CS2720 Practical Software Development

STL Tutorial Spring 2011

Instructor: Rex Forsyth
Office: C-558
E-mail: forsyth@cs.uleth.ca
Tel: 329-2496

Tutorial Web Page: http://www.cs.uleth.ca/~forsyth/cs2720/lab/lab.html

Standard Template Library

Composed of three core parts

- Containers
- Algorithms
- Iterators

These work together to provide tools for problem solving

Container Classes

- A **container** is an object that contains other objects. Example: array, vector, string
- A **container class** provides services such as
  - insertion
  - deletion
  - searching
  - sorting
  - testing for membership
  - …

Containers

Objects that contain other objects. There are 3 main types of containers

1. Sequence
   - kept in ordinal sequence
   - position is independent of the value stored in it
   - several inserts provided
   - vector, deque, list
2. Associative
   - kept in sorted order
   - position is determined by the value stored in it
   - only one insert provided
   - $<$ must be overloaded
   - order of insertion does not matter
   - set, multiset, map, multimap

3. Special
    - basic_string - strings of any type
    - bitset - usually hexadecimal with bit operators

Algorithms

Act on contents of containers

Include capabilities for

- initialization
- sorting
- searching
- transforming

Iterators

Like pointers

Allow movement through a container

Allow access to individual elements in the container

Five types provided

- Random Access - stores and retrieves values; elements may be accessed randomly
- Bidirectional - stores and retrieves values; forward and backward access
- Forward - stores and retrieves values; forward access only
- Input - retrieve values but not store; forward access only
- Output - store values but not retrieve; forward access only

Others

- allocators
  - every container must have one
  - manages the memory
  - default one is provided which is generally sufficient

Function Objects

- predicates
  - special function which returns true or false
  - unary - has one argument
  - binary - has two arguments
  - arguments are always objects of the same type that the container holds
  - arguments are always *first, second*
- comparison functions
  - returns true or false
  - must always have 2 arguments, not necessarily objects
- other functions
  - has one parameter, return type may vary
  - eg print, square, change

Pairs

- a simple container type
- has the following
  - constructors
  - a method to create a pair `make_pair`
  - two public data members `first` and `second`
- #include <utility>

Examples

```
pair<name,int> p1;
p1 = make_pair(name("Jill Smith"), 120);

pair<name,int> p2(name("Joe Blow"), 170);
```

p1.first is a name

p2.second is an int

these are public so can be used as any other variable

use **typedef** to simplify declarations

```
typedef pair<name,int> person;

person p3;
```

can build arrays, vectors, lists etc. which contain pairs

Iterators

- all container classes define the following
  - iterator
  - const_iterator
  - reverse_iterator
  - const_reverse_iterator
- **vector** and **deque** iterators are random access
- all others are bidirectional
- to declare an iterator you must use the scope resolution operator since the iterator is a public member of the container class
- vector<Term>::iterator vit;
- vector<Term>::reverse_iterator rit;
- use **typedef** to simplify declarations
  - typedef vector<Term>::iterator vIter;
  - vIter vit;

All container classes provide the following methods in both const and non-const versions

- begin() - returns an iterator to the first element
- end() - returns an iterator to the element one past the last element
- rbegin() - returns an iterator to the first element of the reversed container, i.e. the last element
- rend() - returns an iterator to the element one past the last element of the reversed container, i.e. one before the first element

Forward iterators provide the following operators

- $*i$ - contents of the iterator; l-value if not const
- i->m - the same as $(*i).m$
- ++i, i++ - pre and post increment operators
- i=j - assignment operator
- i==j - equality operator
- i!=j - inequality operator

Bidirectional iterators provide all the foward iterator operators plus

- --i, i-- - pre and post decrement operators

Random access iterators provide all of bidirectional operators plus

- $i[n]$ - access the nth element
- i+=n , i-=n - increment/decrement the iterator n elements
- $i + n$, $i - n$ - return the incremented/decremented iterator
- $i - j$ - return the number of elements between i and j
- i<j, i<=j, i>j, i>=j - ordering operators

