by Jeffreys Copeland and Haemer

ALEX GROSS

"A picture is worth a thousand words."
– Anonymous

"Talks, speeches, articles and resolutions should all be concise and to the point. Meetings also should not go on too long."
– Mao Tse-tung

Slides

Last time, we talked about photographs, so this time we thought we'd discuss slides. Now that laptop computers are ubiquitous, people have defeated the idea of traveling light by choosing to cart around 25-pound projectors that connect to their 10-pound laptops to do presentations. The current tool of choice for generating presentations is Microsoft Corp.'s PowerPoint. If you like WYSIWYG word processing systems, and would rather use your mouse than your keyboard, PowerPoint might be the tool for you.

But back when overhead projectors were predominantly used, people usually brought in black-and-white slides–viewgraphs–for presentations, assuming there would be a projector on-hand when they got there. The tool of choice for producing viewgraphs–for us, at least–was the `-mv` macro package for `troff`. Because we would rather edit the map than the territory, we almost always prefer to use `troff` or `TeX` for any text-processing exercise. (And because we'd rather carry a file folder of viewgraphs than 35 pounds of computer and projector, we still favor the old method.) Unfortunately, `-mv` required a fair amount of markup. Fortunately, Perl guru Tom Christiansen wrote an alternative called `perlpoint`, an obvious pun on PowerPoint.

There are a number of other ways to attack the problem. As UNIX bigots, we've always favored any approach that uses a tools-and-pipes solution. Certainly, the `-mv` solution does that: we can build diagrams and tables into our slides using the `troff` preprocessors `pic` and `tbl`.

The other end of the spectrum–the wrong approach, in our view–would be to take some text marked up for slides, have a huge markup language that allowed for tables, pictures and equations, and directly generate PostScript from that language. (In this world-view, the PowerPoint solution is perhaps a step worse because you don't even get to edit the markup. But we realize that this is a religious battle in which we may be adherents of a shrinking cult.)

The tools-and-pipes solution allows us some extra flexibility. We can make some decisions about the structure of the slides–a title, two levels of bullet items, some font changes and so on–and decide what the text markup for those will be. Given a fixed set of markup rules–for example, a title line is bracketed by `<TITLE>...</TITLE>`–we can then build a filter to translate each markup element into the appropriate `troff` directives. Even better, because the structure of `troff` allows us to encapsulate a series of directives into a macro, we can simplify our markup translator by pushing most of the `troff` work into a macro package. Later, if we wish to change the appearance of our slides, we can modify the macro package without having to change anything about the markup of the slide text or the filter. We can do this because what we've used to mark up the slides is structural information, not the procedure for generating each element.

If you've followed our discussions of Perl's `pod`–that is, "Plain Old Documentation"–language for documentation, you'll realize that it's exactly this kind of markup. It generally uses structural markup without worrying about the way in which each title or heading is going to be rendered. This is important because `pod` can be, and normally is, translated into a number of different forms. For example, there are `pod2html`, `pod2man` and `pod2tex` filters, and each will have a different way of rendering some text in italic.

This is an example of the virtue of documented intermediate formats. By having the input and output formats of `TeX`, `troff`'s `-man` macros and HTML all carefully documented, we can generate filters to be added before or after existing tools.

Back to Tom Christiansen's solution: It uses the UNIX-like approach of a Perl filter along with a `groff` (the GNU version of `troff`) macro package for formatting slides. We've recently done some work on the macro package to add some features, so we thought we'd discuss the whole package. Look for it on the Comprehensive Perl Archive Network (CPAN) at `http://www.cpan.com/`. Because `perlpoint` uses a simple structured markup, it's also easy to build a filter to convert `perlpoint` input to HTML. (Such a filter exists, called `pp2html`, but even though it's part of the `perlpoint` distribution we won't discuss it here.)

