# Lecture 24

- map<U,V>
- Iterators
- Algorithms
- STL tips



# map<Key,Data[,Compare]>

- #include <map>
- Sorted-pair-unique associative container
- Associates keys with data
- Value-type is **pair<const Key, Data>**
- Insert/delete operations **do not invalidate** iterators



# map Example

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

typedef map<string, int> Month2Days;
Month2Days m2d;

m2d["january"]   = 31; m2d["february"] = 28;
m2d["march"]     = 31; m2d["april"]    = 30;
m2d["may"]       = 31; m2d["june"]     = 30;
m2d["july"]      = 31; m2d["august"]   = 31;
m2d["september"] = 30; m2d["october"]  = 31;
m2d["november"]  = 30; m2d["december"] = 31;

string m = "june";
Month2Days::iterator cur = m2d.find(m);
if (cur != m2d.end()) {
   cout << m << " has " << (*cur).second << " days" <<
endl;
} else
   cout << "unknown month: " << m << endl;
```

# Commonly used map members

- iterator **begin**() : returns iterator to first pair
- iterator **end**() : returns iterator to end (past last pair)
- size_type **size**() : # of pairs in map
- bool **empty**() const : true iff map is empty
- void **erase**(iterator pos) : removes pair at position pos
- pair<iterator, bool> insert(const Key&):
  - inserts key, returns iterator and true iff new
- void **clear**() : erase all pairs
- iterator **find**(const Key& k) :
  - looks for key k, returns its position if found, and end() otherwise
- Data& **operator[]**(const Key& k) :
  - returns the data associated with key k;
  - if it does not exists inserts default data value!

# Iterators

- Generalization of pointers
- Often used to iterate over ranges of objects
  - iterator points to object
  - the incremented iterator points to the next object
- Central to generic programming:
  - **interface** between containers and algorithms
  - algorithms take iterators as arguments
  - container only needs to provide a way to **access its elements** using iterators
  - allows to write **generic algorithms** operating on many different containers such as vector and list



# Iterator Concept Hierarchy

- **Input Iterator, Output Iterator**
  - permit single pass (like reading/writing file)
  - read or write access, resp. - writing to input iterators not supported, nor reading from output iterators
- **Forward Iterator**
  - can be used to step through a container several times (read or write)
  - only ++ supported (e.g. singly linked list)
- **Bidirectional Iterator**
  - motion in both directions (++ --, e.g. doubly linked list)
- **Random Access Iterator**
  - allows adding of offsets to interators [ e.g. *(it+5) ]



# Ranges

- Most algorithms are expressed in terms of iterator ranges **[begin, end)**
- Empty iff begin = end
- If **n** iterators in a range, then [begin, end) represents **n+1** locations. **Crucial!**
- E.g. linear search (find) must be able to return some value to indicate an unsuccessful search



# set Algorithm Example

```
string a[4] ={ "banana", "apple", "pear",
"orange" };
string b[4] ={ "green", "red", "orange", "blue" };

set<string> sa(a,a+4);     // creates set from a[]
set<string> sb(b,b+4), sc; // set from b[], result

set_intersection(sa.begin(), sa.end(),
                 sb.begin(), sb.end(),
                 inserter(sc, sc.begin()));

// computes intersection of sa and sb and stores
// result in sc:  "orange"

set<string>::iterator it, en;
it = sc.begin(); en = sc.end();
for (; it != en; ++it) cout << *it << endl;
```

# insert_iterator

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

list<int> l;

l.push_back(3);

insert_iterator<list<int> > ii(l, l.begin());
*ii = 0;  // insert 0 before l.begin()
*ii = 1;  // insert 1 before l.begin()
*ii = 2;  // insert 2 before l.begin()

// 2 1 0 3
```

- keeps track of the container c and insertion point p
- *ii = x performs action p = c.insert(p, x); ++p;



# reverse_iterator

- iterator adaptor that enables backwards traversal of a range using operator++

```
#include <iterator>

vector<int> v;
typedef vector<int>::reverse_iterator vrit;

v.push_back(1); v.push_back(2);

vrit rit = v.rbegin();
vrit rend = v.rend();

// traverse v backwards
while (rit != rend) { cout << *rit++ << endl; }

// 2 1
```



# Non-Mutating Algorithms:

Work on a range and do not change elements

- **for_each** - apply a function to each element
- **find** - find an element
- **equal** - checks whether two ranges are the same
- **count** - count elements equal to value
- **search** - search for a sub-sequence
- ...



