initialize vector with a string array

std::vector<std::string> vecOfStr(arr,

arr + sizeof(arr) / sizeof(std::string));

for (std::string str : vecOfStr)

std::cout << str << std::endl;

}

void example4() {

// Create an std::list of 5 string objects

std::list<std::string> listOfStr;

listOfStr.push\_back("first");

listOfStr.push\_back("sec");

listOfStr.push\_back("third");

listOfStr.push\_back("fouth");

// Initialize a vector with std::list

std::vector<std::string> vecOfStr(listOfStr.begin(), listOfStr.end());

for (std::string str : vecOfStr)

std::cout << str << std::endl;

// Initialize a vector with other string object

std::vector<std::string> vecOfStr3(vecOfStr);

for (std::string str : vecOfStr3)

std::cout << str << std::endl;

}

int main() {

example1();

example2();

example3();

example4();

return 0;

}

**Importance of Constructors while using User-Defined Objects with std::vector?**

For User-Defined classes, if Copy Constructor and Assignment Operator are public then only one can insert its object in std::vector.

**This is because of two reasons**,

* All STL contains always store the copy of inserted objects, not the actual ones. So, whenever we insert any element or object in a container then its copy constructor is called to create a copy, and then this copy is inserted in the container.
* While insertion in std::vector it might be possible that storage relocation takes place internally due to insufficient space. In such cases, the assignment operator will be called on objects inside the container to copy them from one location to another.

**How to use vectors efficiently in C++?**

* Vector will be more efficient if elements are inserted or removed from the back end only.
* Set the storage of the vector initially using the reserve() member function.
* Instead of adding a single element in multiple calls, a large set of elements is added in a single call

**When to choose deque over vector:**

One should choose deque over vector if he wants to either add or delete from both the ends like implementing a Queue.

**When to choose vector over deque:**

One should choose a vector if insertion or deletions are required mostly in end like implementing a Stack.

1. **List:** The List is a sequential container that allows non-contiguous memory locations. It is implemented by using a double-linked list.

**List faster compared to other sequential containers (vectors, forward\_list, deque, array) in terms of insertion and deletion.**

**we should use this class instead of a double-linked list because it is well-tested and bench of available functions**.

Lists are sequence containers that allow non-contiguous memory allocation. As compared to the vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about a doubly linked list.

**Few available operations:** assign(), front(), back(), empty(), size(), clear(), insert(),

emplace(),push\_back(), pop\_back(),push\_front(), pop\_front(),reverse, sort, merge, splice, unique,

remove, remove\_if, size.

1. **forward\_list:** Forward list in STL implements a singly linked list. Introduced from C++11, forward lists are more useful than other containers in insertion, removal, and moving operations (like sort) and allow time-constant insertion and removal of elements.

It differs from the list by the fact that the forward list keeps track of the location of only the next element while the list keeps track of both the next and previous elements, thus increasing the storage space required to store each element. The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.

Forward List is preferred over the list when only forward traversal is required (same as the singly linked list is preferred over doubly linked list) as we can save space. Some example cases are, chaining in hashing, adjacency list representation of the graph, etc.

**Operations on Forward List:**

i**) assign():** This function is used to assign values to the forward list, its other variant is used to assign

repeated elements and using the values of another list.

Example:

// C++ code to demonstrate forward list and assign()

#include <forward\_list>

#include <iostream>

using namespace std;

// Driver Code

int main()

{

// Declaring forward list

forward\_list<int> flist1;

// Declaring another forward list

forward\_list<int> flist2;

//Declaring another forward list

forward\_list<int> flist3;

// Assigning values using assign()

flist1.assign({ 1, 2, 3 });

// Assigning repeating values using assign()

// 5 elements with value 10

flist2.assign(5, 10);

//Assigning values of list 1 to list 3

flist3.assign(flist1.begin(), flist1.end());

// Displaying forward lists

cout << "The elements of first forward list are : ";

for (int& a : flist1)

cout << a << " ";

cout << endl;

cout << "The elements of second forward list are : ";

for (int& b : flist2)

cout << b << " ";

cout << endl;

cout << "The elements of third forward list are : ";

for (int& c : flist3)

cout << c << " ";

cout << endl;

return 0;

}

**ii) push\_front:** This function is used to insert the element at the first position on the forward list. The

value from this function has copied the space before the first element in the container.

**iii) emplace\_front():** This function is similar to the previous function but in this no copying operation

occurs, the element is created directly at the memory before the first element of the forward list.

**iv) pop\_front():** This function is used to delete the first element of the list.

// C++ code to demonstrate the working of

// push\_front(), emplace\_front() and pop\_front()

#include <forward\_list>

#include <iostream>

using namespace std;

// Driver Code

int main()

{

// Initializing forward list

forward\_list<int> flist = { 10, 20, 30, 40, 50 };

// Inserting value using push\_front()

// Inserts 60 at front

flist.push\_front(60);

// Displaying the forward list

cout<< "The forward list after push\_front operation : ";

for (int& c : flist)

cout << c << " ";

cout << endl;

// Inserting value using emplace\_front()

// Inserts 70 at front

flist.emplace\_front(70);

// Displaying the forward list

cout << "The forward list after emplace\_front "

"operation : ";

for (int& c : flist)

cout << c << " ";

cout << endl;

// Deleting first value using pop\_front()

// Pops 70

flist.pop\_front();

// Displaying the forward list

cout << "The forward list after pop\_front operation : ";

for (int& c : flist)

cout << c << " ";

cout << endl;

return 0;

