SPECTRAL:
TECHNOLOGIES

WEEK 3

std-like containers and its' internals

SPECTRAL:
TECHNOLOGIES

Why bother

- The most common part of any c++-program
- You may not use std exactly, but probably use some similar implementation with the same data structures (vectors, lists, hashmaps, sets, etc.)
- An incorrect usage can easily lead to performance bottlenecks
- In order to avoid this you have to understand what happens inside.

SPECTRAL:
TECHNOLOGIES

Non-performance concerns

- Iterators stability
  - Do the container's iterators remain valid after insertions/deletions?
- Reference stability
  - What does happen with pointers/references to container's elements after insertions/deletions?
- Exceptions safety
  - What if an exception is throwed during some insertion/deletion?
- Thread safety (not discussed today)

SPECTRAL:
TECHNOLOGIES

array

- `std::array<T, N>` is a wrapper around `T data[N]` , so the data stored on stack (unless its a part of some structure on heap).
- Use it only if you know the exact amount of elements you need (and its constant) and its relatively small.
- The main advantage comparing to vector is that it doesn't allocate:

```
for (int i = 0; i < a_lot_of_iterations; ++i) {
    std::array<int, 10> data;
    // much better than std::vector<int> data(10);
}
```

- This is rare though since usually you can reuse the same array over and over and you can preallocate vector for this.

SPECTRAL:
TECHNOLOGIES

vector

- `vector<T>` is a wrapper around `T *data` + some logic about resizing when needed.
- Allocates it's data on heap (you can pass your own allocator)
- Data is stored continiously.
- As long as you create it once and don't resize it, vector doesn't introduce any performance penalty comparing to `array` .
- The main difference between implementations is the expansion const (x2 by default, x1.5 is popular too)}

SPECTRAL:
TECHNOLOGIES

vector, things to consider

- Any insertion/deletion can invalidate all the iterators and references.
- `vector` never shrinks in almost all of the implementations (use shrink_to_fit if needed, `clear` doesn't free it's memory)
- If you have a type `T` with a custom move-constructor, make sure its marked as noexcept if you do a lot of `push_back` s.
- `vector<bool>` isn't a vector of bools.
- vector is the most cache-friendly data structure, so using it as a map/set sometimes can be faster than an actual map and set.

SPECTRAL:
TECHNOLOGIES

list/forward_list/intrusive_list

- Regular single-linked or double-linked list
- Any insertion creates a new node (+1 allocation), so all iterators and references remain valid whatever happens. You don't even need copy- or move-constructor for your type.
- Due to it's really bad data locality you almost never want to use it.
- And even if you have to, you probably look for `intrusive_list` (not in std)

SPECTRAL:
TECHNOLOGIES

string

- You may think of it as aт `vector<char>` , but thats a half of the story.
- There are 2 main optimizations: COW and SSO. Only the last one is used in std currently.
- SSO simplified

```
class string {
   union {
      ShortStringRepresentation, // on stack (like char data[16])
      LongStringRepresentation // on heap (like vector<char>)
   }
};

std::string a("hello"); // no allocation
```

- tradeoff between memory consumption and the length of a string you can still store on stack

SPECTRAL:
TECHNOLOGIES

string

- `c_str()` should be 0-terminated, so the internal buffer is 0-terminated too.
- In libstdc++ `std::string` is 32-bytes and stores strings up to 16 (including 0-byte) symbols on stack. Folly fits about 20 symbols and has 3 representations.
- Work with it as with `vector` except for sso.
- Most of the times you actually want to use `string_view` .

SPECTRAL:
TECHNOLOGIES

deque

- Greatly differs from deques you usually find in other languages.
  - `[ ]` is O(1)
  - As long as you modify it only at the front/back, the references (not iterators!) remain valid.
- `std::deque` is also called a paged vector.
- front/back operations allow types without copy- or move constructors.
- `libstdc++` has 512-byte block.
  - 4096 in libc++
  - 8 in MSVC
- You can't write an efficient cross-platform code with std::deque.

set/map

- Usually a red-black tree.
- Any insertion is an allocation.
- Same guarantees as of a list in terms of iterators, references and exceptions.
- Its not worth to change an implementation (e.g. use treap/AVL/etc) most of the time, rather consider the following:
  - `flat_map` , `flat_set` (not in std) if the elements' count is small enough
  - passing a custom allocator
  - cache-friendly alternative like `absl::btree_map`

hashset/hashmap

- There are 2 different ways to implement a hash table: chaining and open addressing.
- C++ standart forces the chaining method (more convinient to use, but way less effective)
- So if you have a lot of operations with hash tables, you want to replace `std::unordered_set/map` in general.

SPECTRAL:
TECHNOLOGIES

How to choose hashmap

- There are tons of implementations, the only way to determine which is better is to benchmark them in **your case**.
- If you don't have time for benchmarks, consider some well-known implementation like `absl::flat_hash_map` . Most likely its gonna be better than std-variant.
- In general chain-based tables provide better guarantees about iterators and references' validity, but slower. Make sure to check which guarantees are provided by your chosen implementation.
- For small datasets vector or map may be faster.

SPECTRAL:
TECHNOLOGIES

Other DS

- `small_vector<T, N>` (not in std) is usually `vector<T>` with SSO up to `N` elements.
- `std::queue` and `std::stack` are an adaptors of `std::deque` (but you can change the internal ds)
- `std::priority_queue` is a wrapper around `std::vector` using `std::push_heap` , `std::make_heap` and `std::pop_heap` .
- `flat_map` , `flat_set` (not in std) are the wrappers around `vector` .

SPECTRAL:
TECHNOLOGIES

General tips 1

- In 95% of cases you either need `vector` (or `small_vector/flat_map/flat_set`) or `hashmap/hashset`.
- Consider using `vector` even if its worse asymptotically in your scenario, but you have small enough datasets. Especially if the majority of operations you do are lookups and iterations.
- Even if you have a lot of insertions/deletions, `vector` can still be better up to hundreds of elements (depends on your workload) due to it's good data locality.
- For stream processing you probably need `deque` for it's asymptotic behaviour.
- Don't use `list` unless its really worth asymptotically. Even then you probably need `intrusive_list`.
- `list` and `deque` allow to put types without copy- or move-constructors.

SPECTRAL:
TECHNOLOGIES

General tips 2

- Most likely there is an already tuned for the typical workload hash table in your company. You may look at `absl` if there is not.
- Allocations (and not ds) are your main enemy in a lot of cases. So instead of implementing your "fast" binary search tree, just replace the default allocator in `std::set`.
- Whenever you use `string` ask yourself whether its possible to replace it with `string_view`.
- Don't forget to pass containers around by references or pointers, a full copy can be quite slow.
- Prefer `emplace` -methods to the regular ones.

SPECTRAL:
TECHNOLOGIES

Conclusion

- There is a way of thinking (especially in the competitive programming) about efficiency of a data structure as an asymptotic complexity multiplied by this ds’s hidden constant. For a example, `vector` has O(1) complexity for `push_back` (amortized though) and 1 is a constant. And then `set` has `O(log n)` for `insert` and 6 as a constant.
- By this logic vector is indeed can be faster. Hidden constant here represent many of the artifacts of real machines executing your code (like caches and etc.)
- Despite giving some intuition this approach isn't really helpful when you worry about latency.
- The only proper way to check which solution is better in your case is to benchmark.

SPECTRAL:
TECHNOLOGIES

Literature

- https://abseil.io/docs/cpp/guides/container
- https://accu.org/journals/overload/30/170/munoz/ (about unordered_map)
- Scott Meyers "Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library"