Input Format and Script

Let's begin with the input format and script, which will allow you to see how things flow. The input format is very simple, as shown in the following self-documenting test file:

```
=A perlpoint Test File

This file is an example of Perlpoint input

* The line beginning with = is the slide title.

* These lines beginning with asterisks are bullet items.

* It is also possible to use font changes in the same
way we do in C<pod>.
For example, I<italics>.

* All work and no play makes a very long slide.

    1) this is a display
    2) test
    3) where we have code
    4) on 4 lines

INDEX
```

The only slightly obscure item in this example is the line containing `INDEX`. It triggers the macros to produce an index of the slides. This is helpful when you're conducting a presentation and someone asks a question about "that slide a few minutes ago labeled Middle East Religions."

How does that get translated into slides? It's pretty simple, really. A command line like the following will do the trick:

```
pp2roff test.pp | groff - | lp
```

The Perl script `pp2roff` deals with the `troff` markup for you, and then `groff` deals with the formatting. The `pp2roff` script–which was originally written by Christiansen, and slightly modified by us–is fairly straightforward (see Listing 1). Here follows a dramatic reading.

We begin with the usual shebang line, `use strict` and `-w`, so Perl will keep us honest about grammar. Line 3 sets Perl's record separator to the empty string, which separates records with a blank line rather than the usual newline. Lines 6 through 9 define the text to be used at the start of each slide: a comment for a separator and the `author` tag from data we set up in lines 4 and 5. Line 10 produces the text to start the package of slides: we read the `slidemacs` macro package–`groff`'s `.mso` directive is the same as `troff`'s `.so` directive but it uses the macro package search rules, so it will begin looking in `/usr/local/share/groff/tmac`.

Our major `while` loop beginning on line 11 reads each record of the input–remember they're now separated by blank lines–and processes them. Line 13 ensures that backslashes aren't eaten unnecessarily by `troff`. The `if` statement at line 15 processes the bullet items tagged in our input file with an asterisk, turning them into `.2` macros for `groff`. The `if` statement on line 20 processes the `INDEX` line in the input.

There's a slightly complicated `if-else` between lines 25 and 43 that bears a little study. In the `if` clause, we take any block of lines beginning with white space, strip off the white space and bracket the lines with a `.b/.e` pair of macros. Why? This handles an inset display–generally, a code example. The `else` clause, on the other hand, deals with `pod`-like font changes, converting double-hyphens into en-dashes.

Fairly simple, right? Which means most of the magic of what's happening is pushed down into the `slidemacs` macro.

Listing 1. The pp2roff Script

```
1   #!/usr/local/bin/perl -w

2   use strict;

3   $/ = '';

4   my $author = (getpwuid($<))[0];
5   my $title = @ARGV ? " \\- $ARGV[0]" : "";
6   my $TOP = <<EOF;
7   .\\"-----------------------------------
8   .au "$author$title"
9   EOF

10  print ".mso slidemacs\n.ns\n";

11  while (<>) {

12     chomp;
13     s/\\/\\e/g;
14     s{^=\s*(.*)}{$TOP\n.c "$1"\n};
15     if (/^\*\s*/) {
16       # handle bullet items
17       s/^  (?=\S)//gm;
18       s{^\*\s*}{.2\n}gm;
19     }

20     if (/^INDEX$/) {
21       # produce an index slide
22       print "\n.IX\n";
23       next;
24     }

25     if (/^[\t ].*\S/s) {
26       s/^    //gm;
27       print <<EO_DISPLAY;
28  .b
29  .sz -7
30  .sp .5
31  $_
32  .sp .5
33  .sz +7
34  .e
35  EO_DISPLAY
36     } else {
37       s{I<(.*?)>}{\\f2$1\\fP}g;
38       s{C<(.*?)>}{\\f(CB$1\\fP}g;
39       s{B<(.*?)>}{\\f3$1\\fP}g;
40       s/\s*--\s*/ \\(en /g
41         if( ! /^.\\\"/ );
42       print $_, "\n";
43     }
44  }
```

The Macros in Question

Space limitations prevent us from doing a complete reading of `slidemacs`, but we'll touch on the high points to give you a flavor for some `troff` tricks you may not have seen. You can collect the entire set of macros from our Web site, `http://alumni.caltech.edu/~copeland/work/`, or from CPAN. We've heavily modified the `slidemacs` macro package from the `perlpoint` distribution.

