# STL

http://www.cs.brown.edu/~jak/proglang/cpp/stltut/tut.html
http://www.cplusplus.com/reference/stl/
http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html

# STL Components

- Container Classes
    - Storage for your data
    - stack, queue, vector, map ...
- Iterator Classes
    - Algorithm Classes
    - sort, find, binary search, next_permutation ...
- Utilities

# STL Containers

- Sequences
    - vector, list, deque ...
- Associative Containers
    - set, map ...
- Container Adaptors
    - Based on deque by default
    - stack, queue, priority queue ...

# vector

- random access ([] operator) like an array
- add / remove an element in constant time at the first or last of vector
- add / remove an element in linear time in the middle of vector
- automatic memory management
- needn’t specify quantity of elements

# vector

```
#include <vector>
```

---

```
void vectest()
{
  vector<int> v;
  v.push_back(10);
  v.push_back(20);
  print (v);
  cout << v.size() << ' ' << v.capacity() << endl;
  v.pop_back();
  print(v);
  cout << v.size() << ' ' << v.capacity() << endl;
}
```

```
10,20,2 2
10,1 2
```

# print - traditional

```
void print(vector<int> v)
{
  for (unsigned int i = 0; i < v.size(); i++)
  {
    cout << v[i] << ",";
  }
}
```

# print - iterators

```
void print(vector<int> v)
{
  for (vector<int>::iterator i = v.begin(); i != v.end(); i++)
  {
    cout << *i << ",";
  }
}
```

- Will only print int vectors

# print - generic

```
template <typename T>
void print(vector<T> v)
{
  typename std::vector<T>::iterator iter;

  for (iter = v.begin(); iter != v.end(); iter++)
  {
    cout << *iter << ",";
  }
}
```

- Will print a vector of any type

# print - foreach

```
#include <boost/foreach.hpp>
#define foreach BOOST_FOREACH
//...

template <typename T>
void print(vector<T> v)
{
  foreach(T i, v)
  {
    cout << i << ",";
  }
}
```

# print - foreach (iterator reference)

```
#include <boost/foreach.hpp>
#define foreach BOOST_FOREACH
//...

template <typename T>
void print(vector<T> v)
{
  foreach(T&i, v)
  {
    cout << i << ",";
  }
}
```

# copy algorithm

```
template <typename T>
void print(vector<T> v)
{
  copy (v.begin(), v.end(), ostream_iterator<T> (cout, "\n"));
}
```

# list

- double-linked list
    - predecessor, successor
- add / remove an element in linear time
- alternative choice, single-linked list
    - slist

# list

```
#include <list>
```

```
template <typename T>
void print(list<T> v)
{
  foreach( T&i, v )
  {
    cout << i << ",";
  }
}
```

```
void listtest()
{
  list<int> la, lb;

  la.push_back(2), la.push_back(1), la.push_back(5);

  lb.push_back(4), lb.push_back(3);

  la.sort();
  lb.sort();
  la.merge(lb);
  print(la);
}
```

```
1, 2, 3, 4, 5,
```

# deque

- Double-Ended Queue
  - usually pronounced [deck] instead of [de-cue]
- random access
  - slower than vector
- add / remove an element in constant time from the first or last position of a deque
- add / remove an element in amortized constant time in the middle of deque

# deque (continued)

- vector is more efficient than deque
    - to sort a deque...
    - copy a deque to vector, sort the vector, and then copy the vector back to the deque

# deque

```
#include <deque>
```

```
template <typename T>
void print(deque<T> v)
{
  foreach( T&i, v )
  {
    cout << i << ",";
  }
}
```

```
void dequetest()
{
  deque<int> dq;
  dq.push_back(3);
  dq.push_front(1);
  dq.insert(dq.begin()+1, 2);
  dq[2] = 0;
  print(dq.begin(), dq.end());
}
```

```
1, 2, 0,
```

# Set

- Sorted Associative Container
    - Compare function
- Unique
    - There are no two same elements.
- add / remove a sorted range in linear time

# set

```
#include <set>
```

```
struct ltstr
{
  bool operator() (const char* s1, const char* s2) const
  {
    return (strcmp(s1, s2) < 0);
  }
};

void settest()
{
  const char* str[6] = {"isomer", "ephemeral", "prosaic",
                        "nugatory", "artichoke", "serif"};

  set<const char*, ltstr> s;
  for (int i=0; i<6; i++)
  {
    s.insert(str[i]);
  }

  print(s);
}
```

```
artichoke, ephemeral, isomer, nugatory, prosaic, serif,
```

