Inside `boost::unordered_flat_map`

Introduction

Starting in Boost 1.81 (December 2022), Boost.Unordered provides, in addition to its previous implementations of C++ unordered associative containers, the new containers `boost::unordered_flat_map` and `boost::unordered_flat_set` (for the sake of brevity, we will only refer to the former in the remaining of this article). If `boost::unordered_map` strictly adheres to the C++ specification for `std::unordered_map`, `boost::unordered_flat_map` deviates in a number of ways from the standard to offer dramatic performance improvements in exchange; in fact, `boost::unordered_flat_map` ranks amongst the fastest hash containers currently available to C++ users.

We describe the internal structure of `boost::unordered_flat_map` and provide theoretical analyses and benchmarking data to help readers gain insights into the key design elements behind this container's excellent performance. Interface and behavioral differences with the standard are also discussed.

The case for open addressing

We have previously discussed why *closed addressing* was chosen back in 2003 as the implicit layout for `std::unordered_map`. 20 years after, *open addressing* techniques have taken the lead in terms of performance, and the fastest hash containers in the market all rely on some variation of open addressing, even if that means that some deviations have to be introduced from the baseline interface of `std::unordered_map`.

The defining aspect of open addressing is that elements are stored directly within the bucket array (as opposed to closed addressing, where multiple elements can be held into the same bucket, usually by means of a linked list of nodes). In modern CPU architectures, this layout is extremely cache friendly:

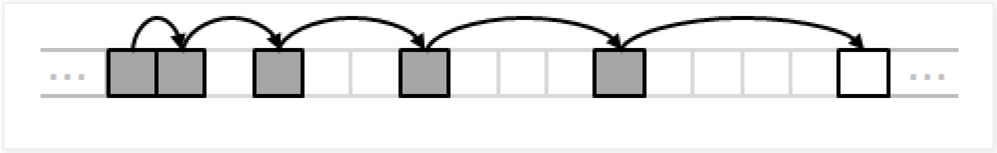
- There's no indirection needed to go from the bucket position to the element contained.
- Buckets are stored contiguously in memory, which improves cache locality.

The main technical challenge introduced by open addressing is what to do when elements are mapped into the same bucket, i.e. when a *collision* happens: in fact, all open-addressing variations are basically characterized by their collision management techniques. We can divide these techniques into two broad classes:

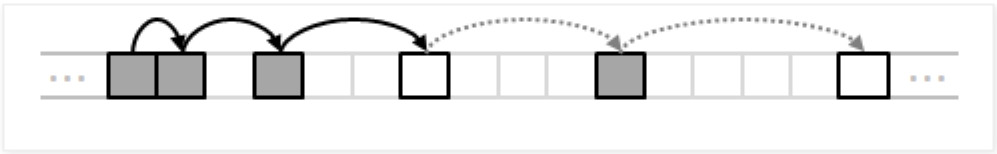
- **Non-relocating:** if an element is mapped to an occupied bucket, a *probing sequence* is started from that position until a vacant bucket is located, and the element is inserted there *permanently* (except, of course, if the element is deleted or if the bucket array is grown and elements *rehashed*). Popular probing mechanisms are *linear probing* (buckets inspected at regular intervals), *quadratic probing* and *double hashing*. There is a tradeoff between cache locality, which is better when the buckets probed are close to each other, and *average probe length* (the expected number of buckets probed until a vacant one is located), which grows larger (worse) precisely when probed buckets are close —elements tend to form clusters instead of spreading uniformly throughout the bucket array.
- **Relocating:** as part of the search process for a vacant bucket, elements can be moved from their position to make room for the new element. This is done in order to improve cache locality by keeping elements close to their "natural" location (that indicated by the hash → bucket mapping). Well known relocating algorithms are *cuckoo hashing*, *hopscotch hashing* and *Robin Hood hashing*.

If we take it as an important consideration to stay reasonably close to the original behavior of `std::unordered_map`, relocating techniques pose the problem that `insert` may invalidate iterators to other elements (so, they work more like `std::vector::insert`).

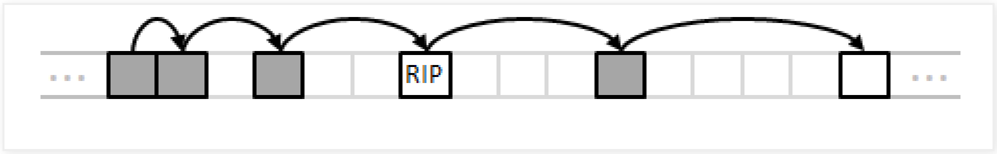
On the other hand, non-relocating open addressing faces issues on deletion: lookup starts at the original hash → bucket position and then keeps probing till the element is found *or probing terminates*, which is signalled by the presence of a vacant bucket:



So, erasing an element can't just restore its holding bucket as vacant, since that would preclude lookup from reaching elements further down the probe sequence:



A common techique to deal with this problem is to label buckets previously containing an element with a *tombstone* marker: tombstones are good for inserting new elements but do not stop probing on lookup:



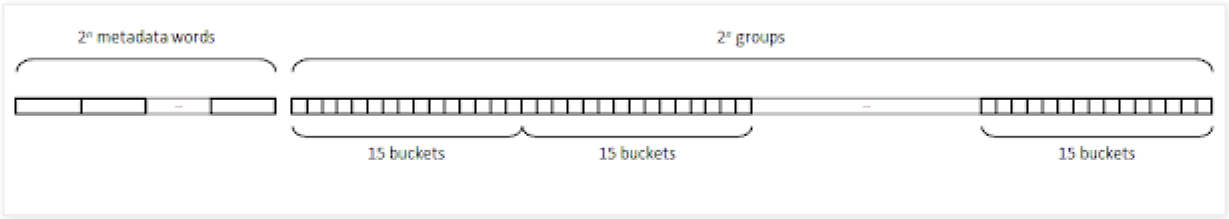
Note that the introduction of tombstones implies that the average lookup probe length of the container won't decrease on deletion —again, special measures can be taken to counter this.

SIMD-accelerated lookup

SIMD technologies, such as SSE2 and Neon, provide advanced CPU instructions for parallel arithmetic and logical operations on groups of contiguous data values: for instance, SSE2 `_mm_cmpeq_epi8` takes two packs of 16 bytes and compares them for equality *pointwise*, returning the result as another pack of bytes. Although SIMD was originally meant for acceleration of multimedia processing applications, the implementors of some unordered containers, notably Google's Abseil's Swiss tables and Meta's F14, realized they could leverage this technology to improve lookup times in hash tables.

