Support Documentation Blog
Try now

Using C++ containers efficiently

Posted by Edouard on 23 Mar 2020

Hello, dear reader! I think you have many assumptions about the performance and usage of data structures in C++. This blog post is me

> To fully enjoy this article, I recommend you, if you need it, quick refreshers about linked lists, double ended queues, hash tables, b trees, and heaps.

Without further ado, let the destruction begin!

Inserting and getting data

Look at the following program:

```
std::vector<std::uint64_t> v;
for(std::uint64_t i = 0; i < 10'000'000; ++i)
{
    v.push_back(i);
}
```

You're probably thinking *"This is so inefficient!"*

Do you think that this program will run faster if:

- We use std::deque instead of std::vector?
- std::list instead of std::vector?
- std::set?
- std::unordered_set?

Let's bench it!

On a Xeon E5-2630 v3, running Windows 10, and using VS 2019:

- `std::vector`: 163,277 us
- `std::deque`: 695,575 us - 4X+ slower
- `std::list`: 1,123,685 us - 9X+ slower
- `std::set`: 3,043,661 us - 18X+ slower
- `std::unordered_set`: 4,979,332 us - 30X+ slower

There is one way, though, to make the program run much faster:

```
std::vector<std::uint64_t> v;
v.reserve(10'000'000);
for(std::uint64_t i = 0; i < 10'000'000; ++i)
{
    v.push_back(i);
}
```

This implementation runs in 53,185 us (3X+ faster). If you're curious, reserve for `std::unordered_set` brings it down to 2,545,083 us, m `std::unordered_set` faster than `std::set`.

In C++ there is only one God, and its name is vector.

Why? Why? Why!!!

The common allocation strategy for `std::vector` is to reallocate always the double of existing usage, quickly bringing down the numb asymptote in O(log n)). For 10M entries, that means we probably do less than 25 allocations (2^24 = 16,77,216).

Even if we must copy everything over at each reallocation, the data is continuous in memory, making copies super-fast (notwithstandin super-fast as well). If we were using `push_front`, vector would do poorly because we would have to move at every insert.

`std::deque` will have 10M/size of the chunk allocations. `std::list` probably at least one allocation per entry, same for set, and `std::u` `std::unordered_set` has the double penalty of needing to resize the array + allocating the node. If we reserve the `std::unordered_s` beat `std::set`.

On top of that, `std::vector` only need to store the entry, deque needs to create the chain between each chunk, list between each entr `std::unordered_set` needs to compute the hash for each entry, create the node, and link to that node.

You can mitigate these problems with an ad-hoc allocator, but you'll never surpass the raw power of `std::vector`.

> If you are curious about benchmarking STL containers, you can see a more this in-depth article

The ideal case

It's good to keep in mind what your computer like (by computer I mean hardware + operating system):

- Continuous memory area to benefit from cache, and then page locality
- If possible, hitting the same memory area again and again (continuous memory helps)
- As few indirections as possible
- As few branches as possible

Size of the object

Should you store the object itself in the collection, or a pointer to the object? In our example, we used a 64-bit integer (8 bytes), and as grows, performance problem can arise. While it may sound beneficial to chose a container that doesn't move objects as you grow (for e linked list), you can also consider to store a pointer to an object (with `std::unique_ptr`).

What is the most efficient will depend on your exact use case.

For insertion

To insert data in a memory structure, the code sample you saw at the beginning is extremely efficient. You append to an existing memo wrote. The page is hot in memory, the address is in the TLB, and the cache can do its work optimally.

You want, as much as possible, to go back to this case. This is why it's generally better to insert data unsorted and sort it after than do

For lookups

This is the ideal entry lookup. You may not like it, but this is what peak performance looks like:

```
std::vector<std::uint64_t> v;
// stuff
// assuming size_t idx is where your entry is
auto x = v[idx];
```

One indirection to a contiguous memory area. And if you recently looked up an entry at a close index, it's in the cache. As much as pos yourself in that case.