Using iterators

- simple pointers can be used as iterators
- sequence containers (vector, deque, list) provide the following (where i,j and k are iterators):
  - s(i,j) - constructor to build a container and insert objects from i to j not including j
  - s.assign(i,j) - make s a container containing objects from i to j not including j
  - s.insert(i,e) - insert element e at iterator i
  - s.insert(i,n,e) - insert n copies of element e at iterator i
  - s.insert(k,i,j) - insert all elments from i to j but not j, at iterator k
  - s.erase(i) - remove element at iterator i
  - s.erase(i,j) - remove all elements from i to j but not j

Examples

```
string names[] = {"Charmaine","Jesse","Hirobumi","Ming","Shu",
                  "Andrew","Tim","Andrew","Chad","Adam","Wen",
                  "Erin","Ye"};
deque<string> d(names+1, names+6);  // simple pointers as iterators
list<string> l(d.begin()+3,d.end());
l.assign(names,names+4);
l.insert(l.begin(),"Rex");
list<string>::iterator it = l.begin();
++it; ++it; ++it;
l.insert(it, 2, "Who");
l.insert(l.begin(), d.begin(), d.end());
it = l.begin();
++it; ++it; ++it; ++it;
l.erase(it);
it = l.end();
--it; --it; --it;
l.erase(l.begin(), it);
```

In addition to the `begin` and `end` functions, 3 other iterators are provided :

- back_inserter - has one parameter, a container to insert into
- front_inserter - has one parameter, a container to insert into
- inserter - has two parameters, a container and an iterator to the container

These can be used with many of the algorithms

Algorithms

- #include <algorithm>
- `copy(first, pastLast, iter)` - copies elements from *first* to *pastLast* into the container beginning at *iter*

```
it = l.begin();
++it;
//copy(d.begin(), d.end(), it);
copy(d.begin(), d.end(), back_inserter(l));
copy(names+8, names+13, front_inserter(d));
it = l.begin();
++it; ++it;
copy(names+11, names+13, inserter(l,it));
```

- `equal(first, pastEnd, iter)` - returns true if the elements in the container starting at *iter*, match those from *first* to *pastEnd*
- `find(first, pastEnd, value)` - returns an iterator to the first occurence of *value* between *first* and *pastEnd*. Returns *pastEnd* if no match is found.
- `sort(first, pastEnd)` - sort using the $<$ operator
- `sort(first, pastEnd, better)` - sort using the *better* function
- `replace(first, pastEnd, oldValue, newValue)`
- `replace_copy(first, pastEnd, iter, oldValue, newValue)`

```
if (equal(names, names+4, l.begin()) ...
sort(names, names+13);
replace(d.begin(), d.end(), string("Tim"), string("Timothy"));
replace_copy(l.begin(), l.end(), d.begin(), string("Who"), string("Unknown"));
vector<string> v;
replace_copy(l.begin(), l.end(), back_inserter(v), string("Who"), string("No Name"));
```

- unary predicate

```
bool firstHalf(const string& n) { return toupper(n[0]) < 'M'; }

cout << count_if(d.begin(), d.end(), firstHalf) << endl;
```

- binary predicate

```
bool dEqual(double x, double y) { return fabs(x-y) < TOLERANCE; }
bool sameExp(const Term& t1, const Term& t2) { return t1.exp() == t2.exp(); }

if (equal(f.begin(), f.end(), g.begin(), dEqual)) ...
if (equal(s.begin(), s.end(), t.begin(), sameExp)) ...
```

- comparison

```
bool smallerExp(const Term& t1, const Term& t2) { return t1.exp() < t2.exp(); }
bool after(const string& n1, const string& n2) { return n1 > n2; }

sort(s.begin(), s.end(), smallerExp);
sort(d.begin(), d.end(), after);
```

- other

```
void print(const string& n) { cout << n ' '; }
string addLetter(const string& n) { return n + n[0]; }

for_each(d.begin(), d.end(), print);
deque<string> funny;
transform(d.begin(), d.end(), front_inserter(funny), addLetter);
```

Function objects

- We can create a class which overloads the function call operator ()

```
class addLetter {

   public:
      string operator() (const string& s) const { return s+s[0]; }
}
```

- We can now declare `function objects` and use them

```
addLetter duh;
cout << duh("Henry") << end;
```

- This class can be used as the parameter to an algorithm

```
transform(d.begin(), d.end(), front_inserter(funny), addLetter());
```

