by Jeffreys Copeland and Haemer

Comparing Text, Part 2

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats, and being a thorn in the side of his local school board.*

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.html`.

Last time, we began building a wrapper for `diff` that allows us to compare running text. Why would we want such a thing? To isolate the differences between two versions of a draft RFC, or two versions of a formatted email message, for example.

Our basic strategy is to strip each of the two files into a list of words, one per line, and do an `sdiff` (or `diff -y`) on them. We then postprocess the differences and fold them back into something readable, with the differences highlighted. Last month, we finished the shell script wrapper, called `redline`, which is very simple. We'll include it here for reference:

```
#! /bin/sh
# $ Id: redline,v 1.1 1997/10/13 14:59:39 jeff Exp $
#   This does a diff on running text,
#   in the same style as a Word or
#   WordPerfect red line comparison.

#   set the cleanup
trap ‘rm -f /tmp/$$*’   EXIT HUP QUIT INT TERM

#   check that file arguments are present
[ -z "$1" -o -z "$2" ]   &&
 echo usage:  $0 file1 file2   &&
 exit

#   begin by breaking files into a word per line
#   ensure that paragraphs are handled nicely
#   whether they’re indented or preceded by
#   blank lines
expand $1 | sed -e ‘s/^   */\
/’  | fmt -2 >/tmp/$$a
expand $2 | sed -e ‘s/^   */\
/’  | fmt -2 >/tmp/$$b

#  now do an sdiff, and collect the differences
diff -y /tmp/$$a /tmp/$$b | expand | reddiff
```

The guts of the whole thing is the `reddiff` program, which formats the differences into something readable. We had just finished with the `parse()` routine from `reddiff` when we ran out of time last month. For review, `parse()` takes a line of input and two `char **`s as arguments. It returns the `sdiff` difference indicator (<, >, `|` or blank) and the words it found on the input line.

Beginning where we left off, let's proceed with the rest of the `reddiff` program. We'll continue by laying out the global declarations:

```
char line[BUFSIZ];
char bufcomm[BUFSIZ];
char bufnew[BUFSIZ];
char bufold[BUFSIZ];

#define ANY(bp)  (*bp)
#define WIDTH   75
```

We need an input line, and buffers for common text–text in the new version and text in the old version. We add a macro for detecting text in the buffer. Because, in the end, we'll be filling the text we gather, we define the width.

Given that, we can begin the main program:

```
main()
{
   char *word1, *word2,  tag;
   char **wp1 = &word1,  **wp2 = &word2;

   bufcomm[0] = bufnew[0] = bufold[0] = 0;
```

We declare pointers to strings and pointers to pointers, and initialize the buffers to the empty string before beginning. Next, we'll begin the main loop:

```
while( fgets(line, BUFSIZ, stdin) != NULL )
{
   tag = parse(line, wp1, wp2);
```

We read each line from the output of `sdiff`. Then we use the routine we wrote last time to extract the words from the input line. It takes a pointer to the line and returns pointers to the two words from its argument list.

We have to deal with a special case next: If we've only got a separator, but no words on the line, we've got a paragraph break:

```
/* special case:  if we only have
a separator, we've
got a paragraph break */
if( ! *word1  &&  ! *word2 )
{
   spill();
   showme("",'x');  /* forces a
          reset of line counts */
   printf((tag == '') ? "\n\n" : "\n%c\n", tag);
}
```

We have two new routines: `spill()`, which outputs the current buffers, and `showme()`, which maintains the state of those buffers and shows them as needed. We'll explore these two routines later.

We now need to deal with the general case of the return from `parse()`:

```
/* store the words on the line */
switch( tag ) {
case ' ':
   if( ANY(bufold) || ANY(bufnew) )   spill();
   addword(bufcomm, word2);
   break;
case '|':
case '<':
case '>':
   if( ANY(bufcomm) ) spill();
   addword(bufold, word1);
   addword(bufnew, word2);
   break;
default:
   fprintf( stderr,"huh? what? %c\n",tag );
   break;
}
```

Each time we need to add a word to a buffer, we use the `addword()` routine. We need to dump the accumulated paragraph buffers at specified times. For example, when we switch from common text to text in either new or old versions, or vice versa, we need to invoke `spill()` to dump the partial paragraph.

