Using TR1's unordered_set and unordered_map

David Kieras, EECS Department, University of Michigan

October 18, 2008

Since the first C++ Standard was approved in 1998, a group of C++ wizards have been working on the open-source Boost library (www.boost.org), with the goal of developing new library facilities good enough to be considered for addition to the C++ Standard Library. The Standard Library Technical Report No. 1 (TR1) specifies several Boost libraries that are scheduled to become part of the official C++ Standard Library in the next version of the Standard (expected to be completed in 2009). The original Standard Library did not include any containers based on hash tables, although hash tables have long been a popular way to get fast lookup times. TR1 specifies some unordered associative containers that are based on hash tables. This document provides a tutorial on `unordered_set` and `unordered_map`, and assumes that you already know how to use `std::set` and `std::map`. There is also `unordered_multiset` and `unordered_multimap`; using these is analogous to the unordered containers covered here and the corresponding ordered containers, and so will not be discussed.

Concept of the TR1 Unordered Containers

Hash Tables

The goal of a hash table container is get constant-time lookup of items, which will be faster than the logarithmic-time binary-tree or binary-search containers or algorithms. This goal can be accomplished by converting the key value for the sought-for item directly to an array index; if the array is big enough to have a cell for every possible key value, the solution is fast and easy. For example, such an arrangement is often used to classify characters for functions such as `isalpha()` and `isdigit()` - an array indexed by the 256 possible character values contains bits that show the attributes of each character. However, this direct approach is not practical if there are an extremely large number of possible keys and only an arbitrary and relatively small subset of them will actually be used. For example, there is no way to create an array big enough to hold all the possible character strings for use in a spell-checking dictionary. A hash table is a compromise: by mapping the key values to a smaller number of index values, we can get constant-time lookups under realistic conditions with a reasonable memory footprint.

A hash table has a set of buckets that can be indexed with an integer. The key value of the item to be stored is *hashed* by a function that converts the value into an integer with a wide range, and the item is placed in the bucket designated by the integer. Typically the index is computed by taking the hash value modulo the number of buckets. Many possible key values will be hashed to the same integer value, so each bucket will contain one or more items, kept in a container such as a list. Each new item placed in the bucket is added to the list. To retrieve an item, the key value of the desired item is hashed, and then the corresponding bucket is searched for the matching item, where "matching" means the item whose key compares as equal to the searched-for value.

As the number of items in the container increases, the hashing and bucket access will be constant-time, but the time to search the bucket for the matching item will be linear with the number of items in the bucket. To ensure that the lookup time remains roughly constant as the number of items increases, the number of buckets needs to increase, so that the number of items in each bucket stays small. Likewise, the hashing function must distribute items evenly among the buckets so that they each contain about the same small number of items. This means that unlike the ordered associative containers in the Standard Library, which have guaranteed logarithmic search times, there are no theoretical performance guarantees with the unordered containers. A poor choice of hash function and too few buckets can produce linear search time instead of constant time.

The two symptoms of a poorly performing hash table are unevenly filled buckets and overly full buckets. As a hash table is filled, its performance can be maintained by *rehashing* the table: increase the number of buckets, and then redistribute the items among the buckets. But this will not make up for a poor choice of hash function. To ensure good hash table performance, you have to test the table with the actual kinds of data involved to see if the number of items in the buckets remains small and equal, and "tune" the hash function if necessary.

Why "unordered?"

The TR1 unordered associative containers were designed as much as possible to have an identical interface to the corresponding ordered container. In particular, just like with the other containers, you can access every item in an unordered container by iterating through it from `begin()` to `end()`. However, because the unordered containers are based on hash-table logic, the contents are not in any natural order given by `operator<` or a supplied ordering function, but rather in a basically arbitrary order resulting from the hashing function and how the hash table buckets are filled. Thus, if you iterate through an unordered container, you will see each item, but in a random-looking order. This is why they are called `unordered_set` and `unordered_map`. [1]

Two more design goals were: First, to automate the rehashing function; if the average number of items per bucket (the *load factor*) gets too large, the container will automatically rehash; alternatively, you can tell the container to rehash to your specifications. Second, to help you "tune" the hash table: you can access the number of buckets, the number of items in each bucket, and the actual items in each bucket to help you troubleshoot and improve your choice of hash function.

