Collections of useful classes for common data structures
Ability to store objects of any type (template)
Containers form the basis for treatment of data structures
Container – class that stores a collection of data
STL consists of 10 container classes:
STL Containers

- **Sequence Container**
  - Stores data by position in linear order:
  - First element, second element, etc.
  - All containers
    - Use same names for common operations
    - Have specific operations

- **Associate Container**
  - Stores elements by key, such as name, ID number or part number
  - Access an element by its key which may bear no relationship to the location of the element in the container

- **Adapter Container**
  - Contains another container as its underlying storage structure
STL Containers

- Sequence Container
  - Vector
  - Deque
  - List
- Adapter Containers
  - Stack
  - Queue
  - Priority queue
- Associative Container
  - Set, multiset
  - Map, multimap
Iterator is an object that can access a collection of like objects one object at a time.

An iterator can traverse the collection of objects.

Each container class in STL has a corresponding iterator that functions appropriately for the container.

For example: an iterator in a vector class allows random access.

An iterator in a list class would not allow random access (list requires sequential access).
Common Iterator Operations

- *  Return the item that the iterator currently references
- ++  Move the iterator to the next item in the list
- –   Move the iterator to the previous item in the list
- ==  Compare two iterators for equality
- !=  Compare two iterators for inequality
Vector Container

- Generalized array that stores a collection of elements of the same data type
- Vector – similar to an array
  - Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector
- Vector vs array
  - Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence
Example:

```cpp
#include <vector>

vector<int> scores (100);  // 100 integer scores
vector<Passenger> passengerList(20);  // list of 20 passengers
```
Vector Container

- Allows direct access to the elements via an index operator
- Indices for the vector elements are in the range from 0 to size() -1
- Example:

```cpp
#include <vector>
vector <int> v(20);
v[5]=15;
```
// constructing vectors
#include <iostream>
#include <vector>
using namespace std;

int main ()
{

    // constructors used in the same order as described above:
    vector<int> first;               // empty vector of ints
    vector<int> second (4,100);      // four ints with value 100
    vector<int> third (second.begin(),second.end()); // iterating through second
    vector<int> fourth (third);      // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    cout << "The contents of fifth are:";
    for (vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
    {
        cout << ' ' << *it;
    }
    cout << '
';

    return 0;
}
List Container

- Stores elements by position
- Each item in the list has both a value and a memory address (pointer) that identifies the next item in the sequence
- To access a specific data value in the list, one must start at the first position (front) and follow the pointers from element to element until data item is located.
- List is not a direct access structure
- Advantage: ability to add and remove items efficiently at any position in the sequence
**Constructors and assignment**
- `list <T> v;`
- `list <T> v(aList);`
- `ll=aList;`

**Access**
- `l.front()` returns the first element in the list
- `l.back()` returns the last element in the list
Insert and Remove

- `l.push_front(value)`
- `l.push_back(value)`

Iterator Declaration

- `list<T>::iterator itr;`

Iterator Options

- `itr = l.begin()` set iterator to beginning of the list
- `itr = l.end()` set iterator to after the end of the list
#include <iostream>
#include <list>
using namespace std;

// Simple example uses type int

int main()
{
    list<int> L;
    L.push_back(0); // Insert a new element at the end
    L.push_front(0); // Insert a new element at the beginning
    L.insert(++L.begin(),2); // Insert "2" before position of first argument
                             // (Place before second argument)
    L.push_back(5);
    L.push_back(6);

    list<int>::iterator i;

    for(i=L.begin(); i != L.end(); ++i) cout << *i << " ";
    cout << endl;
    return 0;
}
Stack Container

- Adapter Container
- These containers restrict how elements enter and leave a sequence
- Stack
  - allows access at only one end of the sequence (top)
  - Adds objects to container by pushing the object onto the stack
  - Removes objects from container by popping the stack
  - LIFO ordering (last end, first out)
// stack::push/pop
#include <iostream>  // cout
#include <stack>     // stack
using namespace std;

int main ()
{
    stack<int> mystack;

    for (int i=0; i<5; ++i) mystack.push(i);

    cout << "Popping out elements...";
    while (!mystack.empty())
    {
        cout << ' ' << mystack.top();
        mystack.pop();
    }
    cout << '\n';

    return 0;
}
Queue Container

Queue
- Allows access only at the front and rear of the sequence
- Items enter at the rear and exit from the front
- Example: waiting line at a grocery store
- FIFO ordering (first-in first-out)
- push (add object to a queue)
- pop (remove object from queue)
#include <iostream>    // cin, cout
#include <queue>        // queue
using namespace std;

int main ()
{
    queue<int> myqueue;
    int myint;

    cout << "Please enter some integers (enter 0 to end):
";

    do {
        cin >> myint;
        myqueue.push (myint);
    } while (myint);

    cout << "myqueue contains: ";
    while (!myqueue.empty())
    {
        cout << ' ' << myqueue.front();
        myqueue.pop();
    }
    cout << '
';

    return 0;
}
Priority queue

- Operations are similar to those of a stack or queue
- Elements can enter the priority queue in any order
- Once in the container, a delete operation removes the largest (or smallest) value
Set Container

Set

Collection of unique values, called keys or set members
Contains operations that allow a programmer to:
- determine whether an item is a member of the set
- insert and delete items very efficiently
```cpp
#include <iostream>
#include <set>
using namespace std;

int main ()
{
    set<string> s;
    cout << "Adding 'Hello' and 'World' to the set twice" << endl;

    s.insert("Hello");
    s.insert("World");
    s.insert("Hello");
    s.insert("World");

    cout << "Set contains:";
    while (!s.empty()) {
        cout << ' ' << *s.begin();
        s.erase(s.begin());
    }

    return 0;
}
```
A multi-set is similar to a set, but the same value can be in the set more than once

Multi-set container allows duplicates
Map Container

- Implements a key-value relationship
- Implements Programmer can use a key to access corresponding values
- Example: key could be a part number such as X89-21 that corresponds to a part: 912 price and MarsOto manufacturer

<table>
<thead>
<tr>
<th>Key</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X45-84</td>
<td>1579</td>
<td>CanOto</td>
</tr>
<tr>
<td>X33-41</td>
<td>589</td>
<td>MilOto</td>
</tr>
<tr>
<td>X89-21</td>
<td>912</td>
<td>MarsOto</td>
</tr>
</tbody>
</table>
Multi-map Container

- Similar to a map container
- Multi-map container allows duplicates
Writing classes that work with the STL

- Classes that will be stored in STL containers should explicitly define the following:
  - Default constructor
  - Copy constructor
  - Destructor
  - operator=
  - operator==
  - operator<

- Not all of these are always necessary, but it might be easier to define them than to figure out which ones you actually need.

- Many STL programming errors can be traced to omitting or improperly defining these methods.