Domain Zone Checking

The subject of this chapter is a program called `checkzonesloaded`. it was provided to me once when I was teaching the class for a large company in Europe; it had been written by one of their employees. The author was not the person who sent it to me, and was not an attendee of the class; it was contributed by one of their co-workers. The co-worker was careful not to identify the program's author, but the program contained a dead giveaway: it cannot possibly work unless it is run by B. Wilkinson. From this, we might infer that it was written by B. Wilkinson.

The purpose of the program is to take a file for a list of a domain names and to check on a locl name serve to make sure the records for each zone are available.

Sometimes programs contributed by class attendees are turgid and unreadable, but often they're quite clear. This one was quite clear. Nevertheless, I was able to cut it down by about half. The complete code is Program ???.

Date Formatting

The first thing that jumped out at me was the `formatDate()` function. It returns a string like `17/03/2003` or like `Mon 17 Mar 2003 02:38:36` depending on whether its argument is the string `short` or `long`. Here's the code:

```
        sub formatDate()
        {
```

```
        my ($format) = @_;

        my $curdate;

        # get date elements
        my @time  = localtime;
        my $secs  = $time[0];
        my $mins  = $time[1];
        my $hour  = $time[2];
        my $dom   = $time[3];
        my $mon   = $time[4] + 1;
        my $month = ("Jan","Feb","Mar","Apr","May","Jun",
                     "Jul","Aug","Sep","Oct","Nov","Dec")[$time[4]];
        my $year  = 1900 + $time[5];
        my $dow   = ("Sun","Mon","Tue","Wed","Thu","Fri","Sat")[$time[6]];

        # now format them as required
        if ($format eq "short"){
            $curdate = sprintf("%02d/%02d/%s", $dom, $mon, $year);
        }
        elsif ($format eq "long") {
            $curdate = sprintf("%s %2d %s %d %02d:%02d:%02d",
                                $dow,$dom,$month,$year,$hour,$mins,$secs);
        }
        else {
            $curdate = "";
        }

        return $curdate;
    }
```

Almost everyone has had the misfortune to write a function of this type. In classes, I always ask the audience how many of them have written a date-formatting function like this one and I always get lots of hands. But there is a better way.

Standard with Perl, since version 5.000, is a big module called `POSIX`. The `POSIX` module contains a lot of standard utility functions that are mandated by the `POSIX` operating system standard but that are not available in Perl's core. For example, Perl has a sine function (`sin()`) and a cosine function (`cos()`) but no tangent function. Where's the tangent function? It `POSIX::tan`. Perl has an `int()` function to discard the factional part of a number, but what if you want the number rounded up to the nearest integer? It's `POSIX::ceil`. (Short for "ceiling".)

Probably the most useful function in `POSIX` is `strftime()`, which is short for "format the time as a string". `strftime()` is analogous to `printf`: you give it a format specifier and a date, and it formats the date according to the specifieer. For example:

```
strftime("%h:%M:%s", localtime);
# returns "02:38:36"

strftime("%A, %d %b", localtime);
# returns "Monday, 17 Mar"
```

In `strftime()` codes, the program's short date format is `%d/%m/%Y` and its long date format is `%a %d %b %Y %T`. Here's a summary of the escape codes we've used:

```
%a      abbreviated name of the weekday
```

`%A`	`full name of the weekday`
`%b`	`abbreviated name of the month`
`%B`	`full name of the month`
`%d`	`day of the month (2-digit)`
`%e`	`like %d, but with leading space instead of leading 0`
`%h`	`hour of the day (2-digit)`
`%m`	`month number (2-digit)`
`%M`	`minute`
`%T`	`24-hour time of day (abbreviation for "%h:%M:%s")`
`%y`	`two-digit year`
`%Y`	`four-digit year`

There are many others; see the manual for full details.

