STL (Standard template Library).

- S'qt is a collection of templete class that prowide data structure such as arrays, vectors etc.
- qt consist Containers, algo & Iterators.
- qt is a generalized library that is independent of datatypes.

Components of STL :-

(1) Containers :-

- qt is a collection of objects of a particular type of ~~do~~ DS.
- we have many types of container class Array, vector, queue, deque, list, map, set etc.
- These containers are generic in nature & implemented as class templates.
- Containers are dynamic in nature & can be used to hold various types of objects.

(a) Sequential containers :-

- ~~Containers~~ that can be accessed in sequential or linear manner. Such as Arrays, vectors, Lists, Deques

(b) Associative Containers:-

- It implement ~~only~~ sorted data structure.
- These containers are fast to search.
- Ex- Map, set, Multimap, multiset etc.
- Implemented in a key/value pair.

(c) Container Adopters:-

- These are sequential containers & implemented by providing a different interface.
- Containers like queue, deque, stack and priority queue are all classified as containers adopters.

(2) Iterators:-

- They are used to traverse through the container objects.
- Similar to indexes that we use to step through the arrays, Iterators act to container class objects and can be used to step through the data.
- Bridge b/w container & algorithm.

- Iterators always point to containers and infact algorithms actually operate on iterators and never directly on containers.

Iterator types:-

- Input Iterator - used in single pass algo.
- Output Iterator - same as Input but not used for transversing.
- Bidirectional Iterator - move both direction
- Forward iterator :- can only used in forward direction, one step at a time
- Random access iterator :- Same as pointer. can be used to access any element randomly.

(3) Algorithm :-

- It is a set of functions or methods provided by stl. that act on containers.
- These are built in functions & can be used directly with the STL containers and iterators instead of writing your own.

STL supports these algo:-

(a) Searching Algo
(b) sorting algo
(c) Modifying & manipulating algo.
(d) Non-modifying algo.
(e) Numeric algo
(f) Min/Max algo.

---

(1) Arrays In STL:-

Syntax:-

```
#include<array>
array<object_type, size> array_name;
```

Ex- `Array<int,5> myarray = {1,2,3,4,5};`

functions supported by array container:-

(a) At:- Return value in array container at given posit

(b) Front:- Return first element

(c) Back - last element.

(d) Fill - Assign a given value to every element

(e) Swap - Swap contents of two array indexwise

(f) Empty - Array container is empty or not, return 0, 1.

(g) Size - No. of element in array container

(h) Max_size - Returns the max size of the array container.

(i) Begin :- Return iterator begining point.

(j) End :- Return the iterator pointing location next to last element in array container.

Vectors In STL :-

Vector is used to store element as array but vector is dynamic u can store element without wasting memory.

Vectors are dynamic array containers that resize it automatically when elements are inserted or deleted

Storage of vector is handled by vector container itself.

Syntax :-

```
#include <vector>
std:: vector<int> my_vec;
```

Vector iterator :

(a) begin() : first element
(b) end () : last element
(c) rbegin() : last element / reverse iterator.
(d) rend() : First element / reverse iterator.

(e) cbegin(): constant iterator pointing to first element in vector container.

(f) cend(): last element in vector container.

(g) crbegin(): reverse constant iterator pointing last element.

(h) crend(): reverse constant iterator pointing first element.

* Use resize() to decrease or increase size of vector.

* max_size() - Return maximum size i.e. maximum no. of elements the vector can hold.

* capacity() - size of storage currently allocated

* empty() - check vector is empty or not.

* shrink_to_fit(): Shrinks the vector capacity to fit the size and discard all the elements.

* reserve(): Reserve the vector capacity to certain n elements.

* vectorname.erase(vec.begin()+1,3); to delete.

* vectorname.insert(vectorname.begin()+index, 20);

* Swap

* Lists in STL :-

- list is a container that overcomes this drawback of the array and vector containers.
- It allow us to insert element anywhere in the lists ~~in STL~~ without calling much of an overhead.
- It is slower than vector.

```
#include <list>

std:: list<Objecttype> listname;
std:: list <int> mylist = {1,2,3,4,5};
```

```
#include <list>
#include <iterator>
using namespace std;
int main() {
  list <int> mylist = {1,1,2,3,5};
  cout << "list elements are: ";
  list<int>:: iterator it;
for(it=mylist.begin(); it != mylist.end(); ++it)
  {
    cout << *it << endl;
```

O/P — 1
1
2
3
5

operations:-

① Insert - insert element to given pos.