Let's begin with the setup. We define whole flock of numeric constants:

```
'\" Parameters:
'nr PW 11i
'\"   PW=paper width
'nr PH 8.5i
'\"   PH=paper height
'nr MB 1i
'\"   MB=margin border
'nr MT 0.5i
'\"   MT=margin text
'nr BW \n(PWu-\n(MBu-\n(MBu
'\"   BW=border width
'nr TW \n(BWu-\n(MTu-\n(MTu
'\"   TW=text width
'nr C  0.5i
'\"   C =corner radius
'nr PT \n(MBu+\n(MTu
'\"   PT=page offset for text
'nr BB \n(PHu-\n(MBu
'\"   BB=bottom of box
'\" 0.7i is amount the logo sticks up over line
'nr CH 0.6i
'\"   CH=height of logo (above base)/2
'nr BT \n(BBu-\n(CHu-1v
'\"   BT=bottom of text
```

Notice that we've calculated some of these from other values. Also, we've used the single-quote mark as an introducer, rather than the normal `troff` dot. This is a carryover from the original and is overkill. As you may know, the single quote performs the directive without doing a line break. For the setup, this is probably unnecessary.

The next step is to set the parameters for `troff`:

```
'pl \n(PHu
'll \n(TWu
'lt \n(Twu
'ev 1
'll \n(TWu
'lt \n(TWu
'ev
'po \n(PTu
'wh \n(BTu fa
'wh 0 hd
.em FL
'ds ff "''' \*(DA '"
'de Ft\" footer
'ds ff \&\\$1
..
```

We're setting up the line length and title length–`.ll` and `.lt`–in both the base and alternate environments. (Environments are `troff`'s way of allowing you to have different setups for the page headers, footers and body text. They can be extended to other uses, too, such as footnotes.) The alternate environment is the one in which we will draw the frame around the box. We set up two traps with the `.wh` directive: one at the top of the page, which invokes the header macro `.hd`, and one that invokes the `.fa` macro at the bottom of the page. We also ensure that we flush the last bullet item by invoking `.FL` as the end-macro. Last, we set the footer text to the date, the `DA` string, but we can reset it with the `.Ft` macro.

From all that setup, we proceed to the `.c` macro, which gets translated from a line starting with an equal sign and begins a slide. That is, `pp2roff` converts a line like

```
=This is a title
```

into

```
.c "This is a title"
```

Note the odd use of the `.FL` macro in the definition of `.c` (more on this later).

```
.de c
.FL \" finish the previous bullet item
.bp
.sp 1v
.in 0
```

```
.na
.\" if we have an argument, it's the
.\" first of a series of slides, else
.\" we just increment the slide number
.\" and print it.
.ce  \" the title line is centered
.ps +3 \" and slightly larger
.ie !@@\\$1@ \{\
.  ds @t "\\$1
.  nr @p 1
\&\\*(@t
.if \\n[index-on] \{\
.da @x  \" -- divert to the index
\&\\*(@t ... \\n%
.br
.da
.\}
.\}
```

Notice that this macro anticipates the possibility of being invoked without an argument. To handle that case, we keep a part number, `@p`, and save the title string in `@t`. We take the slide title argument and append it to the `@x` diversion in which we are caching the titles. (For the uninitiated, a `troff` diversion captures text into a buffer, but doesn't print it. It's a trick that allows you to save the text for a footnote, an index or other text that is gathered out of sequence with its printing. Because you can measure the aggregate height and width of the captured text, it also allows you to decide when you need to end the body text so the whole footnote will fit on the page.)

Also notice that we've committed to using `groff` with `slidemacs` by not using a two-character register name, which traditional `troff` requires. Finishing up the `else` side of the `if`

```
.el \{\
.  nr @p \\n(@p+1
\&\\*(@t (part \\n(@p)
.\}
.ps
..
```

we've incremented the part number and generated a title line with the original title and part number, for example, "Sample Slide (part 3)."

In practice, we only use one level in the bullet lists on our slides. So even though we have different levels of bullet in the macro package, the one we translate asterisks into from our `perlpoint` input is `.2`:

```
.de 2
.KP
.in 0.5i
.ti -\w'\(bu\h@0.15i@'u
\(bu\h@0.15i@\c
.if !@@\\$1@ \&\\$1
..
```

This is fairly simple: we do a negative indent slightly wider than a bullet, we put out the bullet and, if there's an argument, we drop that text next to the bullet. If there's no argument, it can follow the macro because we have not done a line break. Note that we didn't use the `.FL` macro, we used `.KP` instead. We'll explain both of these tricks now.

