Tiny Tools

Gerard J. Holzmann
Jet Propulsion Laboratory, California Institute of Technology

Many programmers like the convenience of integrated development environments (IDEs) when developing code. The best examples are Microsoft's Visual Studio for Windows and Eclipse for Unix-like systems, which have both been around for many years. You get all the features you need to build and debug software, and lots of other things that you will probably never realize are also there. You can use all these features without having to know very much about what goes on behind the screen.

And there's the rub. If you're like me, you want to know precisely what goes on behind the screen, and you want to be able to control every bit of it. The IDEs can sometimes feel as if they are taking over every last corner of your computer, leaving you wondering how much bigger your machine would have to be to make things run a little more smoothly.

So what follows is for those of us who don't use IDEs. It's for the bare metal programmers, who prefer to write code using their own screen editor, and who do everything else with command-line tools. There are no real conveniences that you need to give up to work this way, but what you gain is a better understanding your development environment, and the control to change, extend, or improve it whenever you find better ways to do things.

Bare Metal Programming

Many developers who write embedded software work in precisely this way. Mission critical flight software development for spacecraft, for instance, is typically done on Linux systems, using standard screen editors and command-line tools. The target code will ultimately execute under real-time operating systems, on custom-built hardware. Both productivity, accuracy, and reliability really matter in these applications, so there must be a case that can be made for the bare metal approach.

Let's consider what types of questions a software developer typically encounters when working on an application. There are actually not that many. The four most common questions are: Where is this variable declared, where is it used, how is this function defined, and where is it used? In an IDE you can click on a name and see its definition pop-up. Or you can click on a function name and see the screen switch to its declaration. That can be an unwelcome context as you're staring at a subtle piece of code that you're trying to understand. It is rather simple to answer these same types of queries with a few small command line tools that you run in their own window, separate from the editor, so that you never lose context.

There are standalone tools that can provide the basic functionality that we need, all in a single tool that works almost like an IDE, but without the editor. A good example is the Cscope tool that was developed by Joe Steffen at Bell Labs in the early eighties [1]. Curiously, the original motivation for that tool was that the use of command-line calls to scan code was too slow, so Joe decided to build a database to speed up the resolution of standard types of queries. Today, though, our machines are fast enough that this performance argument no longer applies, unless the information you need is buried so deeply in a

directory hierarchy that you may do better to refactor your code first, before trying to add to it. This means that we can avoid having to construct and store a database, and keep it up to date, before we can answer routine queries about our code.

To become a good bare metal programmer you must be comfortable with shell programming, and at least the standard set of core Unix text-processing tools `grep`, `sed`, and `awk`. I always use the `bash` shell that is currently the default on Linux systems, but almost any other modern shell will do. The text-processing tools are used for quickly extracting information from potentially large numbers of files, and presenting it to you in the form that is most useful.

Whenever I'm asked to evaluate the C source code for an application, there are a couple of simple queries that I often use to get a first impression of the overall quality. They are hard to replicate in an IDE. For example, most coding standards for safety critical software have the rule that all `switch` statements contain a `default` clause. For example, this is rule 16.4 in the most recent MISRA-C guidelines for critical code [2]. How hard is this to check? You can fire up a serious static source code analyzer to do the check, or you can type these two queries:

```
$ cat `find . –name “*.c” –print` |
     grep –c –e switch
1065
$ cat `find . –name “*.c” –print` |
     grep –c –e default
809
```

We get two numbers that should be fairly close if the rule is followed. The check is of course not precise, because the keywords switch and default could also appear in strings or in comments, but that is not likely to dominate the results. We'll get back to more precise ways to check these types of things shortly.

It's just as simple to quickly check for uses of `goto` or `continue` statements that many coding standards also frown upon, the use of `union` data structures or compiler dependent `pragma` directives, or risky calls to routines such as `strcpy` instead of `strncpy`.

It gets a little harder if we want to check for a rule that is very similar to the use of a `default` clause in `switch` statements. How, for instance, would we check if every if-then-else-if chain always ends with a final else. This is Rule 15.7 in MISRA-C [2]. The reason for this rule is again to make sure that there are no accidently missed cases if the final else is missing. There is such a missed case in the following code fragment when both c1 and c2 are false:

```
if (c1) { s1; } else if (c2) { s2; }
```

In this case we can't just look for the keyword combination "else if" because there's more context that needs to be taken into account here. Static analyzers normally don't check for these patterns either, so we'd have to come up with an alternative. It is not that hard to write such a check, using only basic command line tools, as we will show below.

