STL Collection Types

- **Template based set of collection classes**
- **STL collection types (container types)**
  - Sequences
    - vector - collection of elements of type T
    - list - doubly linked list, only sequential access
    - deque - fast insert at either end
  - Associative containers
    - map
    - set

Template

- Skeleton class for multiple types

```
template < class T >
class Value
{
        T _value;
        public:
        Value ( T value ) { _value = value; }

        T getValue ();

        void setValue ( T value );
};

template < class T >
T Value<T>::getValue () { return _value; }

template < class T >
void Value<T>::setValue ( T value ) { _value = value; }

Value<float> values[10];  // array of values of type float
```

Iterators

- **Allows access of elements in a collection**
  - forward iterator
  - bi-directional iterator
  - random access iterator
    - bi-directional + constant time access

STL Collections Methods

- **General class methods**
  - empty – True if collection is empty
  - size – number of elements in the collection
  - begin – start of collection in forward iterator
  - end – one past the last in forward iterator
  - rbegin – last of collection in backward iterator
  - rend – one before the start in backward iterator
  - clear – erases all elements
  - erase – erase an element or range of collection
  - Insert – insert an element

Add and Remove

- **vector, deque, list**
  - front - get a reference to first element
  - back - get a reference to last element
  - push_front - add to the start (NOT for vector)
  - push_back - add to the end
  - pop_front - remove from the start (NOT for vector)
  - pop_back - remove from the end

Iterators

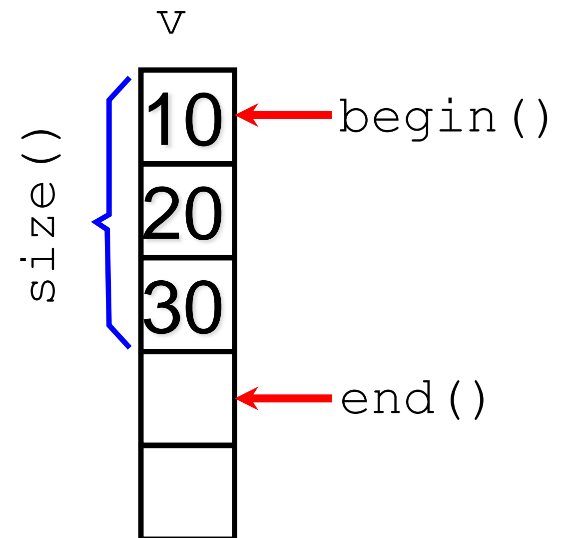
```
vector<int> v;

// input from console
while (cin >> input)
    v.push_back(input);

// sort elements in vector
sort(v.begin(), v.end());

// print out - version 1
for (int i=0; i<v.size(), i++)
    cout << v[i] << "\n";

// print out - version 2
vector<int>::iterator ii;
for (ii=v.begin(); ii!=v.end(); ii++)
    cout << *ii << "\n";
```



Operator [ ]

- **Vector, deque, map can use [ ] to access element**
  - Similar semantics in C array element access
  - Not available for list
- **at - similar to operator [ ]**
  - `v.at(index)` vs. `v[index]`
  - Available for vector, deque
  - Perform bound checking unlike operator [ ]

Vector

- **Automatically take care of resizing**
- **Constructor**
  - vector<T> vec;   // empty vector
  - vector<T> vec(10);   // capacity of 10 elements
  - vector<T> vec(10, 1); // and initialized with 1
- **Add / remove elements**
  - push_back(e), pop_back()
  - insert(pos, e), erase(pos)

Vector

- capacity - allocate space of vector

```
#include <iostream>
#include <vector>
using namespace std;

main()
{
   vector<int> iv1;
   vector<int> iv2(10);

   cout << "iv1: capacity = " << iv1.capacity() << "\n";
   cout << "iv2: capacity = " << iv2.capacity() << "\n";

   iv1.push_back(10); // can increase size and capacity
   iv2[0] = 10;
   iv2.push_back(20);  // iv2[1] or iv2[10]
   iv1[1] = 20;        // segmentation fault
}
```

List

- **Doubly linked list**
- **Constructor**
  - `list<T> li;            // empty list (size = 0)`
  - `list<T> li(10);        // list with size = 10`
  - `list<T> li(10, 1);     // and initialized with 1`

- **Add / remove elements**
  - push_front(e), push_back(e), insert(pos, e)
  - pop_front(), pop_back(), erase(pos)

List

- **Test if list is empty**
  - // std::list<int> il(100);
  - il.size() takes O(n)
  - il.empty() takes O(1)   // why?

- **Useful algorithms for list**
  - sort() on list - uses O(nlogn) algorithm
  - reverse() on list - reverse the order
  - unique() on list - remove repeating elements

List

- **insert (*pos*, *e*) : insert *e* before position *pos***

```
#include <iostream>
#include <list>
using namespace std;

main()
{
    list<int> L;

    L.push_back(0);
    L.push_back(0);
    L.insert(++L.begin(), 2);
    L.push_back(5);
    L.push_back(6);

    list<int>::iterator i;
    for (i=L.begin(); i!=L.end(); i++) cout << *i << " ";
    cout << endl;
}
```

Set

- **Sorted Associative Container**
  - Sorted at insert by using default compare-operator
- **Unique**
  - No two elements are same
- **Fast add & remove**
  - Add/remove a sorted range in linear time

Set

- **Sorted output with iterator**

```
#include <iostream>
#include <set>
using namespace std;

int main(){
    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);

    set<int>::const_iterator it;
    it = myset.begin();
    while (it != myset.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;
}
```

Map

- **Associative container**
  - <key, value> pair
  - Unique element (key)
  - Sorted on key value

Map

- **Associative key can be any type**

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main(){
    map<string, double> market;
    market["apple"]= 2.30;
    market["orange"] = 1.20;
    market["melon"] = 3.30;

    map<string, double>::const_iterator it;
    it = market.begin();
    while (it != market.end()) {
        cout << it->first << ": " << it->second << endl;
        ++it;
    }
    cout << endl;
}
```

[Lab – Practice #1]

- **Implement contact list**
  - Use the STL map
  - Contact list has four functions
    - Show all contacts in ascending order
    - Find a contact by name
    - Add a contact
    - Delete a contact by name
  - Input :
    - show / find (*name*) / add (*name*) (*number*) / delete (*name*)

[Lab – Practice #1]

```
$ ./contacts
input: add mike 010-1111-2222

input: add john 010-2222-3333

input: show
john 010-2222-3333
mike 010-1111-2222

input: find john
010-2222-3333

input: add john 010-4444-2222
john is already in contact list

input: delete mike

input: show
john 010-2222-3333
```