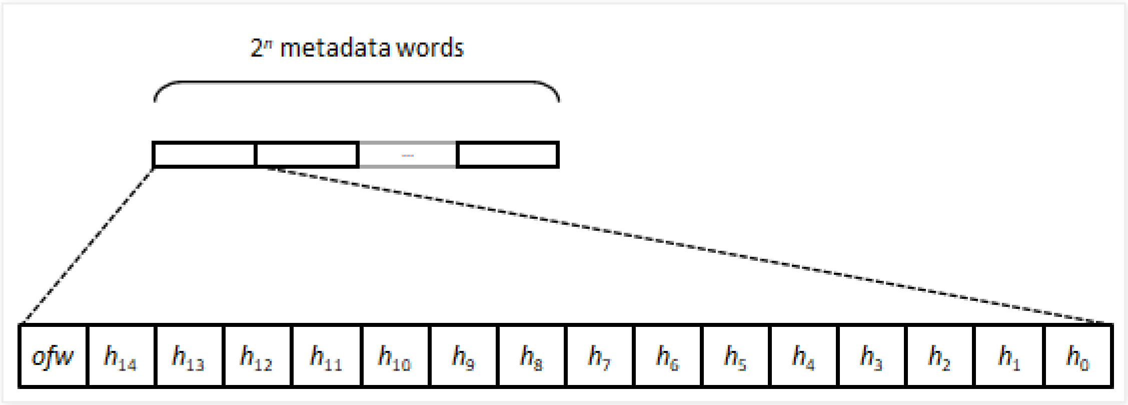
The key idea is to maintain, in addition to the bucket array itself, a separate *metadata array* holding *reduced hash values* (usually one byte in size) obtained from the hash values of the elements stored in the corresponding buckets. When looking up for an element, SIMD can be used on a pack of contiguous reduced hash values to quickly discard non-matching buckets and move on to full comparison for matching positions. This technique effectively checks a moderate number of buckets (16 for Abseil, 14 for F14) in constant time. Another beneficial effect of this approach is that special bucket markers (vacant, tombstone, etc.) can be moved to the metadata array —otherwise, these markers would take up extra space in the bucket itself, or else some representation values of the elements would have to be restricted from user code and reserved for marking purposes.

`boost::unordered_flat_map` data structure



`boost::unordered_flat_map`'s bucket array is logically split into 2^n groups of $N = 15$ buckets, and has a companion metadata array consisting of 2^n 16-byte words. Hash mapping is done at the group level rather than on individual buckets: so, to insert an element with hash value h, the group at position $h / 2^{W-n}$ is selected and its first available bucket used (W is 64 or 32 depending on whether the CPU architecture is 64- or 32-bit, respectively); if the group is full, further groups are checked using a quadratic probing sequence.

The associated metadata is organized as follows (least significant byte depicted rightmost):



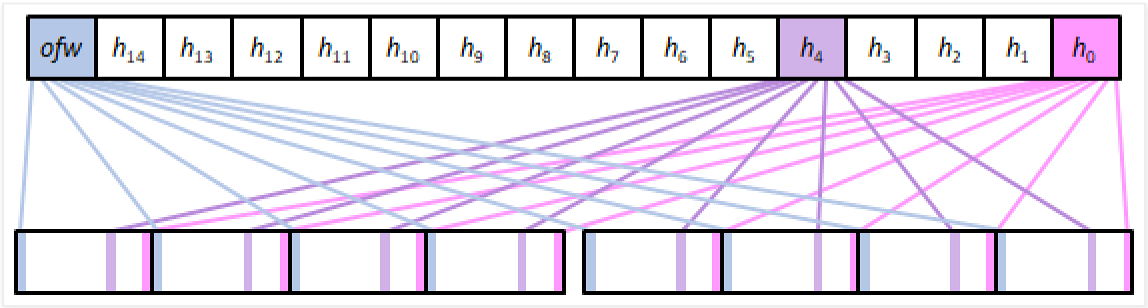
h_i holds information about the i-th bucket of the group:

- 0 if the bucket is empty,
- 1 to signal a *sentinel* (a special value at the end of the bucket array used to finish container iteration).
- otherwise, a reduced hash value in the range [2, 255] obtained from the least significant byte of the element's hash value.

When looking up within a group for an element with hash value h, SIMD operations, if available, are used to match the reduced value of h against the pack of values $\{h_0, h_1, ... , h_{14}\}$. Locating an empty bucket for insertion is equivalent to matching for 0.

ofw is the so-called *overflow byte*: when inserting an element with hash value h, if the group is full then the (h mod 8)-th bit of *ofw* is set to 1 before moving to the next group in the probing sequence. Lookup probing can then terminate when the corresponding overflow bit is 0. Note that this procedure removes the need to use tombstones.

If neither SSE2 nor Neon is available on the target architecture, the logical organization of metadata stays the same, but information is mapped to two physical 64-bit words using *bit interleaving* as shown in the figure:



Bit interleaving allows for a reasonably fast implementation of matching operations in the absence of SIMD.

Rehashing

The maximum load factor of `boost::unordered_flat_map` is 0.875 and can't be changed by the user. As discussed previously, non-relocating open addressing has the problem that average probe length doesn't decrease on deletion when the erased elements are in mid-sequence: so, continously inserting and erasing elements without triggering a rehash will slowly degrade the container's performance; we call this phenomenon *drifting*. `boost::unordered_flat_map` introduces the following anti-drift mechanism: rehashing is controled by the container's *maximum load*, initially 0.875 times the size of the bucket array; when erasing an element whose associated overflow bit is not zero, the maximum load is decreased by one. Anti-drift guarantees that rehashing will be eventually triggered in a scenario of repeated insertions and deletions.

Hash post-mixing

It is well known that open-addressing containers require that the hash function be of good quality, in the sense that close input values (for some natural notion of closeness) are mapped to distant hash values. In particular, a hash function is said to have the *avalanching property* if flipping a bit in the physical representation of the input changes all bits of the output value with probability 50%. Note that avalanching hash functions are extremely well behaved, and less stringent behaviors are generally good enough in most open-addressing scenarios.

Being a general-purpose container, `boost::unordered_flat_map` does not impose any condition on the user-provided hash function beyond what is required by the C++ standard for unordered associative containers. In order to cope with poor-quality hash functions (such as the identity for integral types), an automatic bit-mixing stage is added to hash values:

- 64-bit architectures: we use the `xmx` function defined in Jon Maiga's "The construct of a bit mixer".
- 32-bit architectures: the chosen mixer has been automatically generated by Hash Function Prospector and selected as the best overall performer in internal benchmarks. Score assigned by Hash Prospector: 333.7934929677524.

There's an opt-out mechanism available to end users so that avalanching hash functions can be marked as such and thus be used without post-mixing. In particular, the specializations of `boost::hash` for string types are marked as avalanching.

