Lecture 23

- Sequence containers
  - list<T>
- Associative containers
  - set<T>
  - Implementation using search trees

list<T>

- #include <list>
- list<T> is a **doubly linked** list
- Allows **forward/backward** traversal
- If backward traversal is not needed, use **slist<T>**
- **Constant time** for insertion/removal of elements anywhere
- Inserting/deleting elements does **not invalidate** iterators

list<T> Example

```
#include <list>
#include <iostream>

using namespace std;

int main()
{
  list<int> l;

  l.push_back(0);
  l.push_front(1);
  l.insert(l.begin(), 2); // same as l.push_front(2)
  // l now 2 1 0

  list<int> x(3, 10); // x is list of 3 tens
  l.splice(++l.begin(), x);  // x now empty

  list<int>::iterator it, en;
  it = l.begin(); en = l.end();
  for (; it != en; ++it) cout << *it << " ";
  // output: 2 10 10 10 1 0
}
```


Commonly used list Members

- iterator **begin**() : returns iterator to first element
- iterator **end**() : returns iterator to end (last element + 1)
- size_type **size**() : # of list elements
- bool **empty**() const : true iff list is empty
- void **push_front**(const T&) : inserts new element at the beginning
- void **push_back**(const T&) : inserts new element at the end
- void **pop_front**() : removes first element
- void **pop_back**() : removes last element
- iterator **insert**(iterator pos, const T&) : inserts element before pos
- void **erase**(iterator pos) : removes element at position pos
- void **reverse**() : reverses list (linear time!)
- void **splice**(iterator pos, list<T>& x) : inserts x before pos, clears x

Other Sequence Containers

- deque<T> ("deck")
  - #include <deque>
  - **double-ended** queue
  - inserting/deleting at both ends takes **amortized constant time**
  - inserting/deleting in the middle: linear time
  - supports **random access,** d[i]
- basic_string<T>
  - #include <string>
  - sequence of characters, string = basic_string<char>
  - similar to vector
  - **many member functions -** insert, append, erase, find, replace...
- for details refer to **www.sgi.com/tech/stl**

Associative Containers

- Support efficient retrieval of elements based on **keys**
- Support insertion/removal of elements
- Difference to sequences: no mechanism for inserting elements at specific locations
- Each element has a key that cannot be modified, thus ***it = x is not allowed** for iterator it.
- But data modification possible through iterator:
  - e.g. map<int,double>::iterator it; ... (*it).second = 3.0;

set<T [, Compare] >

- #include <set>
- Simple unique associative container
  - Keys are the elements themselves
  - No two elements are the same
- Internally, sets are represented as search trees
  - Elements are stored in nodes
  - implicitely sorted by **functor** less<T> (default) or Compare if provided. These classes define operator() with two arguments (const T& a, const T& b) and return true iff a is less than b
  - logarithmic time for find / insert / delete
- Inserting/deleting elements does not invalidate iterators

set Example

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

set<int> s;

s.insert(0); s.insert(2);
s.insert(1); s.insert(0); // populate s

set<int>::iterator it, end;
it = s.begin(); end = s.end();
for (; it != end; ++it) cout << (*it) << ' ';

if (s.find(0) != s.end()) cout << "foo";
// output: 0 1 2 foo
```

Comparison Functor for Associative Containers

Binary relation < must be a **strict weak ordering**, i.e.

< is a partial ordering:

irreflexivity:	for all x: x < x false (important!)
antisymmetry:	for all x,y: if x < y then !(y < x)
transitivity:	for all x,y,z: if x < y and y < z then x < z

and: **equivalence is transitive**
[for all x,y: x and y equivalent :<=> !(x<y) and !(y<x)]

(Total order: above + "equivalent = identical")

E.g.: < for int and string are total orders
=> no special comparison functor needed

Example: set with Functor

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

struct Foo { int x, y; };

struct CompFoo {
   // return true iff a < b (lexicographic order)
   bool operator()(Foo &a, Foo &b) {
      if (a.x < b.x) return true;
      if (a.x > b.x) return false;
      return a.y < b.y;
   }
};

set<Foo,CompFoo> foo_set; // set of Foos

somewhere in set<T,Comp> implementation:
    Comp f; ... if (f(a, b)) ... // a < b
```


Commonly used set operations

- iterator **begin**() : returns iterator to first element
- iterator **end**() : returns iterator to end (last element + 1)
- size_type **size**() : # of set elements
- bool **empty**() const : true iff set is empty
- pair<iterator, bool> **insert**(const T& x) :
  - inserts element; if new, returns (iterator,true) - otherwise (?,false)
  - e.g. pair<iterator, bool> p = s.insert(5); if (p.second) { // new ...
- void **erase**(iterator it) : removes element pointed to by pos
- void **clear**() : remove all elements
- iterator **find**(const T& x) const :
  - looks for x, returns its position if found, and end() otherwise
- **set_union**(), **set_intersection**(), **set_difference**()
  - set algorithms (see example ->)