

This file is copyright © 2006 Mark Jason Dominus.
Unauthorized distribution in any medium is
absolutely forbidden.

1. Can't Happen
 2. Comments
 3. Declarations
 4. GetDateNow()
 5. Main flow
 6. ProcessAlertLog()
 7. AlrtError()
 8. StartStopRep()
-

Oracle Log Analysis

When I review these programs, I usually make several passes.

Sidebar: Hard copy

I always start by printing out the code on paper. It's easier to read things on paper than on the screen. While I was working on *Higher-Order Perl*, I mentioned to my editor that I found many more errors when I read the manuscript in hard copy form than when I read it on the screen. He said that was true for everyone, and professional copy editors *always* work from hard copy.

Another benefit of the hard copy is that it gives me a place to make written notes. I usually print the code on one side of the page only, which leaves the backs for longer notes and scribbly diagrams. I find that having several different color pens or highlighter markers is handy; when I do, I note the errors in red, possible problems in orange, places where I have questions in purple, and things that look wrong but aren't in blue. This may be excessively elaborate; a blue ballpoint works well too.

I preprocess the source code with a program called `a2ps` that adds page numbers and filename labels to each page. I'm always tempted to cut corners and send the source code directly to the printer, but I really need the page numbers. If the pages of the program get mixed up, I have an awful trouble sorting them out again.

The first pass is just a quick scan. I glance over the program to see if anything jumps out. After that, I make a longer pass, usually in straight top-to-bottom order, reading the code line-by-line, trying to understand what it does and how, and making notes as I go. That's the same way I do things as a maintenance programmer when I'm asked to maintain something new.

In this case, the quick pass turned up several points of interest. Most obviously:

```
303         sub SplitDate {
312         ...
313             if ( $mnt eq 'Jan'){ $mt = 1 }
314             elsif ( $mnt eq 'Feb'){ $mt = 2 }
315             elsif ( $mnt eq 'Mar'){ $mt = 3 }
316             elsif ( $mnt eq 'Apr'){ $mt = 4 }
```

```

317         elsif ( $mnt eq 'May' ){ $mt = 5 }
318         elsif ( $mnt eq 'Jun' ){ $mt = 6 }
319         elsif ( $mnt eq 'Jul' ){ $mt = 7 }
320         elsif ( $mnt eq 'Aug' ){ $mt = 8 }
321         elsif ( $mnt eq 'Sep' ){ $mt = 9 }
322         elsif ( $mnt eq 'Oct' ){ $mt = 10 }
323         elsif ( $mnt eq 'Nov' ){ $mt = 11 }
324         elsif ( $mnt eq 'Dec' ){ $mt = 12 };

```

This case of **repeated code** is also a good example of how **repeated code harbors bugs**, and bugs that are hard to see. Line 314 will never be executed. This won't affect the behavior of the program, but it's unquestionably erroneous.

As it is so often, the right thing to do here is to use a table:

```

my %m2n = (Jan => 1, Feb => 2, Mar => 3, Apr => 4,
          May => 5, Jun => 6, Jul => 7, Aug => 8,
          Sep => 9, Oct => 10, Nov => 11, Dec => 12);
$mt = $m2n{$mnt};

```

and 13 lines have become 4.

Another problem with the original code is that it has `elsif` without `else`. And, as before, the question to ask is "What would happen if none of the cases were true?"

Can't Happen

It's tempting to say that that "can't happen", but that is nearly always the wrong answer. There are different degrees of "can't happen". In this case, the `else` clause could be reached only if the argument to `SplitDate()` were malformed. And of course, that can't happen, because function arguments are never, ever in the wrong format. Har! The original code should have had:

```

else { die "Malformed month '$mnt' in date argument '$sdat'" }

```

or something of the sort. With the table approach, this becomes:

```

my %m2n = (Jan => 1, Feb => 2, Mar => 3, Apr => 4,
          May => 5, Jun => 6, Jul => 7, Aug => 8,
          Sep => 9, Oct => 10, Nov => 11, Dec => 12);
$mt = $m2n{$mnt}
|| die "Malformed month '$mnt' in date argument '$sdat'";

```

Comments

This book has a lot of negative advice about comments. But here's a comment I really like:

```

303 sub SplitDate {
304
305     # Wed Jul 27 17:07:45 2005
306
307     my $sdat = shift ;

```

Sidebar: Comments

I have a love-hate relationship with comments. All too often, comments are used as a crutch to excuse unclear or badly-written code. People write bad code that they *know* is bad, and then shrug and throw in a comment to make themselves feel better. All too often, people trying to establish coding standards or guidelines are distracted by comment policy to the detriment of more substantive matters.

