
Merging Reports

Our first example is really excellent. It’s a common little system administration utility. Its job is to scan two input directories for files, and copy the files into an output directory, which it creates for the purpose. I love this program because it is a two-headed monster. What is disgusting and frightening about two-headed monsters? Well, they have two heads, and two is too many heads! You are only supposed to have one head. Almost everything in the program is repeated, once for the first input directory, and then again for the second input directory--it has two of everything. As a result, the program is just about twice as big as it should have been.

The complete code is on page Program ???---that’s the "before" version. The "after" version is on page Program ???. When I cite code in the upcoming discussion, I’ll include line numbers only if I’m quoting the original code.

On lines 19 and 20 we see the heart of the flaw in the program. The programmer made his big mistake right here:

```
19      $InputDIR1 = "./wuexport/";
20      $InputDIR2 = "./cuexport/";
```

By using two unrelated scalars instead of an array, the programmer made it difficult to perform the same

operations on each variable. Then he had to write each piece of code twice, once for `$InputDIR1` and once for `$InputDIR2`. So this is the two-headed sperm that gave rise to the two-headed monster.

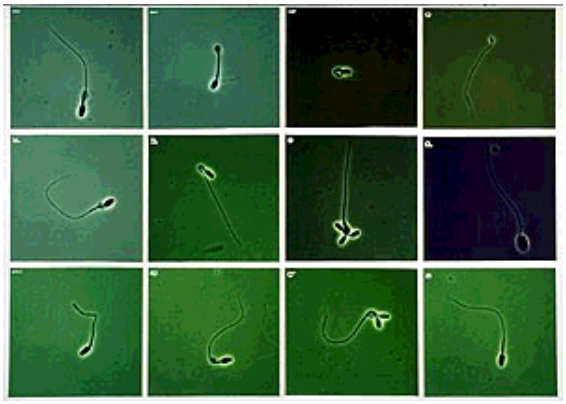

figure 1.1: `sperm`

This is a red flag, the red flag of **families of variables**. The corresponding Good Advice says that we should consider replacing the family with an array. But there are only two variables. Isn't it silly to have an array with only two elements? Perhaps, but not as silly as repeating every piece of code in the program twice! And anyway, there might someday be a third input directory. What are we going to do then, repeat everything in the program a third time? No, let's try the array and see where it takes us.

```
@InputDIR = ("./wuexport", "./cuexport");
```

I trimmed the trailing slashes from the strings here. That was for two reasons. One reason was a matter of principle: the strings are supposed to be directory names, and the trailing slashes are not part of the directory names. The other reason was less accessible: my years of experience say that that's usualy the thing to do. We'll see why shortly.

`GrabFileList()`

Now let's take a loot at one of the program's functions, say `GrabFileList`, and see what changes we'll have to make to it to make it work with `@InputDIR`:

```
113     sub GrabFileList {
114        opendir (FILELISTDIR1,$InputDIR1);
115        @FileList1 = readdir (FILELISTDIR1);
116        closedir (FILELISTDIR1);
117        opendir (FILELISTDIR2,$InputDIR2);
118        @FileList2 = readdir (FILELISTDIR2);
119        closedir (FILELISTDIR2);
120     }
```

This becomes:

```
sub GrabFileList {
for my $dir (@InputDIR) {
  opendir (FILELISTDIR,$dir);
  push @FileList, map "$dir/$_", readdir FILELISTDIR;
  closedir FILELISTDIR;
}
```

```
    }
```

So we have 7 lines that become 5. Okay, big deal. I cut out two lines of code. That does not seem like much. But another way to look at it is to see that I cut out 28% of the code. That sounds much more impressive! But the real point here is that the transformation was easy and almost mechanical. Get rid of family of scalars, replace with array. Get rid of repeated code, replace with `for` loop on array. Code cut 28%. If you could get rid of 28% of your entire program that easily, you would be overjoyed, wouldn't you? And once you have read this book you will find that you probably *can* get rid of 28% of your entire program that easily.

You probably noticed that I introduced another change to the function that I haven't mentioned yet. What is the `map` doing there in the middle?

Implicit Relationships

Here is the problem. There are two kinds of input files. Some of them came from one of the input directories, and some came from the other. Later, the program will need to copy these files, so it will need to remember which files came from which directory. Formerly, this information was *implicit*. There was a convention that names in one array represented files that came from one directory, and names in another array represented files that came from the other directory. But there was nothing in the program that explicitly said so. Every implicit relationship is one more thing that the maintenance programmer must remember. It's also one more chance to screw up. You have to be very careful whenever you're moving data in and out of the `@FileList1` and `@FileList2` arrays, because if you copy data out of those arrays into the wrong place, you might forget something important about it, something that is not actually recorded anywhere. It's better to make relationships explicit wherever possible.

You could put in a comment to explain. But a comment should be the explanation of *last* resort. It's better when things explain themselves. That is what the `map` is trying to do. If we have a list of filenames, some of which are from one directory and some from another directory, what's the best way to keep track of which directory each is from? Simple! Attach the directory name to each filename, making the filenames into full paths. That is just what full paths are for, to say what directory a certain file resides in. Now it is *impossible* to lose track of which directory each file came from, because some of the filenames look like `"./wuexport/carrots"` and others look like `"./cuexport/carrots"`. And the maintenance programmer cannot possibly get confused. Suppose the maintainer is running the program under the debugger, and prints out the contents of some filename variable. In the old program, there is nothing about the data to suggest what directory that filename came from. The maintainer will have to trace backwards through the code, hoping to discover the origin of the data. With the new design, the debuuger wil say something like `"./cuexport/carrots"` and there is no question.

So that's what the `map` is about. (If you haven't seen `map` before, it is simple. It is just like a miniature `for` loop. You give it an expression, `"$dir/$_"` in this case, and a list of values, `readdir DIR` in this case, and it loops over the list, applying the expression to each element in turn, and making up a list of the resulting values. In this case it is appending `"$dir/"` to the beginning of each filename.)

Backing Up

Now I have a confession to make. The `map` approach is not such a good fit for this particular task. Later

on, a function called `CopyFiles()` will try to insert another component, `$NextOut`, between the filename and the directory name:

```
 47         $x2 = $InputDIR1.$NextOut."/".$CopyList;
```

We could fix up `CopyFiles()` to take apart the path and then reassemble it with `$NextOut` in the middle. But that seems wasteful. Why assemble the path when we know we are just going to have to disassemble it again? A better approach is to keep the file lists from the various `@InputDIR`s separate to begin with.