If we use `strftime()` to handle the formatting chores, most of `formatDate()` can be eliminated:

```
sub formatDate()
{
    my ($format) = @_;
    my %formats = (short => "%d/%m/%Y",      # 17/03/2003
                   long => "%e %d %b %Y %T", # Mon 17 Mar 2003 02:38:36
                   );
    return strftime($formats{$format} || "", localtime);
}
```

18 lines have become 6. It has also become easy to extend `formatDate()` to handle other formats; just insert more entries into the `%formats` hash:

```
my %formats = (
  short   => "%d/%m/%Y",       # 17/03/2003
  natural => "%c",             # (locally preferred date format)
  long    => "%a %d %b %Y %T", # Mon 17 Mar 2003 02:38:36
  letter  => "%A, %e %B",      # Monday,  4 March
);
```

Note also how clear the explanatory comments are. Something like `Mon 17 Mar 2003 02:38:36` is much easier to read and understand than something like `dow dom mon year time`, because it's easier to see than to think. I think this also illustrates what I call the *first maxim of documentation*, which says that an ounce of examples is worth a pound of specifications.

The Main Loop

Now I'll tackle the main part of the program. The first red flag I saw was:

```
 69      if (exists $domains{$domain}){
 70     next;
 71      }
 72      else {
           ... the entire rest of the program ...
 98      }
```

This is **The Condition that Ate Michigan**. The `else` part of the `if` block has swallowed the *entire* program whole.

figure 3.1: `Michigan`

The easy rewrite in this case is to turn the `if` test into a statement modifier:

```
next if exists $domains{$domain};

... the rest of the program ...
```

If we do this, everything moves closer to the left-hand margin. When you read, your eye likes to move down smoothly down the left-hand margin. Things are harder to see when they are farther to the right, so you should try to **keep normal control flow close to the left-hand margin**. This minimizes eye movement and improves readability; it’s why newspaper copy, which has a lot of small type, is set in narrow columns.

Sidebar: The Condition that Ate Michigan

Here's another example of The Condition that Ate Michigan

```
Subject: Apply Regex In Place
Message-ID: <1YVv7.64617$0x.22955187@typhoon.southeast.rr.com>

sub each_file {
  if ( -T $_ ) {
    print "processing $File::Find::name\n";
    if ( open F, "< $_" ) {
      my $s = <F>;
      close F;
      if ( open F, "> $_" ) {
        $s =~ s/COLUMN1/COLUMN1/ig;
        print F $s;
        close F;
      }
      else {
        print "Error opening file for write: $!\n";
      }
    }
    else {
      print "Error opening file for read: $!\n";
    }
  }
}
```

Here the entire function has been gobbled up by the ravenous `if -T` condition. Readability suffers as a result. Quick! Does the `if -T` have an `else` clause?

Instead, we can write:

```
Subject: Apply Regex In Place
Message-ID: <1YVv7.64617$0x.22955187@typhoon.southeast.rr.com>

sub each_file {
  return unless -T $_;

  print "processing $File::Find::name\n";
  ...
}
```

and it becomes immediately apparent that the function simply ignores non-text files.

Sidebar: The Great Pyramid

figure 3.2: `pyramids`

The very day after I first recognized this red flag, I met the following example in `comp.lang.perl.misc`:

```
Subject: Quicker way pyramidsto loop through directories?
Message-ID: <9q1f65$ht6@newton.cc.rl.ac.uk>

if(opendir(DIR, "f:/list-logs"))
{
  while($dir = readdir(DIR))
```

```
          {
            if(-d "f:/list-logs/$dir" and $dir ne "." and $dir ne "..")
            {
              if(opendir(SUB, "$listlogs/$dir"))
              {
                while($file = readdir(SUB))
                {
                  if($file =~ /\.log.+/i)
                  {
                    $total += -s "$listlogs/$dir/$file";
                  }
                }
                closedir(SUB);
              }
            }
          }
          closedir(DIR);
        }
```

Wowzers! That’s a lot of context you have to remember by the time you get to the pharaoh’s burial chaber in the center. Perhaps this is better:

```
           opendir(DIR, "f:/list-logs" or die ...;
           while($dir = readdir(DIR))
           {
             next if $dir eq "." or  $dir eq "..";
             next unless -d "f:/list-logs/$dir";

             opendir(SUB, "$listlogs/$dir") or next;

             while($file = readdir(SUB))
             {
               next unless $file =~ /\.log.+/i;
               $total += -s "$listlogs/$dir/$file";
             }
             closedir(SUB);
           }
           closedir(DIR);
```

(If you disagree, that’s OK.)