Rob Pike, co-author of *The Unix Programming Environment*, says:

=beginquotation

I tend to err on the side of eliminating comments, for several reasons. . . if the code is clear . . . it should explain itself.

=endquotation

Some people scoff at the suggestion that clear code explains itself. But I suspect those people don't know what clear code looks like.

It's not a perfect comment, but unlike many comments its presence is far better than its absence would have been. It enables the reader to see at a glance what format the input is expect to be in, and to check whether the following code processes it correctly:

```
310     my ( $dt, $mnt, $dn, $drest, $yr) = split /\s+/, $sdat ;
311     my ( $hr, $mi, $sec ) = split /:\/, $drest ;
```

Without the comment, the reader would have to figure out where `splitDate()` was called from, where its argument was manufactured, and then compare the (possibly far-away) manufacturing code with the parsing code. With the comment, you can check it and move on. Some comments can be obviated by improvements to the code, but I can't think of any improvement to the code that would obviate this one.

The only thing I think this comment needs is to be moved closer to the thing that it documents:

```
my $sdat = shift ; # Wed Jul 27 17:07:45 2005
```

Declarations

Had I been writing this, I would have done the whitespace a little differently, and I would have formatted the debugging message differently. But these things aren't very important. The only other change I feel strongly about here is to eliminate line 308:

```
308     my $mt ;
```

There's no longer any reason to declare the variable at the top of the function; now we can declare it where it's used:

```
my $mt = $m2n{$mnt} || die "Date '$sdat' malformed";
```

At the very least, moving the declaration to the place where the variable is defined reduces the scope of the variable, which is never a bad thing. The maintenance programmer trying to understand the function of the variable has that much less code that they must consider.

With this last change, the function is now 9 lines long, down from 19.

GetDateNow()

The function `GetDateNow()` spans lines 71-145. 36 of these lines are blank. This is not a good use of whitespace. Properly used, whitespace can communicate information about program structure, as it does in ordinary prose. Here, it just takes up valuable real estate. The function could have fit in a single screen, so that the reader's eye could take it in all at once, and refer back to the beginning while reading the end. But the excessive whitespace spreads the functions' code over two screens. In the quotations in this section, I'll omit the excess blank lines.

The most important thing wrong with this function is the way that it does calendar arithmetic:

```

98         my $datev = `date +%a %b %e %m %Y` ;
100         print "The date is $datev\n";
101
102         my ($dy, $mth, $dd, $md, $yd ) = split/\s+/, $datev ;
104         my ($y, $m, $d) = Add_Delta_Days( $yd, $md, $dd, -$value) ;
105
106         $daynum = $d ;
108         my $wday = Day_of_Week( $y, $m, $d ) ;
110         my $day2 = Day_of_Week_to_Text( $wday ) ;
112         my $m2 = Month_to_Text ( $m ) ;
114         $day = substr( $day2, 0, 3 ) ;
116         $mon = substr( $m2, 0, 3 ) ;
```

This is a lot of work, most of which is unnecessary. Date formatting is a common source of overwritten code. Many of the problems programmers have with date formatting can be repaired with a little well-applied advice.

Dates come in essentially three different formats, which I will refer to with the clever names "format A", "format B", and "format C".

Format A is the format used internally by the operating system. On Unix systems, format A is a simple count of the number of seconds that have elapsed since the beginning of 1970, but nearly all computer systems have something similar. Format A is unreadable, but is easy to calculate with:

```
1080891516    # Seconds since 1970
```

Clearly, this is useless for user interaction; if you tell someone to meet you for lunch at 1080891516, they have no idea when to show up. But it has some tremendous advantages over other formats. Do you want to know which of two dates is earlier? Just compare the two numbers; the lower number is earlier. Do you want to know how much time elapsed between the two dates? Just subtract; that is the elapsed time in seconds.

Are you worried about leap days and leap seconds and daylight savings time and time zones and other oddities perpetrated by civil authorities? Format A ignores all these; 1080891516 is the same real moment in all countries everywhere, and one second later is 1080891517.

Format C just the opposite. It is friendly to people and opaque to the computer:

```
Fri Apr  2 02:38:36 2004
```

It is difficult to compare and calculate with format C dates. Calculating one second after `Fri Apr 2 02:38:36 2004` is a minor nuisance; calculating one second after `Fri Apr 2 23:59:59 2004` is a bigger nuisance; calculating one second after `Sun Apr 4 01:59:59 2004` is a tremendous nuisance. [1] In fact, about the only thing that you can do really easily with a format C date is print it out!

