A STL Hash Table Implementation

Bob Fraley
Hewlett-Packard Company
fraley@hpl.hp.com

This paper describes an implementation of an STL hash table.

1.0 Introduction

This paper describes an implementation of hash tables with the intent of verifying the proposed hash table extension that has been submitted to the C++ standards committee. My objectives in making this implementation were these:

- Test the proposed interface, and extend it as appropriate.
- Utilize STL structures in its construction
- Structure the program as best possible to simplify maintenance

The last point was important for me, since I knew that many changes would take place during development.

The interface that we defined closely resembles that of Associative Containers, except that conditions which require ordering of the elements were removed for hash tables, and complexity requirements of many operations were reduced from logarithmic to constant time.

This implementation works in constant time. That is, when the number of entries increases, the time to insert a new entry or find an existing entry remains within a constant bounds. The exception to this is that as the number of entries increases, the hash table size (number of buckets) must also increase. In this implementation the change in size takes place in a single insertion. The cost of this expansion, amortized over the number of insertions required to make it happen, is still an amortized constant. This one insertion, however, is definitely costly.

There are, of course, alternatives which will eliminate this occasional overhead. The one provided within this implementation is to periodically check the number of entries against the number of hash buckets, and call the resize operation when appropriate. This resize can take place when there is no load on the system (at night, for example). By adjusting the parameters appropriately, it becomes extremely unlikely that the resizing would ever occur automatically.

A more graceful alternative is to change the resizing algorithm to expand a single bucket on each insertion operation. This spreads the overhead of resizing over a number of inserts, avoiding the spike in insert time. The cost is a smaller increase in insert time, plus some additional overhead in searching the table while the resize is still in progress. This algorithm could be added, but is not part of this implementation at this time.

2.0 Hash Table Data Structures

The hash table is implemented as a list of entries, plus a vector of pointers into the list. A doubly-linked list is used, as that is the most convenient list available in the STL package. This implementation of hash tables therefore offers bidirectional iterators, which goes beyond the stated requirements for hash tables.

Issue

> Is this of any value to anyone? While there are some implementation complexities in extending this model to a singly linked list while preserving constant time for erase, it is possible, and should be done if there is no need for the reverse iterators.

Entries in the list consist of the user data (or pair<Key, Data> for maps) and a bucket number. The vector contains iterators pointing to the first list entry for each bucket, or list.end() if the bucket is currently unused.

The find(key) operation hashes the key value, then compares the key with the key of each entry in the bucket. When the stored bucket entry changes from the key's bucket, the search stops.

Following the model of set and map, the hash_multiset and hash_multimap change the insertion procedures to allow entries of equal keys to be inserted. Entries having equal keys are placed sequentially in the bucket. On resize, the order of equal entries is maintained. This leads to poor performance if a large number of equal entries are placed into the table. An improved data structure is under development.

Control of automatic resizing has been placed in the constructor. The current parameter structure is a first idea, and has not been refined. As my ideas for gradual resizing change, this may affect the parameters. Having read an article which increases the hash buckets by factors of 2, the growth size is selected as a power of 2. The trigger didn't seem to need the increased accuracy that floating point would offer, so an integer was selected.

3.0 Hash Functions

The hash function was included in the implementation with the expectation that some users would not want to provide their own. The initial hash function came from an article in the August 1994 C++ Report. This function computes a 2-byte hash function where the high-order byte depends only on the odd-number characters and the other only on the even-number characters. While this is satisfactory when there are an odd number of buckets (taking the traditional modulus function to scale down the hash value), it isn't compatible with gradual expansion schemes which grow the hash table by a factor of 2. (Since the low byte is independent of the first character, small tables would map all 1-character keys into a single bucket.)

To overcome both the bucket coverage issue and the size of hash value issue, I devised an alternative but similar algorithm. It works a bit slower, but seems to have slightly better coverage. Another approach, where each partial result is multiplied by a prime and added to the next character, has also been tried, is yet a bit slower, and provides slightly better coverage in most cases.

Issue

> I don't have very good tests for hash function analysis. If anyone is equipped to test these functions, I'd appreciate hearing about the results. If anyone can contribute a better hash function or algorithm, I'd appreciate that as well.

One unusual characteristic of this algorithm is that it computes a random shuffle of the integers from 0 to 255 before starting. There is no reason to do this; the shuffle could be pre-computed. I suspect that variations that I observe in timings arise because some shuffles are better for character string data than others. It would be nice to capture an effective shuffle and hard-wire it into the hash function.

There are two versions of this function available: one which works on null-terminated strings, the other on a data structure whose length is known. A third version is provided for Borland strings. When the key is a pointer (other than pointer to `char`,) be sure to provide your own hash function.

By the way, the hash functions were created as classes with in invoke operator, for no particular reason.

Issue

> Does anyone see an advantage to providing these as classes with invoke operations? Would pointers to functions be better?

I would appreciate any feed back on either design or implementation, including bug fixes.

4.0 Performance

Only an informal performance study has been done on hash tables. The study compared the performance of hash_set with that of set. A file containing 1483 unique words was used as input. To get larger tables, multiple copies of the file were used, changing some characters. Here are some preliminary results:

Table 1: Ratio of set to hash-set times, no resizing needed, 5000 buckets

# of entries	1483	2966	4422	8832	8832 (12017 buckets)
insertion	7.1	5.96	5.48	4.67	5.61
search	9.17	7.18	6.74	6.75	7.96
erase	1.0	1.45	1.29	1.57	1.2

Selecting a smaller table size to begin yields a smaller ratio of insert times, never worse than 3.72 in my trials.

An initial study using unsigned long parameters shows:

Table 2: Ratio of set to hash-set times, no resizing needed

# of entries	2000	10000	12000
insertion	1.2	1.8	1.9
search	1	1.5	2.0

In these results the hash function time was a substantial part of the result. The study was made with the number of buckets approximately equal to the number of entries. The standard deviation was large. The clock granularity prevented reading the results in some of the runs of 2000 entries.

No attempt was made to explain the answers shown in either table. Further study is needed.