To simplify using the unordered containers, the implementation must supply some default hash functions for the built-in types and the Standard Library `std::string` and `std::wstring` (wide string). You can supply your own hashing function for these or class types in the form of a function object class. In addition, by default the equality operators for these types are used in the search through the buckets for the matching item.

Using the Unordered Containers

How to access tr1::unordered_set and tr1:: unordered_map.

gcc 4.x comes with Boost's version of the TR1 library. The following #includes and using directives make the facilities described in this handout available in gcc 4.x. Other compiler/library environments will differ.

```
#include <tr1/unordered_set>
#include <tr1/unordered_map>

using std::tr1::unordered_set;
using std::tr1::unordered_map;
```

Using the containers with default hash functions

If you need a container to hold a built-in type of item or a `std:string` type, and are content to use the default hash functions and equality operators, then it could not be simpler. Just declare and use them analogously to the corresponding ordered containers. For example, here is a sketch of using an `unordered_set` of `int`:

```
unordered_set<int> uset;
...
uset.insert(int_value);    // will automatically rehash as needed
...
unordered_set<int>::iterator it = uset.find(int_value);
if(it != uset.end())
    cout << "found!" << endl;
else
    cout << "not found!" << endl;
...
for(unordered_set<int>::iterator it = uset.begin(); it != uset.end(); ++it)
    cout << *it << endl; // items appear in arbitrary order
```

[1] The names "hash_set" and "hash_map" might have been better, but they were already taken by widespread implementations incompatible with the proposed TR1 Standard.

Here is a sketch of using an unordered map of `string` to `int`, with an initial typedef to make life easier:

```
typedef unordered_map<string, int> Umap_t;
...
Umap_t umap;
...
umap[string_var] = int_var;
umap.insert(Umap_t::value_type("ddd", 4)); // use the std typedef for the pair

Umap_t::iterator it = umap.find(string_var);
if(it != umap.end())
    cout << "key: " << it->first << " has: " << it->second << endl;
else
    cout << "not found!" << endl;

for(Umap_t::const_iterator it = umap.begin(); it != umap.end(); ++it)
    cout << "[" << it->first << "," << it->second << "] " << endl;
```

Note that the pairs will appear in the output with the keys in some arbitrary order, depending on the hash function and how the buckets are filled.

Supplying a hash function and equality comparison

You can supply your own hash functions and equality comparisons, and in general, you must supply them for class types of your own. To specify the hash function, you define a function object class whose `operator()` accepts a `const` reference to the *key type* and returns a value that is of type `std::size_t`. The hash table code takes this returned integer value modulo the current number of buckets in the table to get the bucket index. Remember that the key type is the same as the object type in the set containers. Then supply this function object class name as the *second* template parameter when you declare the unordered container; the container template will create and use a function object of this class for the hash function instead of the default one.

When the bucket is searched for the matching item, the hash table needs to compare two keys for equality. If the key type has an `operator==`, it will be used by default. Otherwise, you must define another function object class whose `operator()` takes two `const` references to the key type and returns a `bool` that is `true` if the two keys should be considered equal. This class should be specified as the *third* template parameter in the declaration.

Here is a code sketch for an unordered_set of a class that defines `operator==`

```
class Thing {
public:
    bool operator== (const Thing&) const;
};

struct Hash_Thing { // hash function object class for Thing
    std::size_t operator() (const Thing& t) const
        { /* compute and return a size_t value using some property of Thing */}
};

unordered_set<Thing, Hash_Thing> uset_of_Things;
```

If the class does not have an equality operator, then here is a code sketch of how you would use an unordered_set:

```
class Thing {
public:
};

struct Hash_Thing { // hash function object class for Thing
    std::size_t operator() (const Thing& t) const
        { /* compute and return a size_t value using some property of Thing */}
};
```

```
struct Equal_Thing {    // equality function object class for Thing
    bool operator() (const Thing& t1, const Thing& t2) const
        { /* return true if t1 should be considered equal to t2 */}
};

unordered_set<Thing, Hash_Thing, Equal_Thing> uset_of_Things;
```

Naturally, your object class will need some kind of public accessors or friend declarations to allow the hash or equality functions to get the information it needs from the class objects. The Appendix has some code that provides a more complete example of providing your own hash functions.