Why did I show you the `map` solution if it turned out to be a bad idea? Because that was what I tried first, and I don't want to pretend to you that I always get the right answer on the first try. It's all too easy to pretend to be a super-genius, and only shows the stuff that works but the truth is that most of us, including me, are not super geniuses, and trying to be a super genius is probably not the best way for us to be effective programmers. I think a better way is to learn to recognize when something is not working well, and to try to change strategies.

Larry Wall, the inventor of Perl, says that the big problem that most language designershave is that they have some language problem that they need to solve, and then they think of a solution, and then they implement it. But what they should be doing is thinking of three or four solutions and then thinking them over and implementing the *best* one. I think programmers need to do the same thing. If a problem crops up, don't just go ahead with your first idea. Try more than one way around it. Often your best ideal will be your second, or third, or fourth.

I don't want you to think that these wonderful programming ideas spring forth out of my head like Athena from the head of Zeus. No, that's not how it works, unless you're a super genius, which I'm not. So I'm showing you my false starts and my not-so-good ideas so that you can remember that even the guy who wrote the book needs to try more than one thing because his first idea wasn't so good.

Anyway, the basic problem here is how to keep track of where the input files came from. The obvious way was to put the filenames together with the directory paths, but as we saw there's a reason why we might not want to do that. My next idea was to use a more complex data structure; the code would look like this:

```
sub GrabFileList {
my %result;
for my $dir (@InputDIR) {
  opendir (FILELISTDIR,$dir);
  push @{$result{$dir}}, readdir FILELISTDIR;
  closedir FILELISTDIR;
}
return %result;
}
```

This produces a hash whose keys are directory names and whose values are lists of files from those directories:

```
( './wuexport' => [ file1, file2, file3... ],
  './cuexport' => [ file6, file7, file8... ],
);
```

This would work OK, and I think a good program could be written to incorporate this solution. But after

a little more thought, I settled on a third solution which I think is simpler. Since we're having trouble keeping the filenames from the varius input directories separate, we'll cut the Gordian knot and solve the problem by not mingling them at all. Instead of asking `GrabFileList()` to read two input directories, and then going to a lot of trouble to avoid mixing up the two lists of files we get, we'll change it so that it reads *one* directory and returns a list of files. We will then pass this list to `CopyFiles()`. We have two input directories? No problem; we'll call `GrabFileList()` twice. The code now looks like this:

```
sub GrabFileList {
  my $dir = shift;
  opendir FILELISTDIR, $dir;
  my @files = readdir FILELISTDIR;
  closedir FILELISTDIR;
  return @files;
}
```

We'll have to call it once for each input directory, somthing like this:

```
for my $dir (@InputDIR) {
  my @files = GrabFileList($dir);
  # ...
  CopyFiles($dir, @files);
}
```

This makes good sense. Both `GrabFileList()` and `CopyFiles()` contained this `for` loop; by abstracting it up into the main program, we can eliminate the two loops from the ftwo functions, resulting in a simple overall control flow.

`CopyFiles()`

Now that we've seen our final, loopless version of `GrabFileList()`, we must modify `CopyFiles()` to match. The original code looks like this:

```
40     sub CopyFiles {
41       @CopyList = @FileList1;
42       foreach $CopyList (@CopyList) {
43         next if ($CopyList =~ /^\./);
44         next if !($CopyList =~ (/[0-9]/));
45         next if !($CopyList =~ (/txt/));
46         $x1 = $InputDIR1.$CopyList;
47         $x2 = $InputDIR1.$NextOut."/".$CopyList;
48         $Tmp = ‘cp $x1 $x2‘;
49         print $x1,"\-\>",$x2,"\n";
50       }
      ...
```

This is only the top half of the function; the bottom half is an identical conjoined twin. Since we're planning to change the function so that it will be called like this:

```
CopyFiles($dir, @files);
```

the modified version of the code looks like this:

```
sub CopyFiles {
  my $dir = shift;
```

```
  foreach my $file (@_) {
    next if ($file =~ /^\./);
    next if !($file =~ (/[0-9]/));
    next if !($file =~ (/txt/));
    $x1 = $dir.$file;
    $x2 = $dir.$NextOut."/".$file;
    $Tmp = ‘cp $x1 $x2‘;
    print $x1,"\-\>",$x2,"\n";
  }
}
```

19 lines of code have become 10.

We are now going to beat this function to death, hammering it until it is barely recognizable.

Say What You Mean

First, notice that `$Tmp` is useless:

```
$Tmp = ‘cp $x1 $x2‘;
```

After this statement is executed, `$Tmp` contains the output of the `cp` command. The `cp` command, however, produces no output. What is `$Tmp` there for, then? I do not know. I thought for a while that it might be preventing a `Useless use of ‘...‘ in void context` warning, but no, it isn’t; there is no such warning. Since `$Tmp` isn’t serving any purpose, let’s get rid of it:

```
‘cp $x1 $x2‘
```

Probably better would have been:

```
system("cp $x1 $x2");
```

The difference is that `‘...‘` asks Perl to run a Unix command and gather up its output, whereas `system(...)` asks Perl to run a Unix command and ignore the output. Since what we want here is to ignore the output, `system()` is mor appropriate. The Good Advice here is **say what you mean**.

The Concatenation Operator

The next bit of code we’ll address is these two lines:

```
56        $x1 = $dir.$file;
57        $x2 = $dir.$NextOut."/".$file;
```

Amazingly, **the concatenation operator** (.) is a red flag. The corresponding Good Advice is **use interpolation instead of operators**. Why? Because unlike the `.` operator, interpolation makes the code look like the string that the program will generate. There’s less stuff that youre brain must filter out:

```
$x1 = "$dir/$file";
$x2 = "$dir/$NextOut/$file";
```

In the original code, you couldn’t see what the output would look like; you would have to inore the extraneous dots and quotation marks in the middle. With the interpolation version, you can look at the code and *see* what the result will be. `$x1` gets `$dir` followed by slash followed by `$file`. `$x2` gets `$dir`

followed by slash followed by `$nextout` followed by slash followed by `$file`. I'll have a lot more to say later on about how **it's easier to see than to think**.

Earlier, I promised to explain why I eliminated the trailing slashes from the names of the input directories. This is the kind of payoff I was hoping for. Without the trimming, the modified code would have had:

```
$x1 = "$dir$file";
$x2 = "$dir$NextOut/$file";
```

This is less clear because you can't tell from looking at it whether there will be a slash inserted after `$dir`. In `$x2`, the contents of `$dir` and `$NextOut` are being joined in the same way as `$NextOut` and `$file`, but they're not written the same way so you can't see it. The Good Advice here is that **things that are the same should look the same**.