Format B is in between these two, and, perhaps surprisingly, is bad for both display *and* calculation:

```
(36, 38, 2, 2, 4, 2004)
```

But we need format B because it's a natural stepping stone on the way from A to C or back.

There's only one other important thing to know about date formatting: Going "downstream", from A to B to C, is easy. Going "upstream", from C to B to A, is much more difficult, and from C to B is most difficult of all.

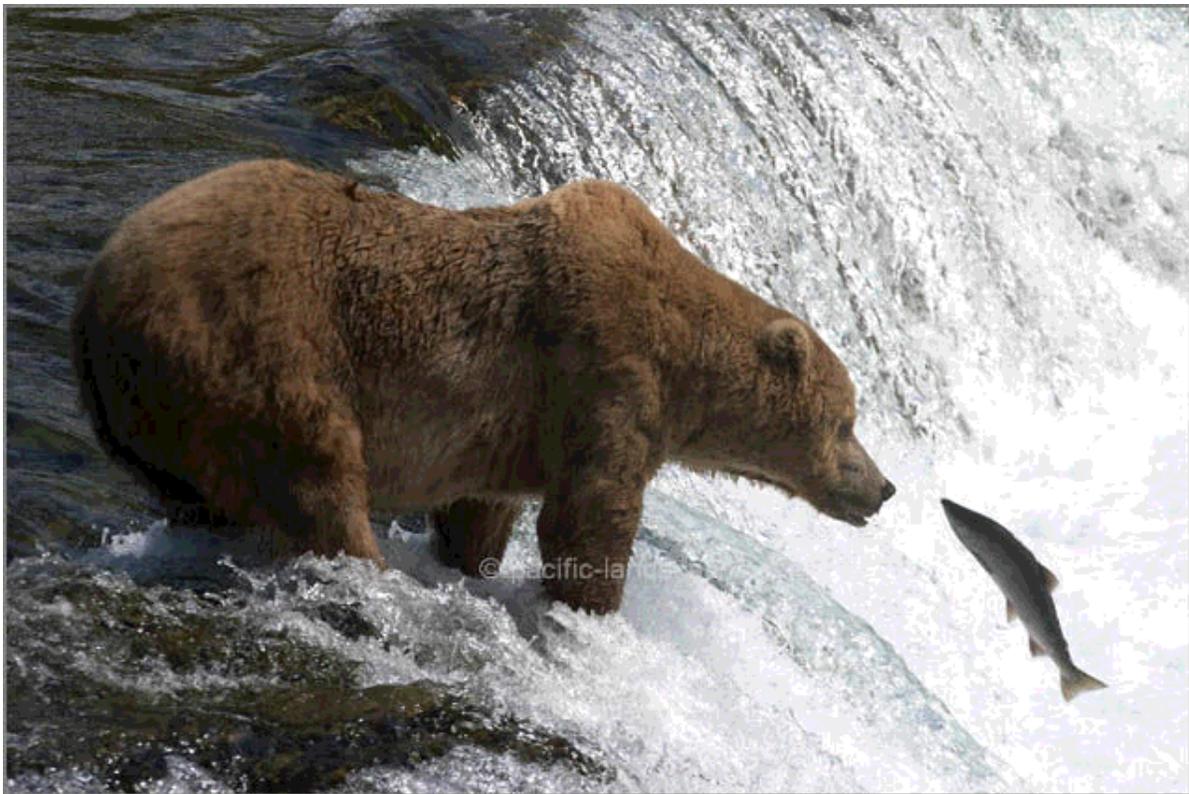


figure 10.1: upstream

If you believe this analysis, the best strategy should be obvious: store dates internally in format A whenever possible, and keep them in format A for as long as possible. Convert to format B and C only when required to produce output.

Sidebar: Date Formatting Functions in Perl

```
A      1080891516
B      (36, 38, 2, 2, 4, 2004)
C      Fri Apr  2 02:38:36 2004
```

- Use format A whenever possible
- To get format A, use Perl's `time()` or `stat()` functions
- To convert A to format B, use `localtime()` and `gmtime()`
- To convert B to format C, use `POSIX::strftime()`
- Try not to go "upstream"
- If you must, use modules like `Date::Parse` and `Date::Calc`
- They are like fish ladders: they permit upstream movement, but that doesn't make it easy

With that analysis in hand, we can look back at `GetDateNow()` and see where the author went wrong:

```
98          my $datev = `date '+%a %b %e %m %Y'`;
```

By calling the Unix `date` program, the author has obtained a human-readable, format-C date, putting the canoe into the water below the falls. Recovering from this mistake requires eight lines of fairly complex code, lines 102--116, and the complex `Date::Calc` module. The program has to work hard to get the components of format B, and then `Date::Calc` returns them in the wrong format anyway, and the program needs to use more code to shorten them.