Providing your own hash function and equality definition for an `unordered_map` would be done in exactly the same way, but the arguments to the function objects will have the type of the *key*. Since map containers often use a built-in type for the key, you are less likely to need to provide a hash or equality function object.

Facilities for tuning the hash function

The unordered containers have an interface that allows you to access or control the hash table parameters, display the number and contents of the buckets, and access and use the hashing function. These operations are done by making use of typedefs, iterators, and functions defined in the container interface. The example code below is for an `unordered_map` container typedef'd as `Umap_t` and named `umap`. The code would be the same for an `unordered_set` except that it would dereference an iterator to access items instead of accessing the first and second members pointed to by the iterator,

The code below illustrates how to get some basic statistics about the container. The load factor is a floating point value which is the average number of items per bucket. The max_load_factor is the load factor which is supposed to trigger a rehashing of the table (although exactly when it will be done depends on the implementation).

```
cout << "number of items: " << umap.size() << endl;
cout << "number of buckets: " << umap.bucket_count() << endl;
cout << "load factor: " << umap.load_factor() << endl;
cout << "max load factor: " << umap.max_load_factor() << endl;[2]
```

You can determine the number of items in each of the buckets with the `bucket_size()` member function that takes a bucket number and returns the number of items in it:

```
for(int i = 0; i < umap.bucket_count(); ++i) {
    cout << "bucket " << i << " contains "
        << umap.bucket_size(i) << " items " << endl;
    }
```

You can see what is in each bucket by iterating through it, using a typedef for a `local_iterator` that points to items in buckets, and a special `begin()` and `end()` accessors that take the bucket number as a parameter and return a `local_iterator` for the bucket. Thus the following prints the contents of each bucket in the entire container:

```
for(int i = 0; i < umap.bucket_count(); ++i) {
    cout << "bucket " << i << " contents: ";
    for(Umap_t::local_iterator it = umap.begin(i); it != umap.end(i); ++it) {
        cout << "[" << it->first << ", " << it->second << "] ";
        }
    cout << endl;
    }
```

[2] The gcc 4.0 implementation used at the time of this writing to prepare these examples produced a compile error in the tr1 library code when this function was called. Similarly, the rehash() function did not appear to be defined.

Finally, you can call the hash function directly to see what it does for key values. The containers have a typedef `hasher` for the type of the hash function, and an accessor `hash_function()` that returns a function pointer to it. For example, we could display the hashed value for each key in an `unordered_map` with:

```
Umap_t::hasher hfunc = umap.hash_function(); // get a pointer to the function

for(Umap_t::const_iterator it = umap.begin(); it != umap.end(); ++it) {
    cout << it->first << " hashes as " << hfunc(it->first) << endl;
    }
```

Performance of the Unordered Containers

The closest competition for the unordered containers are the corresponding ordered containers. The `set` and `unordered_set` containers were compared using the gcc 4.0 implementation. To study the effect of comparison time, the containers held either `Cheap` objects whose key was an integer, or `Expensive` objects, whose key was a long string whose initial characters were always the same. The keys were hashed with function objects that used the same code as the default hash functions in gcc 4.0. The integer hash was simply to convert the key value to `size_t`, while the string keys were hashed using an algorithm that combined all of the characters into a single number. The Appendix contains the classes and hash functions used in this comparison. A million searches were done, distributed evenly over all the objects in the container, for container sizes of {10, 50, 100, 500, 1000, 5000, 10,000, 50,000, 100,000, 500,000, 1,000,000}, and the run times collected on a Mac Pro (quad Intel processor 2.66 GHz, 2 GB) otherwise lightly loaded.

Figure 1 shows the time required for a million searches as a function of number of items in the container for `set` and `unordered_set` containers of `Cheap` and `Expensive` objects. The logarithmic search time for the `set` container is clear. The `unordered_set` times are much faster and almost flat, but increase in a step-like fashion possibly due to issues in the rehashing or to page faults from the large amount of memory required at these sizes. A similar effect step effect appears to be taking place with the set container. However. when the number of items is at all large, the unordered container is the winner by far.

