I. Introduction to Templates

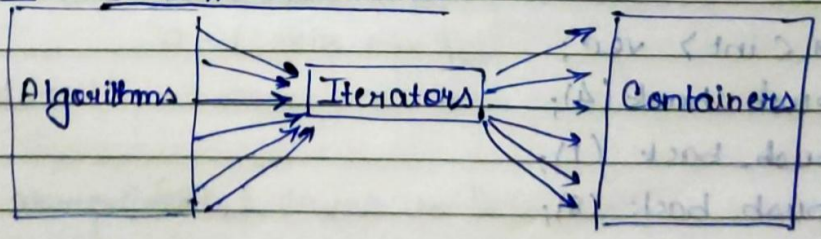
```
template <typename T>
T square(T x){
  return x*x;
}
```

```
template <int val>
void addVal(int i) {
  cout << val+i << endl;
}
```

```
template <typename T>
class BoVector {
  T arr[1000];
  int size;
public:
  BoVector(): size(0) {}
  void push(T x) { arr[size] = x; size++;}
  T get(int i) const { return arr[i]; }
  int getsize() const { return size; }
  void print() const { for (int i=0; i<size; i++) {cout<<arr[i]<<endl;}}
};
```

```
template <typename T>
BoVector<T> operator*(const BoVector<T>& rhs1, BoVector<T>& rhs2)
{ BoVector<T> ret;
  for (int i=0; i< rhs1.getsize(); i++){
    ret.push(rhs1.get(i) * rhs2.get(i));
  }
  return ret;
}
```

II. STL #1: Overview



```
#include <tuple>
tuple<int, int, int> tpl;
get<0>(tpl) = 1;
get<1>(tpl) = 2;
get<2>(tpl) = 3;
```

```
tpl = {1,2,3}
cout << get<0>(tpl);
```

Date / /
Page
STUDY BUDDIES

III. STL #2: Sequence Containers

① Containers:

- Sequence containers (Implementation: Array & linked list)
  - vector, deque, list, forward list, array
- Associative containers (binary tree)
  - set, multiset
  - map, multimap
- Unordered containers (hash table)
  - unordered set / multiset
  - unordered map / multimap

② STL Headers:

```
#include <vector>
#include <deque>
        <list>
        <set>          // set and multiset
        <map>          // map and multimap
        <unordered_set>    // unordered set/multiset
        <unordered_map>    // unordered map/multimap
        <iterator>
        <algorithm>
        <numeric>      // some numeric algorithm
        <functional>
```

③ Vector:

{only STL that gets ~~contiguous memory locations~~}

```
vector<int> vec;      // vec.size() = 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);     // vec: {4, 1, 8};   vec.size() = 3
```

```
// Vector specific operations:
cout << vec[2];  // 8 (no range check)
cout << vec.at(2); // 8 (throw range_error exception if
                   // out of range

for (int i=0; i<vec.size(); i++)
        cout << vec[i] << " ";

for (vector<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr)
        cout << *itr << " ";

for (it : vec)
        cout << it << " ";

// Vector is a dynamically allocated contiguous array in memory
int* p = &vec[0];
cout << p[2];  // 8

// Common member functions of all containers.
// vec: {4, 1, 8}
if (vec.empty()) { cout << "Not possible.\n"; }

cout << vec.size();  // 3

vector<int> vec2(vec); // Copy constructor; vec2: {4, 1, 8}

vec.clear(); // Remove all items in vec; vec.size() == 0

vec2.swap(vec); // vec2 becomes empty, and vec has 3 items
```

- Properties of Vector:
  1. fast insert/remove at the end: O(1)
  2. slow insert/remove at the beginning: O(n) or in the middle
  3. slow search: O(n)

(4) Deque

```
deque<int> deq = {4, 6, 7};
deq.push_front(2);
deq.push_back(3);

cout << deq[1];  // 4
```

- Properties of deque:
  1. Fast insert/remove at the beginning and the end: O(1)
  2. slow insert/remove in the middle: O(n)
  3. slow search: O(n)

