**STL(Standard Template Library)**

Set of C++ template classes to provide common programming data structures and functions

4 components:

1. **Containers** 
   1. generic class templates
   2. Stores collection of data
   3. 7 STANDARD FIRST CLASS CONTAINERS
   4. 7 HEADER FILES
      1. *#include <vector>*
      2. *#include <list>*
      3. *#include <iterator>*
      4. *#include <deque>*
      5. *#include <stack>*
   5. 3 CONT ADAPTOR CLASSES
   6. 4 types:
      1. **Sequence Containers**
         1. *Syntax : Container<Dtype>VarName*
         2. Iterable
         3. Implementation Access Sequential
            1. **Vectors**

*#include <vector>*

Dynamic arrays; Container *Automatically Resizes* at insertion deletion

In contiguous Storage;Iterator Access;

Insertion@end- differential time- extension need

Removal constant time; Insertion Removal @ beginning / middle-linear time.

Associated Functions :

**Element Access**

**at(g) =** reference to element at g

**Reference operators : [g]**

Eg:vector<int> myVector = {10, 20, 30, 40, 50};

cout << "Element at index 2: " << myVector[2];

**front() =** reference to first element

**back() =** reference to last element

**data() =**  gives you a pointer to the start of the memory block where the elements of the vector are stored;allowing direct manipulation of its elements with caution.

**Moderators**

**assign() =** new value to vector; replaces old value

**[Disc : Not used in Array]**

//*insertion*

**push\_back() =** push from last pos

**push\_front() =** push from first pos

//*deletion*

**pop\_back() =** pop from last pos

**pop\_front() =** pop from first pos

**insert() =** from any pos

**erase() =** from any pos

**swap() =** any 2 vectorssame type diff size

**clear() =** all el of vector

**emplace() =** extends vector from front

**emplace\_back() =** extends vector from back

**Iterators(ptrs\*)**

**begin()**

**end()**

**rbegin()**

**rend()**

**cbegin()**

**cend()**

**crbegin()**

**crend()**

NOTE:

c - constant

r - reverse

Begin - first el

End - theoretical element that follows the last el in vector

**Capacity**

**size() =** No of el

**resize() =** resizes to fit n el

**max\_size() =** max no of el that can be contained

**capacity() =** storage space allocated to no of el present

**reserve() =**

**shrink\_to \_fit() =** shrinks to fit req size and destroys all el beyond capacity

**empty() =** checks if empty

* + - * 1. **Array**

Operations :

**at()**

**get<0>(ar)**

**[]**

*array<int,6> ar = {1, 2, 3, 4, 5, 6};* OR array<int, 6> ar

*ar.swap(ar1);*

does not support any insertion or deletion

class template are based on arrays = vector & dequeue

| **BASIS** | **C** | **C++** |
| --- | --- | --- |
| **KNOWS ITS OWN SIZE** | **NO** | **YES** |
| **NEED TO SEND len AS SEPARATE PARAMETER** | **YES** | **NO** |
| **RISK OF ARRAY BEING DECAYED INTO A PTR** | **YES** | **NO** |
| **MORE EFFICIENT, LIGHT-WEIGHT AND RELIABLE** | **NO** | **YESS** |

*Non-contiguous memory allocation*

* + - * 1. **Deque**

Double ended queues/extraction-contraction

=fast insertion & deletion

*#include <list>*

*#include <iterator>*

* + - * 1. **List (Doubly linked list)**

Slow Traversal; Fast insertion deletion

*Syntax for traversal : Container\_list<Dtype> :: iterator it\_Name;*

*Without iterator LL traversal is not possible  
it\_Name is used as for loop variable*

*Syntax for printing : cout << '\t' <<* ***\*it****;*

*Container\_list*

**.reverse()**

**.sort() =** ascending order

**.resize()**

**.unique() =**  removes duplicates

**.merge()**

**splice() =** transfer el from 1 list to another

* + - * 1. **Forward List (Singly linked list)**
    1. **Container Adaptors(Derived Containers)**
       1. provide a different interface for sequential containers
       2. *#include <stack>*
       3. **Stack :**
          1. container adaptors with LIFO
          2. uses an **encapsulated** object of either **vector or deque or list**
          3. Use functions like : (Time Complexity : O(1))

**empty() = bool**

**size()**

**top()**

**push()**

**pop()**

**swap()**

**emplace()**

* + - 1. **Queue**
      2. **Priority Queue**
    1. **Associative Containers**
       1. designed to support direct access to elements using **keys**
       2. DS : tree for fast searching, deletion and insertion
       3. Slow random access
       4. Inefficient sorting
          1. **Map =** key-value pairs, sorted by unique keys

**map<val\_dtype,key\_dtype>Var\_Name;**

**[note : char\* whereas int normal for val\_dtype]**

**map(const key\_compare & comp)**

pair<iterator, bool> insert(const value\_type& x)

Inserts x into the map

***insert***

**iterator insert(iterator pos, const value\_type& x)**

Inserts x into the map, using pos as a hint to where it will be inserted

**void insert(iterator, iterator)**

Inserts a range into the map

***erase***

**void erase(iterator pos)**

Erases the element pointed to by pos

**size\_type erase(const key\_type& k)**

Erases the element whose key is k

**void erase(iterator first, iterator last)**

Erases all elements in a range

**iterator *find*(const key\_type& k)**

Finds an element whose key is k.