Statistical properties of `boost::unordered_flat_map`

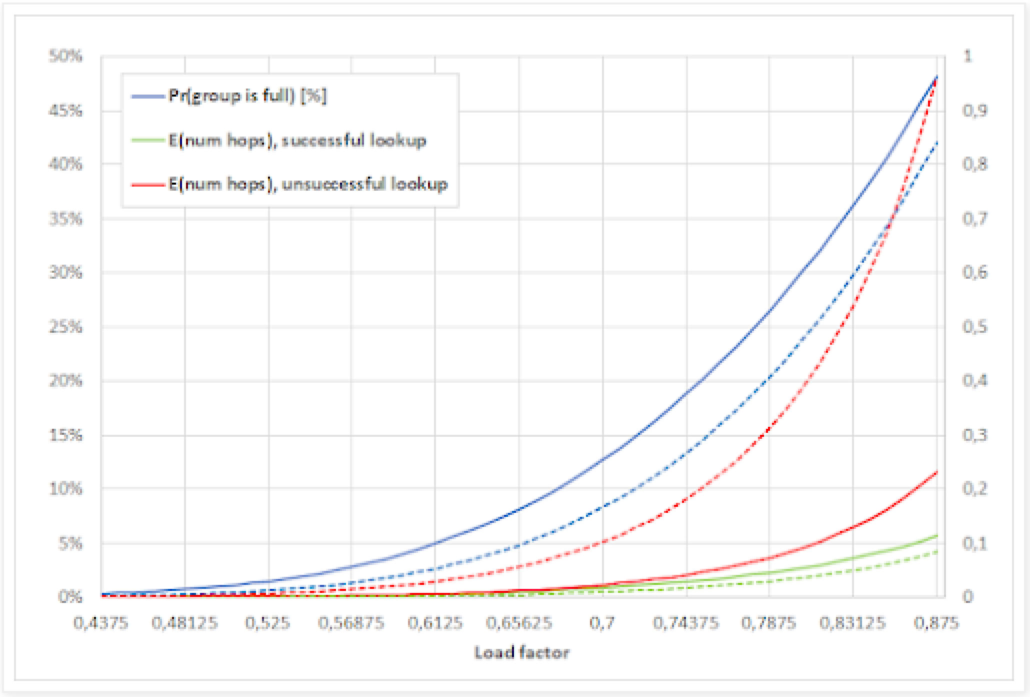
We have written a simulation program to calculate some statistical properties of `boost::unordered_flat_map` as compared with Abseil's `absl::flat_hash_map`, which is generally regarded as one of the fastest hash containers available. For the purposes of this analysis, the main design characteristics of `absl::flat_hash_map` are:

- Bucket array sizes are of the form 2^n, $n \geq 4$.
- Hash mapping is done at the bucket level (rather than at the group level as in `boost::unordered_flat_map`).
- Metadata consists of one byte per bucket, where the most significant bit is set to 1 if the bucket is empty, deleted (tombstone) or a sentinel. The remaining 7 bits hold the reduced hash value for occupied buckets.
- Lookup/insertion uses SIMD to inspect the 16 contiguous buckets beginning at the hash-mapped position, and then continues with further 16-bucket groups using quadratic probing. Probing ends when a non-full group is found. Note that the start positions of these groups are not aligned modulo 16.

The figure shows:

- the probability that a randomly selected group is full,
- the average number of hops (i.e. the average probe length minus one) for successful and unsuccessful lookup

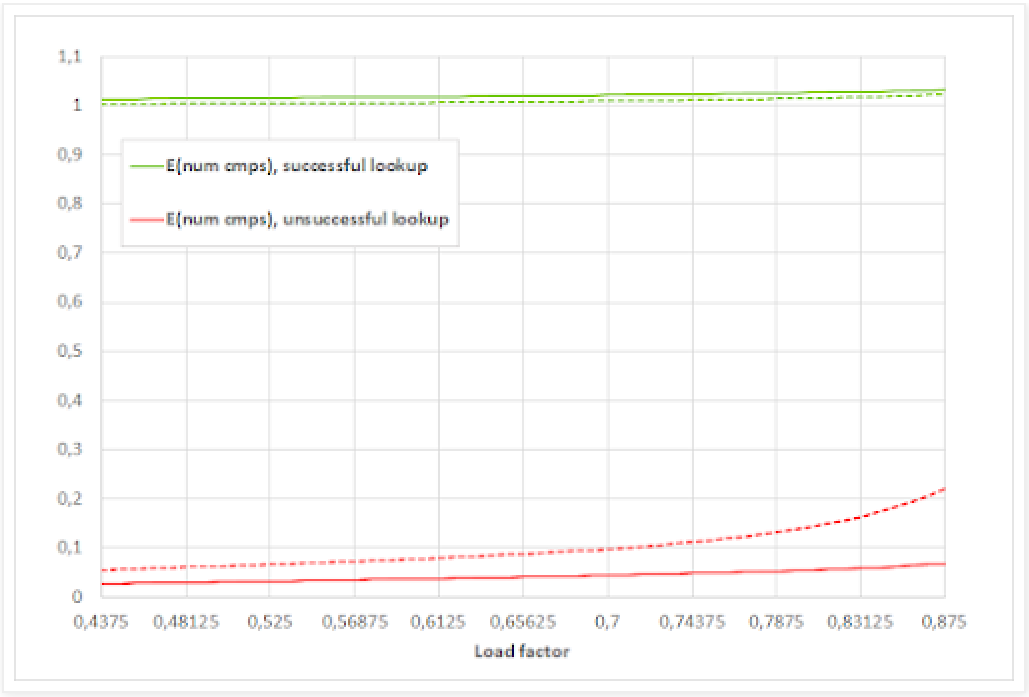
as functions of the load factor, with perfectly random input and without intervening deletions. Solid line is `boost::unordered_flat_map`, dashed line is `absl::flat_hash_map`.



Some observations:

- *Pr*(group is full) is higher for `boost::unordered_flat_map`. This follows from the fact that free buckets cluster at the end of 15-aligned groups, whereas for `absl::flat_hash_map` free buckets are uniformly distributed across the array, which increases the probability that a contiguous 16-bucket chunk contains at least one free position. Consequently, *E*(num hops) for successful lookup is also higher in `boost::unordered_flat_map`.
- By contrast, *E*(num hops) for *unsuccessful* lookup is considerably lower in `boost::unordered_flat_map`: `absl::flat_hash_map` uses an all-or-nothing condition for probe termination (group is non-full/full), whereas `boost::unordered_flat_map` uses the 8 bits of information in the overflow byte to allow for more finely-grained termination —effectively, making probe termination ~1.75 times more likely. The overflow byte acts as a sort of Bloom filter to check for probe termination based on reduced hash value.

The next figure shows the average number of actual comparisons (i.e. when the reduced hash value matched) for successful and unsuccessful lookup. Again, solid line is `boost::unordered_flat_map` and dashed line is `absl::flat_hash_map`.



E(num cmps) is a function of:

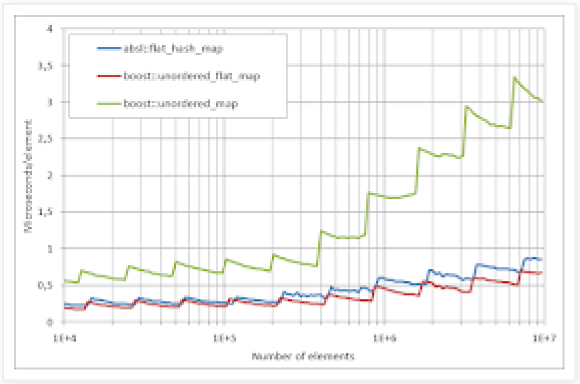
- *E*(num hops) (lower better),
- the size of the group (lower better),
- the number of bits of the reduced hash value (higher better).