- Any class which overloads the function operator may be used

Predefined function objects

- The C++ Standard defines many template function objects
- #include <functional>

```
plus<int> add;
plus<char> catenate;
cout << add(3,4) << ' ' << catenate('a','A') << endl;

vector<bool> bVec(d.size());
fill(bVec.begin(), bVec.end(), true);
transform(d.begin(), d.end(), l.begin(), bVec.begin(), less<string>());
```

Predefined function objects

Function Name	**Explanation**
`plus`	creates function objects that returns `a + b`
`minus`	creates function objects that returns `a - b`
`multiplies`	creates function objects that returns `a * b`
`divides`	creates function objects that returns `a / b`
`modulus`	creates function objects that returns `a % b`
`negate`	creates function objects that returns `-a`
`equal_to`	creates function objects that returns `a == b`
`not_equal_to`	creates function objects that returns `a != b`
`greater`	creates function objects that returns `a > b`
`less`	creates function objects that returns `a < b`
`greater_equal`	creates function objects that returns `a >= b`
`less_equal`	creates function objects that returns `a <= b`

`logical_and`	creates function objects that returns `a && b`
`logical_or`	creates function objects that returns `a || b`
`logical_not`	creates function objects that returns `!a`

Predefined function adapters

- may only be used with function objects derived from standard classes `unary_function` or `binary_function`
- adapt one function object to perform differently
- `f1` denotes a function object with one paramater; `f2` denotes a function object with two parameters.

Adapter Name	**Explanation**
`bind1st(f2,x)`	Changes f2 into a function object with one parameter. The first parameter is bound to the value x.
`bind2nd(f2,x)`	Changes f2 into a function object with one paramter. The second parameter is bound to the value x.
`not1(f1)`	Gives a function that returns the negated value of a call to f1.
`not2(f2)`	Gives a function that returns the negated value of a call to f2.
`ptr_fun(f)`	Gives a function object that calls the function f.

```
transform(d.begin(), d.end(), l.begin(), bVec.begin(), not2(less<string>()));

cout << count_if(d.begin(),d.end(),bind2nd(less<string>(),"M"))) << endl;

cout << count_if(d.begin(),d.end(),bind1st(less<string>(),"M"))) << endl;

// assume f contains doubles
find_if(f.begin(), f.end(), bind2nd(ptr_fun(dEqual),17.45));
```

Some vector class methods and operators

Method prototype/operator	Explanation	Example
`vector<T>()`	default constructor; creates an empty vector to hold elements of type `T`	vector<int> vec1;
`vector<T>(const vector& v)`	copy constructor; creates a copy of `v`	vector<char> vec2(vec1);
`vector<T>(int n)`	creates a vector with `n` elements	vector<float> vec1(10);
`vector<T>(int n, const T& t)`	creates a vector with `n` elements each with a value `t`	vector<char> vec1(10,'f');
`=`	assign a vector to this vector	vec1 = vec2
`int size()`	returns the number of elements in this vector	i = vec1.size()
`void resize(int n)`	changes the number of elements to `n`. If `n` is larger than the current size, the vector's size is enlarged. If smaller, the vector is truncated.	vec1.resize(15)

`void resize(int n, const T& t)`	changes the number of elements to `n`. If `n` is larger than the current size, the vector's size is enlarged and all new elements at set to `t`. If smaller, the vector is truncated.	vec1.resize(15,5)
`int capacity()`	returns the current capacity for this vector	i = vec1.capacity()
`void reserve(int n)`	changes the capacity to `n` if `n` is larger than the current capacity. Otherwise the capacity is unchanged.	vec1.reserve(20)
`T& at(int pos)`	returns the element at position `pos`. Remember that vectors start at position 0! This is an **l-value**, that is, it can be used on the left hand side of an assignment statement.	c = vec1.at(5) vec1.at(5) = 't'
`[ ]`	same as `at` except that it may not be checked.	c = vec1[5] vec1[5] = 't'
`bool empty()`	returns true if `size()` == 0.	if (vec1.empty()) ...