**data\_type& operator[](const key\_type& k)**

Returns a reference to the object that is associated with a particular key.

If the map does not already contain such an object, operator[] inserts the default object data\_type()

**count()**

* + - * 1. **Multimap =** key-value pairs, sorted by keys
        2. **Set = unique** keys, sorted by keys

VS

* + - * 1. **Multiset =** keys, sorted by keys
        2. empty() = bool
        3. size()
        4. front() = reference to first element
        5. back() = reference to last element
        6. swap()
        7. .reverse()
        8. .sort() = ascending order
    1. **Unordered Associative Cont**

1. **Algorithms**
   1. Generic function templates
   2. Operate on containers
   3. Types
      1. **Retrieve or Non-mutating Algorithms**
         1. **Adjacent\_find –adj pairs**
         2. **Count-occurrence of a value**
         3. **Count\_if—no.of elements that matches a predicate**
         4. **Equal-if two ranges are equal**
         5. **Find-**
            1. first occurrence of a value
            2. InputIterator find (InputIterator first, InputIterator last, const T& val);
            3. int myints[] = { 10, 20, 30, 40 };
            4. int \* p = find (myints, myints+4, 30);
         6. **Find\_end**
         7. **Find\_first\_of()**
         8. **Find\_if()-** find the elements that matches a predicate
         9. **For\_each()-** apply an operation to each element
         10. **Mismatch()**
         11. **Search\_ch()**
         12. **Search\_n()**

i = search(v1.begin(), v1.end(), v2.begin(), v2.end());

* + 1. **Mutating Algorithms**
       1. **Copy()**
       2. **Copy\_backward()**
    2. **Sorting Algorithms**
       1. sort(s,s+4,compare\_string);
       2. sort (first,first+5);

sort (second,second+5);

merge (first,first+5,second,second+5,v.begin());

* + - 1. for\_each (InputIterator first, InputIterator last, Function fn);

for\_each/transform(inches, inches+5, in\_to\_cm);

* + - 1. OutputIterator transform (InputIterator first1, InputIterator last1, OutputIterator result, UnaryOperation op);
    1. **Set Algorithms**
    2. **Relational Algorithms**

1. **Iterators**
   1. Provide interface for containers and algo
   2. Facilitate movement through containers
2. **Functions**

Functor : Allows instance obj of a class to be called as ordinary func

**// C++ program to illustrate the iterators in vector**

*//header*

#include <iostream>

#include <vector>

*//std*

using namespace std;

int main()

{//vector intialization

vector<int> g1;

for (int i = 1; i <= 5; i++)

g1.push\_back(i);

*[used to add elements at the end of a vector, increasing its size by one. It is commonly used to dynamically grow a vector.]*

cout << "Output of begin and end: ";

for (auto i = g1.begin(); i != g1.end(); ++i)

*[allows the compiler to automatically deduce the data type of a variable based on its initializer. It is commonly used to simplify code, especially when dealing with complex iterator types or other lengthy type names.]*

cout << \*i << " ";

cout << "\nOutput of cbegin and cend: ";

for (auto i = g1.cbegin(); i != g1.cend(); ++i)

cout << \*i << " ";

cout << "\nOutput of rbegin and rend: ";

for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)

cout << \*ir << " ";

cout << "\nOutput of crbegin and crend : ";

for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)

cout << \*ir << " ";

return 0;

}

**// C++ program to illustrate DEQUE**

#include <iostream>

#include <deque>

using namespace std;

void showdq(**deque <int> g**)

{

**deque <int> :: iterator it;**

for (it = g.begin(); it != g.end(); ++it)

cout << '\t' << \*it;

cout << '\n';

}

OR

void show(const **deque<int>& myDeque**) {

for (const **auto& element : myDeque**) {

cout << element << " ";

}

cout << endl;

}

int main()

{

deque <int> gquiz;

gquiz.push\_back(10);

gquiz.push\_front(20);

gquiz.push\_back(30);

gquiz.push\_front(15);

cout << "The deque gquiz is : ";

showdq(gquiz);

cout << "\ngquiz.size() : " << gquiz.size();

cout << "\ngquiz.max\_size() : " << gquiz.max\_size();

cout << "\ngquiz.at(2) : " << gquiz.at(2);

cout << "\ngquiz.front() : " << gquiz.front();

cout << "\ngquiz.back() : " << gquiz.back();

cout << "\ngquiz.pop\_front() : ";

gquiz.pop\_front();

showdq(gquiz);

cout << "\ngquiz.pop\_back() : ";

gquiz.pop\_back();

showdq(gquiz);

return 0;

}