Standard Template Library

Consists of containers, generic algorithms, iterators, function objects, allocators ~~and~~ ~~adaptors~~ and adaptors.

Container:
- is an object that holds other objects.
  e.g. list, vector etc.
- ~~These are~~ Abstract and heavily depends on template
- Common use of container is to iterate through the container, looking at the elements one after anothers. Done by defining an iterator class appropriate to the kind of containers.

Iterator:
- used to ~~nav~~ navigate containers without the program having to know the actual type used to identify elements.
- Iterator is container specific.

```
vector <int> v;  ⇒ } ~~Cont~~ Containers are implemented
                   }  via template class definition.
vector <Student> l; }

vector <int> :: iterator p; } Iterator is container
                               specific. operⁿ.
                               depends on type of
                               container.
```

Generic Algorithms: A container needs to be supported by basic operations ~~to~~ like finding its size, capacity, copying, sorting, searching etc. for this purpose, function templates are provided.

Function Objects: its a mechanism by which user can customize the behaviour of standard algorithm.

Allocator: ~~Responsible~~ Handles physical memory allocation.

Adaptors: provides restricted interface to a container. They do not provide iterators. used only through ~~Specilar~~ Specialised interfaces.

vector, list, deque → basic container.

stack, queue → adaptors

~~Sequential Containers~~

vector

- expandable array, but growth may requiring lots of copying.
- Random access

#include <vector>

vector<int> v(30); | initialized by default value of data type. for int, it is zero.
→ vector of 30 int

~~Initialized~~

vector<int> v(4, 10, 3); | size 3, initialized values are 4, 10, 3.

vector<int> v(&x[i], &x[j]) ⇒ &x[i], &x[j] ⇒ iterators

Suppose, x is an existing vector, then v is initialized with x[i], x[i+1] ... x[j-1]
& [x[i], x[j])

- vectors can be assigned to each other
- subscript operator to access individual element. ([ ])
- also at()
- ==, !=, <, <=, >, >= } to compare the vectors. Corresponding elements are compared.
- v1.swap(v2) → to swap
- size() returns size
- empty() whether empty or not.
- insert(posn, data source) posn ⇒ iterator where to insert
- erase() erase all elements
- erase(posn) • erase(posn1, posn2) → prior to posn2

- pop_back() ⇒ Removes last element (it is not returned)
- front() ⇒ returns 1st element
- back() ⇒ returns last element
- push_back(data source) ⇒ appends data

```
vector <int> v;
    v[0] = —; // error as v is empty vector
  v.push_back(—); // o.k.
```

- resize(n) → new size n, addl. elements default value
- resize(n, data source) → new size n, addl. elmt. will have the value same as source.

- begin() → iterator to 1st element
- end() → " to last + 1
- rbegin(), rend() } Reverse

list

#include <list>

- like, doubly linked list
- preferred when deletion & insertion are frequent.

list <string> l1, l2(10), l3(5, "abc");

l1 ⇒ empty list, l2 of 10 empty string, l3 of 5 string each abc

list <string> l (it1, it2); | initialized with elements pointed by it1 to prior to it2.

to compare { = (assignment list to list)
==, !=, <, >, <=, >=

like vectors
- erase()
- erase(posn)
- erase(posn, begin, end)
- size()
- swap()
- insert() & its variants
- back()
- front()
- empty()
- resize()

begin()
end()
rbegin()
rend()
} Returns iterators

④

ⓐ for vector and list if elements are not basic type then for Comparison Operator overloading is read.

- push_front()
- push_back()
- pop_front() } Removes value (not returned)
- pop_back() }
- remove(source) → all occ. of source is removed.

- sort()
- unique() can work on sorted list. for multiply occurring elements one element is retained.
- merge() → l1.merge(l2) to merge sorted lists. if not ordered, o/p may not be completely ordered.

merge(), sort() requires < and == are defined on the data elements. for user defined Objects ⇒ these must be defined. (Returning boolean)

Stack (limited interface)

empty()
Size()
top() returns 1st element
push(_) ⇒ adds at top
void pop() → Removes top element

Queue [limited interface]

```
#include<queue>                 queue<string> q1;
                                  q1.push(—);
                                     !
empty()
size()
front() ⇒ returns 1st element, also can be reassigned
void pop() ⇒ Removes 1st element (not returned)
back() ⇒ Returns last element, also can be reassigned
push(source) ⇒ appends

      cout << q1.front();
      q1.front() = "—";

   while (!q1.empty())  →   while (q1.size())
   {                          {
      cout << q1.front();       =
        q1.pop();             }
   }
```

Priority Queue

#include <queue>

Similar to queue but inserted according to priority.

smaller value ⇒ < ~~higher~~ lower priority.

to change priority definition operator< to be overloaded.

```
push(data) ⇒ adds to desired location
top()  ⇒ returns top (highest priority) element
void pop()  ⇒ Removes top element.
size()        ~~priority~~
empty()
```

```
#include
priority_queue <string> q;

    q.push(—);
        ⋮

while(q.size())
  { cout<<q.top()<<"\n";
     q.pop();
  }
```

provides string in reverse order (higher ascii value, higher priority)

if required in ascending order ⇒

```
class Test
   { string w;
   public:
        ~~string~~ Test(string s)
                { w=s;
                }
      bool operator<(Test t)
        { return (w > t.s)
        }
   };
priority_queue <Test> q;
          ⋮
```

deque

double ended queue

```
# include <deque>
```

deque () → initializes an empty deque

deque(argument) → initializes with another deque as argument.

