CSCI-1200 Data Structures — Fall 2020
Lecture 23 – Hash Tables, Part 2

Review from Lecture 22 & Lab 12
- “the single most important data structure known to mankind”
- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- Collision resolution: separate chaining
- Using a hash table to implement a set/map
  - Iterators, find, insert, and erase

Today’s Lecture
- Using STL’s for_each
- Something weird & cool in C++... Function Objects, a.k.a. Functors
- Hash Tables, part II
  - STL’s unordered_set (and unordered_map)
  - Hash functions as functors/function objects (or non-type template parameters, or function pointers)
  - Collision resolution: separate chaining vs. open addressing

23.1 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 alot 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::endl; });
```
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!
23.2 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.

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

23.3 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:

```cpp
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:

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

If we don’t specify a 3rd argument:

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

This is what STL does by default:

```cpp
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.

```cpp
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 (but for any type T)!
23.4 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:

```cpp
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:

```cpp
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.

```cpp
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;
}
```

“Weird Things we can do in C++” Finished – Now back to Hash Tables!

23.5 Hash Table in STL?

• The Standard Template Library standard and implementation of hash table have been slowly evolving over many years. Unfortunately, the names “hashset” and “hashmap” were spoiled by developers anticipating the STL standard, so to avoid breaking or having name clashes with code using these early implementations...

• STL’s agreed-upon standard for hash tables: unordered_set and unordered_map

• Depending on your OS/compiler, you may need to add the -std=c++11 flag to the compile line (or other configuration tweaks) to access these more recent pieces of STL. (And this will certainly continue to evolve in future years!)

• For many types STL has a good default hash function, so in those cases you do not need to provide your own hash function. But sometimes we do want to write our own...
23.6 Writing our own Hash Functions or Hash Functors

- Often the programmer/designer for the program using a hash function has the best understanding of the distribution of data to be stored in the hash function. Thus, they are in the best position to define a custom hash function (if needed) for the data & application.

- Here’s an example of a (generically) good hash function for STL strings:
  
  Note: This implementation comes from [http://www.partow.net/programming/hashfunctions/](http://www.partow.net/programming/hashfunctions/)

```cpp
unsigned int MyHashFunction(std::string const& key) {
    unsigned int hash = 1315423911;
    for(unsigned int i = 0; i < key.length(); i++)
        hash ^= ((hash << 5) + key[i] + (hash >> 2));
    return hash;
}
```

- Alternately, this same string hash code can be written as a functor – which is just a class wrapper around a function, and the function is implemented as the overloaded function call operator for the class.

```cpp
class MyHashFunctor {
    public:
        unsigned int operator() (std::string const& key) const {
            unsigned int hash = 1315423911;
            for(unsigned int i = 0; i < key.length(); i++)
                hash ^= ((hash << 5) + key[i] + (hash >> 2));
            return hash;
        }
};
```

- Once our new type containing the hash function is defined, we can create instances of our hash set object containing `std::string` by specifying the type `MyHashFunctor` as the second template parameter to the declaration of a `ds_hashset`. E.g.,

```cpp
ds_hashset<std::string, MyHashFunctor> my_hashset;
```

23.7 Using STL’s Associative Hash Table (Unordered Map)

- Using the default `std::string` hash function.
  - With no specified initial table size.
    ```cpp
    std::unordered_map<std::string,Foo> m;
    ```
  - Optionally specifying initial (minimum) table size.
    ```cpp
    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.
    ```cpp
    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”.
    ```cpp
    std::unordered_map<std::string,Foo,decltype(&MyHashFunction)> m(1000, MyHashFunction);
    ```

- Using a home-made `std::string` hash functor or function object.
  - With no specified initial table size.
    ```cpp
    std::unordered_map<std::string,Foo,MyHashFunctor> m;
    ```
  - Optionally specifying initial (minimum) table size.
    ```cpp
    std::unordered_map<std::string,Foo,MyHashFunctor> m(1000);)
    ```

- Note: In the above examples we’re creating an 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.
23.8 How do we Resolve Collisions? METHOD 1: Separate Chaining

NOTE: We used this method in the last lecture & in lab!

- Each table location stores a linked list of keys (and values) hashed to that location. Thus, the hashing function really just selects which list to search or modify.

- This works well when the number of items stored in each list is small, e.g., an average of 1. Other data structures, such as binary search trees, may be used in place of the list, but these have even greater overhead considering the (hopefully, very small) number of items stored per bin.

23.9 How do we Resolve Collisions? METHOD 2: Open Addressing

- Let’s eliminate the individual memory allocations and pointer indirection / dereferencing that are necessary for separate chaining. This will improve memory / data access performance.

- We will directly store the data (key/key-value pair) in the the top level vector, and store at most one item per index/location.

- When the chosen table index/location already stores a key (or key-value pair), we will seek a different table location to store the new value (or pair).

- Here are three different open addressing variations to handle a collision during an insert operation:
  - Linear probing: If i is the chosen hash location then the following sequence of table locations is tested (“probed”) until an empty location is found:
    
    \[(i+1)\%N, (i+2)\%N, (i+3)\%N, \ldots\]
  
  - Quadratic probing: If i is the hash location then the following sequence of table locations is tested:
    
    \[(i+1)\%N, (i+2^2)\%N, (i+3^2)\%N, \ldots\]

    More generally, the jth “probe” of the table is \( (i + c_1 j + c_2 j^2) \mod N \) where \( c_1 \) and \( c_2 \) are constants.

  - Secondary hashing: When a collision occurs a second hash function is applied to compute a new table location. If that location is also full, we go to a third hash function, etc. This is repeated until an empty location is found.

    We can generate a sequence/family of hash functions by swapping in a fixed random-like sequence of big (prime?) constants values into the same general function structure.

- For each of these approaches, the find operation follows the same sequence of locations as the insert operation. The key value is determined to be absent from the table only when an empty location is found.

- When using open addressing to resolve collisions, the erase function must mark a location as “formerly occupied”. If a location is instead marked empty, find may fail to return elements that are actually in the table. Formerly-occupied locations may (and should) be reused, but only after the find search operation has been run to completion to determine the item is definitely not in the table.

- Advantages of open addressing:
  - No linked lists! No pointers! It’s faster! (Indirect memory accesses are slow!)

- Problems with open addressing:
  - Fails completely when the table is full. **So we MUST resize before it gets full!**
  - Slows dramatically when the table is nearly full (e.g. about 80% or higher). This is particularly problematic for linear probing.
  - Memory cache performance can be poor when we are jumping around unpredictably in the top level array.
  - Cost of computing new hash values (linear < quadratic < secondary hashing).
  - Careful testing and parameter tuning is necessary to achieve optimal memory/speed performance.