Using format A makes this all much easier. We can use `time()` to get the current time, and then a simple subtraction to go back in time by the correct number of days: [2]

```
my $date = time() - $value * 86400;          # Format A
```

`time()` returns a format-A date, which is easy to calculate; once we have the date we want, we can use `localtime()` to go downstream to format B, getting the day, day-of-week, and month components:

```
my ($yday, $daynum) = (localtime($date))[6,3];      # A -> B
```

Sidebar: `POSIX::strftime`

Since 5.000, Perl has come with a module called `POSIX` which provides functions that are standard on POSIX-compliant systems but that are not built into Perl. For example, Perl includes `sin()` and `cos()` but not `tan()`, so `POSIX` provides `POSIX::tan()`. Probably the most useful function provided by this module is `strftime()`.

`strftime()` (the name means "format the time as a string; the `f` is for "format") takes a format string, analogous to the format string taken by `printf()`, and the date output from `localtime()` or `gmtime()`, and formats the date as specified by the format string. For example:

```
strftime("%A, %e %B %Y", localtime(1080891516))
```

yields "Friday, 2 April 2004", and

```
strftime("%H:%M:%S %d/%m/%y", localtime(1080891516))
```

yields "02:38:36 02/04/04".

Finally, `POSIX::strftime()` converts 5 to "Fri" and 9 to "Oct":

```
my $day = strftime("%a", localtime($date));      # B -> C
my $mon = strftime("%b", localtime($date));      # B -> C
```

```
$pattern = "$day\\s+$mon\\s+$daynum" ; # B -> C
```

I wanted to trim this still more, by letting `strftime()` put the pieces together for us; that's what it's for, after all:

```
my $date = time() - $value * 86400;
my $pattern = strftime "%a\\s+%b\\s+%e", localtime($date);
```

Ten lines would become two. But it's likely that this has introduced a bug. When the day of the month is between 1 and 9, the new code generates " 6" but the old code generated just "6". (It's also possible that I've fixed a bug here, not introduced one, but we should give the benefit of the doubt to the original programmer, who has actually seen the input data, and who sometimes writes `%e`, which generates " 6", and sometimes writes `%d`, which generates "06". `strftime()`, unfortunately, has no option to produce a one-or-two-digit day of the month. One solution is to do something like this:

```
my $date = time() - $value * 86400;
$daynum = (localtime($date))[3];
my $pattern = strftime "%a\\s+%b\\s+$daynum", localtime($date);
```

An alternative is to introduce a `mystrftime()` function that is just like `strftime()` except that it also accepts a `%D` escape code that does what we want. This is easier than it sounds:

```
sub mystrftime {
    my ($fmt, @time) = @_ ;
    my $day = $time[3];
    $fmt =~ s/%D/$time[3]/g;
    strftime($fmt, @time);
}
```

Then the code in `GetDateNow()` becomes:

```
my $date = time() - $value * 86400;
my $pattern = strftime "%a\\s+%b\\s+%D", localtime($date);
```

This technique is also useful for extending functions like `pack()` and `printf()`.

Sidebar: Interpolation

Often, when I present the advice that **the concatenation operator** is a red flag, people are concerned about code like this:

```
$p_wt = "The pumpkin weighs " . $pumpkin->weight . " pounds.\n";
```

"How do I avoid the dot operator here?" they ask. Well, you don't. Red flags are not prohibitions; they're just advice that you should look to see if there's might be an easy way to get a quick improvement. In this case, we look, and there isn't any easy way, so we leave the dot operators alone.

The other case that seems to worry people is this one:

```
$err_msg = $n_errors . "s have occurred";
```

Here we can't get rid of the dot operator in the obvious way:

```
$err_msg = "$n_errorss have occurred";
```

Because this tries to interpolate `$n_errorss`, rather than interpolating `$n_errors` followed by a literal `s`. Perl has a special syntax for avoiding this problem:

```
$err_msg = "${n_errors}s have occurred";
```

But this is unquestionably worse than the original code we were trying to fix. Getting rid of the simple and well-known dot operator, at the cost of introducing the obscure and little-known curly brace syntax, is not a win.