This code is far from the worst example of the problems of the dot operator. Here's an extremely common example:

```
$html = "<a href='".$url."'>".$hot_text."</a>";
```

Here it's quite difficult to disentangle the quotation marks. Is `$url` in quotes, or not? To answer, you have to run the Perl lexer algorithm in your head. Have you ever seen the code for the Perl lexer? It's an eight-thousand-line C program that starts off with a comment that says "It all comes from here, the stench and the peril."

If we use interpolation here instead of the dot operator, we get this:

```
$html = "<a href='$url'>$hot_text</a>";
```

Now you can see at a glance that the `$url` is in single quotes.

Returning to `CopyFiles()`, I'd like to get rid of the variables `$x1` and `$x2`, which are used here:

```
58        $Tmp = 'cp $x1 $x2';
```

My idea is that this would become:

```
system("cp $dir/$file $dir/$NextOut/$file");
```

and then you not only get rid of the two variabes but also make it possible to see at a glance what command is actually being executed. But I can't quite get rid of `$x1` and `$x2`, because they are used again a little later:

```
59        print $x1,"\-\>",$x2,"\n";
```

So while I'm mulling over what to do, I will fix the variable names. You've probably already been propagandized about **use sensible variable names**, so I'll spare you the tirade. I have an idea that the topics under the heading of "programming style" have gotten less and less until all that's left is a lot of tedious crap about naming variables, how to format your comments, and proper indentation. But it wasn't always so. Good prose style manuals cover a lot of tedious crap about punctuation, but they also cover ineresting matters of how to choose the right word and how to write clearly. Programming style

discussions used to have some interesting substance in them, but not any more, it seems. Anyway, I am going to change `$x1` and `$x2` to `$s` (for "source") and `$d` for ("destination"). `$s` and `$d` may not be the best names in the world, but they must be better than `$x1` and `$x2`.

```
my $s = "$dir/$file";
my $d = "$dir/$NextOut/$file";

system("cp $s $d");
```

Perl Has Too Much Punctuation

Now I'm going to say something that might shock you: Perl has too much punctuation. (Gasp!) We don't like to say so, but it's true. Puntuation in Perl is like kudzu, the weed that ate Georgia:

figure 1.2: kudzu

Isn't that a fine picture? I always wonder what that object in the middle is. Is it a car? A house? We'll never know, because it's covered in punctuation marks. Just like this next line of code:

```
59        print $x1,"\-\>",$x2,"\n";
```

A particularly important piece of Good Advice for Perl is **avoid excess punctuation**. Compare that line with this version:

```
print "$s->$d\n";
```

This is another nice example of why you should **use interpolation instead of operators**. We have reduced the punctuation in this line by nearly 50%, from 15 to 8 characters.

People sometimes worry that the `->` in the line will have some sort of dereferencing or method-calling effect. But it doesn't. Don't we need backslashes in there somewhere? No, we don't. The line is doing just what it appears to be doing: it prints two strings with an arrow in between.

Superstitious Parentheses

Come to think of it, what were the backslashes for in the original code? I think they betray an interesting psychological problem. The programmer here is not sure what is special inside of double-quotes. In a firghtening and poorly--understood world, we fall back on superstition. The backslashes here are pure superstition. "If you break a mirror, throw salt over your left shoulder to avoid bad luck. And try sprinkling your strings with backslashes."

But we don't have to fall back on superstition. We can look our fear straight in the eye, swallow hard, and bravely run the following command:

```
perl -le 'print "-->>->>->->--->->>"'
```

And what terrible things go wrong? Nothing. It prints:

```
-->>->>->->--->->>
```

I guess there was no monster under the bed after all.

We're engineers, not cave-dwellers, so we should act like it. The Good Advice is **don't be superstitious**. With a little practice, you can learn to notice when you're avoiding some construction because of superstitious fear, and then you can overcome that fear.

Incidentally, no double-quoted string does anything unusual unless it contains one of `$`, `@`, `\`, or `"`. If it doesn't contain any of those four, it behaves exactly the same as a single-quoted string.

Returning to the program, I would *still* like to get rid of `$s` and `$d`, for reasons cited earlier. But I can't, because they are needed in the diagnostic line

```
print "$s->$d\n";
```

But I got to wondering: is the format of the diagnostic important? Probably not; probably it is just there so that the programmer running the program can see that something is happening. I may be wrong about this, of course it sometimes happens that someone is assigned to reimplement a thirty-year-old COBOL program, and the output of the new program must be precisely the same. It's possible that by changing the format of this diagnostic message, I am breaking the program. But I don't think that's the way to bet, and if the exact format of the message *isn't* important, then I would prefer to do it like this:

```
my $command = "cp $dir/$file $dir/$NextOut/$file";
system($command);
print "$command\n";
```

With this change, not only is the format of the complete command visible at a glance in the source code,

but it is visible at a glance in the diagnostic output. If someone other than the original author sees the diagnostic output, they won't have to guess what the arrow means; the program is printing out exactly the commands it is executing.

One of the attendees of one of my talks suggested this alternative:

```
system("cp -v $dir/$file $dir/$NextOut/$file");
```

Here the `cp` command itself is responsible for printing the diagnostic, and two more lines of code go away.

The last part of the function that we haven't touched is the part that validates the filenames:

```
53                next if ($file =~ /^\./);
54                next if !($file =~ (/[0-9]/));
55                next if !($file =~ (/txt/));
```

Let's **avoid excess punctuation** here and get rid of the superfluous parentheses. Beginning programmers like to put in excess parentheses, because they are worried about precedence effects. I can sympathize with this---Perl's operator precedence is excessively complicated. But here the inner pairs of parentheses are pure superstition. They are not surrounding any operators that could cause a precedence conflict.

