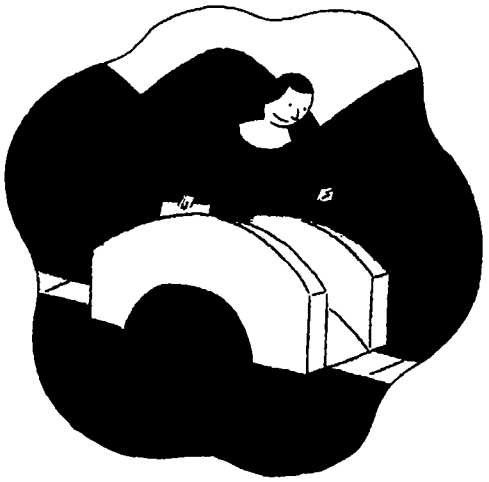
A Bridge Column

by Jeffreys Copeland and Haemer

Happy new year! Our last two columns covered diaries, appointments and to-do lists. This column will bridge that topic and our next one by finishing up some odds and ends for to-do lists and beginning some background discussion in preparation for next time.



Checkboxes

In the troff versions of diary and to-do examples, we check off boxes as we complete items. How do we label the items as complete? Well, we can print our to-do logs out and manually check off items that we have completed. Or we can rely on troff and the fact that a square root sign looks like a check mark. We rely on the -mm macro package list facility. Each task in the list is a separate list item. We specify a tagged list with a square as the default tag: .ML\(sq. Next we use an alternate tag for items that we have completed, relying on the troff overstrike facility: .LI.\o'\(sq\(sr'.

```
.de todo@done\"mark an item "done"
\(sq \h'-1m'
..
.ds dn \o'\(sq\(sr'
.de DN
.LI \*(dn
..
.ML \(sq
.LI
this has yet to be done
.DN
```

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

```
Here's a done item.
.LE
```

Macro

Having put together a facility for marking up the to-do list, it would be useful to have a simple method of marking the items. Fortunately, we can use the macro facility of vi to set up a one-key tag:

```
map q O.DN^[^M
```

One Shots

As you'll recall, we have a file events, which contains a list like this:

```
09/07/95
    November RS column due
    Take cat to vet

09/21/95
    Gillian's birthday

10/06/95
    run off to join the circus

12/25/95
    half day off
```

It would be useful to have a simple appt script to add items to events for us, which we would use in the form:

```
appt 11/20/95 language release
```

```
#! /usr/local/bin/perl
# add a one-shot entry (an appointment)
# to our events file; assume that date
# is the first argument

open( O, ">>events" );

$date = shift(@ARGV);

# now we need to do some massaging of
# the elements of the date, to ensure
# that we can find them again later,
# by converting "6/5" to "06/05"
@datebits = split(/\//, $date);
$datebits[0] =~ s/^[1-9]$/0$&/;
$datebits[1] =~ s/^[1-9]$/0$&/;
$date = join('/',@datebits);


# here we should check for a valid date

print O "\n$date\n @ARGV\n";
```

This script is pretty straightforward. We begin by opening the events file for appending. We grab the date as the first argument from the command line. Notice that we go through some machinations to convert a date of the form 6/5 to 06/05, so that it can be recognized by the todo script that we showed you last month. Lastly, we append the remaining arguments and the date as a separate paragraph to events. (Exercise for the reader: Add code to validate the date.)

Old Events

It would also be useful to have a script to purge the events list; otherwise the file will grow without bounds as we add events over time. This is a little more complicated than the previous script.

```
#! /opt/local/bin/perl
# purge the events file of items
# that are in our past, so the
# file doesn't grow without bounds

open( I, "events" );
$old_RS = "";
$/ = "";
@events = <I>;
$/ = $old_RS;
close I;

($month, $day, $year) = split('/', `date +%D/` );
# (why the extra '/'? to prevent $year
#  from bodily including a '%n')

# century turning alert!
for( $i = 10; $i < $year; $i++ ) {
  @events = grep(!/[0-9][0-9]\/[0-9][0-9]\/$i/,
                 @events);
}

for( $i = 1; $i < $month; $i++ ) {
  $p = sprintf( "/02d/[0-9][0-9]/$year", $i );
  @events = grep( !/$p/, @events );
}

for( $i = 1; $i < $day; $i++ ) {
  $p = sprintf( "$month/%02d/$year", $i );
  @events = grep( !/$p/, @events );
}

open( O, ">events" );
print O @events;
exit;
```

We begin by opening the events file and reading the entire file into an array @events. We use a blank line as a

record separator by setting the value of `$/` to null.

We use a variant of our massive date trick from `todo` to get the month, day and year. We remove from the array any date before the current year using the `perl` function version of `grep()`. Note the comment in the code: This first `for()` loop will fail at the turn of the century. (Reader exercise: How to fix this?) Similarly, we remove any date containing an earlier month in the current year, and lastly any date earlier in the current month. We complete the task by writing out `@events` as trimmed.

HTML has suddenly become the markup language of choice for many applications.