We see then that `boost::unordered_flat_map` approaches `absl::flat_hash_map` on *E*(num cmps) for successful lookup (1% higher or less), despite its poorer *E*(num hops) figures: this is so because `boost::unordered_flat_map` uses smaller groups (15 vs. 16) and, most importantly, because its reduced hash values contain $\log_2(254) = 7.99$ bits vs. 7 bits in `absl::flat_hash_map`, and each additional bit in the hash reduced value decreases the number of negative comparisons roughly by half. In the case of *E*(num cmps) for unsuccessful lookup, `boost::unordered_flat_map` figures are up to 3.2 times lower under high-load conditions.

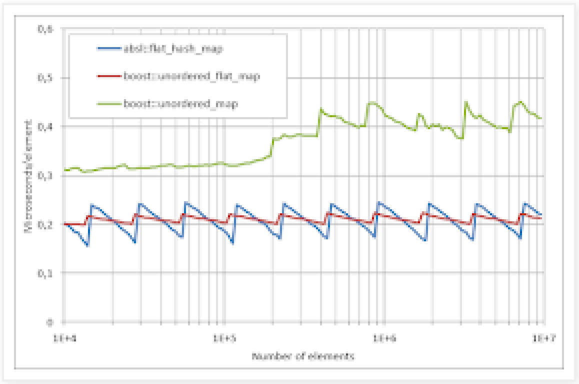
Benchmarks

Running-*n* plots

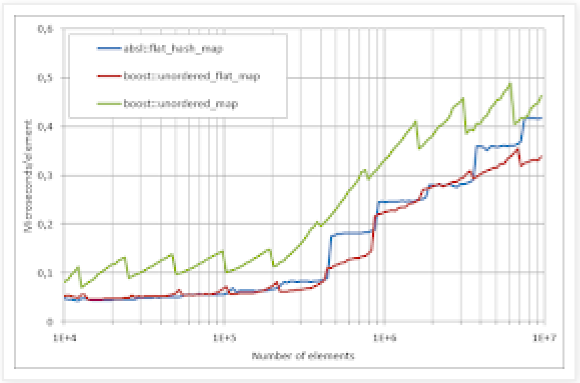
We have measured the execution times of `boost::unordered_flat_map` against `absl::flat_hash_map` and `boost::unordered_map` for basic operations (insertion, erasure during iteration, successful lookup, unsuccessful lookup) with container size *n* ranging from 10,000 to 10M. We provide the full benchmark code and results for different 64- and 32-bit architectures in a dedicated repository; here, we just show the plots for GCC 11 in x64 mode on an AMD EPYC Rome 7302P @ 3.0GHz. Please note that each container uses its own default hash function, so a direct comparison of execution times may be slightly biased.



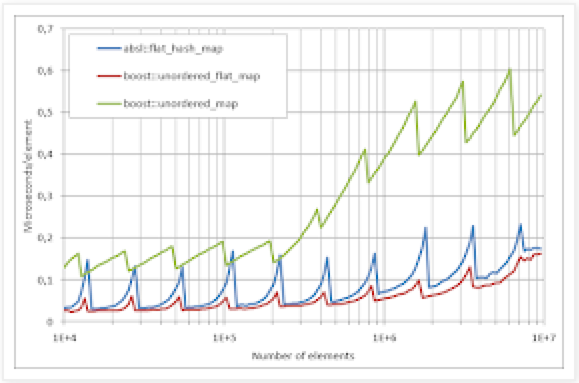
Running insertion



Running erasure



Successful lookup



Unsuccessful lookup

As predicted by our statistical analysis, `boost::unordered_flat_map` is considerably faster than `absl::flat_hash_map` for unsuccessful lookup because the average probe length and number of (negative) comparisons are much lower; this effect translates also to insertion, since `insert` needs to first check that the element is not present, so it internally performs an unsuccessful lookup. Note how performance is less impacted (stays flatter) when the load factor increases.

As for successful lookup, `boost::unordered_flat_map` is still faster, which may be due to its better cache locality, particularly for low load factors: in this situation, elements are clustered at the beginning portion of each group, while for `absl::flat_hash_map` they are uniformly distributed with more empty space in between.

`boost::unordered_flat_map` is slower than `absl::flat_hash_map` for runnning erasure (erasure of some elements during container traversal). The actual culprit here is iteration, which is particularly slow; this is a collateral effect of having SIMD operations work only on 16-aligned metadata words, while `absl::flat_hash_map` iteration looks ahead 16 metadata bytes beyond the current iterator position.

Aggregate performance

Boost.Unordered provides a series of benchmarks emulating real-life scenarios combining several operations for a number of hash containers and key types (`std::string`, `std::string_view`, `std::uint32_t`, `std::uint64_t` and a UUID class of size 16). The interested reader can build and run the benchmarks on her environment of choice; as an example, these are the results for GCC 11 in x64 mode on an Intel Xeon E5-2683 @ 2.10GHz:

```
std::string
               std::unordered_map: 38021 ms, 175723032 bytes in 3999509 allocations
             boost::unordered_map: 30785 ms, 149465712 bytes in 3999510 allocations
        boost::unordered_flat_map: 14486 ms, 134217728 bytes in 1 allocations
                  multi_index_map: 30162 ms, 178316048 bytes in 3999510 allocations
              absl::node_hash_map: 15403 ms, 139489608 bytes in 3999509 allocations
              absl::flat_hash_map: 13018 ms, 142606336 bytes in 1 allocations
       std::unordered_map, FNV-1a: 43893 ms, 175723032 bytes in 3999509 allocations
     boost::unordered_map, FNV-1a: 33730 ms, 149465712 bytes in 3999510 allocations
boost::unordered_flat_map, FNV-1a: 15541 ms, 134217728 bytes in 1 allocations
          multi_index_map, FNV-1a: 33915 ms, 178316048 bytes in 3999510 allocations
      absl::node_hash_map, FNV-1a: 20701 ms, 139489608 bytes in 3999509 allocations
      absl::flat_hash_map, FNV-1a: 18234 ms, 142606336 bytes in 1 allocations
```

```
std::string_view
               std::unordered_map: 38481 ms, 207719096 bytes in 3999509 allocations
             boost::unordered_map: 26066 ms, 181461776 bytes in 3999510 allocations
        boost::unordered_flat_map: 14923 ms, 197132280 bytes in 1 allocations
                  multi_index_map: 27582 ms, 210312120 bytes in 3999510 allocations
              absl::node_hash_map: 14670 ms, 171485672 bytes in 3999509 allocations
              absl::flat_hash_map: 12966 ms, 209715192 bytes in 1 allocations
       std::unordered_map, FNV-1a: 45070 ms, 207719096 bytes in 3999509 allocations
     boost::unordered_map, FNV-1a: 29148 ms, 181461776 bytes in 3999510 allocations
boost::unordered_flat_map, FNV-1a: 15397 ms, 197132280 bytes in 1 allocations
          multi_index_map, FNV-1a: 30371 ms, 210312120 bytes in 3999510 allocations
      absl::node_hash_map, FNV-1a: 19251 ms, 171485672 bytes in 3999509 allocations
      absl::flat_hash_map, FNV-1a: 17622 ms, 209715192 bytes in 1 allocations
```

```
std::uint32_t
       std::unordered_map: 21297 ms, 192888392 bytes in 5996681 allocations
     boost::unordered_map:  9423 ms, 149424400 bytes in 5996682 allocations
boost::unordered_flat_map:  4974 ms,  71303176 bytes in 1 allocations
          multi_index_map: 10543 ms, 194252104 bytes in 5996682 allocations
      absl::node_hash_map: 10653 ms, 123470920 bytes in 5996681 allocations
      absl::flat_hash_map:  6400 ms,  75497480 bytes in 1 allocations
```

```
std::uint64_t
       std::unordered_map: 21463 ms, 240941512 bytes in 6000001 allocations
     boost::unordered_map: 10320 ms, 197477520 bytes in 6000002 allocations
boost::unordered_flat_map:  5447 ms, 134217728 bytes in 1 allocations
          multi_index_map: 13267 ms, 242331792 bytes in 6000002 allocations
      absl::node_hash_map: 10260 ms, 171497480 bytes in 6000001 allocations
      absl::flat_hash_map:  6530 ms, 142606336 bytes in 1 allocations
```

```
uuid
       std::unordered_map: 37338 ms, 288941512 bytes in 6000001 allocations
     boost::unordered_map: 24638 ms, 245477520 bytes in 6000002 allocations
boost::unordered_flat_map:  9223 ms, 197132280 bytes in 1 allocations
          multi_index_map: 25062 ms, 290331800 bytes in 6000002 allocations
      absl::node_hash_map: 14005 ms, 219497480 bytes in 6000001 allocations
      absl::flat_hash_map: 10559 ms, 209715192 bytes in 1 allocations
```