}

|  |  |
| --- | --- |
| **Vector** | **List** |
| It has contiguous memory. | While it has non-contiguous memory. |
| It is synchronized. | While it is not synchronized. |
| The vector may have a default size. | The list does not have a default size. |
| In vector, each element only requires space for itself only. | In the list, each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list. |
| Insertion at the end requires constant time but insertion elsewhere is costly. | Insertion is cheap no matter where in the list it occurs. |
| The vector is thread-safe. | The List is not thread-safe. |
| Deletion at the end of the vector needs constant time but for the rest, it is O(n). | Deletion is cheap no matter where in the list it occurs. |
| Random access to elements is possible. | Random access to elements is not possible. |
| Iterators become invalid if elements are added to or removed from the vector. | Iterators are valid if elements are added to or removed from the list |

1. **Deque:** Double-ended queues are sequence containers with the feature of expansion and contraction on both ends. They are like vectors but are more efficient in the case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.

Double Ended Queues are basically an implementation of the data structure double-ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both ends.

The functions for deque are the same as a vector, with the addition of push and pop operations for both front and back.

**The time complexities for doing various operations on deques are**-

**Accessing Elements- O(1)**

**Insertion or removal of elements- O(N)**

**Insertion or removal of elements at start or end- O(1)**

**Properties of Deque:**

Deque is a generalized version of the queue that allows us to insert and delete the element at both ends. It does not follow FIFO (first in first out) rule.

**Applications of Deque:**

* It is used in job scheduling algorithms.
* If supports both stack and queue operations.
* The clockwise and anti-clockwise rotation operations in deque are performed in O(1) time which is helpful in many problems.

**Real-time Application of Deque:**

In a web browser’s history, recently visited URLs are added to the front of the deque and the URL at the back of the deque is removed after some specified number of operations of insertions at the front.

Storing a software application’s list of undo operations.

**Advantages of Deque:**

You are able to add and remove items from both the front and back of the queue.

Deques are faster in adding and removing elements to the end or beginning.

The clockwise and anti-clockwise rotation operations are faster in a deque.

**Disadvantages of Deque:**

Deque has no fixed limitations for the number of elements they may contain. This interface supports capacity-restricted deques as well as deques with no fixed size limit.

They are less memory efficient than a normal queue.

**Associative containers:** Associative containers allow us to store elements in sorted order

i) set ii) map iii) multimap iv) multiset

1. **Set:** The 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.

**header file** : #include <set>

**syntax**: set<int> Set;

**Properties**:

* + **Storing order**: The set stores the elements in sorted order.
  + **Values Characteristics**: All the elements in a set have unique values.
  + **Values Nature**: The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element. Thus, the values are immutable.
  + **Search Technique**: Sets follow the Binary search tree implementation.
  + **Arranging order**: The values in a set are unindexed.

**The time complexities for doing various operations on sets are**:

**Insertion of Elements – O(log N)**

**Deletion of Elements – O(log N)**

1. **unordered Set:** An unordered set is 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.

The unordered\_set can contain a key of any type – predefined or user-defined data structure but when we define the key of the type user defines the type, we need to specify our comparison function according to which keys will be compared.

**syntax : unordered\_set <string> stringSet ;**

**Sets vs Unordered Sets**

Set is an ordered sequence of unique keys whereas unordered\_set is a set in which keys can be stored in any order, so unordered. Set is implemented as a balanced tree structure which is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(log n) while for unordered\_set, it is O(1).

**Difference between Set and Unordered Set**

|  |  |
| --- | --- |
| **Set** | **Unordered Set** |
| Set stores elements in a sorted order | Unordered Set stores elements in an unsorted order |
| Set uses Binary Search Trees for implementation | Unordered Set uses Hash Tables for implementation |
| More than one element can be erased by giving the  starting and ending iterator | We can erase that element for which the iterator position is given |
| set<datatype> Set\_Name; | unordered\_set<datatype> UnorderedSet\_Name; |

1. **MultiSet:** Multisets are a type of associative container similar to the set, with the exception that multiple elements can have the same values.

**Insertion of Elements- O(log N)**

**Accessing Elements – O(log N)**

**Deleting Elements- O(log N)**

1. **unordered\_multiset:** To handle this duplication unordered\_multiset should be used, it can store duplicate elements also. Internally when an existing value is inserted, the data structure increases its count which is associated with each value. As the count of each value is stored in unordered\_multiset, it takes more space than unordered\_set (if all values are distinct).

The internal implementation of unordered\_multiset is the same as that of unordered\_set and also uses a hash table for searching, just the count value is associated with each value in the former one. Due to hashing of elements, it has no particular order of storing the elements so all element can come in any order but duplicate element comes together. All operation on unordered\_multiset takes constant time on average but can go up to linear in the worst case.

1. **Map:** 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.

set and map in STL are similar in the sense that they both use Red Black Tree (A self-balancing BST).

Note that the time complexities of search, insert, and delete are O(Log n).

**Creating std::map objects:** std::map<std::string, int> mapOfWords;

**Inserting data in std::map:**

**mapOfWords.insert(std::make\_pair("earth", 1));**

**mapOfWords.insert(std::make\_pair("moon", 2));**

**We can also insert data in std::map using operator [] i.e. mapOfWords["sun"] = 3;**

Array/vector - Time complexity memory complexity

Insertion/deletion - End O(1) O(n)

Insertion- middle/deletion - O(n)

serch - O(1)

Deque- Insertion begin/End - O(1) , middle - O(n) O(n)

List : Insertion begin/End - O(1) , middle - O(n)