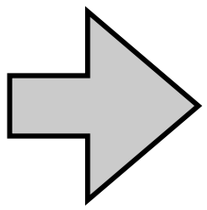
# set foreach

```
template <typename T, class Comp>
void print(std::set<T, Comp> s)
{
  foreach( T i, s )
  {
    cout << i << ",";
  }
}
```

# Set initialisation

```
const char* str[6] = {"isomer", "ephemeral", "prosaic",
                      "nugatory", "artichoke", "serif"};
```

```
for (int i=0; i<6; i++)
{
  s.insert(str[i]);
}
```



```
set<const char*, ltstr> s(str, str + 6);
```

# map

- Sorted Associative Container
- Pair Associative Container
- Unique Associative Container
- Dictionary

# map

```
#include <map>
```

```
typedef map<string, int> MyMap;

void print(MyMap m)
{
  foreach (MyMap::value_type &i, m)
  {
    cout << i.second << endl;
  }
}
```

```
void maptest()
{
  MyMap grade;
  grade["Mark"] = 95;
  grade["Edward"] = 87;
  grade["Louise"] = 66;
  grade["Allen"] = 76;
  print(grade);
  cout << grade["Allen"] << endl;
}
```

```
76
87
66
95
76
```

# stack

- last in first out, LIFO
- unable to access elements except the top of stack
    - you cannot traverse a stack
- implementation based-on deque by default

# stack

```
#include <stack>
```

```
void stacktest()
{
  stack<int> s;
  s.push(8);
  s.push(5);
  s.push(6);
  cout << s.top() << endl;
  s.pop();
  cout << s.top() << endl;
}
```

```
6
5
```

# queue

- fast in first out, FIFO
- unable to access elements except the top of queue
    - you cannot traverse a queue
- implementation based-on deque by default

# queue

```
#include <queue>
```

```
void queuetest()
{
  queue<int> q;
  q.push(8);
  q.push(5);
  q.push(6);
  cout << q.front() << endl;
  q.pop();
  cout << q.front() << endl;
}
```

```
8
5
```

# Priority Queue

- fast in first out, FIFO
    - top is always the biggest in the queue
- unable to access elements except the top of queue
    - you cannot traverse a queue
- implementation based-on deque by default

# priority_queue

```
#include <queue>
```

```
void prioritytest()
{
  priority_queue <int, vector<int>, greater<int> > pq;

  pq.push(2);
  pq.push(5);
  pq.push(3);
  pq.push(1);
  cout<<"pq contains " << pq.size() << " elements.\n";

  while (!pq.empty())
  {
    cout << pq.top() << endl;
    pq.pop();
  }
}
```

```
pq contains 4 elements.
1
2
3
5
```

# STL Iterator

- Every STL container holds a nested iterator class

# iterators

```
void iteratortest()
{
  vector<int> v;
  int ary[6] = {1, 4, 2, 8, 5, 7};
  for (int i = 0; i < 6; v.push_back(ary[i++]));

  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
  {
    cout << (*it) << " ";
  }
  cout << endl;
  copy (v.begin(), v.end(), ostream_iterator<int>(cout, " "));
}
```

```
1 4 2 8 5 7
1 4 2 8 5 7
```

# STL Algorithms

- Linear Search
    - find,find_if,find_first_of...
- Subsequence Matching
    - search, find_end ...
- Counting Elements
    - count,count_if
- for_each
- Comparing Two Ranges
    - equal, mismatch ...

# STL Algorithms

- Copy Ranges
    - copy,copy_backward
- Swapping Elements
    - swap, swap_ranges ...
- Replacing Elements
    - replace, replace_if, replace_copy ...
- Permuting Elements
    - reverse,rotate,next_permutation.

# STL Algorithms

- Random Shuffling and Sampling
  - random_shuffle ...
- Generalized Numeric Algorithms
  - inner_product, adjacent_difference ...
- Sorting Ranges
  - sort,stable_sort,is_sorted,nth_element...
- Operation in Sorted Ranges
  - binary_search,merge,equal_range...