Here’s another variation on that innermost loop:

```
             for (grep /\.log.+/i, readdir(SUB))
             {
               $total += -s "$listlogs/$dir/$_";
             }
```

Which of the three variations do you like best? Which do you like least? Remember to **try it both ways**.

Skipping Duplicate Items

The program now has:

```
        # ignore this one if it has already been processed
        next if exists $domains{$domain};

        $domains{$domain} = $domain;
```

This is a fairly ordinary use of a hash to recognize duplicate items, but it confused me. What is the hash value used for? I investigated, and found that it wasn’t used for anything. I would have been less confused by:

```
        $domains{$domain} = 1;
```

One very frequently-seen idiom for this kind of task is:

```
        next if $seen{$domain}++;
```

Also, domain names are case-insensitive, so we probably want:

```
        next if $seen{lc $domain}++;
```

Loop Hoisting

Next, I got to wondering about this:

```
         77     # set up resolver object
         78        my $res = new Net::DNS::Resolver;
         79     $res->nameservers($NAMESERVER);         # search 1 ns only
```

The main loop builds a new `Net::DNS::Resolver` object for each input line. Why do it repeatedly?

Code in a loop that doesn't depend in any way on the data being looped over is called *invariant*. Invariant code can often be *hoisted* out of the loop:

```
           # set up resolver object
           my $res = new Net::DNS::Resolver;
           $res->nameservers($NAMESERVER);              # search 1 ns only

        43 print "Start: ".formatDate("long")."\n";
           ...
```

Here we've removed the initialization of the `$res` object from the loop, butting it up toward the top of the program. This makes the loop easier to understand by making it smaller; it may also be fractionally more efficient.

File-Scope `my` Variables

Since I'm in the vicinity, let's look at lines 36--41:

```
         36  my %seen;
         37  my $domain;
         38  my $rr;
         39  my $query;
         40  my $records;
         41  my $null;
```

We've seen this before. Why are all these variables declared at file scope? To answer this question, I searched the program to find out where the variables were used. It turned out that `$rr`, `$records`, and `$null` were *not* used, so that's three more lines of code gone with no gross cost at all.

`$domain` is private to the main loop, so that is where it should be declared. We can eliminate the file-scope `my` declaration completely just by slapping a `my` on the first appearance of `$domain` inside the loop. Line 64 becomes:

```
        64       [C[my]C] $domain = $inprecord[0];
```

`$query` is the same:

```
        82      [C[my]C] $query = $res->query($domain, "NS");
```

Lines 37--41 have gone away completely.

Diagnostics

We’ve seen diagnostics like these before:

```
         43  print "Start: ".formatDate("long")."\n";

        105  print "End: ".formatDate("long")."\n";
```

They are being sent to `STDOUT`, but it’s almost always preferable to send diagnostics to `STDERR`. That way, if the program’s output is directed into a file, the diagnostics don’t vanish into the file; they stay on the terminal where someone can see them. If the program’s output is directed into a pipe to another program, the diagnostics don’t go into the pipe where they will confuse the other program or have to be filtered out; they stay on the terminal where someone can see them. This is why `STDERR` was invented. So I would prefer:

```
             [C[warn]C] "Start: ".formatDate("long")."\n";

             [C[warn]C] "End: ".formatDate("long")."\n";
```

Hardwired Source and Destination

```
         27  my $DOMLIST = "/export/home/bwilkins/dnsmu/scripts/testzones.2";
         28  my $OUTLIST = "/export/home/bwilkins/dnsmu/scripts/testzones.3";

         45  open( DOMFILE, "<$DOMLIST" ) || die "Can’t open the file $DOMLIST:$!\n"
         46  open( OUTFILE, ">$OUTLIST" ) || die "Can’t open the file $OUTLIST:$!\n"
         52  while (<DOMFILE>)

         97          print OUTFILE "$outrecord\n";
```

Since the program is hardwired to get its input from one particular file, and to write its output to one particular file, no matter what, it can be run only by B. Wilkinson, who happens to own those two files. Nobody else can use it. This is how we can infer that the author is B. Wilkinson.