deque (n, argument) → n elements, all value argument

begin()
end()
rbegin()
rend()
} Returns iterator.

front()
back()
} Returns 1st & last element, also can be reassigned.

size()
empty()
swap()
push_back(source)
push_front(source)
pop_back()
pop_front()
insert() & its variants.
erase()

pair

#include <pair> Can store two elements called first and second.

pair< type1, type2> x (value1, value2);

x.first }
x.second } to get the values

x.first = — }
x.second = — } to set the values

pair <–, –> y;

Can also be assigned as || y = pair <–, –> (value1, value2);

map

implements sorted associative array.

include <map>

A map is filled with key/value pairs. key used for looking up the information belonging to the key. value is the associated information.

- Each key can be stored only once.
- ~~[illegible]~~

map < ~~string~~ type1, ~~int~~ type2 > m1; (empty map)
⇓ key ⇓ value

to create a value for such map element

map<type1, type2> :: value_type(value1, value2)

insert (argument) ⟶ map element

- create an empty map
- Insert element with map element created ~~as~~ using value_type() as argument.

Or pair <—, —> p[] = { pair <—, —>, pair <—, —>, ... }

~~then~~ map <—, —> Object(&p[i], &p[j])
iterators [start, end)

map <—, —> x(y)
↳ initiates with this map y.

begin() }
end() } returns
rbegin() } iterators
rend() }

empty()
size()
swap()

- Subscript operator Object[key] = value;
  - can be used to reassign.
  - can also be used to access

- Insert (argument) ⟹ Returns
  map element
  pair <iterator, bool>

~~map~~

points to data element in the map

true, if value was inserted or not.
true ⇒ key does not exist
false ⇒ key already exists

~~map <—, —>::iterator~~
~~p = insert(—)~~

map <—, —> ~~xxx~~ m;

pair <map <—, —>:: iterator, bool> r;

~~e~~ ~~insert~~ ~~m.insert~~

r = m. insert (map <—, —>:: value_type(—,—));

r.first → first
iterator   key of element

r.first → second
iterator   value

r. second ⇒ boolean.

- variants of insert()

- ~~xxx~~ erase (key) → iterator
- erase (position)
- find (key) → Returns iterator pointing to element, if not found returns end()

Count (key) → 1 if key is available in map else 0.

multimap
→ allows multiple entries of ~~key~~ same key.

set
set of values   set<—> :: x;
↑ not pair

multiset

hash-map

#include <hash-map>

hash-map implements an associative array in which the key is stored according to hashing scheme.

map ⇒ stores sorted keys. But faster method of storing keys is to use hashing.

hash fn. on key provides the index in table where keys are stored.

Constructor of hash-map needs a key-type, a value type, an object creating a hashvalue for the key, and an object ~~[illegible]~~ comparing two keys for equality.

hash functions are available for char const* keys and for all scalar numerical types short, int etc. including char.

for other datatype a hash function and an equality test must be implemented. (one can use function objects).

for such cases ~~two~~ key and value is sufficient. unless hash fn. and equality checks are customized.

~~[illegible]~~

```
hash-map <keytype, value type>                default hash fn., ==
hash-map <keytype, value type, hash fn.>      default ==
hash-map <key type, valuetype, hf, eq. chk>   hf(), eq.chk()
```

class eqr

```
class Equal
  {
   public:
      size_t operator()(keytype t1, keytype t2)
         { // write code to chk. equality of t1, t2
              and return true if equal else false
         }
  }

 class hashfn
   {
     public:
       size_t operator() (key type t1)
```
~~keytype~~

```
          {
           //apply hash fn. as per design
             & return the index
          }
   }
```

map < keytype, value type, hashfn, Equal> m;

~~m[key val] = value~~

```
      m[key1] = val1     } assigning
           ,
           ,
   Cout << m[key1] ;     } retrival
```

String #include <string>

→ String x("___");

String x ;

String y(x)

Char const * c = ~~Str~~ x . c_str()

↓ Stringobject

returns c like array of cha.

x[index] ⟶ no bound check

x . at(index) ⟶ bound checked

x is an object, & x (not allowed)

two strings can be compared using
==, !=, <, <=, >, >=

or x.compare(y)

to concatenate: x + y
x.append(y)

insert(posn., char)

swap()
erase()
size() empty()

begin()
end()
rbegin()
rend()
} iterators.

replace(arg1, arg2, arg3, arg4) → index of 1st char. to be used in arg3

arg1 ↓ posn. of 1st char to be replaced.

arg2 ↓ no. of char. to be replaced.

arg3 ↓ replacement text string / char const*

String substr(start posn., no. of characters)

String substr(start posn.) till end

String substr() copy & returns

→ find(string) → if found, returns position (of 1st char.)
if not found returns
string::npos

searching substring

~~rfind(string) → [illegible]~~

find_first_of(string) → not substring matching any of the given cha.

find_first_not_of("—")

find_last_of("—")

find_last_not_of("—")