`B::Deparse`

Perl comes with a module called `B::Deparse` that can be a good cure for precedence superstition. After Perl compiles your program, `B::Deparse` gets control and tries to turn it back into Perl source code; this tells you what Perl thought your program meant; the output is guaranteed to be equivalent to the input. So we can run this:

```
perl -MO=Deparse -e 'next if !($file =~ (/txt/));'
```

and, because the output is this:

```
next if not $file =~ /txt/;
```

we know that it is safe to rewrite the original code as `next if not $file =~ /txt/;`

`B::Deparse` also has a `-p` option, which stands for "**P**ut in **p**recedence-**preserving p**airs of **p**arentheses." If we want to know whether something like `$c += $s =~ /.../` does what we think, we can use `-p`:

```
perl -MO=Deparse,-p -e '$c += $s =~ /.../'
```

The output shows that `$s` is matched against the pattern, and then the true-or-false result of the match is added to `$c`:

```
($c += ($s =~ /.../));
```

In the case of lines 53--55, we can make even the most superstitious programmer happy:

```
next if $file =~ /^\./;
next if $file !~ /\d/;
```

```
        next if $file !~ /txt/;
```

Now there can't possibly be a precedence conflict, because each line has only one operator!

The first of these lines may have a small bug. It is probably intended to skip over the `.` and `..` entries for the input directory and its parent directory. But it also skips over other files, such as `.netscape`, should ther ebe such a file in the input directory. And had `$file` contained a full path, such as `./wuexport/foo`, the file would have been skipped also.

All this suggests to me that this test is in the wrong place. Why are `.` and `..` in the `@FileList` in the first place? It shouldn't be the job of `CopyFiles()` to weed out bad filenames; its job should be to copy the files it is told to copy. It's the job of `GetFileList()` to manufacture the list of files; why is it including files in this list that we don't want to copy? We should fix `GrabFileList()` to return the correct list in the first place:

```
sub GrabFileList {
  ...
  my @files = grep !/^\./, readdir FILELISTDIR;
  ...
```

However, supposing that the intention here was to skip only `.` and `..`, the test is still wrong. Every Perl programmer seems to have their own favorite way to write this test, and I don't think it matters much which one you choose. My own preference is for:

```
my @files = grep { $_ ne '.' && $_ ne '..' } readdir FILELISTDIR;
```

I should probably have moved the other two tests (`next if $file !~ /\d/` and `next if $file !~ /txt/`) into `GrabFileList()` as well, but I forgot to do this when I was first writing up this program. That is okay. You do not have to make a program perfect the first time you work on it. It is enough to make it better. There is nothing wrong with making more improvements on a second pass.

At this point, `CopyFiles()` looks like this:

```
sub CopyFiles {
 my $dir = shift;
 foreach $file (@_) {
   next if $file !~ /\d/;
   next if $file !~ /\.txt$/;
   my $command = "cp $dir/$file $dir/$NextOut/$file";
   system($command);
   print "$command\n";
 }
}
```

Global Variables

The final major problem here is the global variable `$NextOut`. Whenever possible, functions should avaoid using global variables. To understand why, we must first understand why we have functions at all.

Why *do* we have functions? It is for *modularity*. Functions can be understood in isolation from the rest of the program. A maintenance programmer trying to change the way a 10,000-line program does its

database lookup will only need to read and understand that 40-line subroutine that actually does the database lookup; they can ignore the other 9,960 lines of the program.

Functions can also be re-used in a modular way. Someone trying to do a similar database lookup in another program can pick up the 40-line function from the first program and drop it down into the second program unchanged. Or, perhaps more likely, they can pick it up and put it into a library from which it can be used by many other programs.

Using global variables in functions defeats both of these benefits. If a function uses a global variable, the maintenance programmer can no longer understand it in isolation. To understand what it does, they will also have to understand the global variable, and the behavior of the global variable might be very complicated and might depend on many of the details of the other 9,960 lines of the program. At the very least, you the maintenance programmer must examine the entire rest of the program to make sure that none of it is modifying that global variable in an unexpected way.

Using global variables in functions also defeats the benefit of modular reuse. A function that uses a global variable cannot be picked up and plopped into another program or into a library; you would also have to carry along all the machinery for maintaining the global variable, and that is not always possible.

So using global variables in functions reduces modularity, defeating the primary reasons for using functions in the first place. Functions should use the usual parameter-passing methods for communication, avoiding **global variables** whenever possible.

The usual prophylaxis for the global-variable-in-function problem is to replace the global vaariable with a new function argument. If we were to do this, then `CopyFiles()` would be called like this:

```
CopyFiles($dir, $NextOut, @files);
```

and its code would look like this:

```
sub CopyFiles {
 my $dir = shift;
 my $NextOut = shift;
 foreach $file (@_) {
   next if $file !~ /\d/;
   next if $file !~ /\.txt$/;
   my $command = "cp $dir/$file $dir/$NextOut/$file";
   system($command);
   print "$command\n";
 }
}
```

But in this case we have a better option:

```
# Usage: CopyFiles(source_dir, destination_dir, files....)

CopyFiles($dir, "$dir/$NextOut", @files);
```

This is simpler and more flexible. Instead of being a function that will copy files from one directory to a subdirectory of that directory, `CopyFiles()` is now a function that will copy files from anywhere to anywhere.

Here's the final result:

```
sub CopyFiles {
 my $src = shift;
 my $dst = shift;
 foreach $file (@_) {
   next if $file !~ /\d/;
   next if $file !~ /\.txt$/;
   my $command = "cp $src/$file $dst/$file";
   system($command);
   print "$command\n";
 }
}
```

Isn't that tidy? We've cut it from 19 lines to 9.

`CreateInput()`

Here's the most bizarre result of the two-headed sperm:

```
32     &CreateInput(1);
33     &CreateInput(2);

63     sub CreateInput {
64       my $What = shift;
65       if ($What == 1) {
66           print "Creating DIR: ",$InputDIR1.$NextOut,"\n";
67           $I = mkdir ($InputDIR1.$NextOut,0777);
68       } else {
69           print "Creating DIR: ",$InputDIR2.$NextOut,"\n";
70           $I = mkdir ($InputDIR2.$NextOut,0777);
71       }
72       if ($I) {
73           print "Success! \n";
74       } else {
75           print "Fail! : $! \n";
76       }
77     }
```

When I was first writing up my conference talk about this program, I had some trouble expressing just why I thought this was ludicrous. I was flabbergasted, and I stood around for quite some time exclaiming "1??" "2???" I finally decided that the best way to get across why I think the 1 and the 2 are strange is with an equivalent but slightly different version. Instead of this:

```
32     &CreateInput(1);
33     &CreateInput(2);
```

why not have this:

```
&CreateInput("The Flesh Blanket");
&CreateInput("Maximo Perez");
```

The 1 and the 2 here are totally arbitrary. They look like numbers, but they aren't. They might as well be the strings `The Flesh Blanket` and `Maximo Perez` for all the difference it makes to this program.