# print algorithm - sequences

```
template <typename T>
void print(vector<T> v)
{
  foreach(T i, v)
  {
    cout << i << ",";
  }
}
```

```
template <typename T>
void print(list<T> v)
{
  foreach( T&i, v )
  {
    cout << i << ",";
  }
}
```

```
template <typename T, class Comp>
void print(std::set<T, Comp> s)
{
  foreach( T i, s )
  {
    cout << i << ",";
  }
}
```

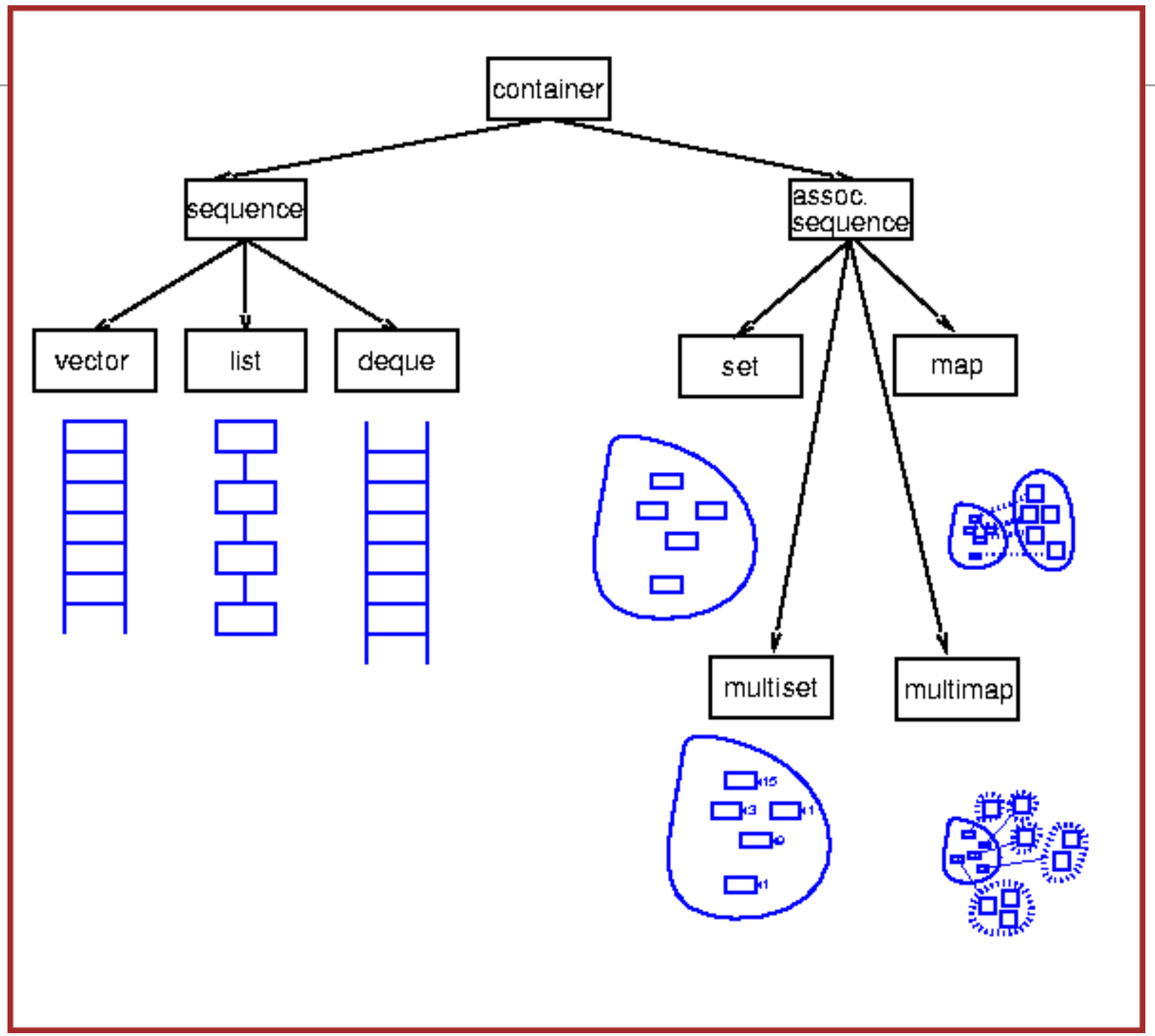
```
template <typename Iterator>
void print (Iterator start, Iterator  end)
{
  while (start != end)
  {
    cout << *start << endl;
    start++;
  }
}
```

```
print (v.begin(), v.end());
```

```
print(la.begin(), la.end());
```

# print algorithm - associative containers

```
template <typename Iterator>
void print (Iterator start, Iterator  end)
{
  while (start != end)
  {
    cout << start->second << endl;
    start++;
  }
}
```


container
sequence
assoc. sequence
vector
list
deque
set
map
multiset
multimap

## Container class templates

**Sequence containers**:

| | |
|---|---|
| **vector** | Vector (class template ) |
| **deque** | Double ended queue (class template ) |
| **list** | List (class template ) |

