Cache_LRU

Data structure that evicts the key-value pair of the least-recently-used (accessed) key when full.

Presented by Terry Janas

Why I needed a Cache_LRU

Application that consumes market data in real-time for all ticker symbols and writes to disk.

Market data for each ticker symbol is written to its own file.

Over 400K unique files per day, but the application can only keep about 4K files open at any given time.

Try to minimize file open/close/write calls.

First implementation attempt

```
template <class K, class V> class Simple_Cache_LRU
{ ...
  struct CacheData { unsigned long access; V Value; }
  unordered_map<K, CacheData> m_data_map;
  map<unsigned long, const K*> m_access_map;
  unsigned long m_access_counter;
  size_t m_maxsize;
  ...
};
```

Each key has an access value. When a new key is added to the cache, it is given an access value of m_access_counter++. When an existing key is accessed, its current access value is updated to be m_access_counter++.

This requires erasing the key from m_access_map by its prior access value, followed by inserting the key into m_access_map with its new access value.

m_access_map.begin() represents the least-recently-used key (via default map ordering of keys).

Second implementation attempt

```
template <class K, class V> class MappedList_Cache_LRU
{ ...
  typedef list< pair<const K*,V> > AccessList;
  AccessList m_access_list;
  unordered_map<K, typename AccessList::iterator> m_data_map;
  size_t m_maxsize;
  ...
};
```

No more access counter. Ordering of keys from least-recently-used to most-recently-used achieved with list. Value in map points to the key-value pair in the list. When a new key is added to the cache, the key-value pair is inserted at the front of m_access_list, and the key and iterator to this pair (e.g. m_access_list.begin()) is added to m_data_map. K* points to K in m_data_map. Accessing an existing key results in the list element being splice()'ed to the front of the list. m_access_list.**rbegin**() represents the least-recently-used key.

Performance is significantly increased from the first implementation attempt. For a <string,int> cache size of 10K keys (all strings with a uniform length of 6 bytes), and cache access/insert/evictions with a normal distribution of approximately 400K keys, the second implemention reduced was at least 15% faster.

Third (most successful) implementation

```
template <class K, class V> class Cache_LRU
{ ...
  typedef list< pair<const K*,V> > AccessList;
  AccessList m_access_list, m_access_list_pool;
  unordered_map<K, typename AccessList::iterator> m_data_map;
  size_t m_maxsize;
  ...
};
```

Added a list pool to the previous implementation. Since the max size of the cache is fixed, we can allocate the list nodes at startup, so pushing/popping items from the cache do not result in object creation/destruction. List elements are splice()'ed between the two lists.

Using the same test in the previous slide, performance was improved by an additional 15% (over 27% from the original implementation).

Questions?