Lecture 13b
The Standard Template Library: Iterators
(and if we get there, algorithms)
November 16th, 2017

Announcements
• PA8 is due this Saturday
• Quiz 10 is due before class, Tuesday after break
• Recitation after break:
  – Installing & using Eclipse for C++
  – The last mandatory recitation

STL
1. Containers
   – Abstractions of data structures
2. Iterators
   – Abstractions of pointers
   – Allow you to iterate through containers
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

Containers
• Containers are data structures
  – capable of holding objects of any data type.
  – But only a single object type per container
  – Of course, polymorphism still works
    • But be careful of splicing
• Containers are templated classes
  – Example: vector
• They hold what you tell them to
  – vector<int> != vector<int*

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>
<tr>
<td>Unordered</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Today’s Topic: Iterators
• Iterators in STL/C++ are similar to iterators in Java
  – They support iteration
  – Within the semantics of a container
• Iterators are abstractions of pointers
  – Classes that overload *, ->, ==, !, and ++
  – Sequential iterators also overload --, +, -
• But first, let’s go back to basics…

• 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
Iterate through an array (the most obvious method)

```cpp
int length = 10;
int* arr = new int[length];
for(int i=0; i < length; i++) {
    arr[i] = ... 
}
```

Iterate through an array (faster – pointer hopping)

```cpp
int length = 10;
int* arr = new int[length];
int* end = arr + length;
for(int* ptr = arr;
    ptr != end;
    ptr++) {
    *ptr = ... 
}
```

STL Iterators

- STL Iterators act like pointers
- Are used as if pointer hopping
- No matter what the data structure

```cpp
set<int> myset = init_fn();
for(set<int>::iterator i = myset.begin();
    i != myset.end();
    i++) {
    ...
}
```

Simpler case: vector<int>

```cpp
vector<int> vec = foo();
for(vector<int>::iterator i = foo.begin();
    i < foo.end();
    i++) {
    ...
}
```

DIY: Lists & List Iterators

```cpp
template<typename TYPE>
class ListElement {
public:
    TYPE value;
    ListElement<TYPE>* next;
    ListElement<TYPE>* previous;
};
```

Lists & List Iterators (cont.)

```cpp
template <typename TYPE>
class iterator {
protected:
    ListElement<TYPE>* ptr;
public:
    TYPE& operator[] () {return (ptr->value);} 
    bool operator == (const iterator<TYPE>& alt) 
    {return (ptr == alt.ptr);} 
    bool operator != (const iterator<TYPE>& alt) 
    {return (ptr != alt.ptr)};
    iterator<TYPE>& operator++ () 
    {ptr = ptr->next; return *this;} 
    iterator<TYPE>& operator-- () 
    {ptr = ptr->previous; return *this;}
};
Why iterators matter

- Yes, I fudged some details on the previous slide
  - It doesn’t handle NULL pointers, for example.
  - But it conveys the key points
- Note that all of the following now work:
  
  ```cpp
  for(vector<int>::iterator i = vec.begin(); i != vec.end(); i++) {
  …}
  for(list<int>::iterator i = lst.begin(); i != lst.end(); i++) {
  …}
  for(set<int>::iterator i = st.begin(); i != st.end(); i++) {
  …}
  for(unordered_set<int>::iterator i = h.begin(); i != h.end(); i++) {
  …}
  ```

Remember This? (simplified)

```cpp
template<typename PIXEL>
class Image {
public:
  Image() {
    allocateMemory(0, 0);
  }
  Image(unsigned int width, unsigned int height) {
    allocateMemory(width, height);
  }
  Image(unsigned int width, unsigned int height, const PIXEL& value);
  Image(const Image<PIXEL>& img);
  ~Image() {
    deallocateMemory();
  }
  Image<PIXEL>& operator = (const Image& img);
  int Width() const {
    return width;
  }
protected:
  unsigned int width;
  unsigned int height;
  unsigned int size;
  PIXEL* data;
  PIXEL** rows;
};
```

Adding Iterators...

```cpp
template<typename PIXEL>
class Image {
public:
  typedef PIXEL* iterator;
  Image() {
    allocateMemory(0, 0);
  }
  Image(unsigned int width, unsigned int height) {
    allocateMemory(width, height);
  }
  Image(unsigned int width, unsigned int height, const PIXEL& value);
  Image(const Image<PIXEL>& img);
  ~Image() {
    deallocateMemory();
  }
  int Width() const {
    return width;
  }
  iterator begin() {return data;}
  iterator end() {return data + size;}

protected:
  unsigned int width;
  unsigned int height;
  unsigned int size;
  PIXEL* data;
  PIXEL** rows;
};
```

Fun with Iterators

```cpp
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;
}
```

Calling Max

```cpp
set<Quagga> herd = foo();
Quagga big =
  *max_element(herd.begin(), herd.end());
-----
Unordered_set<Animal> herd = bar();
Animal big =
  *max_element(herd.begin(), herd.end());
-----
vector<double> dvvec = foo(); // longer than 10
double big =
  *max_element(dvvec.begin(), dvvec.begin()+10);
```

Calling Max (cont.)

```cpp
int length = 10;
int* arr = initialize(length);
int big = *max_element(arr, arr + length);
-----
ifstream int_str(filename);
std::istream_iterator<int> iit(int_str);
std::istream_iterator<int> eof;
int big = *max_element(iit, eof);
-----
std::istream_iterator<int> iit(std::cin);
std::istream_iterator<int> eos;
int big = *max_element(iit, eos);
```
CS253 Fall 2017 © Bruce Draper

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

  ```cpp
  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;
  }
  ```

CS253 Fall 2017 © Bruce Draper

List of Examples

- 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 functor predicate as an optional 3rd argument
    - Example: sort
  - Others have a _if version that takes a functor predicate
    - Example: count_if

CS253 Fall 2017 © Bruce Draper

List of Examples (cont)

- random_shuffle
- sort (c or predicate versions)
- stable_sort (c or predicate versions)
- partial_sort (c or predicate versions)
- partial_sort_copy (c or predicate versions)
- nth_element (c or predicate versions)
- merge (c or predicate versions)
- inplace_merge (c or predicate versions)
- includes (one sorted sequence includes another; c or predicate versions)
- set_union (c or predicate versions)
- set_intersection (c or predicate versions)
- set_difference (c or predicate versions)
- set_symmetric_difference (c or predicate versions)
- min_element (c or predicate versions)
- max_element (c or predicate versions)
- min (of two; c or predicate versions)
- max (of two; c or predicate versions)
- next_permutation
- prev_permutation

CS253 Fall 2017 © Bruce Draper

STL Function Calls

- Take-away messages:
  - If you want to do something common, its probably in <algorithms>
  - 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

CS253 Fall 2017 © Bruce Draper

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

// 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…

Copy

// 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

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

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

Fixed Code

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