Practical Programming Methodology

(CMPUT-201)

Michael Buro

Lecture 22

- STL Overview
- Sequence Containers vector<T>, list<T>
- Associative Containers set<T>, map<U,V>
- Iterators
- Algorithms

STL Overview Cont.

- Part of the C++ standard library
- Several implementations exist
- g++ comes with SGI version
- Web-sites
  - SGI STL site: `www.sgi.com/tech/stl` – good!
    STLport site: `www.stlport.org`
- Good Books
  - Josuttis: "The C++ Standard Library"
  - Meyers: "Effective STL"

Sequence Containers

`vector<T>`

- Vector template, dynamic array functionality
- Element type is T

`list<T>`

- Doubly linked list template
- Data associated with node is T

vector<T>

- `#include <vector>`
- Sequence that allows random access to elements of type `T` by index
- Simplest STL container, often most efficient one
- Amortized constant time insertion/deletion at the end
- Linear time insertion/removal anywhere else
- Vectors can grow and shrink
- Compatible with arrays: elements are laid out consecutively in memory
- Iterators that refer to vector elements are invalidated after insert/delete operations

vector Example

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

int main()
{
  const int N = 1000;
  vector<int> v;  // empty integer vector

  v.reserve(N);   // reserve memory for N elements
  // saves time and memory! v.size() still 0

  // append N elements
  for (int i=0; i < N; ++i) v.push_back(i);

  // add up all elements, array syntax
  int s = 0;
  for (size_t i=0; i < v.size(); ++i) sum += v[i];

  // remove all elements one by one back to front
  for (int i=0; i < N; ++i) v.pop_back();
  assert(v.empty());
  // v is destroyed here; if v contains pointers,
  // destructors are not called on the objects!
}
```

Frequently Used vector Member Functions

```
iterator begin()   : returns iterator to first element
iterator end()     : returns iterator to end (last element+1)

size_type size()   : # of elements in vector
bool empty() const : true iff vector is empty
                     (faster than !size())

void push_back(const T&) : inserts new element at the end
                           (amortized constant time)
void pop_back()    : removes last element

reference operator[](size_type n):
                     returns n-th element (starts with 0)
reference back()   : returns reference to last element

void clear()              : remove all elements
void erase(iterator pos)  : removes element at position pos
void reserve(size_type n) : allocates memory for n elements
bool operator==(const vector&, const vector&) : equality
```

list<T>

- #include <list>
- list<T> is a doubly linked list
- Data type associated with nodes is T
- 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> Examples

```
#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()+1, x);  // x now empty

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

Frequently Used list Member Functions

```
iterator begin() : returns iterator to first element
iterator end()   : returns iterator to end (last element+1)

size_type size()   : # of list elements (linear time!)
bool empty() const : true iff list is empty

void push_front(const T&) : inserts new element at the front
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 in front of 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 in front of pos, clears x
```

Other Sequence Containers

deque<T> ("deck");

- #include <deque>
- double-ended queue; supports random access: d[i]
- inserting/deleting at both ends takes amortized constant time
- inserting/deleting in the middle: linear time

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 visit 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 and const T& b and return true iff a < 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 = 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 not $(y < x)$

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 $:\Leftrightarrow$ not $(x < y)$ and not $(y < x)$]

Total order: above + "equivalent = identical"

E.g.: $<$ for int and string are total orders $\rightsquigarrow$ 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()(const Foo &a, const 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
```

Frequently Used set Functions

```
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)
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 ops.
```

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;
```

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)`)

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

Set Algorithms

```
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()));
// inserter_iterator maintains an insert position in the cont.
// each assignment *it++ = v; inserts v at current position

// computes intersection of sa and sb and stores
// result in sc: "orange"
set<string>::iterator it = sc.begin(), end = sc.end();
for (; it != end; ++it) cout << *it << endl;
```

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
```

Non-Mutating Algorithms

Work on range but 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 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 Functions

```
sort : sorts elements in ascending order

lower_bound, upper_bound : find first/last position to insert
                           value in a sorted range without
                          violating order in logarithmic time

merge : merge sorted ranges into one

includes : check if one range is contained in another

set_union, set_intersection, set_difference,
set_symmetric_difference : set operations
...
```

sort

```
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.
```

What else is there in STL?

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`