CSE687 - Object Oriented Design

Standard Template Library

Jim Fawcett
Spring 2009

Some Definitions

- vector, string, deque, and list are ***standard sequence containers***.
- set, multiset, map, and multimap are ***standard associative containers***.
- ***Iterators***:
  - ***Input iterators*** are read only – each iterated element may be read only once.
  - ***Output iterators*** are write-only – each iterated element may be written only once.
  - ***Forward iterators*** can read or write an element repeatedly. They don't support operator--() so they can only move forward.
  - ***Bidirectional iterators*** are like forward iterators except that they support moving in both directions with operator++() and operator--().
  - ***Random access iterators*** are bidirectional iterators that add the capability to do iterator arithmetic – that is they support *(it + n);
- Any class that overloads the function call operator - operator() - is a functor class, and we refer to its instances as functors or function objects.

Computational Complexity

- Constant time refers to operations that do not depend on the number of elements stored in a container.
  - Inserting an element into a list is a constant time operation. Finding the location at which to insert is a linear time operation.
- Logarithmic time refers to operations that need time to run that grows as the logarithm of the number of elements in the container.
  - A logarithmic operation on a container with 1,000,000 takes 3 times as long to complete as that operation of a container with 1,000 elements.
- Linear time refers to operations that require computation time that grows proportionally to the number of elements in the container.

STL Supports Guaranteed Complexity for Container Operations

- ***Vectors and Deques:***
  - Insertion is a linear time operation.
  - Accessing a known location is constant time.
  - Searching an unsorted vector or deque is a linear time operation.
  - Searching a sorted vector or deque should be a logarithmic time operation (use binary_search algorithm to ensure that it is).
- ***Lists:***
  - Insertion is a constant time operation.
  - Accessing a known location and searching, whether sorted or not, is linear time, with the exception of the end points, which can be accessed in constant time.
- ***Sets and Maps:***
  - Insertion and accessing are logarithmic time operations.
  - Searching should be a logarithmic time operation (use member function find, etc., to ensure that it is).

STL Supports Guaranteed Complexity for Container Operations

- ***Unordered_set and Unordered_map***
  - Lookup, insertion, and deletion are constant time operations

STL Header Files for Containers

<deque>	deque<T>	Double ended queue, fast insert/remove from either end, indexable
<list>	list<T>	Doubly linked list, fast insert/erase at current location and either end, slow traversal
<map>	map<key, value> multimap<key,value>	Associates values with sorted list of keys, fast insert/remove, fast access with index, fast binary search. Map is indexable
<queue>	queue<T> priority_queue<T>	First in, first out queue Efficient insertion, removal of largest
<set>	set<T> multiset<T>	Set of sorted keys, fast find/insert/remove
<stack>	stack<T>	Last in, first out queue
<vector>	vector<T>	Slow insert/delete except at end, fast access with index. Slow find.

STL Header Files for Containers

`<array>`	`array<T>`	Fixed array of elements of type T
`<unordered_set>`	`unordered_set<T>`	Unordered collection, constant time lookup, insertion, removal
`<unordered_map>`	`unordered_map<k,v>`	Unordered key/value collection, constant time lookup, insertion, removal

Other STL Header Files

<algorithm>	find, find_if, search, copy, fill, count, generate, min, sort, swap, transform, …	applied to a container over an iteration range
<functional>	bind1st, bind2nd, divides, equal_to, greater, less, negate, minus, multiplies, plus, …	passed to an algorithm instead of using function pointers.
<iterator>	operator+, operator=, operator++, operator--, operator*, operator->, …	defines current location, range of action on a container or stream
<memory>	allocator, operator==, operator!=, operator=, operator delete, operator new	supports redefinition of allocation policy for containers
<numeric>	Accumulate, product, partial sum, adjacent difference	applied to a container over an iteration range
<utility>	pair, operator!=, operator<=, operator>, operator>=	pair class and global operators

