Inside STL: The unordered_map, unordered_set, unordered_multimap, and unordered_multiset

devblogs.microsoft.com/oldnewthing/20230808-00

August 8, 2023

Raymond Chen

The C++ standard library provides hash-based associative containers `unordered_map`, `unordered_set`, `unordered_multimap`, and `unordered_multiset`.

All of these collections are hash tables with different payloads. The `unordered_map` and the `unordered_multimap` use a `std::pair<Key, Value>` as the payload, whereas the the `unordered_set` and the `unordered_multiset` use a `Key` as the payload.

Conceptually, the hash table consists of a bunch of buckets, and each bucket contains a linked list of the nodes that fall into that bucket. (This design is known as *open hashing* or *separate chaining*.) However, that's not how the information is structured internally, because iterating through a traditionally-structured hash table requires more state than a single pointer: When you reach the end of each hash chain, you need some other information to tell you which chain to enumerate next.

C++ standard library implementations instead structure the hash table like this:

```
struct hashtable
{
    using hint = std::list<payload>::iterator;

    std::list<payload> list;
    std::vector<hint> buckets;
};
```

The `list` is a linked list of payloads, sorted by bucket. The `buckets` is a vector of iterators (pointers) into the list that tells you where each bucket begins. Each bucket implicitly ends when the next bucket begins, or (for the last bucket) when the end of the list is reached.

For example, suppose we have a hash table with four buckets, using the identity function as the hash function, and using "mod 4" as the bucket mapping function. Suppose that this hash table contains the values 2, 5, 6, 7. A traditional version of that hash table would look like a vector of lists:

buckets					
0					
1	•	→	5		
2	•	→	2	→	6
3	•	→	7		

However, in reality, the hash table looks like this:

5	→	2	→	6	→	7
↑		↑				↑
[0] [1]		[2]				[3]

In this diagram, we see that

- Bucket 0 consists of all the elements until we reach the start of bucket 1. But the start of bucket 1 is the same as the start of bucket 0, so bucket 0 is empty.
- Bucket 1 consists of all the elements until we reach the start of bucket 2, so that's just `5`.
- Bucket 2 consists of all the elements until we reach the start of bucket 3, so that's `2` and `6`.
- Bucket 3 consists of all the elements until we reach the end of the list, so that's `7`.

Another way of looking at this is that all of the bucket linked lists have been concatenated into a single linked list, and the `buckets` vector tells you how to break the giant list into individual bucket lists.

Internally, the standard library implementations often use bespoke versions of `list` and `vector` instead of the public ones. Furthermore, gcc and clang use a singly-linked list instead of the doubly-linked `std::list` I used here for expository purposes. The hash table-based collections do not support reverse iteration, so a forward list is sufficient.

When you're debugging, you're not really interested in the buckets. You just want to see what's in the hash table. To do that, dump the list. The Visual Studio debugger has a nice visualizer for these collections, but here's how you dig in. (Again, the hash object, the equality object, and the allocator are typically empty classes, so they are often stored as compressed pairs with other stuff.)

```
0:000> ?? s
class std::unordered_set<int,std::hash<int>,std::equal_to<int>,std::allocator<int> >
   +0x000 _Traitsobj       :
std::_Uset_traits<int,std::_Uhash_compare<int,std::hash<int>,std::equal_to<int>
>,std::allocator<int>,0>
   +0x008 _List            : std::list<int,std::allocator<int> >
   +0x018 _Vec             :
std::_Hash_vec<std::allocator<std::_List_unchecked_const_iterator<std::_List_val<std::
 >,std::_Iterator_base0> > >
   +0x030 _Mask            : 7
   +0x038 _Maxidx          : 8
0:000> ?? s._List
class std::list<int,std::allocator<int> >
   +0x000 _Mypair          :
std::_Compressed_pair<std::allocator<std::_List_node<int,void *>
>,std::_List_val<std::_List_simple_types<int> >,1>
0:000> ?? s._List._Mypair
class std::_Compressed_pair<std::allocator<std::_List_node<int,void *>
>,std::_List_val<std::_List_simple_types<int> >,1>
   +0x000 _Myval2          : std::_List_val<std::_List_simple_types<int> >
0:000> ?? s._List._Mypair._Myval2
class std::_List_val<std::_List_simple_types<int> >
   +0x000 _Myhead          : 0x000001b8`ad8a9280 std::_List_node<int,void *>
   +0x008 _Mysize          : 4
```

Okay, we finally dug into the `_List` to the point where we found the list head. Now we can use the usual list dumping commands to inspect the linked list.

```
0:000> dl 0x000001b8`ad8a9280
000001b8`ad8a9280  000001b8`ad8a9640 000001b8`ad8a84c0
000001b8`ad8a9290  baadf00d`baadf00d abababab`abababab ← garbage sentinel node
000001b8`ad8a9640  000001b8`ad8a92d0 000001b8`ad8a9280
000001b8`ad8a9650  baadf00d`00000002 abababab`abababab ← 2
000001b8`ad8a92d0  000001b8`ad8a8470 000001b8`ad8a9640
000001b8`ad8a92e0  baadf00d`00000003 abababab`abababab ← 3
000001b8`ad8a8470  000001b8`ad8a84c0 000001b8`ad8a92d0
000001b8`ad8a8480  baadf00d`00000005 abababab`abababab ← 5
000001b8`ad8a84c0  000001b8`ad8a9280 000001b8`ad8a8470
000001b8`ad8a84d0  baadf00d`00000007 abababab`abababab ← 7
```

The Windows debugger just dumps all the nodes right next to each other, so it's hard to see the boundaries between them. Here's what it looks like after I've added some spaces and annotations:

```
0:000> dl 0x000001b8`ad8a9280
000001b8`ad8a9280  000001b8`ad8a9640 000001b8`ad8a84c0
000001b8`ad8a9290  baadf00d`baadf00d abababab`abababab ← garbage sentinel node
                            ^garbage

000001b8`ad8a9640  000001b8`ad8a92d0 000001b8`ad8a9280
000001b8`ad8a9650  baadf00d`00000002 abababab`abababab ← 2
                            ^value

000001b8`ad8a92d0  000001b8`ad8a8470 000001b8`ad8a9640
000001b8`ad8a92e0  baadf00d`00000003 abababab`abababab ← 3
                            ^value

000001b8`ad8a8470  000001b8`ad8a84c0 000001b8`ad8a92d0
000001b8`ad8a8480  baadf00d`00000005 abababab`abababab ← 5
                            ^value

000001b8`ad8a84c0  000001b8`ad8a9280 000001b8`ad8a8470
000001b8`ad8a84d0  baadf00d`00000007 abababab`abababab ← 7
                            ^value
```

All of the techniques we used when debugging lists work here too, so I won’t repeat them.