C++ STL – Performance and Memory-Footprint

Estimating Performance (Basics)

Performance estimations may come from a theoretical analysis,

- either based on heuristics on
  - expensive operations vs.
  - cheap operations
- or on an assembler code analysis
  - adding the clock cycles for the instructions to be executed.

It is clear that the first approach leaves room for vague assumptions and hence can give only a first idea, but is not very accurate.

But the second approach too has its difficulties for common CPU designs with several levels of on-chip memory caches, making algorithms with a good cache hit rate more efficient, often by an order of magnitude or even more.*

*: E.g. the linear traversal of an `std::array` or an `std::vector` will typically be much faster compared to traversal of an `std::forward_list`, an`std::list`, or even an `std::unordered_set`, as only the hash-buckets are stored in adjacent memory but the actual data is in a linked list (per hash bucket) and hence spread out over memory. Of course, from the theoretical point of view both have O(N) performance … only that the "Big-O" is much smaller for data structure in contiguous memory.

Measuring Performance

The alternative approach is to determine performance through a series of measurements,

- either based on the central taken into considered for solving the problem data structures in isolation.
- or based on "the real thing", i.e. the whole application
    - potentially instrumented to get performance data of the parts of interest, or
    - by comparing several competing solutions.

Great care must be taken in the first case, not to draw conclusions from the parts scrutinized in isolation, that do not extrapolate to the whole.

To get results applicable to "real-life" usage of an application, also care must be taken to do the analysis in a real-life context with real-life data.*

*: E.g. register assignment for frequently accessed variables may be quite different for an algorithm tested in isolation, compared to the same algorithm linked with a large program. Also sorting step might perform much worse for "nearly sorted data" compared to random data, so evaluating it with random data makes no sense if this is not typical for the real employment.

Estimating Memory-Footprint (Basics)

Estimating memory footprint on a theoretical base can be rather exact.

The only pitfalls are:

- Not taking into regard aggressive compiler optimization.
- Not considering possible clever tricks, applied by an implementation, maybe with compile time meta programming.

Also the contrary is possible: Underestimating the complexity of a general solution and assuming "too simple" data structures.

Note that the STL is generally specified in a way that allow extremely efficient solutions for the problems at hand (e.g. container iterators).

[!] This is even true if safety is traded in for speed, though Undefined Behaviour is often slightly misunderstood: it is no hindrance for the implementor of a C++ translation system to generate extra tests, not **mandated** by the Standard.

Measuring Memory-Footprint

Measuring footprint is often possible with very simple means:

- Applying the `sizeof` operation to some class
    - directly tells the size it uses in memory,
    - including any amount of padding necessary for alignment.
- For dynamically allocated data there is
    - typically a slight overhead with respect to additional memory required,
    - but the runtime performance degradation is often much more pressing to avoid unnecessary heap usage.

Frequently Applied Improvements

Some frequent improvements, also applied in modern STL implementations, are:

- Selecting a more efficient data structure – i.e. purging member data – when possible.
  - This is e.g. the case for [Stateless Allocators].
- The [Small Buffer Optimization] (or SBO) i.e. for data structure using heap allocation
  - to store small amounts of data in the holder object
  - instead of pointers (as done in the general case).
- Lazy evaluation or transparently avoiding copies (in the hope such will finally turn out to be dispensable).
- Transparently splitting the work to a number of threads.*

*: In fact, the specification of some STL algorithms effectively gives that much leeway, but of course for CPU-bound tasks this pays only on multi-core or hyper-threaded hardware architectures.