(5) List (Doubly linked list)

```
list<int> mylist = {5, 2, 9};
mylist.push_back(6);   // mylist: {5, 2, 9, 6}
mylist.push_front(4);  // mylist: {4, 5, 2, 9, 6}

list<int>::iterator itr = find(mylist.begin(), mylist.end(), 2);
                                                  // itr → 2
mylist.insert(itr, 8);  // mylist: {4, 5, 8, 2, 9, 6}
                        ↳ O(1), faster than vector/deque
itr++;                  // itr → 9
mylist.erase(itr);      // mylist: {4, 5, 8, 2, 6}

mylist1.splice(itr, mylist2, itr_a, itr_b);   // O(1)
                ↳ Some people use list just for this purpose
```

- Properties:
  1. fast insert/remove at any place: O(1)
  2. slow search: O(n) → slower than vector (not contiguous)
  3. no random access, no [] operator → to do with loading memory into cache

⑥ Forward list

- You can only traverse from beginning to the end, not the reverse

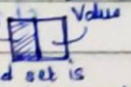
⑦ Array

```
array<int, 3> a = {3, 4, 5};
a.begin();
a.end();
a.size();
a.swap();
array<int, 4> b = {3, 4, 5}; // error: different types
```

IV. STL # 3: Associative Containers

① (Associative) container

- Always sorted, default criteria is <
- No push_back(), push_front()



(and set is special case of map)

② Set → No duplicates

```
set<int> myset;
myset.insert(3);  // myset: {3}
myset.insert(1);  // myset: {1,3}
myset.insert(7);  // myset: {1,3,7}.  O(log(n))
```

* [Logarithmic time searching is the most important feature of associative containers] STUDY BUDDIES
sequence containers don't even have find() member function

```
set<int>::iterator it;
it = myset.find(7); // O(log(n)), it points to 7

pair<set<int>::iterator, bool> ret;
ret = myset.insert(3);      // no new element inserted
if (ret.second == false)
    it = ret.first;         // it now points to element 3

myset.insert(it,9); // myset: {1,3,7,9}
// points to 3, * acts as a hint for insertion. O(log(n)) => O(1)
myset.erase(it);    // myset: {1,7,9}

myset.erase(7);     // myset: {1,9}
```

{ None of the sequence containers provide this kind of erase }

③ Multiset

- Set that allows duplicated items

```
multiset<int> myset;
```

- set/multiset : value of the elements cannot be modified
  i.e., `*it = 10;` not allowed → (it is read-only)

- Properties:
  1. Fast search : O(log(n))
  2. Traversing is slow (compared to vector and deque)
  3. No random access, no [] operator

- class template does not infer parameter types, a function template does

④ **Map** → No duplicated key

```
map<char, int> mymap;
mymap.insert(pair<char,int>('a', 100));
mymap.insert(make_pair('z', 200));

map<char, int>::iterator it = mymap.begin();
mymap.insert(it, pair<char, int>('b', 300)); // "it" is a hint

it = mymap.find('z'); // O(log(n))

for( it = mymap.begin(); it != mymap.end(); it++)
    cout << (*it).first << "=>" << (*it).second << endl;
```

⑤ **Multimap**

- Map that allows duplicated keys

```
multimap<char, int> mymap;
```

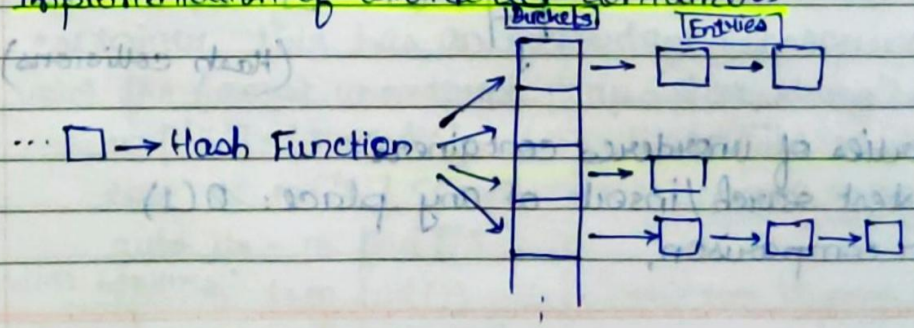
- Map / multimap : keys cannot be modified; type of *it : pair<const char, int>

