CSCI-1200 Data Structures — Spring 2018
Lecture 21 – Functors & Hash Tables, part II

Review from Lecture 20

- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- Collision resolution: separate chaining vs open addressing
- STL's `unordered_set` (and `unordered_map`)

Today's Lecture

- Using STL's `for_each`
- Something weird & cool in C++... Function Objects, a.k.a. *Functors*
- Continuing with Hash Tables...
  - STL's `unordered_set` (and `unordered_map`)
  - Hash functions as functors/function objects
- STL Queue and STL Stack
- Definition of a Binary Heap
- What's a Priority Queue?

21.1 Using STL's `for_each`

- First, here's a tiny helper function:

```
void float_print (float f) {
  std::cout << f << std::endl;
}
```

- Let's make an STL vector of floats:

```
std::vector<float> my_data;
my_data.push_back(3.14);
my_data.push_back(1.41);
my_data.push_back(6.02);
my_data.push_back(2.71);
```

- Now we can write a loop to print out all the data in our vector:

```
std::vector<float>::iterator itr;
for (itr = my_data.begin(); itr != my_data.end(); itr++) {
  float_print(*itr);
}
```

- Alternatively we can use it with STL's `for_each` function to visit and print each element:

```
std::for_each(my_data.begin(), my_data.end(), float_print);
```

  Wow! That's a lot less to type. Can I stop using regular `for` and `while` loops altogether?

- We can actually also do the same thing without creating & explicitly naming the `float_print` function. We create an *anonymous function* using *lambda*:

```
std::for_each(my_data.begin(), my_data.end(), [](float f){ std::cout << f << std::end; });
```

  Lambda is new to the C++ language (part of C++11). But lambda is a core piece of many classic, older programming languages including Lisp and Scheme. Python lambdas and Perl anonymous subroutines are similar. (In fact lambda dates back to the 1930's, before the first computers were built!) You'll learn more about lambda more in later courses like CSCI 4430 Programming Languages!

21.2 Function Objects, a.k.a. *Functors*

- In addition to the basic mathematical operators `+ - * / < >` , another operator we can overload for our C++ classes is the *function call operator*.

  Why do we want to do this? This allows instances or objects of our class, to be used like functions. It's weird but powerful.

- Here's the basic syntax. Any specific number of arguments can be used.

```
class my_class_name {
public:
  // ... normal class stuff ...
  my_return_type operator() ( /* my list of args */ );
};
```

21.3 Why are Functors Useful?

- One example is the default 3rd argument for `std::sort`. We know that by default STL's sort routines will use the less than comparison function for the type stored inside the container. How exactly do they do that?

- First let's define another tiny helper function:

```
bool float_less(float x, float y) {
  return x < y;
}
```

- Remember how we can sort the `my_data` vector defined above using our own homemade comparison function for sorting:

```
std::sort(my_data.begin(),my_data.end(),float_less);
```

  If we don't specify a 3rd argument:

```
std::sort(my_data.begin(),my_data.end());
```

  This is what STL does by default:

```
std::sort(my_data.begin(),my_data.end(),std::less<float>());
```

- What is `std::less`? It's a templated class. Above we have called the default constructor to make an instance of that class. Then, that instance/object can be used like it's a function. Weird!

- How does it do that? `std::less` is a teeny tiny class that just contains the overloaded function call operator.

```
template <class T>
class less {
public:
  bool operator() (const T& x, const T& y) const { return x < y; }
};
```

  You can use this instance/object/functor as a function that expects exactly two arguments of type `T` (in this example `float`) that returns a bool. That's exactly what we need for `std::sort!` This ultimately does the same thing as our tiny helper homemade compare function!

21.4 Another more Complicated Functor Example

- Constructors of function objects can be used to specify *internal data* for the functor that can then be used during computation of the function call operator! For example:

```
class between_values {
private:
  float low, high;
public:
  between_values(float l, float h) : low(l), high(h) {}
  bool operator() (float val) { return low <= val && val <= high; }
};
```

- The range between `low` & `high` is specified when a functor/an instance of this class is created. We might have multiple different instances of the `between_values` functor, each with their own range. Later, when the functor is used, the query value will be passed in as an argument. The function call operator accepts that single argument `val` and compares against the internal data `low` & `high`.
- This can be used in combination with STL's `find_if` construct. For example:

```
between_values two_and_four(2,4);

if (std::find_if(my_data.begin(), my_data.end(), two_and_four) != my_data.end()) {
  std::cout << "Found a value greater than 2 & less than 4!" << std::endl;
}
```

