by Jeffreys Copeland and Haemer

ALEX GROSS

"UNIX…is not so much a product as it is a painstakingly compiled oral history of the hacker subculture. It is our Gilgamesh epic."
– Neal Stephenson, "In the Beginning was the Command Line," 1999

Back to Basic(s)

We have been working with UNIX for so long that we sometimes take it for granted. This column is for people who have been so immersed in non-UNIX systems that they haven't yet advanced to the 1970s. Yes, we mean in systems like Windows and MVS.

Lottery Numbers

A couple of months ago, we received a note from a Romanian friend who wanted to dip her toes into UNIX. She had spent most of her professional career working on DOS/Windows boxes, and was ready to try something new. She wrote to us, asking for advice.

A good place to start, we decided, was to attack a simple problem she'd already tried elsewhere.

Why a simple problem? Brian W. Kernighan and Dennis M. Ritchie explain, in the first section of the first chapter of *The C Programming Language* (published by Prentice Hall Inc., now in its second edition, 1988, ISBN 0-13-110370-9):

1.1 Getting Started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

Print the words

`hello, world`

This is the basic hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

And it's not even as easy as Kernighan and Ritchie make it sound. Our friend recounts an early experience:

I remember when I first met Visual C++, I only wanted to make a simple random lottery number generator. My first steps into the IT world were FORTRAN, COBOL, assembly, all of them on a mainframe. After many years spent far from the IT world, I came across a PC with this idiotic Windows system. I had to learn C, so I thought of this little program. OK, I thought of it, read a book of C (Ritchie and Kernighan)–of course, I didn't read it all, I tried to follow the steps. Alas! By no means could I have guessed where the C compiler was. I had to first make a project, then change all kinds of settings that were "Chinese" to me, or better said, I was "Chinese" to them. In order to get that beast to write the lottery numbers on the screen (ON A BLACK SCREEN!), I had to waste days on end, helpless and clueless, thinking I must be the last idiot on the planet. Finally, I managed it: using Borland C++ 3.0, of course, because there was no rand() *function in Visual C++.*

This problem was well-defined, so it seemed worth trying on UNIX in order to show her what a UNIX-y solution would look like.

Our first question was, of course, "What does a lottery ticket look like?" Romanian lottery tickets, it turns out, contain three picks. Each lottery pick is made up of half a dozen, unique, random numbers from the set {1… 49}, and each ticket requires three picks.

Here's an example:

```
15 42 16 28 7 40
8 13 34 31 20 17
18 16 38 49 10 12
```

(We suspect that any single, matching pick is a winner, but because we're not going to actually buy a ticket, not having the Romanian lei to spread around, we don't care if we're wrong.)

/bin/sh

Here's what we did, step-by-step:
First, we wanted to count to 49:

```
$ seq 49
1
2
3
...
```

Then, we needed 49 random integers:

```
$ for i in $(seq 49) do; echo $RANDOM; done
12978
17637
31835
...
```

Next, we added line numbers:

```
$ for i in $(seq 49); do echo $RANDOM; done | nl
1  21931
2  9178
3  24191
...
```

Then, we sorted them numerically by the second field:

```
$ for i in $(seq 49); do echo $RANDOM; done |
> nl | sort -n +1
   41  623
   46  984
   28  1596
   ...
```

Next, to get our seven out of 49, we grabbed the top seven:

```
$ for i in $(seq 49); do echo $RANDOM; done |
> nl | sort -n +1 | head -7
   33  786
   34  1384
   16  1414
   11  2496
   23  2684
   35  3180
   3  3404
```

And, of course, we just wanted the indices:

```
$ for i in $(seq 49); do echo $RANDOM; done |
> nl | sort -n +1 | head -7 | awk '{print $1}'
30
34
25
1
11
31
40
```

Well, that did what we wanted, so we made it into a program:

```
$ echo "for i in \$(seq 49); do echo \$RANDOM; \
done \
| nl | sort -n +1 | head -7 | \
awk '{print \$1}'" > lottery
$ chmod +x lottery
$ lottery
1
11
42
36
17
14
41
```

Voilà.