Each container uses its own default hash function, except the entries labeled `FNV-1a` in `std::string` and `std::string_view`, which use the same implementation of Fowler–Noll–Vo hash, version 1a, and `uuid`, where all containers use the same user-provided function based on `boost::hash_combine`.

Deviations from the standard

The adoption of open addressing imposes a number of deviations from the C++ standard for unordered associative containers. Users should keep them in mind when migrating to `boost::unordered_flat_map` from `boost::unordered_map` (or from any other implementation of `std::unordered_map`):

- Both `Key` and `T` in `boost::unordered_flat_map<Key,T>` must be MoveConstructible. This is due to the fact that elements are stored directly into the bucket array and have to be transferred to a new block of memory on rehashing; by contrast, `boost::unordered_map` is a *node-based* container and elements are never moved once constructed.
- For the same reason, pointers and references to elements become invalid after rehashing (`boost::unordered_map` only invalidates iterators).
- `begin()` is not constant-time (the bucket array is traversed till the first non-empty bucket is found).
- `erase(iterator)` returns `void` rather than an iterator to the element after the erased one. This is done to maximize performance, as locating the next element requires traversing the bucket array; if that element is absolutely required, the `erase(iterator++)` idiom can be used. This performance issue is not exclusive to open addressing, and has been discussed in the context of the C++ standard too.
- The maximum load factor can't be changed by the user (`max_load_factor(z)` is provided for backwards compatibility reasons, but does nothing). Rehashing can occur *before* the load reaches `max_load_factor() * bucket_count()` due to the anti-drift mechanism described previously.
- There is no bucket API (`bucket_size`, `begin(n)`, etc.) save `bucket_count`.
- There are no node handling facilities (`extract`, etc.) Such functionality makes no sense here as open-addressing containers are precisely *not* node-based. `merge` is provided, but the implementation relies on element movement rather than node transferring.

Conclusions and next steps

`boost::unordered_flat_map` and `boost::unordered_flat_set` are the new open-addressing containers in Boost.Unordered providing top speed in exchange for some interface and behavioral deviations from the standards-compliant `boost::unordered_map` and `boost::unordered_set`. We have analyzed their internal data structure and provided some theoretical and practical evidence for their excellent performance. As of this writing, we claim `boost::unordered_flat_map`/`boost::unordered_flat_set` to rank among the fastest hash containers available to C++ programmers.

With this work, we have reached an important milestone in the ongoing Development Plan for Boost.Unordered. After Boost 1.81, we will continue improving the functionality and performance of existing containers and will possibly augment the available container catalog to offer greater freedom of choice to Boost users. Your feedback on our current and future work is much welcome.

Posted by Joaquín M López Muñoz at 1:23 PM 4 comments :

Sunday, October 2, 2022

Deferred argument evaluation

Suppose our program deals with heavy entities of some type `object` which are uniquely identified by an integer ID. The following is a possible implementation of a function that controls ID-constrained creation of such objects:

```
object* retrieve_or_create(int id)
{
  static std::unordered_map<int, std::unique_ptr<object>> m;

  // see if the object is already in the map
  auto [it,b] = m.emplace(id, nullptr);
  // create it otherwise
  if(b) it->second = std::make_unique<object>(id);
  return it->second.get();
}
```

Note that the code is careful not to create a spurious object if an equivalent one already exists; but in doing so, we have introduced a potentially inconsistency in the internal map if object creation throws:

```
// fixed version

object* retrieve_or_create(int id)
{
  static std::unordered_map<int, std::unique_ptr<object>> m;

  // see if the object is already in the map
  auto [it,b] = m.emplace(id, nullptr);
  // create it otherwise
  if(b){
    try{
      it->second = std::make_unique<object>(id);
    }
    catch(...){
      // we can get here when running out of memory, for instance
      m.erase(it);
      throw;
    }
  }
  return it->second.get();
}
```

This fixed version is a little cumbersome, to say the least. Starting in C++17, we can use `try_emplace` to rewrite `retrieve_or_create` as follows:

```
object* retrieve_or_create(int id)
{
  static std::unordered_map<int, std::unique_ptr<object>> m;

  auto [it,b] = m.try_emplace(id, std::make_unique<object>(id));
  return it->second.get();
}
```

But then we've introduced the problem of spurious object creation we strived to avoid. Ideally, we'd like for `try_emplace` to **not** create the object except when really needed. What we're effectively asking for is some sort of technique for *deferred argument evaluation*. As it happens, it is very easy to devise our own:

```
template<typename F>
struct deferred_call
{
  using result_type=decltype(std::declval<const F>()());
  operator result_type() const { return f(); }

  F f;
};

object* retrieve_or_create(int id)
{
  static std::unordered_map<int, std::unique_ptr<object>> m;

  auto [it,b] = m.try_emplace(
    id,
    deferred_call([&]{ return std::make_unique<object>(id); }));
  return it->second.get();
}
```

`deferred_call` is a small utlity that computes a value upon request of conversion to `deferred_call::result_type`. In the example, such conversion will only happen if `try_emplace` really needs to create a `std::pair<const int, std::unique_ptr<object>>`, that is, if no equivalent object was already present in the map.