  eg. (*it).first = 'd'; will give error

V. STL #4: Unordered Containers

Introduced in C++ 11

① **Implementation of unordered containers**



② Unordered set

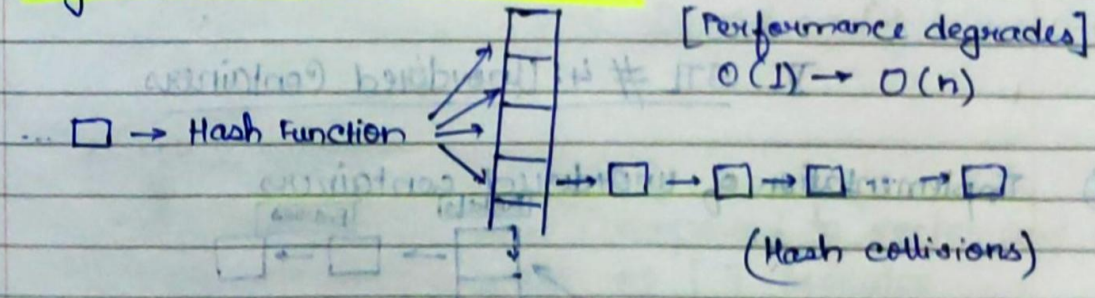
```
unordered_set <string> myset = {"red", "green", "blue"};
unordered_set <string>:: const_iterator itr = myset.find("green");
if (itr != myset.end())                                    // O(1)
    cout << *itr << endl;                                  (amortized constant time)
myset.insert("yellow");                                    // O(1)

vector <string> vec = {"purple", "pink"};
myset.insert(vec.begin(), vec.end());
// Hash Table specific APIs                                (ratio = total no of elements / total no of buckets)
cout << "load_factor = " << myset.load_factor() << endl;
string x = "red";
cout << x << " is in bucket #" << myset.bucket(x) << endl;
cout << "Total bucket #" << myset.bucket_count() << endl;
```

③

- **Unordered multiset**: Unordered set that allows duplicated elements
- **Unordered map**: Unordered set of pairs
- **Unordered multimap**: Unordered map that allows duplicated keys

④ Degraded unordered containers



⑤ Properties of unordered containers:

1. Fastest search / insert at any place: O(1)
   For comparison,

Amortized constant time

{Guaranteed Logarithmic time}

- Associative container takes $O(\log(n))$
- vector, deque takes $O(n)$
- list takes $O(1)$ to insert, $O(n)$ to search

2. Unordered set / multiset: element value cannot be changed
   Unordered map / multimap: element key cannot be changed

⑥ Associative Array

- [There is no separate container by this name, but it can be implemented with map or unordered map]

```
unordered_map<char, string> day = {{'S', "Sunday"}, {'M', "Monday"}};
cout << day['S'] << endl;
cout << day.at('S') << endl;

vector<int> vec = {1, 2, 3};
vec[5] = 5;   // Misbehave at runtime
```

But for associative array,

```
day['W'] = "Wednesday";   // Inserting {'W', "Wednesday"}
day.insert(make_pair('F', "Friday"));   // Inserting {'F', "Friday"}
```

However,

```
day.insert(make_pair('M', "MONDAY"));   // Fail to modify, it's an unordered_map
day['M'] = "MONDAY";   // succeed to modify
```

→ Subscript operator

★ Compiler won't catch it, It just doesn't do anything at all

- Subscript operator provides a write access to the container, this has an important consequence →

```
void foo(const unordered_map<char, string>& m) {
    m['S'] = "SUNDAY";   // Won't compile as m is const
    cout << m['S'] << endl;   // Won't compile as it assumes we'll write into m
    auto itr = m.find('S');
    if (itr != m.end())
        cout << (*itr).second << endl;
}
foo(day);
```