`T& front()`	returns the first element of the vector	i = vec1.front()
`T& back()`	returns the last element of the vector	i = vec1.back()
`void push_back(const T& t)`	appends `t` to the back of this vector. The size is incremented.	vec1.push_back(3.4)
`void pop_back()`	removes the last element from this vector. The size is decremented.	vec1.pop_back()
`void clear()`	removes all elements from this vector. The size is 0.	vec1.clear()
`vector<T>(iterator f, iterator l)`	creates a vector with a copy of the data from `f` to `l`	int a[] = {1,2,3,4,5,6,7,8,9,10}; vector<int> v(a, a+10);
`iterator begin()`	returns an iterator to the first item	v.begin()
`iterator end()`	returns an iterator to one past the last item	v.end()
`const_iterator begin() const`	same as begin but it is an r-value only	v.begin()
`const_iterator end() const`	same as end but it is an r-value only	v.end()
`reverse_iterator rbegin()`	returns an iterator to the first item in the reversed vector (i.e the last item)	v.rbegin()
`reverse_iterator rend() const`	returns an iterator to one past the last item in the reversed vector	v.rend()

`void insert(iterator p, const T& x);`	insert `x` before `p`	v.insert(v.end(), 5)
`void insert(iterator p, iterator f, iterator l);`	insert the range [f..l] before `p`	v.insert(v.end(), a, a+5)
`void insert(iterator p, int n, const T& x);`	insert `n` copies of `x` before `p`	v.insert(v.end(),7, 5)
`void erase(iterator p);`	erase the item at `p`	v.erase(v.begin())
`void erase(iterator f, iterator l);`	erase the all items from `f` to `l`	v.erase(v.begin(),v.end())
`void swap(vector<T>& vec);`	swap this vector with `vec`	v.swap(vec2);

```
vector<Item> v;
typedef vector<Item>::iterator vIter
vIter start = v.begin();
vIter stop = v.end();
vIter it = start+4;
it+=3;

//traverse vector with index
for (size_t i = 0; i < v.size(); i++)
    cout << v[i] << ' ';

// traverse vector with iterator
for (vIter p = start; p != stop; ++p)
   cout << *p << ' ';

// traverse using for_each assuming we have defined an action function
for_each(start, stop, action);

// partial traverse
for_each(it, it+6, action);

// reverse transversal
for_each(v.rbegin(), v.rend(), action);
```

deque

- random access iterators
- allows insertion and deletion at both ends
- has all the vector method except `capacity` and `reserve`
- two additional methods `push_front` and `pop_front`
- #include <deque>

Stack

#include <stack>

- no iterator access
- Has the following functions which are implemented from deque
  - size
  - empty
  - top
  - push
  - pop

Queue

#include <queue>

- no iterator access
- Has the following functions which are implemented from deque
  - size
  - empty
  - front
  - back
  - push
  - pop

Priority Queue

- #include <queue>
- same operations as queue
- popping order is determined by priority
- the operator < must be overloaded

List

Has the following functions which are implemented from deque

- all the members of the deque class except `[ ]` and `at`
- allows efficient insertion and deletion at any position in the list
- bidirectional iterators
- #include <list>
- other functions

```
splice(iterator p, list& l, iterator p1)
splice(iterator p, list& l, iterator p1, iterator p2);
remove(const T& x);
unique();
merge(list& l);
reverse();
sort();
```

Map

- dictionary or table or associative array
- key value associates with a specific value
- consists of `pairs`
- #include <map>
- same functions as vector
- one new function `find(key)`
- can use subscript operator with the key
  - if key is not a valid subscipt, it is inserted with default value
- no duplicate keys are allowed
- bidirectional iterators
- multimap - allows duplicate keys

Some Examples

```
#include <map>

map<name,date> birthdays;
birthdays.insert(make_pair(name("Jill Wilson"),date(2000,11,10)));
birthdays.insert(make_pair(name("Thomson, Jenny"),date(1981,1,3));

birthdays.insert(make_pair(name("Jill Wilson"),date(1975,10 21)));
birthdays[name("I Am Lazy Boy")] = date(1982,12,1);

birthdays[name("Jill Wilson")] = date(2000,11,11); // makes a change

name who = "Jill Wilson";
cout << birthdays[who] << endl;
cout << birthdays[name("Not Here")] << endl;  // causes an insert

// use typedef to declare iterators for simplicity sake
typedef map<name,date>::iterator mIter;
mIter it = birthdays.find(who);
if (it != birthdays.end())
   cout << who << " has birthday " << it->second << endl;
else
   cout << who << " is not in the birthday list" << endl;

for (mIter it = birthdays.begin(); it != birthdays.end(); ++it)
    cout << it->first << " -- " << it->second << endl;
```