You need three sets? Sure. We just start the process again by running it three times:

```
$ for i in $(seq 3); do lottery; done
6
24
25
49
1
35
21
25
43
28
2
23
14
32
15
17
44
43
33
13
29
```

But we don't like that format. So let's put each pick on a separate line:

```
$ for i in $(seq 3); do lottery | fmt; done
22 12 17 44 47 23 21
27 8 17 34 35 33 18
4 17 32 16 47 25 6
```

(Oh heck, we just looked back at the problem specification and realized we only want six random numbers between 1 and 49, not seven. Whatever shall we do? We'll leave this as an exercise for even the least experienced of our readers.)

We could do all this interactively, and nearly instantly, because the UNIX shell lets you recall and edit command lines. Although POSIX only guarantees `vi`-like editing commands, the shells we've used also provide an `emacs`-like mode in case you like that better.

When this feature first became widely available in the late 1980s, it quickly changed the way we interacted with the shell.

Our typical approach is now to do much of our shell-level programming on the fly. At each step, we recall and edit a previous command, mostly just appending a new filter to do something new to the data.

Because a good UNIX filter takes its input from `stdin` and writes to `stdout`, we often test what we write from the keyboard and watch the results on the screen. When we are finally satisfied, we capture what we have been doing and turn it into an executable shell script. (We use this same process in a substantial amount of our Perl programming.)

UNIX is full of tiny tools that filter and transform text. The sense we get when we're programming is one of popping together little, existing tools, looking at the output, and then adding some new transformation to get to the next stage.

Production Code

But what if we wanted "production" code?

The answer is: we'd do the same thing. For a straightforward task like this, the most important factor to consider is development time. As Tom Christiansen says:

Q: What's the difference in speed between an application in Perl and an application in C++?

A: About three weeks. :-)

It is, however, worth enhancing our program a bit. Listing 1 shows a production version that adds comments, does its own formatting and takes an optional command-line argument to specify how many picks to generate.

One noteworthy feature of this code is the internal documentation. We have simply co-opted Perl's `pod` format. The various `pod` tools encourage you to keep documentation and code in the same file so they'll stay in synch, and produce a wide variety of documentation formats from a single, well-defined input format. They work for our shell script, too, because the `exit` statement at the end of our executable code prevents the shell from trying to interpret the documentation, while the various `pod` tools (`pod2html`, `pod2latex`, `pod2man`, `pod2text`, `pod2usage` and `pod select`) will ignore everything before the first `pod` directive.

Listing 1. Production Code

```
#!/bin/sh
# Romanian lottery program:
#    print lottery tickets with NPICK picks (default 3)
#    each pick is 6 random, non-repeated integers from 1..49
# $Id: lottery,v 1.6 1999/11/27 00:24:46 jsh Exp $

pick1() {
  RANDOM=$RANDOM          # reset the seed
for i in $(seq 49)        # 49 random numbers
do
  echo $RANDOM
done |
nl |                # index the numbers
sort -n +1 |        # randomize indices by sorting random data
head -6 |               # grab the first 6
awk 'print $1' |        # now throw away the sort key
fmt                 # and put all 6 on one line
}

case $# in
 0) N=3 ;;
 1) N=$1 ;;
 *) echo "usage: $0 [npicks]" 1>&2 ; exit 1 ;;
esac

for j in $(seq $N)
do
 pick1
done

exit

##############################################################

=head1 NAME

lottery - print "6 from 49" lottery picks

=head1 SYNOPSIS

B<lottery [npicks]>

=head1 DESCRIPTION

B<lottery> prints B<seq> lottery picks, one per line (default: 3)

Each pick is six, non-repeated integers out of 1..49.

=head1 SEE ALSO

sh(1)

=head1 AUTHORS

Jeffrey L. Copeland <copeland@alumni.caltech.edu>
Jeffrey S. Haemer <jsh@usenix.org>
```

A Good Programmer Can Write FORTRAN in any Language

If you've tried running this code, you may see that we've glossed over a step. In our examples, we've used a utility called `seq`. This trivial, yet amazingly useful UNIX tool isn't found in the POSIX standard, or even on most UNIX distributions. It should be clear what it does: `seq` counts. We first saw `seq` used in *The UNIX Programming Environment* by Brian W. Kernighan and Rob Pike (published by Prentice Hall, 1984, ISBN 0-13-937681-X), but we now use it so much that we'd be lost without it. In fact, we were, just the other day.