- Alternatively, we could create the functor without giving it a variable name. And in the use below we also capture the return value to print out the first item in the vector inside this range. Note that it does not print all values in the range.

```
std::vector<float>::iterator itr;
itr = std::find_if(my_data.begin(), my_data.end(), between_values(2,4));
if (itr != my_data.end()) {
  std::cout << "my_data contains " << *itr
            << ", a value greater than 2 & less than 4!" << std::endl;
}
```

21.5 Using STL's Associative Hash Table (Map)

- Using the default `std::string` hash function.
  - With no specified initial table size.
    `std::unordered_map<std::string,Foo> m;`
  - Optionally specifying initial (minimum) table size.
    `std::unordered_map<std::string,Foo> m(1000);`
- Using a home-made `std::string` hash function. Note: We are required to specify the initial table size.
  - Manually specifying the hash function type.
    `std::unordered_map<std::string,Foo,std::function<unsigned int(std::string)> > m(1000, MyHashFunction);`
  - Using the `decltype` specifier to get the "declared type of an entity".
    `std::unordered_map<std::string,Foo,decltype(&MyHashFunction)> m(1000, MyHashFunction);`
- Using a a home-made `std::string` hash functor or function object.
  - With no specified initial table size.
    `std::unordered_map<std::string,Foo,MyHashFunctor> m;`
  - Optionally specifying initial (minimum) table size.
    `std::unordered_map<std::string,Foo,MyHashFunctor> m(1000);`
- Note: In the above examples we're creating a association between two types (STL strings and custom `Foo` object). If you'd like to just create a set (no associated 2nd type), simply switch from `unordered_map` to `unordered_set` and remove the `Foo` from the template type in the examples above.

21.6 Additional STL Container Classes: Stacks and Queues

- We've studied STL vectors, lists, maps, and sets. These data structures provide a wide range of flexibility in terms of operations. One way to obtain computational efficiency is to consider a simplified set of operations or functionality.
- For example, with a hash table we give up the notion of a sorted table and gain in find, insert, & erase efficiency.
- 2 additional examples are:
  - **Stacks** allow access, insertion and deletion from only one end called the *top*
    - There is no access to values in the middle of a stack.
    - Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
    - All stack operations are $O(1)$

- **Queues** allow insertion at one end, called the *back* and removal from the other end, called the *front*
  - There is no access to values in the middle of a queue.
  - Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
  - All queue operations are $O(1)$

21.7 Exercises: Tree Traversal using a Stack and Queue

Given a pointer to the root node in a binary tree:

- Use an STL `stack` to print the elements with a pre-order traversal ordering. *This is straightforward.*
- Use an STL `stack` to print the elements with an in-order traversal ordering. *This is more complicated.*
- Use an STL `queue` to print the elements with a breadth-first traversal ordering.

21.8 What's a Priority Queue?

- Priority queues are used in prioritizing operations. Examples include a personal "to do" list, what order to do homework assignments, jobs on a shop floor, packet routing in a network, scheduling in an operating system, or events in a simulation.
- Among the data structures we have studied, their interface is most similar to a queue, including the idea of a `front` or `top` and a `tail` or a `back`.
- Each item is stored in a priority queue using an associated "priority" and therefore, the `top` item is the one with the lowest value of the priority score. The `tail` or `back` is never accessed through the public interface to a priority queue.
- The main operations are `insert` or `push`, and `pop` (or `delete_min`).

21.9 Some Data Structure Options for Implementing a Priority Queue

- Vector or list, either sorted or unsorted
  - At least one of the operations, `push` or `pop`, will cost linear time, at least if we think of the container as a linear structure.
- Binary search trees
  - If we use the priority as a `key`, then we can use a combination of finding the minimum key and erase to implement `pop`. An ordinary binary-search-tree insert may be used to implement `push`.
  - This costs logarithmic time in the average case (and in the worst case as well if balancing is used).
- The latter is the better solution, but we would like to improve upon it — for example, it might be more natural if the minimum priority value were stored at the root.
  - Next lecture, we will achieve this with binary *heap*, giving up the complete ordering imposed in the binary *search tree*.