You'll remember that we did some hand-waving above in the `.c` macro, which starts a slide, allowing us to continue onto a new slide with an incremented part number. This feature lets us continue adding items to our slides without having to worry about them overflowing. The other thing required to make this feature work is to capture the bullet item that would have forced the overflow onto a new page and wait until the page break to print it. In other words, each bullet item must be captured and cannot be printed until we determine whether it will fit on the page. The `.KP` macro starts the capture and the `.FL` macro does the flush:

```
.de KP \" start a keep for a new bullet item
.FL
.di @k
..
```

The new bullet item is saved into diversion `@k`, but only after we've flushed the old bullet item with `.FL`:

```
.de FL \" flush the current bullet item
.br
.if !@\\n(.z@@ \{\
.  br
.  di \" close the diversion
.  \" bullet item diversion
.  if \\n(dnu>\\n(.tu  .PF
.  nf
.  in 0
.  @k \" now print the bullet item
.  in
.  fi
.\}
..
```

Remember that we're invoking `.FL` before we start the new bullet. We don't even bother if there is no active diversion. It is a peculiarity of `troff` that the name of the current diversion is stored in a *number* register named `.z`. So if the string contained in `\n(.z` is empty, we have no diversion and, hence, no bullet item in progress.

On the other hand, if that register does contain something–presumably `@k`, the name of the bullet diversion–we close the diversion. If it is too long to fit on the page–that is, if the diversion's height in the `dn` register is more than the distance to the bottom-of-page trap in register `.t` –we invoke `.PF` to finish the page. In any event, we print the last bullet item by invoking the diversion as though it were a macro.

The `.PF` macro is pretty simple:

```
.de PF \" page flush for overflow slide
```

```
.c
.sp 1v
..
```

We rely on the ability of `.c` to remember the last title it was given, and use it to begin the new page. This is an implicit insertion of a line like the following into our input:

```
= Test slides (part 2)
```

The `.sp` generates the blank line we normally insert after a title.

Finishing Up

We only have a few more things to do to finish up the macro package. The index is fairly simple:

```
.de IX \" print index
.nr index-on 0   \" don't index the index
.FL
.c "Slide Index"
.sp
.nf
.@x
.nr index-on 1   \" turn indexing back on
..
```

We turn off indexing by resetting the `index-on` number register. This prevents our title line for the index slide from appearing in the index. Then we flush the last slide, which actually begins the index slide. To produce the body of the index slide, we just spill the contents of the `@x` diversion. We finish by turning the `index-on` register back on, in case we have more slides to come.

Last, we need to take a brief look at how we format code. As we noted earlier, we bracket code with a pair of `.b` and `.e` macros:

```
.de b
.ft CB
.ps 23
.nr X 8*\w' 'u
.ta +\\nXu +\\nXu +\\nXu +\\nXu \
  +\\nXu +\\nXu +\\nXu +\\nXu
.nf
..
.de e
.ft R
.ps 23
.vs 23
.fi
..
```

The `.b` macro forces us into Courier Bold font, sets the point size, gives us tab stops at every eight spaces and sets us to no-fill mode. The inverse `.e` macro returns us to the body Roman font, regular point size and fill mode. (OK, our code is not actually presented in a different point size, but we can change these macros depending on the availability of fonts on our printer.)

We've shown you some of the `perlpoint` package. It fills an important niche now that with the advent of `groff`, the `-mv` macro package is usually not available. Mostly, this has been a tutorial in setting up `troff` macros. Take this knowledge and your own needs for presentation slides, and extend `perlpoint` for your own niche. Then donate your extension back to the Perl community by sending it to the CPAN or to Tom Christiansen. Or as the Sirius Cybernetics robots say, "Share and enjoy."

Next time, we will continue the theme with a different purpose-built macro package, which will allow you to amaze your friends in an entirely different way.

Until then, happy trails.

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`*.*