The whole thing is deeply puzzling. If you pass the function one argument, it does one thing; if you pass

it the other argument, it takes a completely different code path. Why have one function? Why not two?

```
&CreateInput1();
&CreateInput2();
```

Or why not zero?

```
  print "Creating DIR: ",$InputDIR1.$NextOut,"\n";
  $I = mkdir ($InputDIR1.$NextOut,0777);
  print "Creating DIR: ",$InputDIR2.$NextOut,"\n";
  $I = mkdir ($InputDIR2.$NextOut,0777);
```

This `if-else` block is a perfect example of why you should **avoid families of variables**. If `$InputDIR1` and `$InputDIR2` had been an array, it would have been obvious to replace the five-line `if-else` block with two lines that use the mysterious `$What` argument as an index:

```
print "Creating DIR: ",$InputDIR[$What].$NextOut,"\n";
$I = mkdir ($InputDIR[$What].$NextOut,0777);
```

The very worst thing about this function, however, is its *name*. The function is called `CreateInput()`. What does it do? It creates the *output* directory. Seriously. The first time I read over this program, I got stuck for ten minutes on the name of this function, wondering what it was that I was missing.

Under the **say what you mean** rule, we are going to call this function `CreateOutputDir()` from now on.

The function's job is to create a directory. So instead of fooling around with all this Flesh Blanket rigamarole, and having it infer the directory name from whether the argument is `The Flesh Blanket` or `Maximo Perez`, let's just *tell* it what directory we want it to create:

```
sub CreateOutputDir {
 my $dir = shift;
 print "Creating DIR: ",$dir,"\n";
 $I = mkdir($dir,0777);
 if ($I) {
     print "Success! \n";
 } else {
     print "Fail! : $! \n";
 }
}
```

Then the calls to the function become:

```
&CreateOutputDir("$InputDir1/$NextOut");
&CreateOutputDir("$InputDir2/$NextOut");
```

Or, once we've replaced the family of variables with an array:

```
for my $dir (@InputDir) {
  &CreateOutputDir("$dir/$NextOut");
}
```

The function is more modular and more reusable, and we didn't need any extra code to make it that way.

Set Then Used Only Once

This is the only place that the `$I` variable appears:

```
$I = mkdir($dir,0777);
if ($I) {
```

The red flag here is **variable use immediately follows assignment**. The variable as assigned, used once right away, but never again. The = symbol in Perl performs a copying operation. The data in the right is copied into the variable on the left. Why make a copy if you are only going to use the information once? There are some legitimate answers to this question. One is that you might want to give the temporary value a name for documentative purposes. Clearly, that wasn't what was going on here, because the name `I` has no documentative value. In this case, the code is better written as

```
if (mkdir($dir,0777)) {
```

eliminating `$I`.

Debugger Variables

The other reason people use variables this way is so that they can place a breakpoint on the `if` in the debugger and examine the value of the condition *before* the branch is taken. This just points up a severe limitation of the debugger. There is no reason why the debugger couldn't allow you to place a breakpoint in an `if` statement after the condition is evaluated but before the branch. Why doesn't the debugger offer this feature? Apparently because nobody thought to implement it. Amazingly, debugger technology has advanced almost exactly zero since 1962; if you read the manual for a 1962 debugger you find *exactly* the same features that we have in the debuggers of today. Our crappy, obsolescent debugger technology is inducing us to write bad code with superfluous variables. In the next year or so, I hope to be able to rework the Perl debugger with new, advanced features, such as the ability to break in the middle of a statement.

Returning to the program, let's **use interpolation instead of operators**, changing this:

```
print "Creating DIR: ",$dir,"\n";
```

to this:

```
print "Creating DIR: $dir\n";
```

Yakkity yak yak yak

Now the function looks like this:

```
sub CreateOutputDir {
 my $dir = shift;
 print "Creating DIR: $dir\n";
 if (mkdir $dir, 0777) {
     print "Success! \n";
 } else {
     print "Fail! : $! \n";
 }
```

```
}
```

I don't want to belabor this too much, but I wonder about those `prints`. Suppose the `mkdir` *does* fail. Is it really appropriate to go ahead with the function? The program's entire purpose is to copy some files to the output directory; if it can't create the output directory, what's the point of continuing? Conversely, what if it does success? Does it really need to announce `Success!`? We're going to see it copying all those files anyway.

Talkative functions can be nice during debugging, but eventually you want them to shut up. Printing your diagnostic messages to `STDOUT` is convenient, but there's a reason why `STDERR` was invented: it's to keep your program's diagnostics separate from its regular output. If I were writing this, I would have done something more like this:

```
sub CreateOutputDir {
 my $dir = shift;
 mkdir $dir, 0777
   or die "Couldn't make dir $dir: $!";
}
```

Structure vs. Function

This is genuine architectural terminology. I used to call these ideas something else, until I discovered that architects have exactly the same ideas.

The understand the difference between structure and function in architecture, consider the simplest possible structure, a tent.

figure 1.3: `teepee`

The purpose of the tent is to keep the rain off of you. It has *structural* and *functional* members that help it do this. The tent cloth is the important part, because the cloth is directly advancing the function of the tent. It's the cloth that actually repels the rain; the cloth is a *functional* member. But the tent has another part, namely the tent pole. The pole is not doing anything to keep the rain off. It is a *structural* member. It is only there to support the functional members---it holds up the cloth.

The pole, like all structural members, is a liability. It costs money. You have to carry it around. You might hit your toe on it in the middle of the night. If you could figure out some way to get the cloth to stay up without a pole, you would. But you can't, so you need the pole.

Sometimes when I teach my Red Flags class at big companies I get lucky and we get a space in a conference room that exemplifies this perfectly:

figure 1.4: `conferenceroom`

Can't you imagine the architect of this building designing the perfect conference room? "Ah!" says the architect. "I'll put in these walls, to keep the other people on the floor from disturbing the training. And I'll put in these beautiful windows, to let in the light. And the crowning touch---I'll stick a giant pillar in the middle of the room!"

No, the pillar is not there to serve the purpose of the conference room. It is there to hold up the building. If we could build the conference room without the pillar, we would. But without it, the building will fall down.

Code in your program similarly serves structural and functional purposes:

```
sub CreateOutputDir {
 my $dir = shift;
 mkdir $dir, 0777
   or die "Couldn't make dir $dir: $!";
}
```

The purpose of this program is to create a directory and to copy some files into it. The `mkdir` is directly advancing that goal; it is functional code. The `die` is also functional: part of the program's function is to issue an appropriate diagnostic if it can't do its job. But the rest of the code is purely structural. It does not advance the real goal of creating a directory; it only advances a secondary goal of having a subroutine.