In a general setting, for `deferred_call` to work as expected, that is, to delay producing the value until the point of actual usage, the following conditions must be met:

1. The `deferred_call` object is passed to function/constructor template accepting generic, unconstrained parameters.
2. All internal intermediate interfaces are also generic.
3. The final function/constructor where actual usage happens asks exactly for a `deferred_call::result_type` value or reference.

It is the last condition that can be the most problematic:

```
void f(std::string);

// error: deferred_call not convertible to std::string
f(deferred_call([]{ return "hello"; }));
```

C++ rules for conversion alows just **one** user-defined conversion to take place at most, and here we are calling for the sequence `deferred_call` → `const char*` → `std::string`. In this case, however, the fix is trivial:

```
void f(std::string);

f(deferred_call([]{ return std::string("hello"); }));
```

Update Oct 4

Jessy De Lannoit proposes a variation on `deferred_call` that solves the problem of producing a value that is one user-defined conversion away from the target type:

```
template<typename F>
struct deferred_call
{
  using result_type=decltype(std::declval<const F>()());
  operator result_type() const { return f(); }

  template<typename T>
  requires (std::is_constructible_v<T, result_type>)
  constexpr operator T() const { return {f()}; }

  F f;
};

void f(std::string);

// works ok: deferred_call converts to std::string
f(deferred_call([]{ return "hello"; }));
```

This version of `deferred_call` has an eager conversion operator producing any requested value as long as it is constructible from `deferred_call::result_type`. The solution comes with a different set of problems, though:

```
void f(std::string);
void f(const char*);

// ambiguous call to f
f(deferred_call([]{ return "hello"; }));
```

There is probably little more we can do without language support. One can imagine some sort of "silent" conversion operator that does not add to the cap on user-defined conversions allowed by the rules of C++:

```
template<typename F>
struct deferred_call
{
  using result_type=decltype(std::declval<const F>()());
  operator result_type() const { return f(); }

  // "silent" conversion operator marked with ~explicit
  // (not actual C++)
  template<typename T>
  requires (std::is_constructible_v<T, result_type>)
  ~explicit constexpr operator T() const { return {f()}; }

  F f;
};
```

Posted by Joaquín M López Muñoz at 6:14 PM   No comments :

Saturday, June 18, 2022

Advancing the state of the art for `std::unordered_map` implementations

Introduction

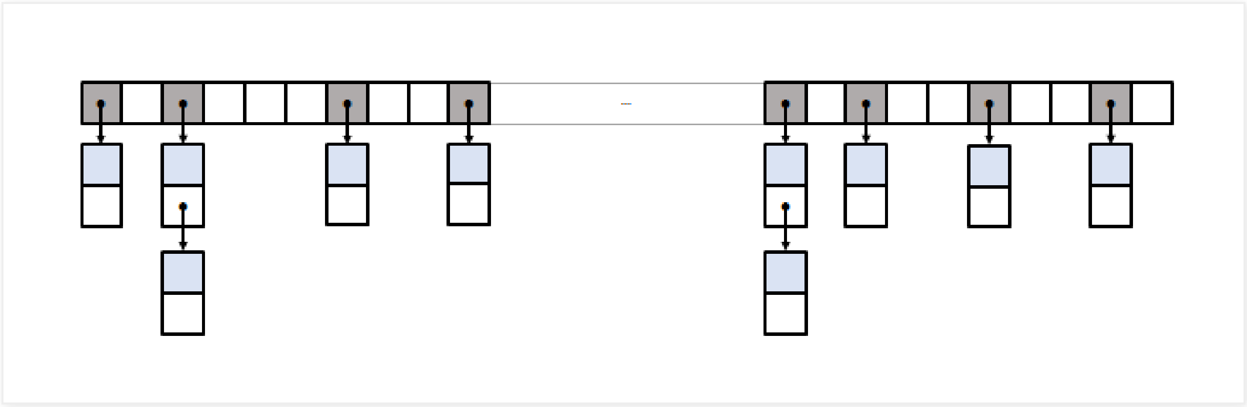
Several Boost authors have embarked on a project to improve the performance of Boost.Unordered's implementation of `std::unordered_map` (and `multimap`, `set` and `multiset` variants), and to extend its portfolio of available containers to offer faster, non-standard alternatives based on open addressing.

The first goal of the project has been completed in time for Boost 1.80 (due August 2022). We describe here the technical innovations introduced in `boost::unordered_map` that makes it the fastest implementation of `std::unordered_map` on the market.

Closed vs. open addressing

On a first approximation, hash table implementations fall on either of two general classes:

- *Closed addressing* (also known as *separate chaining*) relies on an array of *buckets*, each of which points to a list of elements belonging to it. When a new element goes to an already occupied bucket, it is simply linked to the associated element list. The figure depicts what we call the *textbook implementation* of closed addressing, arguably the simplest layout, and among the fastest, for this type of hash tables.



- *Open addressing* (or *closed hashing*) stores at most one element in each bucket (sometimes called a *slot*). When an element goes to an already occupied slot, some *probing* mechanism is used to locate an available slot, preferrably close to the original one.

Recent, high-performance hash tables use open addressing and leverage on its inherently better cache locality and on widely available SIMD operations. Closed addressing provides some functional advantages, though, and remains relevant as the required foundation for the implementation of `std::unodered_map`.

Restrictions on the implementation of `std::unordered_map`

The standardization of C++ unordered associative containers is based on Matt Austern's 2003 N1456 paper. Back in the day, open-addressing approaches were not regarded as sufficiently mature, so closed addressing was taken as the safe implementation of choice. Even though the C++ standard does not explicitly require that closed addressing must be used, the assumption that this is the case leaks through the public interface of `std::unordered_map`:

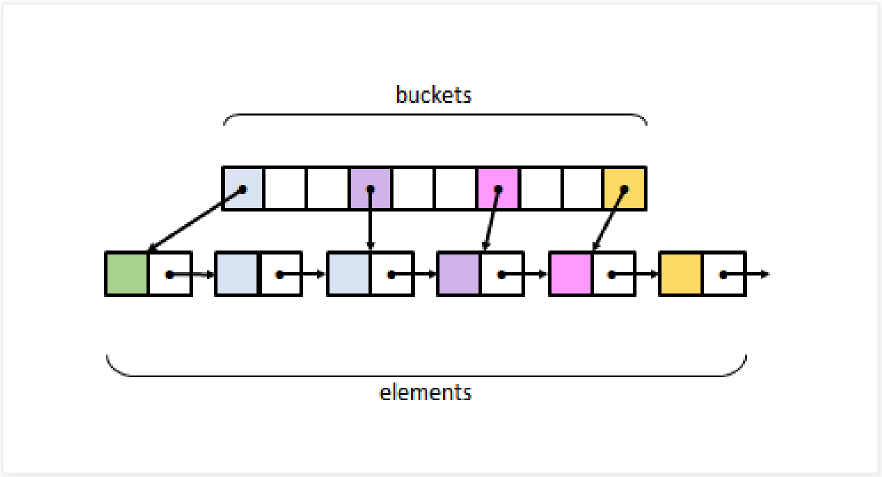
- A bucket API is provided.
- Pointer stability implies that the container is node-based. In C++17, this implication was made explicit with the introduction of `extract` capabilities.
- Users can control the container load factor.
- Requirements on the hash function are very lax (open addressing depends on high-quality hash functions with the ability to spread keys widely across the space of `std::size_t` values.)