I learn something amazing from every program I read. Reading this program taught me a new and better solution to a common case of this interpolation problem. The author of this program wants to generate a string like this:

```
my $tmpfilealrt = "alert_{$daynum} . "_{$day} . "_{$mon}.log" ;
```

We can't replace the dot operators with interpolation:

```
my $tmpfilealrt = "alert_{$daynum_{$day_{$mon}.log" ;
```

This would interpolate `$daynum_` and `$day_` instead of the variables we want. We could use the curly brace notation, but as usual, that's not an improvement:

```
my $tmpfilealrt = "alert_{$daynum}_{$day}_{$mon}.log" ;
```

But the author of this program has found a solution that I think *is* an improvement, one that I've never seen anywhere else:

```
133 my $tmpfilealrt = "alert_{$daynum}\_{$day}\_{$mon}.log" ;
```

It does what's needed, uses less punctuation, and should be immediately clear to almost everyone what is happening.

When I saw this, my mouth hung open and I wondered why I hadn't been doing that for years.

We should also make the corresponding change in the `else` block:

```
123     } else {  
125         my $dfmt = `date +%a %b %e` ;  
127         ( $day, $mon, $daynum ) = split /\s+/, $dfmt ;  
129         $pattern = "$day\s+$mon\s+$daynum" ;
```

This would become:

```
    } else {  
        $daynum = (localtime())[3];  
        my $pattern = strftime "%a\s+%b\s+$daynum", localtime();
```

We will need to make some corresponding changes farther down:

```
133     my $tmpfilealrt = "alert_${daynum}_${day}_${mon}.log" ;
```

This becomes:

```
     my $tmpfilealrt = strftime("alert_${daynum}_%a_%m.log", localtime($date
```

The following line is never used, so we can eliminate it:

```
136     my $dpattern = "$day\\s+$mon\\s+$daynum" ;
```

The bottom part of the function still has:

```
my $errdate = `date +%d%m%Y` ;
chomp ($errdate) ;

my $errorfile = "day_alrt_err_" . "$errdate.log" ;
my $reportfile = "day_alrt_rep" . "$errdate.log" ;
```

The code in the top half of the function had logic that would look for dates in the past if a suitable value was supplied, but that logic is missing here, so we have a maintenance programmer's quandary: is this omission intentional? Is it a bug or a feature? I *think* it is a bug. If so, we should use:

```
my $errdate = strftime("%d%m%Y", localtime($date));
```

And if not:

```
my $errdate = strftime("%d%m%Y", localtime());
```

Note that the %d format is correct here.

This test is probably wrong:

```
if ( $opt =~ /\d/ ) {
    $value = $opt ;
    ...
}

my $date = time() - $value * 86400;
```

The intention seems to be to use the value of `$opt` directly as a date offset, if it looks like a numeral. But the test that's used, `$opt =~ /\d+/` is wrong for that; it will succeed on `Yobgogle2theMaxDude!!!`, for example. This is probably a bug, although nobody will notice it unless the user enters something weird. Still, we should repair the test:

```
if ( $opt =~ /^\d+$/ ) {
```

Sidebar: Refactoring

Why is *refactoring* called that? Consider this code:

```
if (condition) {
  A;
  B;
} else {
  A;
  C;
}
```

We can rewrite this in an abbreviated form that eliminates the duplicated A:

```
A;
if (condition) {
  B;
} else {
  C;
}
```

This transformation is analogous to the rewriting of the algebraic expression $AB + AC$ as $A(B+C)$; here we have "factored out" the repeated factor A. Programming constructions obey algebraic laws just as algebraic expressions do, with sequencing taking the role of multiplication, branching the role of addition, and looping the role of exponentiation.

The code now looks like this:

```
if ( $opt ) { print "The option is defined\n" } ;
if ( $opt ) {
  if ( $opt =~ /^\\d+$/ ) {
    $value = $opt ;
  } else {
    print "How many days prior do you want to search upon before the current
    chomp ( $value = <> );
  }
  my $date = time() - $value * 86400;
  $daynum = (localtime($date))[3];
  $pattern = strftime "%a\\s+%b\\s+$daynum", localtime($date);
} else {
  $daynum = (localtime())[3];
  $pattern = strftime "%a\\s+%b\\s+$daynum", localtime();
}

my $tmpfilealrt = strftime "alert_$daynum\\_%a_%b.log", localtime();
my $errdate = strftime "%d%m%Y", localtime();
my $errorfile = "day_alrt_err_$errdate.log";
my $reportfile = "day_alrt_rep_$errdate.log";

return $pattern, $tmpfilealrt, $errorfile, $reportfile ;
```