② insert(position, no. of element, element)
↓ default 1.

```
int main() {
    list<int> mylist = {1,1,2};
    list<int>::iterator it = mylist.begin
    advance(it, 3); // iterator to 4th pos
    mylist.insert(it, 3);

    for(it = mylist.begin(); it != mylist.end(); ++it){
        cout << *i << " ";
```

O/P - 1 1 2 3

(2) push_back : Add new element at the end of list
(3) push_front : Add new element at the front of list.

```
my list.push_back(5);
my list.push_front(0);
```

(4) pop_back :- mylist.pop_back();
pop_front :- mylist.push_back();

(5) size - no. of elements in list
(6) empty - check empty or not
(7) erase - remove a element or range of element.
(8) clear - Remove all element make size 0.

```
mylist.size(); mylist.empty(); mylist.erase(it);
mylist.clear();
```

(9) front, Back
(10) swap - swap content of one list to another
(11) reverse.- (12) sort.

```
mylist.swap(oddlist); oddlist.sort()
```

* splice :- used to transfer the contents of one list to another list at specified position.

```
splice (position, list);

mylist = {1, 1, 8, 13}
seclist = {2, 3, 5}
mylist.splice (it, seclist); // it = 2 ++.
o/p = 1 1 2 3 5 8 13
```

* merge :- transfer the content of one list to another
Both lists are needed to be in sorted order

```
mylist.merge (seclist);
o/p - 1 1 2 3 5 8 13
```

* STACK & QUEUE IN STL :-

STACK
```
#include <stack>
stack<objecttype> stackname
```
Push
pop
top
empty
size

Queue
```
#include <queue>
queue<objtype> queuename
```
Push
pop
front
back
empty
size

MAP in STL :- key/~~pair~~value pair

```
#include <map>
map <keytype, valuetype> map_name;
map<int, int> mymap = {{1,10}, {2,20}}
```

operations :-

* at and [] - used to access the element of map.

If accessed key is not present in map then 'at' throws exception whereas [] operator insert new key if accessed key is not present.

* begin : first element in map
* end.

```
map<int, int> mymap {{1,10},{2,20}, {3,30}};
map<int, int>::iterator it;
cout<<"\t key \t value \n";
for (it = mymap.begin; it != mymap.end; ~~it~~ it){
    cout <<'\t'<< it->first <<'\t'<<
        it->second <<'\n';
    Cout << endl;
```

o/p - key	Value
1	10
2	20
3	30

```
mymap.at(2) = 10
mymap[3] = 10
```

* insert (key, value) –
  mymap.insert(pair<int,int>(1,1));
* erase –
  mymap.erase(3);
* empty –
* size – mymap.size();
* max_size – Return max size the map can hold.
* clear – mymap.clear(); size = 0

Multimap :–

Same as map but having multiple with same key.

```
#include<map>
multimap<int,int> mymap;
mymap.insert(pair<int,int>(1,1));
                          (2,3));
                          (3,5));
                          (3,7));
```

map.erase(3); 2 removed.

Unordered map :–

In Unordered map as the name suggest the elements or keys can be in order.

Internally it is implemented as 'Hash table

#include <unordered_map>
unodered_map < key type, value type > umap name;

* unordered_map<string, int > umap;

umap ["RED"] = 1;
umap ["BLUE"] = 2;
umap ["GREEN] = 3;

umap.insert(make_pair ("YELLOW", 4));

unordered_map <string, int> :: iterator it;
for (it = umap.begin(); it != umap.end(); ++it){
    cout << it -> first << it -> second;

cout << umap.bucket_count() // 11
cout << umap.bucket_size(2); // 0

* begin, end, at
* bucket_count
bucket - no. of elements with key that is located on map.
bucket_size -
count - no. of element

Iterators -

stack, queue, Priority queue — No iterator

Vector, deque — Random access

List, map, multimap, set, multiset

↓

Bidirection.

Numeric Algorithm: -

#include <numeric>

(i) accumulate

accumulate (first, last, sum); ——

sum of all variable from first to last and sum is a variable in which sum stored

(ii) Partial_sum

partial_sum (first, last, b);

int A[] = {21, 25, 64, 32}; int b[4];

partial_sum (A, A+4, b);

O/P - 21, 46; 110, 142

* iota - Fill a range with successive increments of starting value.
* reduce:- similar to accumulate, except out of order.
* inner_product - computes the inner product of two ranges of elements.
* adjacent_diff - compute diff. b/w adjacent element in range.

* Inclusive_scan — Similar to partial sum, includes the ith i/p element in the ith sum

* Exclusive_scan — Similar to partial sum exclude

Non-modifying —

① count — int values[] = {5,2,5}
int count_5 = count(values, values+2, 5)
cout << count_5; // 2

② equal
③ mismatch — return a pair of iterators when two iterators are compared and a mismatch occurs.
④ search
⑤ search_n — searches in a given ~~sequence~~ range for sequence of count value.