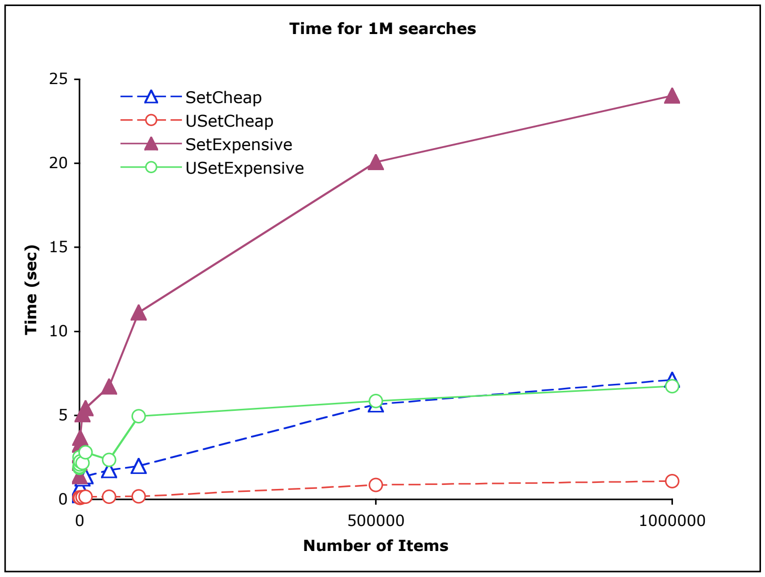

Figure 1. Search time for set and unordered_set of Cheap and Expensive objects, full range.

Figure 2 shows the same data, but zoomed in on the smallest values and using a logarithmic scale for the number of items. At 10 and 50 items, the `unordered_set<Expensive>` is actually slightly slower than `set<Expensive>`. This is probably because hashing the strings requires iterating through each character and doing some numerical work, and well as searching through the buckets for a matching item (which requires comparing all of the string characters). This is a relatively large amount of overhead for a small container. In contrast, the set container requires only a logarithmic number of comparisons, and no final comparison of equality, enabling it to do relatively well at a small size. This advantage disappears at 100 items. In contrast, `unordered_set<Cheap>`, involving only integer hashing, was faster even at the smallest container sizes.

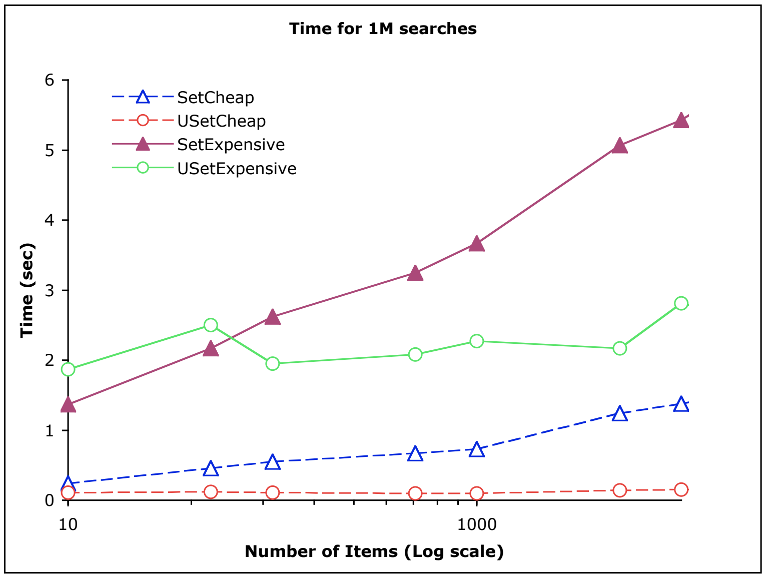

Figure 2. Search time for set and unordered_set of Cheap and Expensive objects, zoomed in to show detail at the smallest values of number of items (shown on a logarithmic scale)

As an additional note for how this particular `unordered_set` default rehashing implementation worked, the load factor was in the range of .90-.99, with the number of buckets always being somewhat more than the container.size. For the Cheap objects, the largest bucket was only 1 item, with 0 the smallest, while for the Expensive objects, the largest bucket size tended to increase with container size to about 8 or 9, with again 0 for the smallest. As the library is further developed, this implementation may change.

