

How I Learned to Stop Worrying and Love High Level Languages

ByteNoise

How I Learned to Stop Worrying and Love High Level Languages

I should probably warn you that I'm writing this as much for myself as anyone else. The next time I read a chapter of Steven Levy's Hackers, this should be enough common sense to remind me that, while the exploits of those people were great at the time, we've moved on.

Back in the nineteen fifties, programming wizards would prove their worth by spending days on end figuring out how to shave another few bytes of code off their programs, lovingly hand written in machine code. These programs were usually written down on paper first, only their final form being actual machine readable code that had been typed into the mainframes affectionately known as hulking giants.

It's now half a century later. As Gordon Moore noted, computers are constantly getting cheaper, smaller, and more powerful. One of the many upshots of this is that a while back, there was a profound shift in the roles of hackers and the machines that executed their programs.

Computing time is now cheaper than programming time.

To put it another way, hackers are finally more valuable than the machines they work with. There has been a gradual shift away

from the mindset of working out the best possible set of painstakingly detailed instructions, instead focusing on writing code that can easily be understood by fellow people.

As a hacker, in a way it almost pains me to say this. I can still see the beauty of elegant code, the aesthetic value of a perfectly optimised routine. However, the era in which such elegance was necessary in the realm of end-user applications has long since passed. Dynamic websites, being computer programs themselves, didn't even arrive until after this transition. Some languages even go as far as to be interpreted, meaning that computers can and do easily process the human-readable source code itself without even requiring it to be compiled into machine code first, let alone optimised by a wizard who writes more efficient machine code than an assembler.

Conditional instructions and loops are now written in neatly indented blocks, instead of having spaghetti-like clumps of old `GOTO` statements crowbarred around them. Functions have replaced subroutines, allowing different parts of the program to be worked on independently of one another, and this trend of separating different parts of the program has been extended even further as object oriented programming has replaced procedural programming. Frameworks go as far as to provide the universally useful half of your code for you, so you can concentrate on writing just the parts that make your program unique rather than having to constantly reinvent the wheel.

Overall, I believe this is a good thing. When people like myself were children in the eighties, our first experiences with computers were the awe of being able to make them do our bidding using simple, high level languages such as BASIC. I doubt I would have understood how amazing it was to have

your very own computer had I actually needed to sit down and learn how to tell it to draw each individual letter one after the other on the grid-like memory map of the screen, compared to the simplicity of telling it to `PRINT "HELLO"`. Taking care of the tedious parts for you is exactly what computers have always been there for in the first place.

As languages move further and further away from directly manipulating hardware, using continually greater levels of abstraction, two advantages occur: firstly, more people can understand the language and therefore appreciate the joys of the discipline of programming, introducing more people to the hacking community; and secondly, code becomes easier to read, maintain and share. People can see what the program does without having to concern themselves with how the result is achieved.

Low level languages still have their uses, of course: on those rare moments it manages to stop being lurid, the demoscene arguably exists as an artform in its own right, pushing the boundaries of what is deemed to be possible; and operating systems have to interface with tangible hardware at lightning-fast speed. However, I believe that modern applications should take advantage of this trend towards languages that are easier to read and write at the cost of being slower for the computer to execute.

As hackers, we should embrace this opportunity to make our craft more accessible to our friends and families. I believe it is time that we should focus on making source code friendly and easy to read. The extra time spent by computers interpreting this code is more than offset by the time saved by us people maintaining and sharing the code, and explaining to laymen just what it is we spend all day doing in the first place.

To put it simply, I'm arguing for pragmatism instead of idealism. Sure, using a framework in a high level language will produce much slower code than handwritten machine code, but it will be quicker to write, easier to maintain and easier to share. Isn't it about time that programming was centered around people instead of machines?