Both the `if` and `else` branches are doing the same thing, so we can factor out the common part. The only difference between them is the way that the date is selected. If `$opt`, the number of days in the past, is false or omitted, it is effectively taken to be zero. But we can achieve the same thing just by defaulting it to zero:

```
my $opt = shift || 0;
```

The `if-else` test goes away:

```
if ( $opt =~ /^\\d+$/ ) {
  $value = $opt ;
}
```

```

}else {
    print "How many days prior do you want to search upon before the current d
    chomp ( $value = <>);
}

my $date = time() - $value * 86400;
$daynum = (localtime($date))[3];
$pattern = strftime "%a\\s+%b\\s+$daynum", localtime($date);

```

Three more lines of code are gone.

The variable \$opt is used only to set \$value, and never again, so we should see if we can merge them. Then we would get:

```

my $opt = shift || 0 ;

unless ( $opt =~ /^d+$/ ) {
    print "How many days prior do you want to search upon before the current
    chomp ( $opt = <>);
}

my $date = time() - $opt * 86400;
my $daynum = (localtime($date))[3];
my $pattern = strftime "%a\\s+%b\\s+$daynum", localtime($date);

my $tmpfilealrt = strftime "alert_$daynum\_a_b.log", localtime();
my $errdate = strftime "%d%m%Y", localtime();
my $errorfile = "day_alrt_err_$errdate.log";
my $reportfile = "day_alrt_rep_$errdate.log";

return $pattern, $tmpfilealrt, $errorfile, $reportfile ;

```

It's a bit puzzling that the \$tmpfilealrt, \$errorfile, and \$reportfile variables disregard the value of \$opt and always use the current date instead of the past date. If that's a bug, it's now much easier to notice and to fix. If it is a bug, we might want to see what happens when we eliminate the repeated call to localtime(\$date):

```

my $opt = shift || 0 ;

unless ( $opt =~ /^d+$/ ) {
    print "How many days prior do you want to search upon before the current
    chomp ( $opt = <>);
}

my @date = localtime(time() - $opt * 86400);
my $pattern = strftime "%a\\s+%b\\s+$date[3]", @date;

my $tmpfilealrt = strftime "alert_$daynum\_a_b.log", @date;
my $errdate = strftime "%d%m%Y", @date;
my $errorfile = "day_alrt_err_$errdate.log";
my $reportfile = "day_alrt_rep_$errdate.log";

return $pattern, $tmpfilealrt, $errorfile, $reportfile ;

```

The code here has been cut from 31 to 13 lines, not counting the diagnostics.

Main flow

I started in on the details of the program, not now let's back up for a higher-level view of what happens.

`GetDateNow()`, which we just saw, generates some patterns and filenames based on the date of interest. These are passed to `ProcessAlertLog()`, which calls `AlrtError()` and `StartStopRep()` to generate some reports. `StartStopRep()` calls `DateDiffRep()` to generate part of its report. Since `ProcessAlertLog()` is next in the calling sequence, that's the next logical part of the program to look at.

ProcessAlertLog()

First there's some trivial stuff up front:

```
151 sub ProcessAlertLog {
152
153     my $falrt = shift ;
154     my $tfalrt = shift ;
155     my $dpat = shift ;
156     my $perrorfile = shift ;
157     my $prepfile = shift ;
158
```

We may as well write this as:

```
sub ProcessAlertLog {
    my ($falrt, $tfalrt, $dpat, $perrorfile, $prepfile) = @_;
```

The next thing the function does is open an input and an output file:

```
163     open ( ALERTLOG, $falrt ) or die "Can't open $falrt due to $!" ;
164
165     open ( TMPALRTLOG, "> $tfalrt" ) or die "Can't open $tmpfilealrt due t
```

Close inspection reveals a violation of G ???:

```
open ( ALERTLOG, $falrt ) or die "Can't open $falrt due to $!" ;
open ( TMPALRTLOG, "> $tfalrt" ) or die "Can't open $tfalrt due to $!" ;
```

The function's main logic is:

```
159 my $LINE ;
166
167 while ( <ALERTLOG> ) {
168     if ( $_ =~ /$dpat/ ){ $LINE = 'yes' } ;
170
171     if ( $LINE && $LINE eq 'yes' ){ print TMPALRTLOG "$_" } ;
172
173 }
174
175 print TMPALRTLOG "Error: ... \n" if ( $LINE && $LINE eq [C['no']C] );
```