**Container adaptors**:

| | |
|---|---|
| **stack** | LIFO stack (class template ) |
| **queue** | FIFO queue (class template ) |
| **priority_queue** | Priority queue (class template) |

**Associative containers**:

| | |
|---|---|
| **set** | Set (class template ) |
| **multiset** | Multiple-key set (class template) |
| **map** | Map (class template ) |
| **multimap** | Multiple-key map (class template ) |
| **bitset** | Bitset (class template) |

| | | | Sequence containers | | | Associative containers | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Headers | | | <vector> | <deque> | <list> | <set> | | <map> | | <bitset> |
| Members | | complex | vector | deque | list | set | multiset | map | multimap | bitset |
| | *constructor* | * | constructor | constructor | constructor | constructor | constructor | constructor | constructor | constructor |
| | *destructor* | O(n) | destructor | destructor | destructor | destructor | destructor | destructor | destructor | |
| | operator= | O(n) | operator= | operator= | operator= | operator= | operator= | operator= | operator= | operators |
| iterators | begin | O(1) | begin | begin | begin | begin | begin | begin | begin | |
| | end | O(1) | end | end | end | end | end | end | end | |
| | rbegin | O(1) | rbegin | rbegin | rbegin | rbegin | rbegin | rbegin | rbegin | |
| | rend | O(1) | rend | rend | rend | rend | rend | rend | rend | |
| capacity | size | * | size | size | size | size | size | size | size | size |
| | max_size | * | max_size | max_size | max_size | max_size | max_size | max_size | max_size | |
| | empty | O(1) | empty | empty | empty | empty | empty | empty | empty | |
| | resize | O(n) | resize | resize | resize | | | | | |
| element access | front | O(1) | front | front | front | | | | | |
| | back | O(1) | back | back | back | | | | | |
| | operator[] | * | operator[] | operator[] | | | | operator[] | | operator[] |
| | at | O(1) | at | at | | | | | | |
| modifiers | assign | O(n) | assign | assign | assign | | | | | |
| | insert | * | insert | insert | insert | insert | insert | insert | insert | |
| | erase | * | erase | erase | erase | erase | erase | erase | erase | |
| | swap | O(1) | swap | swap | swap | swap | swap | swap | swap | |
| | clear | O(n) | clear | clear | clear | clear | clear | clear | clear | |
| | push_front | O(1) | | push_front | push_front | | | | | |
| | pop_front | O(1) | | pop_front | pop_front | | | | | |
| | push_back | O(1) | push_back | push_back | push_back | | | | | |
| | pop_back | O(1) | pop_back | pop_back | pop_back | | | | | |
| observers | key_comp | O(1) | | | | key_comp | key_comp | key_comp | key_comp | |
| | value_comp | O(1) | | | | value_comp | value_comp | value_comp | value_comp | |
| operations | find | O(log n) | | | | find | find | find | find | |
| | count | O(log n) | | | | count | count | count | count | count |
| | lower_bound | O(log n) | | | | lower_bound | lower_bound | lower_bound | lower_bound | |
| | upper_bound | O(log n) | | | | upper_bound | upper_bound | upper_bound | upper_bound | |
| | equal_range | O(log n) | | | | equal_range | equal_range | equal_range | equal_range | |
| *unique members* | | | capacity<br>reserve | | splice<br>remove<br>remove_if<br>unique<br>merge<br>sort<br>reverse | | | | | set<br>reset<br>flip<br>to_ulong<br>to_string<br>test<br>any<br>none |

Amortized complexity shown. Legend: O(1) constant < O(log n) logarithmic < O(n) linear; *=depends on container

**Container adaptors**:

| | | | Container Adaptors | | |
|---|---|---|---|---|---|
| Headers | | | `<stack>` | `<queue>` | |
| Members | | | **stack** | **queue** | **priority_queue** |
| | *constructor* | * | constructor | constructor | constructor |
| capacity | `size` | O(1) | size | size | size |
| | `empty` | O(1) | empty | empty | empty |
| element access | `front` | O(1) | | front | |
| | `back` | O(1) | | back | |
| | `top` | O(1) | top | | top |
| modifiers | `push` | O(1) | push | push | push |
| | `pop` | O(1) | pop | pop | pop |

STL Generic Algorithms on Forward Containers