So only one of the three lines here is functional. The Good Advice is to **minimize structure**. If the structural parts were eliminated, this would be a one-line function.

One-line functions

A one-line function that is called from only one place is almost entirely useless.

Multiline functions called from only one place are *not* useless; they can be used to give a simple name to a large operation, and to break the code into more manageable chunks. And single-line functions aren't useless if they are called from many places, because they are abstracting a common operation. But a single-line function that is called from one place does not deliver either of those benefits. At best, it derives some documentative value from its name. Countervailing against that is the consideration that the name might not communicate the function's behavior as clearly as its single line of code would have.

In this program, we have:

```
for my $dir (@InputDir) {
  &CreateOutputDir("$dir/$NextOut");
}

sub CreateOutputDir {
 my $dir = shift;
 mkdir $dir, 0777
   or die "Couldn't make dir $dir: $!";
}
```

But getting rid of the one-line function `CreateOutputDir()` leaves us with:

```
for my $dir (@InputDir) {
  mkdir "$dir/$NextOut", 0777
 or die "Couldn't make dir '$dir/$NextOut': $!";
}
```

and so five lines become two. (Or six become three, depending on how you count.) The new code is better in all ways. We did lose the old reporting behavior, but I think that is a benefit.

`FindLastOut()`

Moving on to the next function, the big problem here is the argument passing and use of **global variables**. `FindLastOut()` depends on two global variables: `$LastOut` and `$OutputDIR`.

```
 97     sub FindLastOut {
 98        opendir (FINDLASTOUT_OUT,$OutputDIR);
 99        my @Files = readdir (FINDLASTOUT_OUT);
100        closedir (FINDLASTOUT_OUT);
101        my $Highest = $LastOut;
102        foreach $File (@Files) {
103                next if ($File =~ /^\./);
104                next if !($File =~ (/[0-9]/));
105                if (int(substr($File,0,5)) >= $Highest) {
106                        $Highest = int(substr($File,0,5));
107                }
108        }
109        print "Last Entry = ",$Highest,"\n";
110        return $Highest;
111     }
```

The usual prophylaxis in this case is to turn the global variables into function arguments, as we did initially with `CreateOutputDir`. That is indeed the right thing to do with `$OutputDIR` in this case.

But what about `$LastOut`? There's no need to make `$LastOut` into an argument because it is always zero! The function becomes:

```
sub FindLastOut {
   my $OutputDIR = shift;
   opendir (FINDLASTOUT_OUT,$OutputDIR);
   my @Files = readdir (FINDLASTOUT_OUT);
   closedir (FINDLASTOUT_OUT);
   my $Highest = 0;
   foreach my $File (@Files) {
           next if $File eq '.' || $File eq '..';
           next if $File !~ /\d/;
           if (int(substr($File,0,5)) >= $Highest) {
                   $Highest = int(substr($File,0,5));
           }
   }
   return $Highest;
}
```

(I've also made some minor changes to the tests in the bottom half of the function.)

The next piece of code I'll address is:

```
104             next if $File !~ /\d/;
105             if (int(substr($File,0,5)) >= $Highest) {
106                     $Highest = int(substr($File,0,5));
107             }
```

This worries me. The code wants `$File` to begin with 5 digits. But the test on line 104 is not testing for that. The **say what you mean** rule says that we should change this to:

```
next if $File !~ /^\d{5}/;
```

which incidentally renders the test for `.` on line 103 unnecessary.

The `int()` is Just Plain Wrong---not a red flag; just a mistake. Perhaps the author thought it was for converting a string to an integer. But Perl automatically converts strings to integers whenever nexessary; `int()` is for converting a fraction to an integer by throwing away the fraction part. I'm replacing this chunk of code with:

```
next if $File !~ /^\d{5}/;
my $n = substr($File, 0, 5);
if ($n > $Highest) {
$Highest = $n;
}
```

I've also changed `>=` to `>`; there was no need to execute the code if the values were equal.

This reads the filename `123456foo` as beginning with `12345` rather than with `123456`. This may or may not be a bug, if it is a bug, it may or may not come up. If it is a potential problem, we should use something like this:

```
next if $File !~ /^(\d{5,})/;
if ($1 > $Highest) {
$Highest = $1;
}
```

or, in place of that `if` block, perhaps something like this:

```
$Highest = $1 if $1 > $Highest;
```

Some people would have preferred this:

```
$Highest = ($1 > $Highest) ? $1 : $Highest;
```

I cannot predict which of these you find clearest. But I know from talking to many audiences over the years that there is a good chance that you will prefer one of them to the others. The Good Advice is to **try it both ways**. Don't just go with the first expression that pops into your head; ask yourself if you can think of any other ways to write the same thing that might be clearer or simpler. And don't just ask yourself; really *try* out some other ways, actually type them onto the screen so you can look at both bits of code side by side. You may find yourself surprised at how much clearer one of the alternatives is than you thought it would be, or how much less so. When you hold the alternatives in your head, you may not be able to compare them accurately. But if you can see them on the screen, you'll see at a glance which one is preferable, because **it's easier to see than to think**.

Here's why **it's easier to see than to think**:

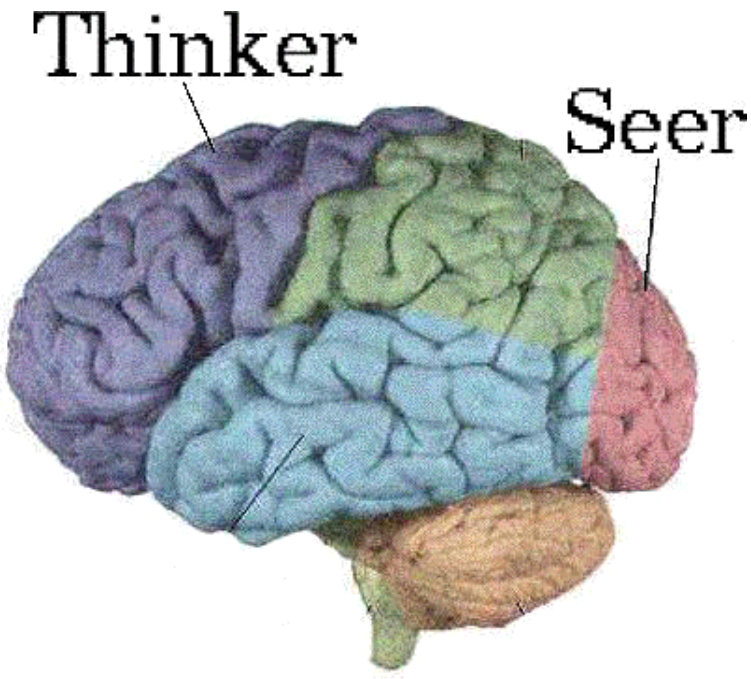

figure 1.5: `lobes`

Your brain has several subsystems. The reasoning is done by a subsystem up in the frontal lobe, but the seeing is done by a different system on the occipital lobe. It's like you have dedicated vision coprocessing hardware in your head. And if you can offload work from the thinker onto the seer, that frees up the thinker to do more thinking. Thinking is hard to do, so we want to offload as much as possible.