Since the only values `$LINE` ever takes on are undefined and `yes`, the condition on line 175 can never be true, and the test for `yes` on line 171 is redundant. This is a good example of the confusion that can

ensue when you don't G ????. The fixed code is simpler and easier to read:

```
my $seen_dpat;

while (<ALERTLOG>) {
    $seen_dpat = 1 if /$dpat/o;

    if ($seen_dpat) { print TMPALRTLOG $_ }
}

print TMPALRTLOG "Error: ... \n" unless $seen_dpat;
```

I've also added the only-once flag to the pattern match, since the pattern won't change over the lifetime of the program.

But there is an alternative way to write this function. A little more thought will reveal that `$seen_dpat` starts off false, and once it is set true, it never changes back again. If this happens, the `/$dpat/` test becomes superfluous since the action it's guarding is redundant. What's *really* going on here is that we're trying to copy all the records from `/$dpat/` onward, so we might try this alternative:

```
while (<ALERTLOG>) { print(TMPALRTLOG), last if /$dpat/o }
print TMPALRTLOG while <ALERTLOG>;
```

The first loop scans the records up to `/$dpat/`; the second one copies the subsequent records. Often this kind of formulation is an improvement on the version with the flag variable. In this case, it's not a clear win, because we've lost the error message if `/$dpat/` doesn't occur at all, and we need to add back the flag variable to rescue it:

```
my $ok;
while (<ALERTLOG>) { $ok = 1, print(TMPALRTLOG), last if /$dpat/o }
print TMPALRTLOG while <ALERTLOG>;
print TMPALRTLOG "Error: Pattern /$dpat/ was not found \n" unless $ok;
```

If we're into obscure features, we could use Perl's little-known "flip-flop" operator:

```
while (<ALERTLOG>) {
    print if /$dpat/o .. eof();
}
```

It's a pity that this operator is so little-known, since it seems to be just what is needed here, and it's not hard to figure out what it means even if it's unfamiliar. We can even get the error message, although the behavior here is even more obscure:

```
while (<ALERTLOG>) {
    print TMPALRTLOG if $seen = /$dpat/o .. eof();
}
print TMPALRTLOG "Error: Pattern /$dpat/ was not found \n" unless $seen;
```

Here we're testing the value of the flip-flop at the last record. If it's still flipped, the last record was printed, and there's no need for the error message.

Related to all this is another error message at the end of the function:

```
183     print "Error: Pattern was not found \n" if (!$LINE eq 'no' );
```

This isn't a red flag; it's just a bug. Or rather, two bugs. First, the test is backwards: it should be `if $LINE eq 'no'`. And second, the precedence is off. The `!` applies here to `$LINE`, not to the result of `eq`. In Chapter ???, we saw how to use the `B::Deparse` module to investigate this sort of thing.

The next line is a real head-scratcher:

```
181 if ( -r $tfalrt ) { &AlrtError ( $tfalrt, $perrorfile ) } ;
```

The test here succeeds if the file `$tfalrt` is readable. But the program has just created that file itself:

```
165 open ( TMPALRTLOG, "> $tfalrt" ) or die ...
```

So what the function is really testing is whether the process' `umask` is set correctly, which surely isn't what was intended.

But even if there is some weird logic behind the test, what will the program do if the test fails? It will skip the call to `AlrtError()`. But that's the function that prints the report, and printing the report is the whole point of the program! So we might guess that the test has never been false, and we should just generate the report:

```
AlrtError( $tfalrt, $perrorfile );
```

AlrtError()

`AlrtError()` contains a puzzle too, involving the use of the `$errorfile` argument:

```
192 my $errorfile = shift ;
196 open ( ERRORALRTLOG, "> $errorfile" ) or die "Can't open $errorfile
205 # print ERRORALRTLOG "$_" } ;
220 close(ERRORALRTLOG) or die "Cold not close tmpalert due to $!\n" ;
```

Nothing is ever written to the file! I have no problem with that, since it seems to have served a diagnostic purpose in the past. But if you're going to comment out the `print`, I think you should comment out the `open` and the `close` too. But I also think that the *right* solution here is just to delete all of it.

Programmers sometimes leave huge messes of commented-out code in their programs, on the theory that they might need it again someday. The right place to keep this kind of lumber is in your source code management system. If you aren't using a source code management system, then the lumber is itself an argument in favor of using one.

