Standard Template Library

Jim Fawcett
Summer 2017

Some Definitions

- vector, string, deque, and list are ***standard sequence containers***.
- set, multiset, map, multimap, unordered_set, unordered_multiset, unordered_map and unordered_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.

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:***
  - Based on Red-Black binary tree.
  - 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***
  - Based on hashtable
  - Lookup, insertion, and deletion are (nearly) constant time operations
  - They are hashed containers, so we get access to an element by computing a hash function on a key which maps to an address in the table. This is constant time. If there is more than one element that hashes to that address then we search a linked list rooted at that address (the elements on this list are referred to as a bucket).
  - So access is nearly constant time.

STL Header Files for Containers

Include	Container Type	Description
<array>	array<T>	Fixed array of elements
<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> mulitmap<key, value>	Associates values with sorted list of keys, fast insert/remove, fast access with index, fast binary search, is indexable with keys.
<queue>	queue<T> priority_queue<T>	First in, first out queue, efficient push and pop. Efficient removal of largest.
<set>	set<T> multiset<T>	Set of sorted keys, fast find/insert/removal.
<stack>	stack<T>	Last in, first out stack.
<unordered_map>	unordered_map<K,V> Unordered_multimap<K,V>	Unordered key/value collection, constant time lookup, insertion, removal.
<unordered_set>	unordered_set<V> unordered_multiset<V>	Unordered collection of values, constant time lookup, insertion, removal.
<vector>	vector<T>	Slow insert, delete, except at the end, fast access with index. Slow find. Capacity grows as needed.

Other STL Header Files

Include	Types	Description
<algorithm>	Find, find_if, search, copy, fill, count, generate, min, sort, swap, transform, …	Applied to a container over an iterator range.
<functional>	Function, bind, divides, equal_to, greater, less, negate, minus, plus, …	Function objects passed to an algorithm to operate on elements of a container.
<iterator>	operator+, operator=, operator++, operator--, operator*, operator->, …	Defines current location, range of action on a container or stream.
<memory>	unique_ptr, shared_ptr, allocator, operator==, operator!=, operator=, operator delete, operator new, …	Provides smart pointer for managing resources on native heap, 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 struct and global operators.

STL Iterators

- Input iterator
  - Read only, move forward
  - Example: istream_iterator
- Output iterator
  - Write only, move forward
  - Examples: ostream_iterator, inserter, front_inserter, back_inserter
- Forward iterator
  - Read and write, move forward
  - Example: forward_list<T>::iterator
- Bidirectional iterator
  - Read and write, move forward and backward
  - Examples: list<T>::iterator, map<K,V>::iterator
- Random access iterator
  - Read and write, random access
  - Examples: vector<T>::iterator, deque<T>::iterator, C++ pointers

STL Functions

- unary functions:
  - Function taking single template argument
  - Will be instantiated with 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);
}
```

  - for_each calls printElem with values from list

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 */
}
```

std::function

adapted from example in https://oopscenities.net/2012/02/24/c11-stdfunction-and-stdbind/

```
#include <functional>
#include <iostream>
#include <string>
#include <vector>

void execute(const
std::vector<std::function<void()>>& fs)
{
  for (auto& f : fs)
    f();
}

void plain_old_func()
{
  std::cout << "\n  I'm a plain old
function";
}

class functor
{
public:
  void operator()() const
  {
    std::cout << "\n  I'm a functor";
  }
};
```

```
int main()
{
  std::vector<std::function<void()>> x;
  x.push_back(plain_old_func);

  functor functor_instance;
  x.push_back(functor_instance);
  x.push_back([]()
  {
    std::cout << "\n  Hi, I'm a lambda
expression";
  });

  execute(x);

  std::cout << "\n\n";
}
```

std::bind

adapted from example in https://oopscenities.net/2012/02/24/c11-stdfunction-and-stdbind/

```
#include <functional>
#include <iostream>
#include <string>
#include <vector>

void execute(const std::vector<std::function<void()>>& fs)
{
  for (auto& f : fs)
    f();
}

void show_text(const std::string& t)
{
  std::cout << "\n  Text: " << t;
}

int main()
{
  std::vector<std::function<void()>> x;
  x.push_back(std::bind(show_text, "Bound function"));
  execute(x);
  std::cout << "\n\n";
}
```

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