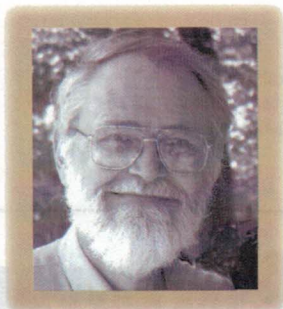


Sometimes the Old Ways Are Best



Brian Kernighan

As I write this column, I'm in the middle of two summer projects; with luck, they'll both be finished by the time you read it. One involves a forensic analysis of over 100,000 lines of old C and assembly code from about 1990, and I have to work on Windows XP. The other is a hack to translate code written in weird language L1 into weird language L2 with a program written in scripting language L3, where none of the L's even existed in 1990; this one uses Linux. Thus it's perhaps a bit surprising that I find myself relying on much the same toolset for these very different tasks.



What's Changed

Bill Plauger and I wrote *Software Tools* in 1975, nine years before *IEEE Software* began publication. Our title was certainly not the first use of the phrase, but the book did help to popularize the idea of tools and show how a small set of simple text-based tools could make programmers more productive. Our toolset was stolen quite explicitly from Unix models. At the time, Unix was barely known outside a tiny community, and even the idea of thinking consciously about software tools seemed new. In fact, we even wrote our programs in a dialect of Fortran because C was barely three years old at the time and we thought we'd sell more copies if we aimed at Fortran programmers.

A lot has changed over the past 25 or 30 years.

Computers are enormously faster and have vastly more memory and disk space, and we can write much bigger and more interesting programs with all that horsepower. Although C is still widely used, programmers today often prefer languages such as Java and Python that spend memory and time to gain expressiveness and safety, which is almost always a good trade.

We develop code differently as well, with powerful integrated development environments (IDEs) such as Visual Studio and Eclipse—complex tools that manage the whole process, showing us all the facets of the code and replacing manuals with online help and syntax completion. Sophisticated frameworks generate boatloads of code for us and glue it all together at the click of a mouse. In principle, we're far better off than we used to be.

But when I program, the tools that I use most often, or that I miss the most when they aren't available, are not the fancy IDEs. They're the old stalwarts from the *Software Tools* and early Unix era, such as `grep`, `diff`, `sort`, `wc`, and shells.

For example, my forensics work requires comparing two versions of the program. How better to compare them than with `diff`? There are many hundreds of files, so I use `find` to walk the directory hierarchy and generate lists of files to work with. I want to repeat some sequence of operations—time for a shell script. And of course there's endless grepping to find all the places where some variable is defined or used. The combination of `grep` and `sort` brings together things that should be the same but might not be—for instance, a variable that's de-

clared differently in two files, or all the potentially risky `#defines`. The language translation project uses much the same core set: `diff` to compare program outputs, `grep` to find things, the shell to automate regression testing.

What We Want

What do we want from our tools? First and foremost is mechanical advantage: the tool must do some task better than people can, augmenting or replacing our own effort. `Grep`, which finds patterns of text, is the quintessential example of a good tool: it's dead easy to use, and it searches faster and better than we can. `Grep` is actually an improvement on many of its successors. I've never figured out how to get Visual Studio or Eclipse to produce a compact list of all the places where a particular string occurs throughout a program. I'm sure experts will be happy to teach me, but that's not much help when the experts are far away or the IDE isn't installed.

That leads to the second criterion for a good tool: it should be available everywhere. It's no help if SuperWhatever for Windows offers some wonderful feature but I'm working on Unix. The other direction is better because Unix command-line tools are readily available everywhere. One of the first things I do on a new Windows machine is install Cygwin so that I can get some work done. The universality of the old faithfuls makes them more useful than more powerful systems that are tied to a specific environment or that are so big and complicated that it just takes too long to get started.

The third criterion for good tools is that they can be used in unexpected ways, the way we use a screwdriver to pry open a paint can and a hammer to close it up again. One of the most compelling advantages of the old Unix collection is that each one does some generic but focused task (searching, sorting, counting, comparing) but can be endlessly combined with others to perform complicated ad hoc operations. The early Unix literature is full of examples of novel shell programs. Of course, the shell itself is a great example of a generic but focused tool: it concentrates on running programs and encapsulating frequent operations in scripts.

It's hard to mix and match programs unless they share some uniform representation of information. In the good old days, that was plain ASCII text, not proprietary binary formats. Naturally, there are also tools to convert nontext representations into text. For my forensics work, one of the most useful is `strings`, which finds the ASCII text within a binary file. The combination of `strings` and `grep` often gives real insight into the contents of some otherwise-inscrutable file, and, if all else fails, `od` produces a readable view of the raw bits that can even be grepped.

A fourth criterion for a good tool is that it not be too specialized—put another way, that it not know too much. IDEs know that you're writing a program in a specific language, so they won't help if you're not; indeed, it can be a real chore to force some nonstandard component into one, like a Yacc grammar as part of a C program.

Lest it seem like I'm only complaining about big environments here, even old tools can be messed up. Consider `wc`, which counts lines, words, and characters. It does a fine job on vanilla text, and it's valuable for a quick assessment of any arbitrary file (an unplanned-for use). But the Linux version of `wc` has been "improved": by default it thinks it's really counting words in Unicode. So if the input is a non-text file, Linux `wc` complains about every byte that looks like a broken Unicode character, and it runs like a turtle as a result. A great tool has been damaged because it thinks it knows what you're doing. You can remedy that behavior with the right incantation, but only if a wizard is nearby.

There has surely been much progress in tools over the 25 years that *IEEE Software* has been around, and I wouldn't want to go back in time. But the tools I use today are mostly the same old ones—`grep`, `diff`, `sort`, `awk`, and friends. This might well mean that I'm a dinosaur stuck in the past. On the other hand, when it comes to doing simple things quickly, I can often have the job done while experts are still waiting for their IDE to start up. Sometimes the old ways are best, and they're certainly worth knowing well. ☞

Brian Kernighan is a professor in the Department of Computer Science at Princeton University and coauthor of several books on languages and programming. Contact him at bwk@cs.princeton.edu.



IEEE Software

Log on to our Web site to

- Search our vast archives
- Preview upcoming topics
- Browse our calls for papers
- Submit your article for publication
- Subscribe or renew

www.computer.org/software