However the source and destination need not be limitations of the program. Perl makes it easy to write programs with flexible input and output conventions.

First, change these:

```
         52  while (<DOMFILE>)

         97          print OUTFILE "$outrecord\n";
```

to these:

```
         while (<>)
```

```
                        print "$outrecord\n";
```

Now the program reads from standard input, or from the files named in `@ARGV`, and writes to standard output. Wilkinson can get the original behavior by executing the command:

```
checkzonesloaded ~/dnsmu/scripts/testzones.2 > ~/dnsmu/scripts/testzones.3
```

If the input or the output files move, the program will works, and other people can run the program in their own files.

If Wilkinson doesn't want to write that entire command line, he should consider something like this:

```
@ARGV = ($DOMLIST) unless @ARGV;
open STDOUT, ">$OUTLIST" if -t STDOUT;
```

This will default the input file list and output destination if they are unspecified. If Wilkinson is willing to type the long command line, we can get rid of the variables `$DOMLIST` and `$OUTLIST` entirely, along with a pile of `open` and `close` machinery.

Warnings

Many people will insist vociferously and dogmatically that you must always use Perl's `-w` flag under all circumstances. I will not so insist. It is a tool, and, like other tools, you should try it out and decided for yourself when its use is appropriate. But it's almost always helpful to run this command:

```
perl -wc yourprogram.pl
```

This performs a basic syntax check and issues static (compile-time) warnings without actually running the program. If there are any warnings, the next step is to try to understand what they mean, and whether they indicate that anything is wrong with the program.

In this case, we get several warnings. The first one is:

```
defined(@array) is deprecated at
  /usr/local/lib/perl5/site_perl/5.6.1/Net/DNS.pm line 137.
        (Maybe you should just omit the defined()?)
```

This is not our problem; it is an error in the `Net::DNS` module.

The next warnings are:

```
main::formatDate() called too early to check prototype
   at checkzonesloaded.pl line 43.
main::formatDate() called too early to check prototype
   at checkzonesloaded.pl line 49.
main::formatDate() called too early to check prototype
   at checkzonesloaded.pl line 105.
```

"Prototype?" What prototype?

It turns out that Perl is complaining about the parentheses in:

```
118  sub formatDate[C[()]C]
119  {
   ...
149  }
```

The Perl syntax for defining functions is:

```
sub formatDate
{
}
```

The extra `()` is interpreted as a *prototype*, describing the format of the intended arguments. In this case, the empty parentheses mean that there will be no arguments. Had this been enforced, the function would have been broken, because we *do* want to call `formatDate()` with no arguments. Fortunately, the prototype was inoperative in this case, because Perl had compiled the calls to `formatDate` before it saw the prototype. By the time it found out that `formatDate` was supposed to be called with no arguments, it was too late to enforce that; all it could do was warn us that it was too late.

Had the prototype been operative, say because the function was defined before the calls, instead of after, we would have gotten a fatal error:

```
Too many arguments for main::formatDate
  at checkzonesloaded line 42, near ""long")"
Too many arguments for main::formatDate
  at checkzonesloaded line 45, near ""short")"
Too many arguments for main::formatDate
  at checkzonesloaded line 77, near ""long")"
Execution of checkzonesloaded aborted due to compilation errors.
```

Perl has too much punctuation, and it’s easy to slip up and put in some extra that isn’t needed, especially if you are used to programming in C, which has a slightly different syntax. I missed this error too, until `-w` pointed it out to me. `-w` is good for catching this sort of thing.

There’s one last warning we get from the `perl -wc` check:

```
Name "main::DOMLIST" used only once: possible typo
  at checkzonesloaded line 103.
```

Perl warns any time you use a variable only once, because such usages are rarely correct. You usually need to access a variable at least twice: once to store a value in it, and once to read the value back again. Or, if the variable is a filehandle, as it is here, you need to mention it once to open it, and then at least once more to read or write it. Here’s the line 103 that Perl is warnings us about:

```
103  close DOMLIST;
```

Here’s where we opened that handle:

```
 45  open( DOM[C[FILE]C], "<$DOMLIST" ) || die "Can’t open the file $DOMLIST
```

Oops!