A few weeks ago, we were asked to help judge a practice ACM programming contest. While waiting for the contestants' entries to be submitted, we did what we do compulsively: write code. We sat down with another judge, Dan Crawl, to explore how we might solve one of the programs in the

shell (more on this anon) and immediately ran into a brick wall: the machine we were working on didn't have `seq(1)`.

When you're hacking, getting there is all the fun. We shifted gears and wrote a `seq`. Listing 2 shows what we came up with.

Here, too, we show our colors. For us, programming in UNIX feels like bolting together prefabricated parts from a brilliantly designed erector set. Many of our workaday tasks can be done in minutes with the right pieces at hand. Sometimes, however, in the midst of our work, we discover we don't have something we need. Rather than go back to the drawing board to redesign a custom solution (say, in Visual C++), we simply pause to manufacture the missing part, then pop it into place.

Occasionally, this approach pushes us over the edge. One of the problems in the programming contest was to count the atoms in a formula, such as tetraethyl lead, Pb(CH3CH2)4, and produce output something like the following:

```
C: 8
H: 20
Pb: 1
```

Our first reaction was to consider a pipeline with these steps:
1. Replace each distinct element with a distinct prime, turn the numbers into exponents, and everything else into multiplications, like this: `2*(3*5^3*3*5^2)^4`.
2. Pipe the expression to `usr/bin/bc` for evaluation.
3. Pipe the result to `/usr/games/factor` for prime factorization.
4. Reverse the transformation in Step 1.

That's deeply warped.

When drunk enough on Green Chartreuse, Haemer will confess to having written an entire object-oriented language in the shell [Jeffrey S. Haemer, "A New Object-Oriented Programming Language: sh," *Proceedings of the USENIX Summer 1994 Technical Conference, Boston, MA.*]

Two (or More) Can Play at this Game

Here's a chance to try your hand. Last week, we got a note from another Windows-bound friend, Michael Mendelson, who said he, too, wanted to start playing with UNIX. When we sent him our Romanian lottery story, he responded with a similar story of his own, culminating with a 154-line C program that he'd written early in his career to randomize the lines of a file. In his note, he says, "I'm sure you could write this one in half the lines from the shell. I'd like to learn some of those tools."

His program reads in a file, shuffles the lines into random order, overwrites the original file with the result and announces the file has been randomized.

How small a shell solution can you come up with? Our first cut took about 15 lines. Let us know how much better you can do. We'll pass on your results to Michael.

Last, we received a note from an observant reader who points out that the anonymous "A picture is worth a thousand words," with which we began our November and December columns ("Pictures," November 1999, Page 38, and "Slides," December 1999, Page 36), is actually a quote from Fred R. Ballard in *Printer's Ink*, March 10, 1927. We're always pleased when our humble efforts reach the desks of the literate and well-read. Happy trails!

Listing 2. Homemade seq

```
#!/usr/bin/perl -w
# $Id: seq,v 1.3 1999/12/13 19:38:13 jeff Exp $

use strict;

my $usage = "usage: $0 [start] end"
 unless @ARGV == 2;
unshift @ARGV, 1 if @ARGV == 1;
die $usage unless (@ARGV == 2);

foreach ($ARGV[0]..$ARGV[1]) {
 print "$_\n";
}

=head1 NAME

 seq - print a sequence

=head1 SYNOPSIS

 seq [start] end

=head1 DESCRIPTION

C<seq> generates sequential integers

By default, it starts counting at 1.
For example,

 $ seq 3 5
 3
 4
 5

=head1 BUGS

By little more than a lucky accident,
C<seq>
does odd, sometimes useful things
with non-integer arguments.
Not obvious that this is a bug.
Try this, for example:

 $ seq cat dog

=head1 AUTHORS

Just a re-implementation of something used
in Kernighan and Pike, but never written
in there.
Maybe it's part of Plan 9 or something.

Dan Crawl <crawl@cs.colorado.edu>
Jeffrey S. Haemer <jsh@usenix.org>
Jeffrey L. Copeland <copeland@alumni.caltech.edu>
```

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.expert.com/pub/Work`.