Correct way to print an element

- Notes about associative array:
  1. Search time: unordered_map, O(1);
     map, O(log(n))
  2. Unordered_map may degrade to O(n)
  3. can't use multimap and unordered_multimap, they don't have [] operator

⑦ Container Adaptors

- Provide a restricted interface to meet special needs
- Implemented with fundamental container classes

1. Stack: LIFO, push(), pop(), top()
2. Queue: FIFO, push(), pop(), front(), back()
3. Priority queue: first item always has the greatest priority
   push(), pop(), top()

⑧ Another way of categorising containers:

1. Array based containers:
   vector, deque
2. Node based containers:
   list + associative containers + unordered containers

Array based containers invalidate pointers:
↓ includes
native pointers, iterators, references

Eg,

```
vector<int> vec = {1, 2, 3, 4};
int *p = &vec[2];          // p points to 3
vec.insert(vec.begin(), 0);
cout << *p << endl;        // undefined behaviour
```

{Insertion invalidates pointer}

Node based containers don't have this problem

VI. STL #5 : Iterators and Algorithms

① Iterators

1) Random Access Iterator : vector, deque, array

```
vector<int>::iterator itr;
itr = itr + 5; // advance itr by 5
itr = itr - 4;
if (itr2 > itr1) ... // compare 2 iterators (which one is in front)
++ itr;        // faster than itr++
-- itr;
```

2) Bidirectional Iterator : list, set/multiset, map/multimap

```
list<int>::iterator itr;
++ itr;
-- itr;
```

3) Forward Iterator : forward_list

```
forward_list<int>::iterator itr;
++ itr;
```

•) Unordered containers provide at least forward iterators. Depending on the implementation, it may provide bidirectional iterators

4) Input Iterator : read and process values while iterating forward

```
int x = *itr;
```

(Subset of functionality of forward iterator)

{you cannot write to it} {can only move forward}

5) Output Iterator : output values while iterating forward

```
*itr = 100;
```

{you can write into a dereferenced output iterator, but you can't read from it}

② Every container has an iterator and a const_iterator

```
set<int>:: iterator itr;
set<int>:: const_iterator citr;  // Read only access to container elements

set<int> myset = {2,4,5, 1,9};
for (citr = myset.begin(); citr != myset.end(); ++citr) {
    cout << *citr << endl;
    // Doing *citr = 3; will give error
}
                    [convenient access of const_iterators]
for_each (myset.cbegin(), myset.cend(), MyFunction);
[Algorithm function of for_each]
```

Will call MyFunction on each element in myset [cannot modify elements in myset]

all containers can be passed to additional function to access the const_iterator

```
// Iterator Functions:
advance(itr, 5);  // Move itr forward by 5 spots
                  For random access iterator, equivalent to
                  itr += 5;
distance(itr1, itr2);  // Measure distance between itr1 and itr2
                       [Convenient for non-random access iterators]
```

③ Iterator Adaptor (Predefined Iterator)

——— A special, more powerful iterator

1) Insert Iterator

```
vector<int> vec1 = {4,5};
vector<int> vec2 = {12, 14, 16, 18};
vector<int>:: iterator it = find(vec2.begin(), vec2.end(), 16);
insert_iterator< vector<int> > i_itr(vec2, it);
copy(vec1.begin(), vec1.end(),  // source
                   i_itr);      // destination
```

// vec2: {12, 14, 4, 5, 16, 18}

Other insert iterators: back_insert_iterator, front_insert_iterator
{inserts at the back} {inserts at the front}

2) Stream Iterator

```
vector<string> vec4;
copy(istream_iterator<string>(cin), istream_iterator<string>(),
     back_inserter(vec4));
copy(vec4.begin(), vec4.end(), ostream_iterator<string>(cout, " "));
```

{iterate through the data that comes in from standard input} {default constructor represents end of the string}
{returns a back_insert_iterator}
{All elements of vec4 are printed in standard out and separated by space}

