Chapter 16

Maps

Data Structures in C++: Using the STL
Timothy Budd

- Associative Memories
- Key/Value Pairs
- Maps & Directories
- Multi-Maps
- Operations on Maps

Associative Memories

- Also called Content Addressable Memory or CAM
- A *key* or *name* is associated with every value stored in the memory
- The key contains information which helps locate the value in the memory
- A *query* presents a key to the CAM and receives the associated value
- Principle on which *cache* memories are based
- First built in hardware, now software implementations are also common
- Mapping of key to location in CAM is performed by a *hash function*

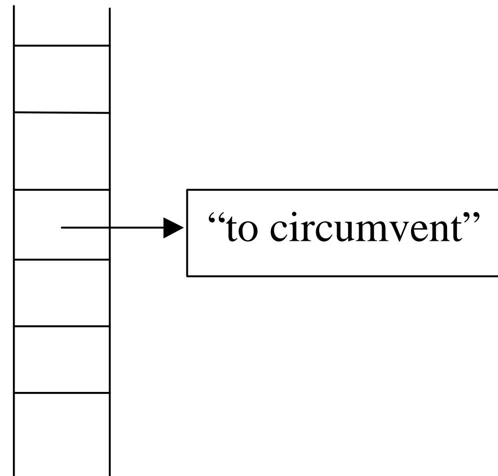
Key/Value Pairs

- Compilers can statically map variable names to addresses (locations in memory) at compile time.
- Suppose we wish to dynamically store and retrieve new values at run time:
  - Could use lists (worst case search $O(n)$)
  - Could use vectors (worst case search is $O(\log n)$, provided we keep it ordered)
  - We can do better, approaching constant search time (i.e. $O(1)$)
- A function which takes a key and produces the location in memory where the value is stored, is called a *hashing* function
  - Hashing functions run in constant time, hence the access time is constant $O(1)$
  - Hashing functions for *maps* must produce *unique* entries for each value stored in the *hash table* (no *collisions* allowed!)

Dictionary Example

Suppose we are to store definitions (as strings) of words in a Hash Table of size n. One useful hashing function would be to view every char. in the key as an 8-bit `int`. Sum all the `ints` and divide it modulo n.

usurp: ‘u’ = 117
‘s’ = 115
‘u’ = 117
‘r’ = 114
‘p’ = 112
total = 575 % n = i



If $h(k)$ produces a collision data is lost!

Maps & Multimaps

- Maps associate only one value with each key, so they don't handle collisions well
- Multimaps can store several values for each key, so collisions are well handled (each table entry basically holds a list!)
- It is the responsibility of the user to determine if the hash function will produce a collision for the set of data to be stored in the hash table

Map Example

```
#include <map>       // declares maps & multimaps
#include <string>    // used in examples below:

// a map, accessed by doubles, containing strings:
map<double, string>  map_one;

// our dictionary example:
map<string, string>  map_two;

// another simple mapping of integers onto integers:
map<int, int>  map_three;

// three typedef's are useful:
map<double, string>::iterator  loc;   // loc is an iterator
map<string, string>::key_type  k;     // k is a string
map<int, int>::value_type  v;         // v is an int

// maps & multimaps support the insert() operation.
// Note: the argument must be a key-value pair,
//         so the two arg constructor is called!
map_three.insert(map_three::value_type(5, 7));

// With maps (but not multimaps), values are accessed
// using the subscript operator:
map_three[1] = 6;
```

```
// Values can be removed by supplying a key:
map_three.erase(1);                // the 6 is now gone!

// or by supplying an iterator:
map<int, int>::iterator  loc = map_three.find(7);
map_three.erase(loc);

// Iterators, once dereferenced, return a pair:

template <class Key_type, class Value_type>
class pair
{
public:
        Key_type  first;
        Value_type  second;

        pair(Key_type initkey, Value_type initval)
                : first(initkey), second(initval) { }
        pair() { }

        bool operator < (pair<Key_type, Value_type>& p)
                { return first < p.first; }
};

// Use the names first, second to access data in the object:
cout << “key = “ << *loc.first << “ value = “ << *loc.second;
```

Implementation of Maps

Since we're short on time, we'll skip this topic!