Modifying —

① reverse — reverse(vec2.begin(), vec2.end());

② swap — a.swap(b)

Min max :—

min(x,y)
max(x,y)

Function Template

Syntax:-

```
template <class T>
return_type function_name(args_list)
{}
```

```
template <class T>
void swap(T& x1, T& x2){
   T temp = x1;
  x1 = x2 ; x2 = temp ; }

main(){
  int i = 10, j = 80;
   swap(i, j);
   char c = 'A' , d = 'B';
    swap(c, d); }
```

* Templates enables us to define generic classes or functions.

* It make it possible to use a single function or a class defination for a variety of data types i.e. to define a family of functions or classes.

* Template is defined with a parameter that would be replaced by specific data type at the time of actual use of class or function.

* It can be considered as a type of macro.

It can be - function templates or class templates.

Need for template function:-

- To find min of two numbers, depending on the data type of no.s, the function have to be declared.

- int min (int a, int b);
  float min(float a, float b);

- This is though the functionality remain the same, separate functions are required for separate data types.

- This repetation of code can be avoided by using template function.

Performance:-

- Compiler doesn't compile any code when it encouter template function defination.

- Compiler generates a specific version of template function corres. to argument types.

- Finally at this point compiler generates the function call code

- Use of template doesn't help save memory because all the diff. versions of function are ultimately generated.

Adv.

* The code can be written with generic version only.
* Template provide reuse of source code
* Templates can significantly reduce source code size & increase code flexibility without reducing type safety.

---

* Template function may be overloaded either by template functions or ordinary function of its name.
* Overloading resolution is accomplish as follows:-
  - Call an ordinary function that has an exact match.
  - Call a template function that could be created with an exact match.
  - Try normal overloading resolution to ordinary functions and call the one that matches.
* error generated if no match found
* No automatic conversion are applied to arguments on template func.

```
template <class T> T Max (T a, T b) {return a>b?a:b;}
void f(int a, int b, char c, char d) {
  int m1 = Max(a, b); // Max (int a, int b)
  char m2 = Max (c, d); // max(char c, char d)
  int m3 = Max (a, c); // error - 'c' is a
                          char & can't be
                          converted to int.
```

Thus a template func can be overridden to perform a specific processing for a particular data type.

Ex :-
```
template <class T>
  void f (T, a)
   cout << " inside temp " << a; }
   void f (char c) {
    cout << " non temp " << c; }
main ()
 { f (10); // inside temp 10
   f ('A'); // non temp A
   f (10.8); // inside temp 10.8.
 }
```

```
template <class T>
T max (T a, T b){
  cout << "templ\n";
   return a>b?a:b;
}

void main(){
    int a=90, b=80;
    char c='a', d='n';
  cout<< max(a,b)<<endl;
  cout<< max(c,d)<<endl;
  cout<< max(a,c)<<endl; //error! can't
                          generate max
                          (int a, char c)
```

```
template <class T> void f1(T); //fine.
template <class T> void f2(T*); //fine
template <class T> T f3(T, int); //error
template <class T, class C> void f4(T, C); //error
template <class T> void f5(const T&, complex); //fine.
template <class T> void f6(vector<list<T>>); //fine
```

* Complex is a class

```
class A { int a1, a2;
  public: A(int x): a1(x), a2(2*x) { }
      int f() { return a1; } };

template <class T1, class T2>
T2 anything(T1 a, T1 b, T2 c)
 {return T2(a+b+c.f()); }

char anything(char b) {return b;}
char anything(int a, int b, int c) { return 'a'; }

main(){
  int x=9, y=10;
  long p=10, q=11;
  char c='z', d='A';
  A a1(60);
  A a_obj = anything(p, q, a1);
  cout<< anything('q');
  cout<< anything(1,2,'q');
```

o/p = q a

Doesn't return anything because parameter of diff. data types. Type conversion is not available in template.

C++11 features.

1. auto:

* int arr[] = {10, 20, 30, 40, 50};
for (int num : arr)
cout << num; // 10 20 30 40 50

Disadv :- (i) Specific index can't be treated.
(ii) Revisiting one or more elements and skipping a group of element can't be done.
(iii) Iteration can't be in reverse order.

* Char str[] = "Hello world"
for (char c : str)
cout << c << " "; // H e l l o w o r l d
for (char c : "Helloworld")
cout << c << " "; // Helloworld

* std::map<int, char> MAP ({{1, 'A'}, {2, 'B'}, {3, 'C'}});
for (auto m : MAP)
cout << m.first << m.second; // 1A 2B 3C

Constexpr

```
constexpr int multiply (int x, int y)
 {
    return x*y;
 }
void main ()
 { const int result = multiply (5,10);
     cout << result;
 }

constexpr long int fib (int n)
 {
   return (n<=1)?n: fib(n-1)+fib(n-2)
 }
 void main
  { // value of res computed at compile time.
    const long int res = fib(30);
     cout <<res;
 }
```

→ always computation is at ~~Runtime~~ compiletime.

