COMP 220: Data Structures **Lecture Notes 04**

Structural Traversal of Collection Types

In these notes we look at structural traversal patterns for our new non-vector typed ADTs.

Loop-Based Traversal

Traversing stacks, queues, maps, and sets is different than traversing a vector or a string. We cannot simply count off index values.

std::stack

You cannot traverse a std::stack without mutating the stack since that would violate the ordering restriction it imposes — LIFO. You must pop the top in order to get to the next. In general, do not count based on the stack size as the the size changes everytime you push and pop.

Figure 1: Top to bottom traversal of a stack

```
std::stack< ... > st;
 ...
while( !st.empty() ){
   ... st.top() ...  //non−destructive top checking

   ... st.pop() ...  // the ++ of stack  traversal
}
```

std::queue

Like stack traversal, queue traversal is a destructive process. You must remove the front in order to get to the next front.

Figure 2: Front to Back traversal of a queue

```
std::queue< ... > qu;
 ...
while( !qu.empty() ){
   ... qu.front() ...  //non−destructive front checking

   ... qu.pop() ...  // the ++ of queue traversal
}
```

std::map

A map only gets a sense of order from its implementation. Abstractly, they are best viewed as a set of key+value pairs. The std::map implementation effectively orders data in key order. The simplest way to traverse a std::map is with the range-based for loop [1]. This loop presents you with a std::pair<kt,vt> where kt is the key type and vt is the value type for the map[2].

Declaring the elements of the map as pairs will present you with copies of the map data.

[1] aka for-each loop

[2] `http://www.cplusplus.com/reference/utility/pair/`

Figure 3: Map Traversal with by-value pairs

```
std :: map<kt,vt> mp;
 ...
// for each key value pair kvp in map mp...
for( std :: pair<kt,vt> kvp : mp ){
      ... kvp. first ...  //copy of the key
      ... kvp.second... //copy of the value
      ... mp[kvp.first ]... //the actual value
}
```

For efficiency reasons or because we intend to do a mutation while we traverse we might want access to the actual key and value data through the pair. Keys cannot be mutated but values can. Thus, we use const type keys and a by-reference pair.

Figure 4: Map Traversal with by-reference pairs

```
std :: map<kt,vt> mp;
 ...
// for each key value pair kvp in map mp...
for( std :: pair<const kt,vt>& kvp : mp ){
      ... kvp. first ...  //const reference to the key
      ... kvp.second... //reference to the the value
      ... mp[kvp.first ]... //same as kvp.second
}
```

Finally, we can avoid the types of the pair altogether by using the C++11 `auto` type[3]. You can even avoid `pair` completely by using C++17's[4] structured bindings feature[5].

[3] You should treat this as a way to simplify complex type declarations and not a way to dodge a lack of knowledge of type

[4] -std=c++1z for older compilers

[5] `auto [k,v] : mp`

std::set

Set traversal is also best accomplished via range based for loops. This traversal is more straight forward than map traversal.

If the copy cost of a by-value traversal is a concern than you can do a constant reference traversal as well.

You cannot to a set traversal that allows you to mutate the set elements. If you need to modify the set contents you should be using

```
std :: map<kt,vt> mp;
 ...
// for each key value pair kvp in map mp...
for( const auto& kvp : mp ){
      ... kvp. first  ...   //const reference to the key
      ... kvp.second...  //reference to the the value
      ... mp[kvp.first ]...   //same as kvp.second
}
```

Figure 5: Read only Map Traversal with const-reference auto-typed pairs

```
std :: set<t> s;
 ...
// for each element of type t in set<t> s
for( t e : s ){
      ... e ...   //copy of current set element e
}
```

Figure 6: Traversal of set by-value

```
std :: set<t> s;
 ...
// for each element of type t in set<t> s
for( const t& e : s ){
      ... e ...   //copy of current set element e
}
```

Figure 7: Traversal of set by const-reference

set removal and insert operations to remove an item and add the "new"value.

Recursion

Stacks and Queues

Once again, we have no ability to access the "rest" of a stack or a queue without discarding the first. The structures must be traversed through mutation. This means recursive procedures for this structures must pass the structures by reference.

Figure 8: Recursive Traversal of Stack data

```
... stackfoo(std :: stack<t>& st, ...) {

  if ( st.empty() ){
     ...
    return ...;
  }

   ... st.top() ...
   ... st.pop() ...
   ... stackfoo(st ,...) ...

  return ... ;
}
```

Figure 9: Recursive Traversal of Queue data

```
... queuefoo(std::queue<t>& qu, ...){

  if ( qu.empty() ){
     ...
    return ...;
  }

   ... qu.front() ...
   ... qu.pop() ...
   ... queuefoo(qu,...) ...

  return ... ;
}
```

Maps and Sets

At this point you do not want to do recursive procedures for maps and sets. It requires utilizing C++ iterators and opens a can of worms we'd rather not open at this point.

Potential Exercises

1. Often it is useful compute how often words appear in a document. The first step of this might be to parse a file word-by-word, producing a vector of all the words that appear in the document one-by-one. Write a procedure that takes in such a vector and prints out each word that appeared followed by how many times that word appeared.

2. The previous question, but sort the words from most to least frequently used. Hint: take the (key,value) pairs from the map and put them into a vector. Then lookup how to use C++ sort[6].

[6] http://www.cplusplus.com/reference/algorithm/sort/

Answers

1.
```
void wc(std::vector<std::string>& words) {
  std::map<std::string,int> counts;

  for (auto& w : words)
    counts[w]++;

  for (auto& [word, count] : counts)
    std::cout << word << ":\t\t" << count << std::endl;

}
```