Anyway, if we go with my idea for more flexible input and output of the previous section, we can get rid of the `open`s and `close`s entirely.

chop

```
chop $_;
```

The `chop` function is almost entirely obsolete. It was originally intended to remove a trailing record terminator sequence from an input line. It was invented on Unix systems, where the terminator sequence was a single `"\n"` character, so `chop` simply removes the last character from its argument. But as Perl spread to other systems, people realized that this wasn’t always the right behavior. On some systems, the terminator sequence isn’t a single character; typically, it’s `"\r\n"`.

Even under Unix, the behavior of `chop` isn’t exactly right. Sometimes a text file doesn’t end with a `"\n"` character; then calling `chop` on the last record discards a legitimate data character and garbles the data.

In Perl 5, the `chomp` function was introduced to repair these defects. It always removes the appropriate record terminator sequence, (as defined by `$/`), but only if there was one to remove. `chop` should almost always be replaced with `chomp`:

```
  chomp $_
```

Variable Use Immediately Follows Assignment

Well, actually not. I thought that was what this was:

```
my @inprecord = split(",", $_);

$domain = $inprecord[0];
```

And I was planning to replace this with:

```
my ($domain) = split(",", $_);
```

but I was mistaken, because `@inprecord` is used later on:

```
    if ($query)
    {
   push (@inprecord, $query->header->aa);
    }
    else
    {
           push(@inprecord, "-");
    }

    my $outrecord .= join (",",@inprecord);
```

Oh well.

Taking Two Steps Forward and One Step Back

But then I got to wondering about the `split` and `join`:

```
my @inprecord = split(",", $_);
```

```
        my $outrecord .= join (",",@inprecord);
```

Why split the record apart if we know that we're just going to have to put it back together again?

Often when you ask that question, the answer is "no reason at all", and you get an easy opportunity to simplify the code; we'll see some later on. But on this case the question has two reasonable answers.

First, we need to split up the record because `$domain` is its first field, and we need to do something with the domain. And second, we need to do the `join` to attach a new field to the end of the record:

```
       push (@inprecord, $query->header->aa);
```

But let's see what happens if we try to avoid the `split-join` anyway; maybe it will turn out to be simpler than what we have. (**Try it both ways**.) We'll deal with the second issue first.

Adding Another Field at the End

The code that deals with this goes like this:

```
        my @inprecord = split(",", $_);
...
        if ($query)
        {
       push (@inprecord, $query->header->aa);
        }
        else
        {
               push(@inprecord, "-");
        }

        my $outrecord .= join (",",@inprecord);

        # print results
        print OUTFILE "$outrecord\n";
```

All we're doing is adding some extra data to the end of each record, so we could replace the code with this:

```
if ($query)
{
  print $_, ",", $query->header->aa, "\n";
}
else
{
  print $_, ",", "-",                "\n";
}
```

Or, in the interests of trying to **avoid excess punctuation**, perhaps this version:

```
if ($query)
{
  print "$_,", $query->header->aa, "\n";
}
else
{
  print "$_,-\n";
}
```

Which do you prefer?

Sidebar: Compression

Factoring out the common code from this pair of blocks gives us something like this:

```
print $_, ",";
if ($query)
{
  print $query->header->aa;
}
else
{
  print "-";
}
print "\n";
```

Or we might even replace the `if`--`else` with `?:` and write:

```
print $_, ",", $query ? $query->header->aa : "-", "\n";
```

Which variation do you think is best?

More Splitting

There were two reasons that we had to split the input records; we dealt with the second which was that we needed to add new data to the end of each record. #1 was that the program needs to do something with the domain field, which is the first field of the input record:

```
        $domain = $inprecord[0];
```

Here we can’t avoid the `split` (at least, not without replacing it with something worse, like a regex match), but we *can* avoid splitting the entire record:

```
my ($domain) = split(",", $_);
```

Perl optimizes this so that instead of splitting it up on every comma it only splits the record on the first comma, which is all that is needed.