STL Iterators

Input iterator	Read only, move forward	istream_iterator
Output iterator	Write only, move forward	ostream_iterator inserter front_inserter back_inserter
Forward iterator	Read and write Forward moving	
Bidirectional iterator	Read and write Forward and backward	list set, multiset map, multimap
Random access iterator	Read and write Random access	C++ pointers vector deque

STL Functions

- unary functions:
  - take single argument of the container’s value_type

```
// unary function
template <typename T>
void printElem(T val) {
  cout << “value is: ” << val << endl;
}

void main( ) {
  list< int > li;
      :
  // unary function used in algorithm
  for_each(li.begin(), li.end(), printElem);
}
```

STL Functions

- predicate:
  - function taking a template type and returning bool

```
// predicate
template <class T>
bool ispositive(T val) { return (val > 0); }

void main( ) {
  list<int> li;
    :
  // return location of first positive value
  list<int>::iterator iterFound =
         find_if(li.begin(), li.end(), ispositive<int>);
}
```

STL Function Objects

- Function objects:
  - class with constructor and single member operator()

```
template <class T> class myFunc {
  public:
    myFunc( /*arguments save needed state info */) { }
    T operator()(/* args for func obj */) {
      /*
        call some useful function with saved
        state info and args as its parameters
      */
    }
  private:
    /* state info here */
}
```

unary_function type

- The unary_function type serves as a base class for functors that will be used in adapters like not1. It supplies traits needed by the adaptors.

  An example use follows on the next slide

```
#include <functional>

template <class Arg, class Result>
struct unary_function{
  typedef Arg argument_type;
  typedef Result result_type;
};
```

STL Function Adapters

- negators:
  - not1 takes unary_function predicate and negates it
  - not2 takes binary_function predicate and negates it

```
 // predicate
template <class T>
class positive : public unary_function
{
  public:
    bool operator()(T val) const { return (val > 0); }
};

void main( ) {
  list<int> li;
    :
 // return location of first positive value
  list<int>::iterator iter =
          find_if(li.begin(), li.end(), positive);

 // return location of first non-positive value
  iter = find_if(li.begin(), li.end(), not1(positive));
}
```

binary_function type

- The binary_function type provides traits needed by binary function adapters, as illustrated on the next slide.

```
#include <functional>

template <class Arg1, class Arg2, class Result>
struct binary_function
{
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```

STL Function Adapters

- binders:
  - bind1 binds value to first argument of a binary_function
  - bind2 binds value to second argument of binary_function

```
void main( ) {

  list<int> li;
    :
  // return location of first value greater than 5
  list<int>::iterator =
     find_if(li.begin(), li.end(), bind2(greater<int>(),5));
}
```

STL Function Objects

arithmetic functions

plus	addition:	x + y
minus	subtraction:	x - y
times	multiplication:	x * y
divides	division:	x / y
modulus	remainder:	x % y
negate	negation:	-x

comparison functions

equal_to	equality test:	x == y
not_equal_to	inequality test:	x != y
greater	greater-than comparison:	x > y
less	less-than comparison:	x < y
greater_equal	greater or equal:	x >= y
less_equal	less or equal:	x <= y

logical functions

logical_and	logical conjunction:	x && y
logical_or	logical disjunction:	x || y
logical_not	logical negation:	!x

Algorithms by Type

compare	`equal, lexicographical_compare, mismatch`
copy	`copy, copy_backward`
heap operations	`make_heap, pop_heap, push_heap, sort_heap`
initialization	`fill, fill_n, generate, generate_n`
merge	`inplace_merge, merge`
min and max	`max, max_element, min, min_element`
permutations	`next_permutation, prev_permutation`
remove	`remove, remove_copy, remove_copy_if, remove_if, unique, unique_copy`

Algorithms by Type (continued)

scanning	accumulate, for_each
Search	adjacent_find, count, count_if, find, find_if, find_first_of, search
set operations	includes, set_difference, set_intersection, set_symmetric_difference, set_union
sorting	nth_element, partial_sort, partial_sort_copy, sort, stable_sort
swap operations	swap, swap_ranges
transformations	partition, random_shuffle, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, stable_partiton, transform

End of Presentation