Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 23

- Associative Container map<U,V>
- Iterators

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>

typedef std::map<std::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;
```

Frequently 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 clear() : erase all pairs
void erase(iterator pos) : removes pair at position pos
pair<iterator, bool> insert(const Key&):
               inserts key, returns iterator and true iff new

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 us to write generic algorithms operating on different containers such as vector and list

Iterator Concept Hierarchy

Input Iterator, Output Iterator

- only 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. `std::slist`)

Bidirectional Iterator

- motion in both directions (++ --, e.g. `std::list`)

Random Access Iterator

- allows adding of offsets to iterators (e.g. `*(it+5)`)

reverse Iterators

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
```

Ranges

- Most algorithms are expressed in terms of iterator ranges `[begin, end)`
- Empty iff `begin() == end()`
- If n iterators are 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