As a result, all standard library implementations use some form of closed addressing for the internal structure of their `std::unordered_map` (and related containers).

Coming as an additional difficulty, there are two complexity requirements:

- iterator increment must be (amortized) constant time,
- `erase` must be constant time on average,

that rule out the textbook implementation of closed addressing (see N2023 for details). To cope with this problem, standard libraries depart from the textbook layout in ways that introduce speed and memory penalties: this is, for instance, how libstdc++-v3 and libc++ layouts look like:



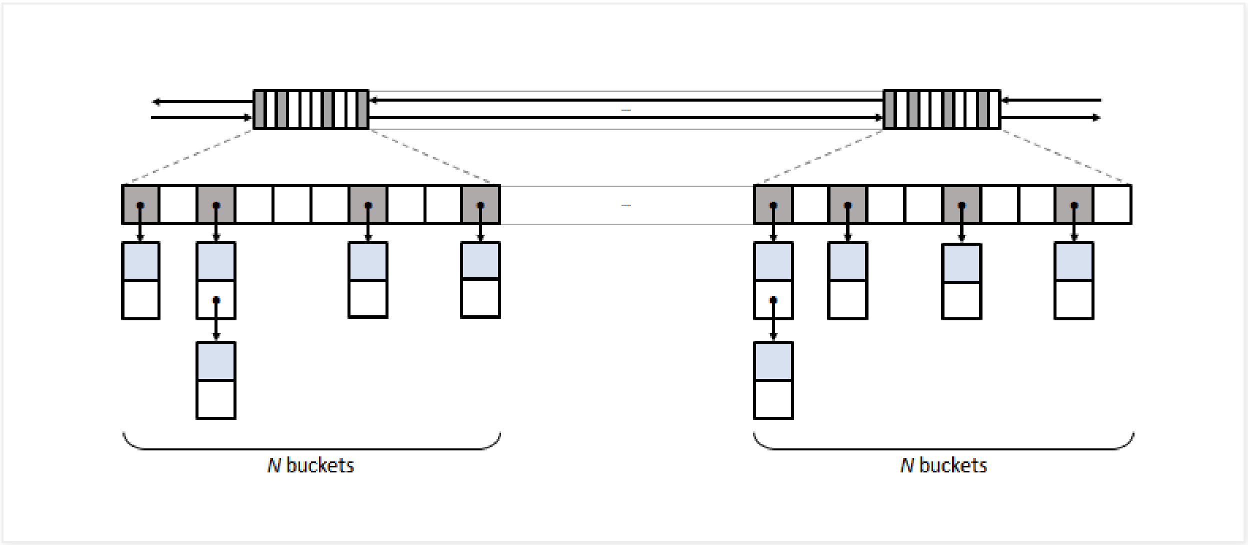
To provide constant iterator increment, all nodes are linked together, which in its turn forces two adjustments to the data structure:

- Buckets point to the node *before* the first one in the bucket so as to preserve constant-time erasure.
- To detect the end of a bucket, the element hash value is added as a data member of the node itself (libstdc++-v3 opts for on-the-fly hash calculation under some circumstances).

Visual Studio standard library (formerly from Dinkumware) uses an entirely different approach to circumvent the problem, but the general outcome is that resulting data structures perform significantly worse than the textbook layout in terms of speed, memory consumption, or both.

Boost.Unordered 1.80 data layout

The new data layout used by Boost.Unordered goes back to the textbook approach:



Unlike the rest of standard library implementations, nodes are not linked across the container but only within each bucket. This makes constant-time `erase` trivially implementable, but leaves unsolved the problem of constant-time iterator increment: to achieve it, we introduce so-called *bucket groups* (top of the diagram). Each bucket group consists of a 32/64-bit bucket occupancy mask plus `next` and `prev` pointers linking non-empty bucket groups together. Iteration across buckets resorts to a combination of bit manipulation operations on the bitmasks plus group traversal through `next` pointers, which is not only constant time but also very lightweight in terms of execution time and of memory overhead (4 bits per bucket).

Fast modulo

When inserting or looking for an element, hash table implementations need to map the element hash value into the array of buckets (or slots in the open-addressing case). There are two general approaches in common use:

- Bucket array sizes follow a sequence of prime numbers p, and mapping is of the form $h \to h \bmod p$.
- Bucket array sizes follow a power-of-two sequence 2^n, and mapping takes n bits from h. Typically it is the n least significant bits that are used, but in some cases, like when h is postprocessed to improve its uniformity via multiplication by a well-chosen constant m (such as defined by Fibonacci hashing), it is best to take the n *most* significant bits, that is, $h \to (h \times m) >> (N - n)$, where N is the bitwidth of `std::size_t` and $>>$ is the usual C++ right shift operation.

We use the modulo by a prime approach because it produces very good spreading even if hash values are not uniformly distributed. In modern CPUs, however, modulo is an expensive operation involving integer division; compilers, on the other hand, know how to perform modulo *by a constant* much more efficiently, so one possible optimization is to keep a table of pointers to functions $f_p : h \to h \bmod p$. This technique replaces expensive modulo calculation with a table jump plus a modulo-by-a-constant operation.

In Boost.Unordered 1.80, we have gone a step further. Daniel Lemire et al. show how to calculate h mod p as an operation involving some shifts and multiplications by p and a pre-computed c value acting as a sort of reciprocal of p. We have used this work to implement hash mapping as $h \rightarrow \text{fastmod}(h, p, c)$ (some details omitted). Note that, even though fastmod is generally faster than modulo by a constant, most performance gains actually come from the fact that we are eliminating the table jump needed to select f_p, which prevented code inlining.

Time and memory performance of Boost 1.80 `boost::unordered_map`

We are providing some benchmark results of the `boost::unordered_map` against libstdc++-v3, libc++ and Visual Studio standard library for insertion, lookup and erasure scenarios. `boost::unordered_map` is mostly faster across the board, and in some cases significantly so. There are three factors contributing to this performance advantage:

- the very reduced memory footprint improves cache utilization,
- fast modulo is used,
- the new layout incurs one less pointer indirection than libstdc++-v3 and libc++ to access the elements of a bucket.