The problem is, very often, you need to *search* for your entry and thus don't know the direct index. You probably know that linear searc with a small number of elements (as explained in this blog post for example).

For deletion

If you want to remove elements a lot, vector isn't your friend, as, like when inserting in front, this means moving elements along. You ca putting a tombstone instead of entries when you want to remove them, but if you have a lot of insertion and deletion, this is a case whe fit.

For destruction

Should you care about how fast destruction your container is? More often than you think.

For example, to properly destroy a linked list, you have to inspect every node, and free it. So, on top of calling the destructor, you have deallocations. On the other side, a vector only requires de-allocating a single continuous memory area and is thus extremely fast (you s destructor of every item).

Destruction speed is essential if your containers are relatively short-lived, for example, created for the duration of a function. You can m extending their lifetimes (re-using the same container), or, by using a different container altogether.

> If destruction takes up significant time, you may want to optimize the destruction of the elements in the collection. If they have a destructor, you may want to see if using a memory arena could be beneficial.

In short

Before doing anything, ask yourself *"Could I not just use an unsorted vector for this?"*

Now we got that out of the way, let's dive into the non-trivial stuff, because, as you can guess, `std::vector` isn't always the solution.

Bettering vector

The fastest allocation you can do is use the stack, as it costs exactly one instruction, moving the stack pointer. That's why, when you kn your container will have, you should use std::array instead of std::vector (or you can use a C array if you're the kind of person who lister on a cassette player, no judgment).

This approach is less attractive as soon as the size your vector exceeds a significant percentage of your stack size, or, when you expect which would prevent efficient move semantics.

To this, you're going to say, *"But! I don't know how large my vector should be at compilation!"*

Most of the time, you don't. But, if you know that very frequently, your vector will not exceed a specific size, then you can use the small

The small vector optimization is when the object is pre-allocated for a certain number of objects and will switch to dynamic memory al exceeds that value. Boost and Folly each offer an implementation (at QuasarDB we use Boost's).

Hash tables

The C++ hash table implementation

The hash table is probably the second most useful data structure after the array. They deliver O(1) lookup with a proper hash function. std::unordered_map and std::unordered_set. If lookup is in the critical path, you can greatly benefit from a hash map with a good hash

That being said, if the hash table is *really* in the performance path, then you will quickly find out that `std::unordered_map`, isn't that gr

This is because the standard is written in such a way that implementations are an array of linked lists. That means that although the cor O(1), you need to do several memory accesses (and indirections) to access your entry. When the hash table lookup is in your critical pa significant slowdown.

This great talk explains how you can build a better hash table. This article shows a benchmark of existing hash tables implementations.

> One crucial thing to keep in mind. Today's efficient hash tables implementations usually store the object within the array, to save and increase cache efficiency. As the size of the object grows, it becomes more efficient to store it in a node, to make re-arrangin Measure and decide accordingly.

At QuasarDB we currently use:

- tbb::concurrent_hash_map when we need concurrent reads and writes
- robin_hood::unordered_map in other cases
- std::unordered_map in legacy code that hasn't been updated yet

Efficiently leveraging your hash tables

Hash tables were initially not part of the C++ standards as they have a certain number of caveats. If you don't know about them, O(1) lc lookup.

On the importance of the hash function

Your hash function needs to be fast and provide great dispersion. Collisions are bad for hash maps as they force on additional comparis you want to ensure different entries have different hashes. For integral types, the hash function is trivial. For strings, Murmurhash or Fa

However, for composed types, there's a catch.

Let's look at this structure

```
struct my_struct
{
    std::string a;
    std::string b;
};
```

Let's say you use a good hash function such as farmhash. You build a custom hash function in such a way:

```
template <>
std::hash<my_struct>
{
    size_t operator()(const my_struct & v) const noexcept
    {
        return farmhash::hash(v.a) ^ farmhash::hash(v.b);
    }
};
```

You used XOR to maximize entropy, but there's a problem. XOR is commutative, meaning the order of the operation doesn't change th hashes are equal, the value will be zero, amplifying collisions.