Another very common thing you need to do when developing or browsing code is to print a suspicious fragment of code, prefixed with a filename and line numbers. In both these cases it helps if we begin by add a few simple extra commands to our tool set. So, let's talk about that first.

Tiny tools can customize and simplify common tasks.

A Survival Kit

When I move to a new system, either because I upgrade my desktop, or when I have to set up a temporary work-environment for myself on someone else's machine, I start by installing a survival kit of small tools that I wrote, that can make life easier. These survival tools are small and simple enough that they are guaranteed to work anywhere. Most of these tools are no more than about 20 lines long, with just two exceptions.

The first exception is a tokenizer for C code of about 600 lines, that I'll talk about shortly. The other is a small emulation of the main features of the `sam` screen-editor that was developed about thirty years ago by Rob Pike at Bell Labs. The `sam` editor is a favorite of many former Bell Labs researchers, although curiously not of Rob himself [3]. The full Unix version of `sam` is about 15K lines of C, but not installed on many systems. My small survival version of the key features is about one tenth that size, and written in Tcl/Tk, which is available on most systems.

Let's talk about some of the other tiny tools in my survival kit. You often want to look at a numbered listing of a fragment of code. There are a few ways to do this with Unix commands, for instance with `pr`:

```
$ pr -T -n $*
```

or similarly with a starting line-number:

```
$ pr -T -n -N 42 $*
```

Why not wrap this into a single command, called `num`, so that you don't have to remember the names of all those options. Here is that command as a tiny tool, applied to itself:

```
$ num num                                   # ONE
 1 #!/bin/sh
 2
 3 # num [nr] [file]*
 4
 5 if [ -f $1 ]
 6 then
 7      pr -T -n $*
 8 else
 9      N=$1
10      shift
```

```
 11       pr -T -n -N $N $*
 12 fi
```

Another tiny tool in my kit is called `line`. All this does is to print a specific line from a file, optionally with a few surrounding lines for context. It uses the `num` script from above. As is not unusual, about half of the tiny script is for error-handling only.

```
$ num line                          # TWO
    1   #!/bin/sh
    2
    3   if [ $# -lt 2  -o  $# -gt 3 ]
    4   then echo "usage: line file linenr [nrlines]"
    5        exit 1
    6   fi
    7
    8   if [ ! -f $1 ]
    9   then echo "error: no such file: $1"
   10        exit 1
   11   fi
   12
   13   n1=$2; n2=$2
   14
   15   if [ $# -eq 3 ]
   16   then n1=`expr $n1 - $3`
   17        n2=`expr $n2 + $3`
   18   fi
   19
   20   sed -n ${n1},${n2}p $1 | num $n1
```

The first two arguments to this script are a filename followed by a line number, and an optional third argument can specify the number of lines that we want to see before and after that line. This allows us to say, for instance:

```
$ line line 9 1
    8   if [ ! -f $1 ]
    9   then echo "error: no such file: $1"
   10        exit 1
```

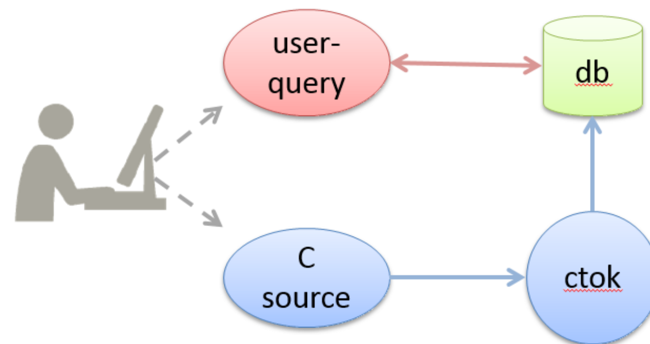
Another even smaller tool that I use on a lot when writing code is called `any`. I use it for quickly finding all locations of a variable name or text string in the source files in the current directory. A basic version of this tool can be written with just a single `grep` command, like this:

```
$ num any                      # THREE
    1 #!/bin/sh
    2 grep -n -e "$1" *.[chyl]
```

There is, however, a flaw. This script works well for longer names, that are relatively unique, but it fails miserably when you need to find all uses of say a single-letter variable named '`i`' or '`j`.' We can do better.

A Tokenizer for C

To solve problems like this I use a small standalone tokenizer for C, called `ctok`, that is remarkably useful in lots of places. Instead of passing the lexical tokens that it recognizes in an input stream to a parser, `ctok` prints this information on the standard output.