Make it terse:

```
copy(istream_iterator<string>(cin), istream_iterator<string>(),
     ostream_iterator<string>(cout, " "));
```

3) Reverse Iterator

```
vector<int> vec = {4, 5, 6, 7};
reverse_iterator<vector<int>::iterator> ritr;
for(ritr = vec.rbegin(); ritr != vec.rend(); ritr++)
    cout << *ritr << " ";   // prints: 7 6 5 4
```

{points to last element in the container}
{points before first element in the container}

4) Move Iterator

④ Algorithms

mostly for loops

"Whenever you see a for loop or while loop in your code, you should seriously consider replacing them with a function call from algorithm."

```
vector<int> vec = {4, 2, 5, 1, 3, 9};
vector<int>::iterator itr = min_element(vec.begin(), vec.end());
                                        // itr => 1

// Note 1: Algorithms always process ranges in a half-open way:
//         [begin, end)
sort(vec.begin(), itr);  // vec: {2, 4, 5, 1, 3, 9}
                         // { 1 is NOT included }
reverse(itr, vec.end()); // vec: {2, 4, 5, 9, 3, 1}; itr => 9

// Note 2:
vector<int> vec2(3);
copy(itr, vec.end(),     // Source
     vec2.begin());      // Destination

// vec2 needs to have
// atleast space for 3
// elements
// otherwise result is undefined behaviour
```

{First range represented by 2 iterators}

{Second range is represented by 1 iterator. The end of the range is inferred base on the size of the first range}

[Sacrificing safety in favour of efficiency and flexibility]

```
// Note 3: To overcome above safety issue (Not efficient)
vector<int> vec3;
copy(itr, vec.end(), back_inserter(vec3); // Inserting instead of overwriting
                                          // Not efficient
vec3.insert(vec3.end(), itr, vec.end()); // Efficient and safe
```

{As it only inserts once}

[Demonstrates that STL often provides many ways of doing the same thing and often times there's only one best way to do it]

```
// Note 4: Algorithm works very well with functions
bool isOdd(int i) {
    return i%2;
}
int main() {
    vector<int> vec = {2,4,5,9,2};
    vector<int>::iterator itr = find_if(vec.begin(),
                                vec.end(), isOdd); // itr -> 5

// Note 5: Algorithm with native C++ array
int arr[4] = {6,3,7,4};
sort(arr, arr+4);          // Pointer can be thought of as
                           // an iterator
```

VII. STL #6: Functors

{Function objects}

① Example:

```
class X {
    public:
    void operator()(string str) {
        cout << "Calling functor X with parameter " << str << endl;
    }
};

int main() {
    X foo;
    foo("Hi");
}
```

Don't mess this syntax with type conversion function: operator string() const { return "x"; } // type conversion function {you put type after operator}

for functor, you put type before operator

{foo is actually an instance of X, but one can use foo as if it is a function}

// Output: Calling functor X with parameter Hi

"Idea of functors: Anything that behaves as a function is a function"

Benefits of functor:
1. Smart function: capabilities beyond operator ()
2. It can remember state
3. It can have its own type

(2) Parameterized Function

```
class X{
  public:
  X (int i) {}
  void operator() (string str) {
     cout << "Calling functor X with parameter " << str << endl;
  }
};

int main()
{
   X (8) ("Hi");
}
```

Why do we want something like this?
Example:

```
int val = 2;
void addVal (int i) {
   cout << i+val << endl;
}
int main () {
   vector <int> vec = {2, 3, 4, 5};
   for_each(vec.begin(), vec.end(), addVal); // vec: {4, 5, 6, 7}
}
```

*Using global variables is a nasty coding practice

convenience symbol
} : line inside int main()

Another solution: Define a template

```
template <int val>
void addVal (int i){
    cout << val+i << endl;
}
} for_each(vec.begin(), vec.end(), addVal<2>);
```

