Lecture 14b
The Standard Template Library:
Algorithms
April 27th, 2016

Announcements

• PA9 is due today
  – Questions?
• PA10 is due one week from today
• Reading Assignment
  – Chapter 10 quiz due before class Friday
• Recitation this week
  – Installing & using google test (from scratch)
• PA8 is graded
  – 10 points submit / 10 points compile
    – 8 tests; 5 points each correctness
    – 8 tests; 5 point each speed
    – Fastest (correct) response: 5 points
    – Diff = median time minus fastest time
    – Times >= fastest + 4*diff get 0 speed points
    – Linear ramp between fastest and (fastest + 4*diff)
    – 50% of correct responses get 3.75 or more points
    – Correct median responses on all answers yields 90
• ACM: virtualization via Docker; CA Technologies

STL

1. Containers
   – Abstractions of data structures
     – Hold a single, arbitrary data type
2. Iterators
   – Abstractions of pointers
     – Allow you to iterate through containers
     – Begin() and end() are pointers to first element and first non-element
3. Algorithms
   – Universal algorithms (e.g. sort)
     • Without reference to object type being acted on
     • Without reference to data structure holding the objects
4. Other
   – Multi-threading
   – Numerics
   – Smart pointers

Choice of Containers

<table>
<thead>
<tr>
<th>Container</th>
<th>Push Back</th>
<th>Push Front</th>
<th>Insert/delete</th>
<th>Index</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>deque</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>list(double)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>set/map</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

• Containers chosen based on complexity of operators
  – Which operators do you use, and how often?
  – List of operators above is partial
  – List of containers above is partial
  – Be aware of hidden costs
    – deques require (slightly) more space than vectors
    – Unordered sets (hash tables) require a lot more space

Example function using iterators

template<typename ITER>
ITER max_element(ITER start, ITER end)
{
    ITER max = start;
    for(ITER i = start; i != end; i++){
        if (*i > *max) max = i;
    }
    return max;
}

STL Algorithms

• Goals:
  – Provide useful basic algorithms (think ‘sort’)
  – Without regard to the data type operated on
  – Without regard to the data structure it is stored in
  – Over a whole container or a fragment thereof
• Example

    Template<typename ITER, typename VALUE>
    VALUE accumulate(ITER start, ITER end, VALUE init_value)
    {
        for(ITER iter=start; iter!=end; iter++)
            init_value += *iter;
        return init_value;
    }
List of Examples

- find (uses ==)
- find_if (function as predicate)
- search (takes in 2 sequences; 4 iterators)
- find_adjacent (function as predicate)
- count (uses ==)
- count_if (function as predicate)
- mismatch (takes in 2 sequences; 4 iterators)
- equal (takes in 2 sequences; 4 iterators)
- search, n (n adj) equal elements; function)
- copy
- copy_backward
- swap
- transform (arbitrary function)
- for_each (arbitrary function)
- replace (uses ==)
- replace_if (function as predicate)
- unique (function as pred; removes non unique elements)
- reverse
- rotate

Wow...

- Take-away messages:
  - If you want to do something common, its probably in <algorithm>
  - You need to be comfortable with iterators to call STL algorithms
  - There is a pattern to STL function calls...
  - Functors are critical to getting the most out of STL
  - In the extreme, almost all loops can be replaced with STL calls
- No, the previous lists are not complete

STL Function Calls

- Many STL functions operate on a single container:
  - Sort, find, count, reverse, rotate, random_shuffle,...
  - These take two iterators as arguments
    - Start: where in the container to begin
    - End: where in the container to end
  - Some allow a function predicate as an optional 3rd argument
    - Example: sort
  - Others have a_if version that takes a function predicate
    - Example: count_if

Functor Examples

```cpp
class Evenp {
public:
    bool operator() (int n)
    { return (n == 2*(n/2)); }
};
class Mod_less {
public:
    mod_less(int mod_base): base(mod_base) {} 
    bool operator< (int a, int b)
    { return ((a % base) < (b % base)); } 
protected:
    int base;
}
```

STL Function Call Examples

```cpp
// initialize data for example
vector<int> vec = initialize();

// count even elements in vec
int n =
    count_if(vec.begin(), vec.end(), Evenp());

// sort the elements in vec
sort(vec.begin(), vec.end());

// stable sort elements by mod 111
stable_sort(vec.begin(), vec.end, Mod_less(111));

// why did I use stable_sort above?
```
STL Function Calls (II)

- Other STL functions map from one container to another
  - Example: copy
  - iter copy(iter start, iter end, iter dest)
  - Return value is end of destination after the copy
- Warning: STL routines do not allocate memory
  - Copy uses assignment (=)
  - Copy increments dest iterator (++)
  - If dest doesn't have enough elements, bad things happen...

```
// possible implementation of copy
// notice that no memory is allocated
template <typename ITER>
ITER copy(ITER start, ITER end, ITER dest)
{
    for(ITER iter = start; iter != end; iter++)
    {
        *dest++ = *iter
    }
}
```

Buggy Code

```cpp
vector<int> vec = initialize();
vector<int> second_vec;

// this crashes
copy(vec.begin(), vec.end(),
     second_vec.begin());
```

Fixed Code

```cpp
vector<int> vec = initialize();
vector<int> second_vec(vec.length());

// this doesn't crash
copy(vec.begin(), vec.end(),
     second_vec.begin());
```

STL Function Calls (III)

- Still other STL functions combine two containers into a 3rd:
  - Example: merge
  - Iter merge(iter start1, iter end1, iter start2, iter end2, iter dest)
  - Optional 6th argument is a functor to replace ==
- Again, STL functions do not allocate memory...

Why so many functors?

- Imagine your goal is to sort Quaggas by weight. You have two options:
  - Overload < for Quaggas
  - Write a functor, pass it to sort
- Imagine your goal is to sort Quaggas by weight in one part of your program, and height in another
  - You need to write functors
  - You cannot re-define < mid program
Eliminating Loops

• STL algorithms can eliminate most loops
• transform
  – Iter transform(iter start, iter end, iter dest, functor fn)
  – Fn can be any functor such that:
    • It is unary on *start
    • Its return type can be stored in *dest
  – Iter transform(iter start1, iter end1, iter start2, iter end2, iter dest)
  – Fn can be any functor such that:
    • It is binary on *start1 and *start2
    • Its return type can be stored in *dest
• for_each
  – Fn for_each(iter start, iter end, functor fn)
  – Fn can be any functor such that:
    • It is unary on *start
  – Fn is for side-effects; its return value is ignored