Set

- like a map but only the keys are stored
- #include <set>
- no duplicate values allowed
- bi-directional iterators
- < operator must be overloaded
- multiset - allows duplicate values

Some Examples

```
#include <set>

set<string> setOne;       // empty set
setOne.insert("square"); setOne.insert("cube"); setOne.insert("circle");
setOne.insert("triangle"); setOne.insert("cube"); setOne.insert("prism");

setOne.erase("circle");
setOne.erase(setOne.begin());

string tmp[] = {"rectangle","cone","rhombus","cylinder", "cube", "sphere"};
set<string> setTwo(tmp+2,tmp+6);

copy(setTwo.begin(),setTwo.end(),inserter(setOne,setOne.begin()));

setOne = setTwo;
```

Algorithms

- void swap(T&, T&)
- const T& min(const T& a, const T& b)
- const T& max(const T& a, const T& b)
- void sort(iterator, iterator)
- void sort(iterator, iterator, comparisonFunction)
- bool binary_search(iterator, iterator, T& key)
- bool binary_search(iterator, iterator, T& key, comparisonFunction)
- iterator find(iterator, iterator, T& key)
- iterator find_if(iterator, iterator, predicate)
- void fill(iterator, iterator, const T& value)
- void reverse(iterator, iterator)
- void random_shuffle(iterator, iterator)
- iterator rotate(iterator first, iterator middle, iterator last)

- iterator remove(iterator, iterator, const T&)
- iterator remove_if(iterator, iterator, predicate)
- void replace(iterator, iterator, const T& oldVal, const T& newVal)
- void replace_copy(iterator, iterator, iterator, const T& oldVal, const T& newVal)
- void replace_if(iterator, iterator, predicate, const T& newVal)
- iterator unique(iterator, iterator)
- iterator unique(iterator, iterator, predicate)
- iterator unique_copy(iterator, iterator, iterator)
- iterator unique_copy(iterator, iterator, iterator,predicate)
- bool lexicographical_compare(iterator, iterator, iterator, iterator);
- bool lexicographical_compare(iterator, iterator, iterator, iterator,comp)

Set Algorithms

- bool includes(iterator first1, iterator last1, iterator first2, iterator last2)
- iterator set_union(iterator, iterator, iterator, iterator, iterator result)
- iterator set_intersection(iterator, iterator, iterator, iterator, iterator)
- iterator set_difference(iterator, iterator, iterator, iterator, iterator)
- iterator set_symetric_difference(iterator, iterator, iterator, iterator, iterator)

Some Examples

```
set<string> union;
set_union(setOne.begin(), setOne.end(),
          setTwo.begin(), setTwo.end()
          inserter(union, union.begin()));

set<string> inter;
set_intersection(setOne.begin(), setOne.end(),
                 setTwo.begin(), setTwo.end()
                 inserter(inter, inter.begin()));

set<string> diff;
set_difference(setOne.begin(), setOne.end(),
               setTwo.begin(), setTwo.end()
               inserter(diff, diff.begin()));

set<string> symDiff;
set_symetric_difference(setOne.begin(), setOne.end(),
                        setTwo.begin(), setTwo.end()
                        inserter(symDiff, symDiff.begin()));

if (includes(diff.begin(), diff.end(), union.begin(), union.end()))
   cout << "diff is a subset of union" << endl;
```

Heap Algorithms

- void make_heap(iterator, iterator)
- void push_heap(iterator, iterator)
- void pop_heap(iterator, iterator)
- bool is_heap(iterator, iterator)
- void sort_heap(iterator, iterator)

Some Examples

```
int array[] = { 30, 45, 35, 13, 50, 18, 27, 20, 29 };
vector<int> v(array, array+9);
vector<int> w(v);

make_heap(array, array+9);

make_heap(w.begin(), w.end());
make_heap(v.begin(), v.end(), greater<int>());

cin >> i;
v.push_back(i); w.push_back(i);
push_heap(w.begin(), w.end());
push_heap(v.begin(), v.end(), greater<int>());

cout << w.front() << ' ' << v.front() << endl;
pop_heap(w.begin(), w.end());
pop_heap(v.begin(), v.end(), greater<int>());

sort_heap(w.begin(), w.end());
sort_heap(v.begin(), v.end(), greater<int>());
```