Now we can eliminate the `@inprecord` variable entirely, getting rid of three more lines of code.

Sidebar: Taking Two Steps Forward and One Step Back

Here's a smaller example of `taking two steps forward and one step back|R`:

```
        54      chop $_;
        55
        56      # write comments lines direct to output file
        57      if (/^#/){
        58     print OUTFILE "$_\n";
        59     next;
        60      }
```

Here we remove the newline from `$_` on line 54 and put it back at line 58. Would it be better to this:

```
                # write comments lines direct to output file
                if (/^#/){
               print;
               next;
                }

                chop $_;
```

I'm honestly not sure. What do you think?

A Note About `split`

I get nervous whenever I see this:

```
        split(",", $_)
```

The reason is that I've seen too many people ask why this doesn't work:

```
        split("|", $_)
```

For example:

```
Date: Mon, 9 Aug 1999 23:33:15 -0400
Subject: Help - Split Function Blowing My Mind Away!!
Message-Id: <a1Nr3.57892$jl.36696862@newscontent-01.sprint.ca>

Please consider the following function:

sub writeEntry
{
   @record = split("|",$_[0]);
   foreach $field (@record)
   {
      print "$field<br>\n";
   }
}

During debugging I have printed the value of $_[0] and it's as
expected.  The value is:

Derek Battams|Mail Address Here|URL Here|Comment Here

When I do the print in the foreach loop I get each character
as an element of the array @record.  Why isn't the string
being split with the pipe as the delimiter?
```

Or for example:

```
        Date: Wed, 06 Dec 2000 02:35:59 GMT
        Subject: Problem with split using |
        Message-Id: <slrn92r9a8.15c.tom.hoffmann@localhost.localdomain>

        When I split the following string on the "|" character, it
        splits into 35 array elements. When I change the "|" to a
        "," and split on the comma, it splits into the 7 fields I
        expect, but I can not figure out why splitting on "|" does
        not give the same result.

        The pertinent code is:

        $unidata = '|0000 0000|CI78-00727|19780126||DV||';
        @unidata = split ("|", $unidata);
```

Or for example:

```
        Date: Wed, 12 Jun 2002 11:19:32 -0400
        Subject: How to split with "|"?
        Message-Id: <3D076684.6060900@mail.med.upenn.edu?

        How to split with "|"?  When I used "@list = split("|",
        $string)", @list turned out to be a single char list.
```

So what’s going on in cases like this one?

```
      Subject: Help - Split Function Blowing My Mind Away!!

      @record = split("|",$_[0]);
```

The quote marks have confused the author. The quote marks make it look as though the first argument to `split` is a string, but it isn’t; it’s a regex. Even if you write it with quote marks, it’s still a regex. And `|` is special in regexes. The `split` call above is treated as if it had been:

```
      @record = split(/|/, $_[0]);
```

and now the problem is apparent; the pattern being split on is empty-string-or-empty-string. Another common problem of this type occurs with `split(".", $var)`, where the `.` is a regex metacharacter that matches any single character.

Returning to `checkzonesloaded`, we have:

```
            split(",", $_)
```

Here Perl pretends that we wrote `split(/,/, $_)` anyway. In this case it didn’t matter, because comma is not special inside of regexes. But it seems simplest to just use `/.../` with `split` to avoid confusion; the first argument is a regex, so we should write it like a regex. So I’ll replace this line with:

```
            split(/,/, $_)
```

There’s one exception to this description of `split`: `split " ", ...` is a weird special case. Instead of behaving like `split / /, ...`, it behaves like `split /\s+/, ...`---except that it also discards leading null fields. This feature was put in to emulate the corresponding feature of `awk`, but the syntax Larry chose isn’t one of his better inspirations.

Summary

The original version of `checkzonesloaded.pl` was 61 lines long; the finished version is only 30. The modified version does the same things, but it’s missing a few potential bugs, it might be a tiny bit faster, and I think it’s easier to read.

Chapter 2 | TOP