Markup Languages

Markup languages are at one end of a spectrum of text formatting tools. The other end is what-you-see-is-what-you-get word processors. Traditional markup languages include such things as `roff`–which is immortalized in Kernighan and Plaugher's *Software Tools* (Addison-Wesley, 1976, ISBN 0-201-03669-X); `troff` (Ossana and Kernighan's reimplementation of the concepts of `roff`); `runoff`–the old favorite from DEC timesharing systems; and TEX which is Knuth's experiment for doing the typesetting of *The Art of Computer Programming*.

Eventually, someone noticed that if you did the markup correctly, you were marking up the document for structure–that is, chapter headings, paragraphs, tables, etc.–rather than layout. Correctly designed, a `troff` macro package should allow you to concentrate on structure, not the minutia of font changes. If you want to change the layout, you can adjust the macro package without disturbing the structure of the document.

A general solution to the problem of separating structure from layout is the family of languages called SGMLs (Standardized General Markup Languages). An SGML is intended to mark up the structure of a document, not its layout. SGML tags are in the form `<foo>` or `</foo>` –where the latter is used for closing a bracketed item. For example, `<Section head>Markup Languages</Section Head>`. (LaTEX afficionados will note that it attempts to do roughly the same thing: It marks up the structure, allowing you to specify the layout by choosing a style, and brackets document elements with the likes of `\begin{table}` and `\end{table}`.)

There is a very common example of an SGML that you're probably using without knowing it: HTML– the Hypertext Markup Language underlying Web pages (Web-surfing is hip enough at the moment that even *Time* magazine has noticed it). HTML has suddenly become the markup language of choice for many applications.

It's instructive to look at some HTML tags and their meanings from a table (see Table 1) partially cribbed from Dougherty, Koman & Ferguson's *The Mosaic Handbook for the X Window System* (O'Reilly, 1994, ISBN 1-56592-095-3).

The tags are case-insensitive. Both `<IMG>` and `<A>` require some additional information: `<IMG>` requires a pointer to the image in question, such as `<IMG SRC="jsh+jic.gif">`; `<A>` requires information about where to chase the link, such as `<A HREF="http://www.dsiegel.com/">`.

Beware, there's a lot of bad HTML

Table 1. HTML Tags

`<HTML>`	Begin document	`<EM>`	Emphasize text
`</HTML>`	End document	`</EM>`	End emphasis
`<HEAD>`	Begin heading text	`<B>`	Begin bold text
`</HEAD>`	End heading text	`</B>`	End bold text
`<TITLE>`	Begin title	`<I>`	Begin italic text
`</TITLE>`	End title	`</I>`	End italic text
`<BODY>`	Begin document body	`<ADDRESS>`	Begin an address block
`</BODY>`	End document body	`</ADDRESS>`	End an address block
`<H1>, <H2>...`	Begin heading level 1, 2...	`<BLOCKQUOTE>`	Begin block quote
`</H1>, </H2>...`	End heading level 1, 2...	`</BLOCKQUOTE>`	End block quote
`<P>`	Paragraph	`<A>`	Begin an anchor–a hypertext link
` `	Line break	`</A>`	End anchor
`<HR>`	Horizontal rule	`<IMG>`	Insert image here

out there. In addition, Netscape has the tendency not to care about line lengths or endings, so many documents on the Web confuse the ASCII, CR and LF characters, or worse, represent the entire document as a single line. California-based typographer David Siegel has some definite views concerning issues of on-line style. His Web page at http://www.dsiegel.com/ has some interesting discussion about the uses and abuses of HTML.

Preprocessors

You should also be familiar with the concept of preprocessors. Earlier we mentioned Kernighan & Plaugher's *Software Tools*, which developed a set of tools in ratfor–a block-structured language which is preprocessed into FORTRAN, a case of transforming one language into another. Table 1 was formatted with the troff preprocessor tbl, which also converts one language to another. We, of course, also have the tried and tested C preprocessor, CPP, which transforms macros into raw C code to be compiled. (Macros are also preprocessed by the formatters troff and TEX as we've already discussed.)

In other applications, we occasionally rely on the same input data to provide two distinct output forms. A simple example of this is nroff's ASCII output versus troff's typeset version of the same text. Similarly, using LaTEX with different parameters to the style specifier, changes the rendition.

In a slightly different vein, there are ways to embed troff source for manual pages within Perl programs. And we've been known to similarly abuse the C preprocessor by embedding manual pages for our utilities within the C source of our programs, surrounded by #ifdef DOC and #endif DOC.

Earlier in this series we demonstrated an equally radical transformation of input data by producing letters and envelopes from the same input text but using different formatting macros.

Next Time

Consider the following problem: HTML, as we have noted, is (in principle at least) a markup language dealing with the structure of a document, not its formatting. Web browsers render that markup into formatted output. How can we process a markup language source into one for a formatter language, or vice versa, without having to build a completely new formatter to do the translation for us? We'll give you 30 days to explore that nasty exercise for yourselves. Until next time... ▲