Amateur vs. Professional: Optimisation

Mark Bools

February 22, 2021

Last Modified: February 22. 2021

Abstract

What distinguishes an amateur from a professional? Let's consider optimisation. (Obviously much of this is opinion.)

Optimisation is a topic often brought up online, and often incorrectly.

People will obsess about trivia, such as how to shave a few keystrokes from their ViM use or how to shave milliseconds from their code run-time, or avoid spawning sub-processes in the Bash scripts. Don't get me wrong, there is a time and place for being concerned about each of these, but it is not something the professional worries about most of the time. The majority of these debates are amateurs trying the flex, showing their 'amazing' skills. In truth, the harder people try to impress like this the less professionals care.

A professional's mantra is, 'get it working'—this is, after all, what we're paid to do.

Let's consider a professional approach to tool use. Yes, you can save time and effort by using your tools efficiently. It *is* worth learning efficient keystrokes on ViM *but* only if you repeat the operation those keystrokes achieve thousands of times. Spending time to learn (and by 'learn' I mean commit to muscle memory so it becomes automatic) a keystroke pattern takes time and effort. The time and effort you commit to this learning had better be less than the time or effort saved by the repeat of those saved keystrokes.

Here's the thing. If you do something often enough your muscle memory will make it efficient over time. If you learn some shorter keystroke for an operation you repeat hundreds of times a day then you may slow a little at first but will quickly save time as your muscle memory improves. No need to spend time actively *trying* to learn it, you're going to just by doing it. And if you don't repeat it often in the course of your working day, then most likely you didn't need to save those keystrokes in the first place[1].

[1] With regards to ViM, learning the general pattern `command\textrightarrow {}movement` is definitely useful, as is learning the how to move about ViM. Trust me, you'll be doing this a lot so forcing yourself to, for example, skip to last line on screen with `L` (rather then repeatedly mashing `j`) is going to stick real quick once you've overcome the short-term slowdown.

When writing code the professional writes first for clarity and to make the code work (yeah, obvious I know). Only once the code is working will they worry about optimising. Arguing over whether a `for` loop is faster than a `while` loop in compiler `xyz` is something best left to amateurs (or compiler designers). Here's the thing, we mere mortals can only know which parts of our code need to be optimised once we can measure its performance. The 'premature optimisation' of code is a leading cause of shitty over-complex code. Some amateur thinks they know that accessing `hash` tables is microseconds slower than `array` accesses in some programming language, so they torture their code to use `array` even though the problem is better suited to `hash` tables. This takes them hours longer to develop and debug, the code is a tangled mess but, 'hey! it runs 3 milliseconds faster than the `hash` version'. Great, but this code is invoked once every minute in real use scenarios. Was all that effort worth it? Is your dog shit code easy to maintain?

Contrast with the professional. Take the problem. Solve the problem in as clear a way as possible. Run the code. Does it work? Yes. Is it fast enough for the situation? Yes. Is it resource efficient enough? Yes. Job done, move on. Result? Clear, maintainable code that satisfies the problem and was quickly developed. It's easier to debug and maintain because it is clearly written.

Suppose one of the conditions is not satisfied? If the clearly written code does not run fast enough? No problem. *Now* the professional starts looking at optimisation. First, the professional does not assume they know where the problem lies, they measure the scale of the problem. Use performance and coverage tools to figure out precisely where the code is being held up. Only then do they start changing things around. It may turn out that the `hash`/`array` access times are an insignificant factor. Let's say we have a loop over some data that's slowing our code. Can I do a simple test to avoid that loop all together in some circumstances? Have I got a loop within a loop (a very common costly process), can I unpack these loops or short-circuit the loops? If so, does this clear my performance bottleneck? If yes, then that's the problem solved. If no, then maybe, just maybe, it's time to use `array` rather than `hash`[2].

At this point the amateur may be feeling very smug because the professional has had to finally admit that 'yes, arrays are an optimization in this case', but in truth the professional approach will produce better code overall and situations requiring this sort of fine optimisation are the exception rather than the rule. In over 30 years of working IT I can count on the fingers of one hand the times that I've had to perform low-level optimisations. Most optimisations have been less obvious but much grosser (like realising that maintaining a larger session pool, or unpacking nested loops, will solve a problem, rather than fiddling with shaving microseconds of my code[3]).

[2]When making this sort of change leave a comment detailing *why* you did it. This is one of the circumstances in which extended comments are invaluable.

[3]Again, don't get the idea you should simply throw resources at a problem! That sort of thinking results in bloated shitty systems too. It's about knowing where best to invest your resources—especially you time.

Does this mean we should avoid learning more efficient ways of working? No, of course not. Just don't waste time fussing the detail too early.