As for memory consumption, let N be the number of elements in a container with B buckets: the memory overheads (that is, memory allocated minus memory used strictly for the elements themselves) of the different implementations on 64-bit architectures are:

Implementation	Memory overhead (bytes)
libstdc++-v3	$16\,N + 8\,B$ (hash caching) $8\,N + 8\,B$ (no hash caching)
libc++	$16\,N + 8\,B$
Visual Studio (Dinkumware)	$16\,N + 16\,B$
Boost.Unordered	$8\,N + 8.5\,B$

Which hash container to choose

Opting for closed-addressing (which, in the realm of C++, is almost synonymous with using an implementation of `std::unordered_map`) or choosing a speed-oriented, open-addressing container is in practice not a clear-cut decision. Some factors favoring one or the other option are listed:

- `std::unordered_map`
  - The code uses some specific parts of its API like node extraction, the bucket interface or the ability to set the maximum load factor, which are generally not available in open-addressing containers.
  - Pointer stability and/or non-moveability of values required (though some open-addressing alternatives support these at the

    expense of reduced performance).
    - Constant-time iterator increment required.
    - Hash functions used are only mid-quality (open addressing requires that the hash function have very good key-spreading properties).
    - Equivalent key support, ie. `unordered_multimap`/`unordered_multiset` required. We do not know of any open-addressing container supporting equivalent keys.
- Open-addressing containers
    - Performance is the main concern.
    - Existing code can be adapted to a basically more stringent API and more demanding requirements on the element type (like moveability).
    - Hash functions are of good quality (or the default ones from the container provider are used).

If you decide to use `std::unordered_map`, Boost.Unordered 1.80 now gives you the fastest, fully-conformant implementation on the market.

Next steps

There are some further areas of improvement to `boost::unordered_map` that we will investigate post Boost 1.80:

- Reduce the memory overhead of the new layout from 4 bits to 3 bits per bucket.
- Speed up performance for equivalent key variants (`unordered_multimap`/`unordered_multiset`).

In parallel, we are working on the future `boost::unordered_flat_map`, our proposal for a top-speed, open-addressing container beyond the limitations imposed by `std::unordered_map` interface. Your feedback on our current and future work is much welcome.

Posted by Joaquín M López Muñoz at 9:16 PM   No comments :

Thursday, March 10, 2022

Emulating template named arguments in C++20

`std::unordered_map` is a highly configurable class template with five parameters:

```
template<
    class Key,
    class Value,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, Value> >
> class unordered_map;
```

Typical usage depends on default values for most of these parameters:

```
using my_map=std::unordered_map<int,std::string>;
```

but things get cumbersome when we want to specify one of the usually defaulted types:

```
template<typename T> class my_allocator{ ... };
using my_map=std::unordered_map<
  int, std::string,
  std::hash<int>, std::equal_to<int>,
  my_allocator< std::pair<const int, std::string> >
>;
```

In the example, we are forced to specify the hash and equality predicate with their default value types just to get to the allocator, which is the parameter we really wanted to specify. Ideally we would like to have a syntax like this:

```
// this is not actual C++
using my_map = std::unordered_map<
  Key=int, Value=std::string,
  Allocator=my_allocator< std::pair<const int, std::string> >
>;
```

Turns out we can emulate this by resorting to *designated initializers*, introduced in C++20:

```
template<
  typename Key, typename Value,
  typename Hash = std::hash<Key>,
  typename Equal = std::equal_to<Key>,
  typename Allocator = std::allocator< std::pair<const Key,Value> >
>
struct unordered_map_config
{
  Key       *key = nullptr;
  Value     *value = nullptr;
  Hash      *hash = nullptr;
  Equal     *equal = nullptr;
  Allocator *allocator = nullptr;

  using type = std::unordered_map<Key,Value,Hash,Equal,Allocator>;
};

template<typename T>
constexpr T *type = nullptr;

template<unordered_map_config Cfg>
using unordered_map = typename decltype(Cfg)::type;

...

using my_map = unordered_map<{
  .key = type<int>, .value = type<std::string>,
  .allocator = type< my_allocator< std::pair<const int, std::string > > >
}>;
```

The approach taken by the simulation is to use designated initializers to create an aggregate object consisting of dummy null pointers: the values of the pointers do not matter, but their types are captured via CTAD and used to synthesize the associated `std::unordered_map` instantiation. Two more C++20 features this technique depends on are:

- Non-type template parameters have been extended to accept *literal types* (which include aggregate types such as `unordered_map_config` instantiations).
- The class template `unordered_map_config` can be specified as a non-type template parameter of `unordered_map`. In C++17, we would have had to define `unordered_map` as

  ```
  template<auto Cfg>
  using unordered_map = typename decltype(Cfg)::type;
  ```

  which would force the user to explicit name `unordered_map_config` in

  ```
  using my_map = unordered_map<unordered_map_config{...}>;
  ```

There is still the unavoidable noise of having to use the `type` template alias since, of course, aggregate initialization is about values rather than types.

Another limitation of this simulation is that we cannot mix named and unnamed parameters:

```
// compiler error: either all initializer clauses should be designated
// or none of them should be
using my_map = unordered_map<{
  type<int>, type<std::string>,
```

```
  .allocator = type< my_allocator< std::pair<const int, std::string > > >
}>;
```

C++20 designated parameters are more restrictive than their C99 counterpart; some of the constraints (initializers cannot be specified out of order) are totally valid in the context of C++, but I personally fail to see why mixing named and unnamed parameters would pose any problem.

Posted by Joaquín M López Muñoz at 11:30 AM   No comments :

Monday, January 17, 2022

Start Wordle with TARES

There have been some discussions on what the best first guess is for the game Wordle, but none, to the best of my knowledge, has used the following approach. After each guess, the game answers back with a matching result like these:

⬛⬛⬛⬛⬛ (all letters wrong),

⬛🟩🟩🟨⬛ (two letters right, one mispositioned),

🟩🟩🟩🟩🟩 (all letters right).

There are 3^5=243 possible answers. From an information-theoretic point of view, the word we are trying to guess is a random variable (selected from a predefined dictionary), and the information we are obtaining by submitting our query is measured by the entropy formula

$$H(\text{guess}) = -\sum p_i \log_2 p_i \text{ bits},$$

where p_i is the probability that the game returns the i-th answer ($i = 1, \dots, 243$) for our particular guess. So, the best first guess is the one for which we get the most information, that is, the associated entropy is maximum. Intuitively speaking, we are going for the guess that yields the most balanced partition of the dictionary words as grouped by their matching result: entropy is maximum when all p_i are equal (this is impossible for our problem, but gives an upper bound on the attainable entropy of $\log_2(243) = 7.93$ bits).

Let's compute then the best guesses. Wordle uses a dictionary of 2,315 entries which is unfortunately not disclosed; in its place we will resort to Stanford GraphBase list. I wrote a trivial C++17 program that goes through each of the 5,757 words of Stanford's list and computes its associated entropy as a first guess (see it running online). The resulting top 10 best words, along with their entropies are:

TARES	6.20918
RATES	6.11622

TALES	6.09823
TEARS	6.05801
NARES	6.01579
TIRES	6.01493
REALS	6.00117
DARES	5.99343
LORES	5.99031
TRIES	5.98875