Conclusion

It is as easy to use the unordered containers as the ordered ones, and they can produce substantial speedup in search times. But to get this benefit, the default or your own hashing functions must work well for the data you have. Fortunately, the TR1 unordered container interface makes it relatively easy to test and tune your hash functions. Keep in mind that many implementations (such as gcc 4.0's) will be new, and so may be buggy.

Where to Read More

To read more about TR1 and the Boost libraries, try the documentation on the boost web site; it ranges from quite cryptic to very clear and useful. The following books were available at the time of this writing:

Becker, P. *The C++ Standard Library Extensions: A Tutorial and Reference*. Addison-Wesley, 2007. Becker is a widely-respected C++ expert, specializing in library implementations. This book gets into how TR1 is implemented more than just how to use it, and so is pretty difficult.

Karlsson, Bjorn. *Beyond the C++ Standard Library; An Introduction to Boost*. This book surveys a large part of the Boost library from the perspective of why and how you should use it to improve your code. TR1 is well covered. Fairly clear and practical, but beware: not all of the examples will compile in gcc's version of TR1, so be sure to try out his exact examples and get them working before going on to use the facilities.

Appendix: Classes used in the Performance Tests

```
using namespace std;

class Cheap {
public:
   Cheap(int i_ = -1) : i(i_) {}
   typedef int key_type;
   key_type get_key() const {return i;}
   bool operator< (const Cheap& rhs) const {return i < rhs.i;}
   bool operator== (const Cheap& rhs) const {return i == rhs.i;}
   friend ostream& operator<< (ostream& os, const Cheap& x);
   void print() const {cout << *this << endl;}
private:
   int i;
};

class Expensive {
public:
   Expensive(int i = -1)
   {
      // name strings the same length, differing only in the last few characters
      ostringstream oss;
      oss << "abcdefghijklmnopqrstuvwxyz" << 10000000 + i;
      s = oss.str();
   }
   typedef string key_type;
   const key_type& get_key() const {return s;}
   bool operator< (const Expensive& rhs) const {return s < rhs.s;}
   bool operator== (const Expensive& rhs) const {return s == rhs.s;}
   friend ostream& operator<< (ostream& os, const Expensive& x);
private:
   string s;
};

// A hash function object class template to be specialized for the different types
template <typename T>
struct MyHash {
   size_t operator()(const T&) const
      {assert(!"MyHash unspecialized operator() has been called!"); return 0;}
};

// a specialization for hashing Cheap objects
// using gcc 4.0 default hash function code for int
template <>
struct MyHash<Cheap> {
   size_t operator()(const Cheap& c) const
     {
      return static_cast<size_t>(c.get_key());   // just use the integer key value
     }
};
```

```
// a specialization for hashing Expensive objects
// using gcc 4.0 default hash function code for std::string
template <>
struct MyHash<Expensive> {
   size_t operator()(const Expensive& e) const
      {
       const string& s = e.get_key();
       size_t result = 0;
        for (string::const_iterator i = s.begin(); i != s.end(); ++i)
          result = (result * 131) + *i;
        return result;
      }
};


// A function object class for testing the performance of an set
// of objects of type T. The constructor takes a vector of objects and
// fills the container with them using the container constructor that takes
// a range specified with two iterators.
template <typename T>
class Set_tester {
public:
   Set_tester(const vector<T>& initial_load) :
      container(initial_load.begin(), initial_load.end())
      {assert(container.size() == initial_load.size());}

   bool operator() (const T& item)
      {return (container.find(item) != container.end());}

private:
   set<T> container;
};

// a similar class for testing an unordered_set container
template <typename T>
class USet_tester {
public:
   USet_tester(const vector<T>& initial_load) :
      container(initial_load.begin(), initial_load.end())
      {assert(container.size() == initial_load.size());}

   // search for the supplied item
   bool operator() (const T& item)
      {return (container.find(item) != container.end());}

private:
   unordered_set<T, MyHash<T> > container;
};
```