This completes the main `while` loop. When we drop out of that loop, we spill the partially accumulated output and finish:

```
   spill();
   printf("\n");
   return(0);
}
```

The main program completed, we can now deal with the service routines.

The Service Routines

We have postulated three routines to deal with the text that's flowing into our filter: `addword()`, which puts words from the input streams into the buffers; `spill()`, which spills out the contents of the stored buffers; and `showme()`, which manages and shows the individual buffers. Let's start with the simplest:

```
void
addword( char *buf, char *word )
{
   /* if the word is null, forget it */
   if( ! *word )
      return;
   /* ensure that we don't overflow the buffer */
   if((strlen(buf)+strlen(word)+2) >= BUFSIZ)
      spill();
   /* now add the stuff to the buffer */
   if( *buf )
      strcat(buf," ");
   /* add extra space at end of sentence */
   if( *buf  &&  buf[strlen(buf)-2] == '.' )
      strcat(buf," ");
   strcat(buf,word);
}
```

Roughly, we just string the word onto the end of the specified buffer, spilling out all the buffers if this one is full and adding an extra space if the word represents the end of a sentence.

Exercise for the reader: Add an extra space at the end of all sentences, including ones that end with exclamations and parentheses.

Next, we can piece together the `spill()` routine, which simply invokes `showme()` for each buffer:

```
void
spill( )
{
   /* output the common buffer, first */
   showme(bufcomm, ' ');
   /* next, do the old text */
   showme(bufold, '<');
   /* finish up with the new text */
   showme(bufnew, '>');
  }
```

We call `showme()` with the buffer and a marker.

"Yeah, yeah, yeah," say those of you who learned discourse from our 12-year-old daughters, "but what does this silly `showme()` routine do?"

That's a good question, and we're glad you asked it. Let's think quickly about how we want the output to appear. We want to fill the lines of the buffers as we output them, preceding each line by a flag character to tell us whether these words appear in both files or just one. For example, using the test files from last time, we'd want output something like this:

```
  The
< Way that can be told of is not the eternal
< Way; The name that can be named is not the
< eternal name. The
  Nameless is the origin of Heaven and Earth;
  The
< Named
> named
  is the mother of all things.
>
  Therefore let there always be non-being, so
  we may see their subtlety, And let there
  always be being, so we may see their outcome.
  The two are the same, But after they are
  produced, they have different names.
> They both may be called deep and profound.
> Deeper and more profound, The door of all
> subtleties!
```

Thus, we start the routine with the usual flock of declarations:

```
/* This routine, which prints the
   actual text, is where the really
   messy formatting stuff happens.
   We need some retained state between
   invocations, and some other stuff. */
void
showme( char *buf, char marker )
{
   char *s, *end;
   /* length of last partial line */
   static int current_length = 0;
   static char last_marker = 0;
```

Because this is the routine that fills the lines for us, we need to keep track of the length of the last partial line, and we need to keep track of the last marker we printed out. If the marker is different between two invocations, we need to start a new line on the output. If we've filled a buffer and spilled it out, we want to know where we stopped on the page, so we don't end up with lines of different lengths.