In other words `my_struct{"a", "b"}` and `my_struct{"b", "a"}`, will yield the same hash, creating collisions, slowing down the looku

This is why you should properly compose hash function. You can use `boost::hash_combine()`, for example. At QuasarDB we have an with constexpr:

```
template <class Integer>
constexpr std::size_t hash_combine(std::size_t seed, Integer v) noexcept
{
    return seed ^ ((static_cast<std::size_t>(v) + std::size_t{0x9e3779b9}) + (seed << std::size_t{6}) + (see
}
```

For strings and blobs we use farmhash, and custom hash functions for any other type.

Reserve!

A hash table is an array of buckets, inserting new entries, resizes this array. That's why it's highly advised to reserve your hash map to th buckets you expect it to hold (assuming your hash function does it job one bucket = one entry). That way, you won't have to pay for a r insert new entries.

When you really need your data ordered

Ordered, or not ordered?

Ordered data is a requirement for a lot of data processing. I'm always dumbfounded by how many software engineering problems ever sorting problems.

Examples when you need data to be sorted: removing duplicates in a collection, data analysis criteria (for example, data lower/higher tl based on a threshold, etc.

It's very important to figure out early on if you need the data in your container to be ordered, because ordered data will limit your choic Again, we tend to overestimate our need for data to be ordered. It can be more efficient to sort the data when you need it to be sorted all the time.

My advice, would be:

- Ensure that ad-hoc sorting is not best for your case
- If that doesn't suffice, maybe a heap can be a good data structure for your use case
- For unicity problems, consider a hash table
- Damn! Looks like you really need a sorted data container

std::map, the average performer

Until C++ 11 the only “official” associative container available was std::map, which is ordered. If you need your data to be ordered, std::m
job. But you should use them only if ordered data is sufficient to solve your problem (performance or otherwise).

These containers are implemented as binary trees and thus incur an allocation of a node at every insertion and have a very cache-unfrie

To be fair, `std::map` has one solid advantage: it performs *ok* accross the board. It doesn’t have the caveat that `std::unordered_map` h
`std::multimap`, you can have entries with similar keys, which is, *very useful*. Since C++ 17 you can also extract nodes to attach them to
making it possible to build temporary maps and merge them.

So, should you use std::map?

Yes if:

- The code isn’t performance critical and you need a sorted container
- You don’t want to depend on third party libraries

For any other case, read on.

Sorted vectors

My favorite structure is the sorted vector. Boost has `flat_map` and `flat_set` (as well as `flat_multimap` and `flat_multiset`).

How does a sorted vector works? Instead of having the items on the nodes of a binary tree, like with a map, the items are stored in a ve
to a predefined criteria. Lookup is performed using binary search which has a O(log n) complexity.

In theory, a sorted vector is as fast as a binary tree, in practice it can be much faster, because:

- Memory is contiguous, making the structure very cache friendly.
- Destruction of the container has almost no overhead (as opposed to the map that needs to delete every node)
- You can insert n elements in a vector as they come, and then sort the vector, instead of making n sorted insertions
- You can give an insertion hint, for example, when you know you are inserting from a sorted source, this moves from O(log n) insert
- You can pre-allocate your sorted vector

In other words, the sorted vector can bring you back to the ideal insertion use case we described at the top.

In my experience, a sorted vector just crushes a binary tree. Most of the time…

What’s the catch?

The catch is that if you can’t insert the data (almost) ordered, and you can’t buffer the insertions, every insert will force you to move ex
in terrible insertion performance.

But fear not, as we can do better that binary trees for that case.

