STL

To initiate an iterator of a container 🡪 they act like pointers but are different like in case of array we have random access of elements.

Container\_type :: iterator i = vec.begin ();

1. Dereferencing works normally 🡪 (\*i) returns value
2. End () points to address next to the last element and begin() points the 1st element.
3. Next(i, n) , prev(i,n) 🡪 returns an iterator after advancing or decrementing n positions from the given base position
4. Advance(i,3) 🡪 changes the value of i to point 3rd next element

TEMPLATES

It’s a generic code and the compiler creates code for the specific data type as per use. Like the sort function, we can apply it to any datatypes like char, int, float etc.

1. Class template 🡪 stack, queue, dequeue
2. Function templates 🡪 sort, liner search

template <typename T>

T my\_func(T int a , T int b)

MACROS 🡪 It doesn’t check type-checking it just blindly searches and replaces the code.

#define new\_line cout<<”\n”;

PAIRS

* Like coordinates, item name and its price, day and max temperature etc.
* There are default constructors in pairs so if we do not store anything it will initialize it with (0,0).

pair<int,int> p2(23,33);

VECTORS

These use dynamically allocated array internally. Everytime the space of allocated is filled the new array of double size is created and all the elements are copied to it and if copying is successful then the previous array is deleted.

It can be used over arrays because 🡪

1. Dynamic size 🡪 we can have a dynamically allocated arrays and can deallocated when no more in use but again we have to predefine the size for allocating this adds to space wastage. If we are storing data of students of a college and if we consider case where number of students keeps on changing, we cannot tell using arrays and end up wasting space.
2. Rich library function
3. No need to pass size 🡪 like in arrays we have to pass size because arrays when passed are received as pointers to the function.
4. A function can be returned from a function (declared inside the function).

* While printing use & so that element is not copied everytime and also it is more efficient if we want to modify the vector because we have passed by reference.

for(int& i : v1)

* Time complexity of most of the operations is O(1) like begin(), end(), insert(), front(), size()
* push\_back() in worst case O(n) average is O(1) and pop\_back() can be considered as O(1)
* insert(), resize() and erase() is O(n)

FORWARD LIST

It is singly linked list implementation.

* assign(), insert\_after() , erase\_after(), push\_front(), pop\_front() is O(1)
* pop\_back() is not possible in O(1) because its single ll and we don’t have address of previous element.
* reverse() and remove() are O(n)

LIST

It is a doubly linked list implementation.

* front(), back(), size(), begin(), end(), erase(), push\_front(), pop\_front(), push\_back(), pop\_back() are O(1).
* reverse(), unique(), remove() are in O(n) because it need traversal.
* Unique function is used to remove consecutive duplicates

DEQUEUE

* Provides functionality of both stack and queue.
* Insert and delete from both ends.(possible using list O(1)).
* Provides random access(not possible in list O(1)).
* It is implemented in a little different way in C++.
* A deque is generally implemented as a collection of memory blocks. These memory blocks contains the elements at contiguous locations.
* Data is first filled in the middle part.

Here it is different from list because the data is not copied while expansion the pointer is passed.

* Insert() , erase() are O(n)
* push\_back(), push\_front(), pop\_back(), pop\_front() are O(1)

HEAP

Creating a heap 🡪O(n)

Removing a item 🡪 O(log n)

Identification🡪

1. k is given with combination of smallest and largest
2. k+ smallest 🡪 use max heap
3. k+ largest 🡪 use min heap

SET

It stores data in sorted form .

Ignores repeated words.