That's why we prefer this:

```
$html = "<a href='$url'>$hot_text</a>";
```

to this:

```
$html = "<a href='".$url."'>".$hot_text."</a>";
```

In the first example, your thinker is working hard to run the Perl lexer algorithm, and that means it's not available to think about something important, like whether the program will work correctly or whether there is a better way to write it. By using the first example, we offload the work of parsing the form of the string into the seer, freeing up the thiner to do something more useful than to count quotation marks.

GetPadString()

I left this function for last because it's my very favorite. Here's the code:

```
122   sub GetPadString {
123      my $Integer = shift;
124      if ($Integer < 10) {
125               return "0000".$Integer;
```

```
126      } elsif (($Integer >= 10) and ($Integer < 100)) {
127              return "000".$Integer;
128      } elsif (($Integer >= 100) and ($Integer < 1000)) {
129              return "00".$Integer;
130      } elsif (($Integer >= 1000) and ($Integer < 10000)) {
131              return "0".$Integer;
132      } elsif ($Integer >= 1000) {
133              return $Integer;
134      }
135   }
```

This is a wonderful example of the world's largest Red Flag:

Repeated Code is a Mistake

Years ago, before I had a red flags class or even had thought of the idea of red flags, I still knew about this largest of all red flags. In those days I used to call it "The Cardinal Rule of Computer Programming":

Repeated Code is a Mistake

Why is repeated code a Mistake? The short answer is that it's a mistake because it is *wasteful*. To understand this fully, we have to indulge in a little bit of philosophy.

Programmers have a big problem. The problem is that they think of code as their product. "I'm a programmer," they say, or sometimes even "I'm a coder." "That means my job is to produce code," they say. No. that is wrong.

Maybe you are working on the Human Genome Project, and your job is to help sequence the human genome. If so, that is your product. The code is structural. If someone could figure out a way to sequence genomes without writing any code, they would. But they haven't, so they write the code. The code is structural. It is not the primary goal; it is only there to advance the primary goal of sequencing DNA fragments.

Of maybe you work for Morgan Stanley or some other financial services company whose mission is to make a zillion dollars by speculating in the mortgage markets. Of morgan Stanley could make a zillion dollars trading mortgages without writing any code, they would, and they would fire all their programmers, because they are not in the business of writing code; they are in the business of making a zillion dollars trading mortgages. Writing the code happens to be the best way to do that, this week.

The programmer's job is not to produce code, but to produce *functionality*. The code itself is structural. It is there to support the functionality, but it does not contribute directly to the function. If we could get rid of it, we would.

In fact, the code is a *liability*. It costs money to write and maintain. Code has negative value.

Twice as much code costs twice as much to write and to test.

It takes twice as long to read and to understand.

It has twice as many chances to have bugs in it.

If it is delivering twice as much functionality, the cost is justified. But if the code is the same as some other code in the same program, it is delivering much less than twice the functionality. You have twice as much code to do the same job, so although the cost is twice as great, the benefit is much less than twice as great. You already got the benefit from the first batch of code; the added benefit from the second batch is nearly zero.

That is why repeated code is wasteful. It has a much higher cost-benefit ratio than unrepeated code.

Repeated code creates maintenance problems. Someone is sure to make an enhancement or a bug fix to one of the copies of the repeated code, but forget to add the enhancement or the bug fix to the other copy or copies. I'm sure this has happened to you sometime.

Repeated code is boring to write. It's hard to keep your brain fully engaged while you are writing it, so it's easier than usual to mess up an introduce errors.

Repeated code is hard to read. Your eyes and your seeing hardware are programmed to skip past visually-similar things without noticing that they are not precisely the same. Here is a symmetric arrangement of dots:

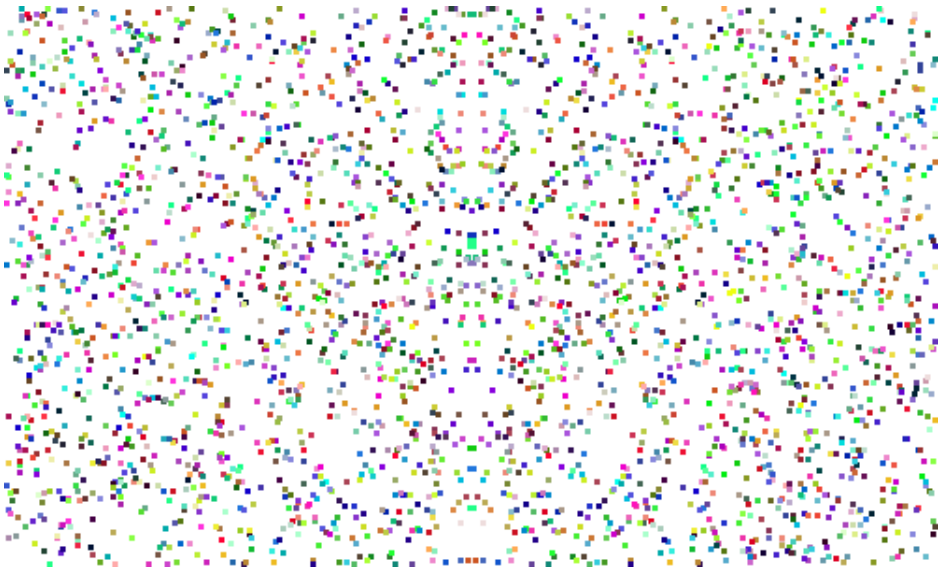

figure 1.6: `blot`

You can see this at a glance, because your seeing hardware is very good at detecting vertical symmetry. It is very important to detect vertical symmetry, because animals are vertical, and if you see something vertically symmetric, it might be an animal looking at you, perhaps because it wants to eat you. It is much better for you to have some false alarms in the vertical-symmetry detection than to fail to see some vertically-symmetric things some of the time, so your seeing hardware is much more likely to detect symmetry where it isn't actually present than to fail to detect it where it is present. The dots above look symmetric even though they are not. Only the middle part is symmetric. The parts on the left and right are totally random. But your seeing hardware sees the symmetry in the middle and jumps to the conclusion that the whole thing is symmetric, and informs your thinking hardware of that conclusion.