B-Trees

The problem with binary trees is that one entry is linked to two other entries down the tree. That means one node per entry: one alloca
poor locality. If you use a B-Tree instead of a binary tree, each node can have several children, minimizing the overhead, and making the
cache friendly.

Google’s Abseil has a great C++ 17 compliant B-Tree implementation.

At QuasarDB, we are big fans of sorted vectors and use them extensively. We currently are not using B-Tree as we haven’t been in a cas
solve with either sorted vector or an unsorted data structure altogether.

The structures we never talk about

At the beginning of this post, we seemed to have destroyed std::list and std::deque. Don’t be fooled by this benchmark, these structure

Showing linked lists the love they deserve

As you get more experience with programming, you may share the sentiment in this article that linked list are useless.

They are not. They are a very useful data structure, once you understand its hidden power: a linked list never invalidates iterators, excep
pointing to deleted elements (obviously!).

You can thus safely use iterators as a proxy for an existing object, making, for example, linked list a great data structure for LRU caches
implemented as a hash map + a linked list). And believe me, as a software project grows, the probability of this project having at least c

1.

std::list is a doubly linked list, meaning each element points to its predecessor and successor. If you only need to traverse the list in a sir
use std::list.

When to use a deque?

We saw in the benchmark that vector outperforms deque for insertion at the end. And you thought deque was optimized for that case

Not exactly. Deque, is great when you need to insert (or remove) at the end *and* the beginning of the container. In addition, deque does
when you insert only at the beginning and the end.

Although deque gives a O(1) asymptotical performance for access, is around twice lower than std::vector as it requires two pointer dere
for each access.

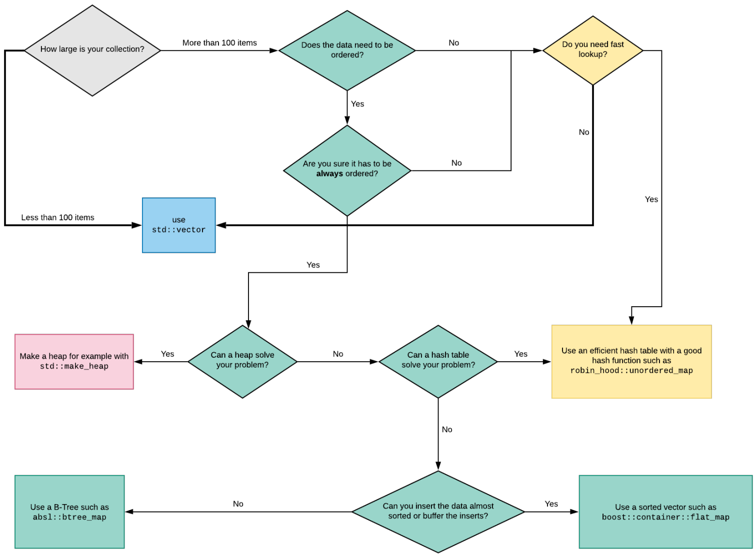
But remember, it's very hard to beat a vector at anything. It's the S&P 500 of C++.

Guidelines you can extrapolate from this

So what's the take away?

- Default to std::vector, if you can know the size at compile time, use std::array
- If you really need the data to be ordered use a sorted vector or a B-Tree
- If you need fast lookup, use a hash table, but be minfdul of the hash function and of the limitations of std::unordered_map

As an added bonus, here's the decision graph we use at QuasarDB. It's not complete (it can't be) as it doesn't include the very special c
want linked lists, deque, prefix trees, etc.



The goal is to help you go in the right direction with the right questions.

As always: conjecture, measure, deduce, decide!

Topics: c++, performance, containers

First Name*

Last Name

Email*

Website

Comment*

protected by reCAPTCHA
Privacy - Terms

Submit Comment

QuasarDB Why QuasarDB Product Resources News Blog Support | Documentation

About Us Partners Careers

 Legal information | Privacy policy Site built by **UNOMENA**