→ Still a problem: A temporary variable is resolved at compile time, so it has to be a compile-time constant. So this won't work:

```
} int x=2;
} for_each(vec.begin(), vec.end(), addVal<x>);
```

Best solution: Using functor

```
class AddValue {
  int val;
  public:
  AddValue (int j): val(j) { }
  void operator() (int j) {
      cout << j+val << endl;
  }
};
} for_each(vec.begin(), vec.end(), AddValue(x));
```

③ Build-in Functors

less	greater	greater_than	less_than
not_equal_to		logical_and	logical_not
logical_or		multiplies	minus
plus	divide	modulus	negate

```
int x= multiplies<int>()(3,4);          // x = 3*4

if (not_equal_to<int>()(x,10))          // if (x != 10)
    cout << x << endl;
```

These are convenient for the algorithm functions

④ Parameter Binding

```
set<int> myset = {2,3,4,5};
vector<int> vec;

// Multiply myset's elements by 10 and save in vec:
transform(myset.begin(), myset.end(),   // source
          back_inserter(vec),           // destination
          bind(multiplies<int>(), placeholders::_1, 10));
                                        // vec: {20,30,40,50}
```

We've a problem { needs a function that takes only one parameter } and { takes 2 parameters }

Soln: We use the bind function →
With placeholders::_1, the first element of multiplies will be substituted with an element from myset

With the bind function, we've another soln to the addVal example in the beginning:

```
void addVal(int i, int val){
    cout << i+val << endl;
}
;  for_each(vec.begin(), vec.end(), bind(addVal,
                placeholders::_1, 2));
```

(5) Convert a regular function into a functor

```
double Pow (double x, double y){
    return pow(x,y);
}
int main()
{
    set<int> myset = {11, 3, 25, 7, 12};
    deque<int> d;
    auto f = function<double (double, double)> (Pow);
    transform(myset.begin(), myset.end(),
              back_inserter(d),
              bind(f, placeholders::_1, 2));
                                    // d: {9, 49, 121, 144, 625}
}
```

(even Pow works here, both without brackets.)

(6) Task: when (x>20) || (x<5), generate need-to-copy array from myset to d

```
set<int> myset = {11, 3, 25, 7, 12};
deque<int> d;
```

(This can also be achieved. Simply replace "transform" with "copy_if" everywhere)

M-1

```
transform(myset.begin(), myset.end(),
          back_inserter(d),
          bind(logical_or<bool>(),
               bind(greater<int>(), placeholders::_1, 20),
               bind(less<int>(), placeholders::_1, 5))
          );
```

M-2

```
bool needCopy(int x){
    return (x>20) || (x<5);
}
transform(myset.begin(), myset.end(), back_inserter(d), needCopy);
```

M-3: lambda function

```
transform(myset.begin(), myset.end(),
          back_inserter(d),
          [](int x){return (x>20)||(x<5);}
          );
```

(7) Why do we need functor in STL?

```
set<int> myset = {11, 3, 25, 7, 12};
// same as:
set<int, less<int>> myset = {11, 3, 25, 7, 12};
```

What if want to sort it by lsb_less criterion?
i.e. myset: {11, 12, 3, 25, 7}

M-1: ✗ Will NOT compile

```
bool lsb_less(int x, int y){
    return (x%10) < (y%10);
}
set<int, lsb_less> myset = {11, 3, 25, 7, 12};
```

↓ requires a function type as the 2nd parameter, not just a function or function pointer

So, we need to define a functor.

M-2: ✓

```
class Lsb_less{
  public:
    bool operator()(int x, int y){
        return (x%10) < (y%10);
    }
};
set<int, Lsb_less> myset = {11, 3, 25, 7, 12};
```

⑧ Predicate

- A functor or function that:
  1. Returns a boolean
  2. Does not modify data

```
class NeedCopy {
public:
    bool operator() (int x) {
        return (x>20) || (x<5);
    }
};
transform(myset.begin(), myset.end(),
          back_inserter(d),
          NeedCopy()
          );
```

Predicate is widely used in STL algorithms, it's mainly used for comparison or condition check