The code for `ctok` is about 600 lines of `lex` input, which compiles into a small standalone executable that has become a core part of my survival kit. Two types of output that `ctok` produces are of interest to solve the variable matching problem are tagged “line” and “ident.”

```
$ ctok main.c
…
line    1399   main.c
ident   3      sym
ident   1      a
…
```

The “line” tags record the name and location in the current file for the information that follows. Lines tagged with “ident” record identifier names. The second field in this case gives the length of the name, and the third field the name itself. This suffices to write a small command-line tool that can accurately locate identifiers in C code, even if they are single-letter names. My version is called `itok`, and it looks as follows:

```
$ num itok                                # FOUR
  1 #!/bin/sh
  2
  3 if [ $# -ne 1 ]
  4 then echo "usage: itok identifier"
  5      exit 1
  6 fi
  7
  8 for f in `ctok *.[chyl] |
  9     awk -v var=$1 '
 10     $1=="line" { lnr = $2; fnm = $3 }
 11     $1=="ident" && $3==var {
 12          printf("%s:%d\n", fnm, lnr)
```

```
 13     }' | sort -u`
 14 do
 15     echo -n `echo $f | sed "s;:.*;:;"`
 16     line `echo $f | sed "s;:; ;"`
 17 done
```

The script uses `awk` to pickup the right information from `ctok` and makes it suitable for passing to the `line` tool that we discussed before. The output looks something like this:

```
$ itok k
...
structs.c:521   int j = i, k;
structs.c:534       { for (k = 0; k < sym->nel; k++)
structs.c:538              (*targ)->lft->val = k;
tl_main.c: 56   int k = 0;
tl_main.c: 65        k++;
tl_main.c: 72        k--;
```

Clearly, a straight `grep` for the letter `k` over the same source files would be quite unhelpful. Using the tokenizer it is easy to build lots of additional tiny checkers, like for the two examples we started with: finding switch statements without a default clause, or if-then-else-if chains that do not end with an else.

The last two tiny tools in my kit that I'll discuss here are really part of the same family. I use them to quickly find the definition of functions or data structures in C source files. It is not hard to write more sophisticated versions of these tiny shell scripts by using the tokenizer as a front-end, but these basic versions already provide most of the needed functionality.

The first is called `ff` (short for find function) to find and print the definition of a function, optionally restricting the search to a specific file:

```
$ num ff                         # FIVE
  1 #!/bin/sh
  2
  3 case $# in
  4 1)  sed -n /\^$1/,/\^}/p *.[chyl]
  5     ;;
  6 2)  sed -n /\^$1/,/\^}/p $2
  7     ;;
  8 *)  echo "usage: ff fctname [filename]"
  9     exit 1
 10     ;;
 11 esac
 12 exit 0
```

This script uses the fact that I always write the name of a function starting on a newline, right after the function type which is also on a line by itself. The end the definition is always a single closing curly brace in the left margin. If you use a different format, you would have to adjust the script to match it, or switch to a `ctok` based version that can remain independent of these types of formatting choices.

Consistency is the first step towards improving code quality.

A close match for this version of `ff` is another tiny script called `ft`, for finding data type or `typedef` definitions. Also here, the simple version below depends on the specific way that I format these definitions. If you use a different format, you will of course have to adjust these scripts to match that. My version looks as follows:

```
$ num ft                    # SIX
 1 #!/bin/sh
 2
 3 case $# in
 4 1) sed -n "/^typedef struct $1/,/^}/p" *.[cdsyhl]
 5    sed -n "/^struct $1/,/^}/p" *.[cdsyhl]
 6    ;;
 7 2) sed -n "/^typedef struct $1/,/^}/p" $2
 8    sed -n "/^struct $1/,/^}/p" $2
 9    ;;
10 *) echo "usage: ft typename [filename]"
11    exit 1
12    ;;
13 esac
14 exit 0
```

The tiny scripts from my survival kit are probably the ones that I use the most each day. Of course, none of this is rocket science, and if these tools don't increase my productivity, at least they make the job of writing code a lot more fun. The last two of the scripts I showed make assumptions about a particular coding style. A nice side-effect of this is that there is a good reason for me to be consistent in the use of that style. A consistent coding style generally improves readability, and one could well make a case that it is also the first step towards improving code quality.

[1] Cscope background, http://cscope.sourceforge.net/history.html

[2] MISRA C:2012, Guidelines for the use of the C language in critical systems. Publ. MIRA Ltd 2013. http://www.misra.org.uk/

[3]The Setup, interview with Rob Pike, https://usesthis.com/interviews/rob.pike/

Acknowledgement
This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

© 2015 California Institute of Technology. Government sponsorship acknowledged.