The main logic of the report-generating function is:

```
199 while ( <ORAALRTLOG> ) {
201     if ( $_ =~ /(ORA-\d+)\s+/ ) {
203         $errors{$1}++ ;
207     }
209 }
210
211 for my $key ( keys %errors ) {
213     my $value = $errors{$key} ;
215     print "The error code $key appears $value times in this section of
```

```
217     }
```

This is quite simple. We could reduce the line count by tinkering with the code; for example, lines 199--203 might be replaced by

```
/(ORA-\d+)\s+/ && $errors{$1}++ while <ORAALRTLOG>;
```

but this does not seem to me to be any improvement. It's only one line of code, but it's one line that is just as complicated as the three lines it was replacing.

I have only two suggestions. One is to eliminate `$value`, which is an example of (some section or chapter):

```
215     print "The error code $key appears $errors{$key} times in this sect
```

The other is to sort the keys so that the error report appears in alphabetical order. The cost is minimal, and the benefit to the reader might be substantial.

StartStopRep()

Funny thing about this function. It's a significant part of the program, about 37 lines long. But the only call to it (in `ProcessAlrtLog()`) is commented out.

I wonder if it was commented out because it didn't work? Maybe we can fix it. As usual, in the absence of any better plan, we'll start at the top:

```
225 sub StartStopRep {
226
227     my $tmpfilealrt = shift ;
228     my $reportfile = shift ;
229     my @lines ;
230     my $stpline ;
231     my $strtline ;
232     my $shdwnlinenum ;
233     my $ndl = 4 ;
234 #   my %match ;
235     my @matimes ;
236     my $counter = 0 ;
237
```

This becomes:

```
sub StartStopRep {
    my ($tmpfilealrt, $reportfile) = @_ ;
    my (@lines, $stpline, $strtline, $shdwnlinenum, @matimes);
    my $ndl = 4;
    my $counter = 0;
```

Even though I don't like talking about variable names, I have to say something about this. The named `$stpline` and `$strtline` really bother me.

I wonder why the author has such a problem with vowels. Why `AlrtError()` and not `AlertError()`? If it needed to be abbreviated---and I'm not saying it did---what was wrong with `AlertErr()`?

Here I would have preferred `$startline` and `$stopline`. If they needed to be abbreviated---and I don't see why they do---why not use `$startln` and `$stopln`---or `$start` and `$stop`? What is really gained by omitting random vowels?

I'd complain about `$shtdwnlinenum` too:

```
251          $shtdwnlinenum = $. ;
```

But this variable is not used anywhere, so we can abbreviate it to nothing.

The main logic goes something like this:

```
245     while ( <ORAALRTLOG> ) {
246
247     my %match ;
    ...
274     $match{START} = $strtline ;
275     $match{STOP} = $stpline ;
276
277         print REPORTLOG "$. => The start time is $strtline and the stop t
278
279         $matimes[$counter] = \%match ;
286     } # End While
```

Once again, we have (some section or chapter). Getting right of `%match` leaves us with:

```
     while ( <ORAALRTLOG> ) {
    ...
        print REPORTLOG "$. => The start time is $strtline and the stop
            $matimes[$counter] = { START => $strtline, STOP => $stpline };
    }
```

And we have removed three lines of code and one variable.

We saw **array length variables** before. Here, the only purpose of `$counter` is to track the length of `@matimes`:

```
236     my $counter = 0 ;
    ...
279     $matimes[$counter] = { START => $strtline, STOP => $stpline }; ;
280     #-D     print "The $counter start is $matimes[$counter]{START}\n";
281     $counter = $counter + 1 ;
```

But arrays in Perl track their own lengths, so as usual we can just use `push()`:

```
push @matimes, { START => $strtline, STOP => $stpline };
#-D print "The " . @matimes . " start is $matimes[-1]{START}\n";
```

Notice that I used **the concatenation operator** in the debugging line here. I can't say this enough: red flags are not prohibitions. They are only signs that you should pause and consider whether there might be a better way to write the code. Here I was not able to think of a better way.

[1] For example, in most of the U.S., the correct answer is `Sun Apr 4 0B<3>:00:00 2004`, but not in Arizona or eastern Indiana...

[2] Purists will observe that this is not always exactly correct. If `$value` happens to occur between midnight and 1 AM on first full day of daylight saving time, then the calculated date will be two calendar days earlier, rather than one; a complementary error can occur during a one-hour window in the autumn. This can be corrected with some effort, and it's still simpler than the original code; or it can be ignored, since it probably doesn't really matter.