Given that, we need to dispose of a special case. If we want to declare that the persistent state should be discarded, the easiest way to do it is to set the last marker to something we won't normally see, for example:

```
/* special for resetting the persistent state */
if( marker == 'x' )   last_marker = 0;
```

Also, if the buffer is empty:

```
/* don't bother if the buffer's empty */
if( ! ANY(buf) )   return;
```

If we are trying to spill a different buffer than we did on the last call to `showme()`, we need to start a new line and reset the marker:

```
/* deal with a new kind of spill */
if( last_marker &&
  ( marker != last_marker ||
    current_length == 0) )
      printf( "\n" );
if( marker != last_marker )
      current_length = 0;
```

Because we've added extra spaces, we may need to skip some of them:

```
/* because we force a second blank
   after each full stop, we may need
   to skip a blank beginning a line */
if( *(s=buf) == ' ' )  s++;
```

We're filling lines with the text from our buffers, so we need to do some line breaking. We do that via a couple of steps in a big `while` loop:

```
/* add some line breaks to the huge
   string and print it */
while( (strlen(s)+current_length) > WIDTH )
{
   /* find the maximal end-of-line,
      and find the preceding space */
   end = s + WIDTH - current_length;
   while( *end != ‘ ‘  &&  end > s )  end--;
   /* we occasionally have too long a word */
   if( end == s )
   {
       printf("\n");
       current_length = 0;
       continue;
   }
```

We go for the longest possible line and backtrack for a space. We do a bit of work for the special case of a word longer than our line length. In each case, we terminate the chunk of buffer with a `NULL` character.

After that, we need to print a segment of the buffer. Again, we do some special-case work if we're dealing with a word longer than our line length:

```
   if( end > s )
   {
       *end = 0;
       printseg(s, marker, ‘\n’, current_length);
       s = end + 1;
       if( *s == ‘ ‘ )  s++;
   } else {
       /* this handwave is for real long words */
       char oops;
       end = s + WIDTH;
       oops = *end;
       *end = 0;
       printseg(s, marker, ‘0, current_length);
       *end = oops;
       s = end;
   }
   current_length = 0;
}
```

We finish up the `showme()` for the buffer by printing the trailing bit of the buffer, saving information about what we just put on the page, and marking this buffer so that we don't reprint it:

```
/* print the last little bit */
if( *s == ‘ ‘ )  s++;
printseg(s, marker, 0, current_length);

/* save data about the last state */
current_length = strlen(s);
last_marker = marker;

/* reset, so that we don’t
   reprint this puppy */
*buf = 0;
}
```

The last bit of code we need to write is a function to print the buffer segment used by `showme()`. It takes the pointer to the buffer; the difference indicator, such as < or >, to prepend to the line; the character to append to the end of the buffer segment; and the current length of the printed line. (The last argument is needed because if there are characters on the line already, we don't need to print the difference marker.)

```
void
printseg( char *s, char marker,
     char terminator, int current_length )
{
 if( !current_length )
     printf( "%c", marker );
 printf( "%s", s );
 if( terminator )
     printf( "%c", terminator );
}
```

With that, we're done, save for some function prototypes and include files at the top of the source. As always, pick up the source code from our Web site if you'd like to try it out for yourself.

Alternate Implementations

As we discussed at the beginning of last month's column, there are several other ways we could have approached this problem. Even using this approach–filtering the output of `sdiff`–there are alternate paths we could have taken.

Much of the code in the program we've just written deals with line filling. We could have relied on `nroff` to do this for us. In other words, the last line of our `redline` script would look something like this:

```
diff -y /tmp/$$a /tmp/$$b | expand |
   reddiff | nroff -mdiff -
```

Exercise for the reader: Build this version of `reddiff` and a `diff` macro package to format its output. (We didn't use this approach in our version because we don't have `nroff` on our DOS laptop machines, and we initially needed the functionality at standards meetings. We also couldn't think of an `nroff`-based version built around anything other than a flock of diversions in the macro package; if *you* can, we'd be interested in seeing it.)

Similarly, we can use the formatting capabilities of `troff` to provide real font changes for the output, for example, rendering the mainline text in a roman font, the new text in italics and the old text overstruck. Exercise for the reader: Provide *this* version of `reddiff`.

Last, we could use the `terminfo` database to render the new and old text in different forms, such as highlighted and underlined, on our terminal screen. Exercise: Revise the code we built in this column to do that.

We don't have a clue what we're going to do next month. Until then, happy trails.