Constexpr variables

constexpr literal-type identifier = constant-expression.

constexpr literal-type identifier {constant-expression};

constexpr literal-type identifier (params);

```
Constexpr float a = 42.0;
Constexpr float b = {108};
Constexpr float c = exp(5,3);
constexpr int d; // error! Not initialized
 int e = 0;
Constexpr int f = e+1;
  //Error! e not a constant expression.
```

* Constexpr functions.

return value can be computed at compile.

```
Constexpr float powerN(float x, int n)
{ return n == 0 ? 1 :
        n % 2 == 0 ? PowerN(x * x, n/2) :
        power(x * x, (n-1)/2) * x;
    };

Constexpr float res = powerN(2,6);
```

* Constexpr with Constructors:—

```
class Rectangle {
  int mLength, mBreadth;
  public:   // A constexpr constructor.
  ConstexprRectangle(int l, int b): mlength(l), mBreadth(b)
  constexpr Rectangle int getarea() const
    { return mlength * mbreadth; }
};

void main(){
     //object is initialized at compile time
  Constexpr Rectangle obj(10,20);
  std:: cout << obj.getArea();
}
```

RTTI

Run time type information:-

- RTTI is a mechanism that allows the type of an object in an inheritance hierarchy to be determined during program execution.

- RTTI was added to cpp lang. because
Many vendors of class libraries were implementing this functionality themselves.

This caused incompabilities b/w libraries. Thus, it became obvious that support for run-time type information was needed at the language level.

- The concept applies to pointers as well as references.

* Need of RTTI:-

```
class shape { int col; public: void paintMe(); };
class circle: public shape { };
class rectangle: public shape { };
main() {
  shape *sh[5]; rectangle r; circle c;
// some elements of sh point to
// circle object
// the remaining point to rectangle object
//All circles to be painted RED -> sh[i]
```

If we need to change color of all circles only then we need to downcast the

base pointer back to that of the correct derived class.

* Three main elements to RTTI :-

(i) The typeid operator :-

used for identifying the exact type of an object.

(ii) dynamic_cast operator :-

used for conversion of polymorphic type.

Type checking & casting at a ~~tg~~ time.

base pointer back to that of the correct derived class.

* Three main elements to RTTI:-

(i) The typeid operator:-

used for identifying the exact type of an object.

(ii) dynamic_cast operator:-

used for conversion of polymorphic type.

Type checking & casting at a ~~to~~ time.

usage of typeid :-

```
#include <typeinfo>
class shape {};        public: void ar() {}
class circle : public shape {};
class rectangle: public shape {};
                       public: void ar(){}
void main(){
    shape * sh[8]; int i;
   for(i=0; i<8, i++){
       if(i%2==0)
          sh[i] = new circle();
       else
          sh[i] = new rectangle();
    }

   for(i=0; i<8; i++)
        cout<<endl<< typeid(sh[i].name())
        << "::" << typeid(*sh[i]).name();
```

o/p = class shape* :: class circle
class shape* :: class rectangle
so on -

dynamic cast

```
for(i=0; i<8; i++){
  if(cir_ptr = dynamic_cast<circle*>(sh[i]))
    cout<<endl<<"dynamic cast::circle";
  else
  cout<<endl<<"dynamic_cast::rectangle";
```

example emphasize.

* RTTI should be used only with polymorphic classes i.e. those which have a virtual function in the base class. (as in second ex)

* In the absence of polymorphism the static type info is used. (as first ex)

* typeid() can be used with built-in types also.

```
cout << endl << typeid(45).name();
```
o/p – int.

```
class employee {
public:
    virtual int salary();
};
class manager : public employee {
public:
    int salary();
    virtual int bonus();
};
void calc(employee *pe) {
    // employee salary calculation
    if(mg)) { // use manager::bonus()
}
    manager *pm = dynamic_cast<manager*>(pe);
else {
    // use employee's member function
    }
```

If the above program, dynamic casts are needed only if the base class employee and its derived classes are not available to users (as in part of a library where it is undesirable to modify the source code).

Otherwise, adding new virtual functions and providing derived classes with specialized defination for those functions is a better way to solve this.

* Upcasting with dynamic_cast means moving a pointer up a class heirarchy.

Code ss:

In this ex.

gPtrTemp->parf() can't be invoked

dynamic cast can be used only with pointers and references to objects.

- Its purpose is to ensure that the result of the type conversion is ~~not~~ valid complete object of requested calls.

- Therefore, dynamic-cast is always successful when we cast a class to one of its base classes.