# for_each

- template <class **InpIterator**, class **UnaryFunc**>
  **UnaryFunc for_each**(**InpIterator** begin,
  **InpIterator** end,
  **UnaryFunc** f)
- applies function or functor f to each element in [begin, end)
- returns the function object after it has been applied to all elements in [ begin, end )

# for_each example

```
#include <set>
#include <algorithm>

struct Add {
   int sum;

   Add() { sum = 0; }
   void operator()(int x) { sum += x; }
};


set<int> s;
s.insert(1); s.insert(2); s.insert(3);

Add f = for_each(s.begin(), s.end(), Add());

cout << f.sum << endl;     // 1+2+3 = 6
```



# for_each Implementation

```
template <typename InputIterator,
          typename Functor>

Functor for_each(InputIterator first,
                 InputIterator end,
                 Functor f)
{
   for (; first != end; ++first) f(*first);
   return f;
}
```



# Mutating Algorithms:

## Work on a range and possibly change elements

- **remove_if** : moves elements for which a predicate is false to front, returns new_end, size unchanged
- **partition** : reorders elements; x with pred(x)=true come first
- **generate** : assigns results of function calls to each element
- **copy** : copies input range to output iterator
- **fill** : assigns a value to each element
- **reverse** : reverses range
- **rotate** : general rotation of range w.r.t. to mid-point
- **random_shuffle** : randomly shuffles all elements
- ... many more



```
#include <algorithm>
struct Even { // functor
  bool operator()(int x) { return (x & 1) == 0; }
};
const int N = 20;
vector<int> v, w;  int a[N];

partition(v.begin(), v.end(), Even());//even | odd
generate(v.begin(), v.end(), rand);

copy(v.begin(), v.end(), w.begin()); // dangerous!
                        // w must be large enough
copy(v.begin(), v.end(), back_inserter(w));//better

fill(v.begin(), v.end(), 314159);

reverse(a, a+N); // array viewed as STL container
rotate(v.begin(), v.begin()+1, V.end()); // "<< 1"
random_shuffle(a, a+N);
```

## Sorting-Related Algorithms

- **sort** : sorts elements in ascending order
- **lower_bound**, **upper_bound** : find values in sorted ranges in logarithmic time
- **merge** : merge sorted ranges into one
- **includes** : check if one range is contained in another one
- **set_union**, **set_intersection**, **set_difference**, **set_symmetric_difference** : set operations
- ...



## Sorting

```
template <typename RandomAccessIterator>
sort(RandomAccessIterator first,
     RandomAccessIterator end);
// uses operator <

template <typename RandomAccessIterator,
          typename StrictWeakOrdering>
sort(RandomAccessIterator first,
     RandomAccessIterator end,
     StrictWeakOrdering less);
// uses comparison functor less
```

- Sorts random access range in ascending order
- Implements 'Introspection sort''which combines quicksort and heapsort
- Worst and average case complexity: O(n log n)
- Fast!



## Sort Examples

```
#include <algorithm>
#include <functional> // for less<T>, greater<T> ...
using namespace std;

vector<int> v(10);
const int N = 20;
int a[N];

generate(v.begin(), v.end(), rand);
generate(a, a+N, rand);

sort(v.begin(), v.end());  // asc., uses <(int,int)
sort(a, a+N, less<int>()); // ascending
sort(v.begin(), v.end(), greater<int>()); // desc.
```



## Where to go from here? STL resources on the web!

- Hashed associative containers
  - e.g. **hash_set**<T, HashFunc, EqualKey>
  - organized as hash tables
  - faster than the standard tree-based containers
  - but need more space
  - see www.sgi.com/tech/stl
- More sorting related functions (stable_sort, merge, ...)
- More C++ libraries at: www.boost.org