All of this means that if you have repeated code, and you mess up and make part of it a little wrong, you are not likely to notice; your eyes are likely to tell your brain that you got it right. There is a bug of this type in `GetPadString()`; did you notice it? [1]

Repeated code harbors bugs. Here's an example I took from usenet: [2]

```
if($match) {
   (($rank = @ranks[0]) && ($percent = "1%")) if $value == 1;
   (($rank = @ranks[1]) && ($percent = "2%")) if $value == 2;
   (($rank = @ranks[2]) && ($percent = "3%")) if $value == 3;
   (($rank = @ranks[3]) && ($percent = "4%")) if $value == 4;
   (($rank = @ranks[4]) && ($percent = "5%")) if $value == 5;
   (($rank = @ranks[5]) && ($percent = "6%")) if $value == 6;
   (($rank = @ranks[5]) && ($percent = "7%")) if $value == 7;
   (($rank = @ranks[6]) && ($percent = "8%")) if $value == 8;
   (($rank = @ranks[7]) && ($percent = "9%")) if $value == 9;
   (($rank = @ranks[8]) && ($percent = "10%")) if $value == 10;
}
```

Several people "corrected" this code to this instead:

```
if ($match) {
  $rank = $ranks[$value-1];
  $percent = "$value%";
}
```

But this code is not equivalent to the original. Notice that the numbers in the first line go 0,1,1, and the numbers in the second line go 1,2,2, but the numbers in the last line go 8,10,10 instead of 8,9,9.

Now, there are two possibilities here. Perhaps the original code had a bug, and the repetition of the subscript 5 was an error. But none of the people looking over and correcting this code pointed that out, which I think proves my point about how hard these things are to see. Or perhaps the original code was *correct*, and all the people following up with "corrections" actually broke the code, because they did not notice the repeated 5---which I think proves my point even more!

Anyway, *someone* had a bug. I was not very proud of our community that day.

Fortunately, it is not only possible to eliminate repeated code, it is usually easy. This is because repeated code is so awful, such a bug problem, that almost *every* feature of *every* programming language is there specifically to help you eliminate repeated code from your programs.

Why does Perl have a `+=` operator? So we can write `$x->{q} += $y` instead of `$x->{q} = $x->{q} + y`, avoiding the repeated expression, avoiding the possibility of messing up one of the two repeated expressions, avoiding the maintenance problems that might occur when someone changes one of the `$x->{q}`'s but not the other, and avoiding forcing the maintenance programmer to compare the two expressions to make sure they are the same.

Why do languages allow you to open a new block wherever you want and declare a new variable which is private to just that block? One reason is so that you can write something like this:

```
{
  my $x = SOME LONG COMPLICATED EXPRESSION;
```

```
            ... $x ...
          ... $x ...
                      ... $x ...
        }
```

instead of repeating the long complicated expression over and over.

Why do languages have `while` and `for` loops? So you can tell the language to repeat some piece of code at run time, instead of having to repeat it textually.

Why do languages have subroutines? Subroutines were invented in the 1950's specifically so that programmers did not have to repeat code in their programs.

Why do object-oriented programming systems support inheritance? So that if two classes need the same method, you do not have to repeat the code for the method in each class; you can move it into a common parent class, and let the two classes inherit it.

Why does Perl have modules? So that not everyone has to write theiur own date parsing routines and include them in their programs and update them every time a performance or a bug improvement comes out.

Why does Unix have pipes? So that not every program has to have sorting built into it; instead, a program can open a pipe to the sorting utility.

Why do operating systems have shared libraries? So that not every program needs to have a copy of `printf()` compiled into it; they can all share one printf, which can be upgraded all at once.

So we see that many many features of languages and operating systems, at many levels, from tiny little language features like the `+=` operator to big important features like shared libraries, are there to help you avoid repeated code. They were put in *specifically* to help you avoid repeating code.

Sidebar: Repeated Code

Here's a nice little repeated code example, again from usenet: [3]

```
...
y/A-Za-z/22233344455566677788899900222333444555666777888999000/;
...
```

When I saw this, I had to grovel over the right-hand side to make sure it listed the same numerals in the same order.

And there is another problem. What is the translation of `mjd`? Quick now!

Here's a better version. Since we are doing the same thing for upper- and lower-case letters, let's make that explicit:

```
y/A-Z/a-z/;
y/a-z/22233344455566677788899900/;
#     abcdefghijklmnopqrstuvwxyz
```

Comments like this have a high information value; notice the way the comment is formatted. Now you can see at a glance that the translation of `mjd` is `653`. And **it's easier to see than to think**.

Here's another possibility:

```
y/A-Z/a-z/;
y{abcdefghijklmnopqrstuvwxyz}
 {22233344455566677788899900};
```

Now the comment is unnecessary. Which do you like best?

Returning to `GetPadString()`, which looked like this:

```
124         if ($Integer < 10) {
125                 return "0000".$Integer;
126         } elsif (($Integer >= 10) and ($Integer < 100)) {
127                 return "000".$Integer;
    ...
```

Probably the best way to write this is with `sprintf()`:

```
return sprintf "%05d", $Integer;
```

Then 10 lines of code become 1.

But perhaps the original author didn't know about `sprintf()`; not everyone does. Perhaps he was a beginner. Fair enough. Even a beginner should not have to repeat code. The language wants you to avoid repeating code. You have almost every feature of the language working to help you. With so much help, there is almost always a solution. The beginner might have written something like this:

```
my $n_zeroes = 5 - length($Integer);
my $zeroes = '0' x $n_zeroes;
return $zeroes . $Integer;
```

And 10 lines became 3.

Okay, perhaps the beginner does not know about the `x` operator. But if you don't, you can still use a loop:

```
while (length($Integer) < 5) { $Integer = "0$Integer" }
```

This is utterly straightforward. While the length of the integer is less than 5, put a zero on the beginning. Even a beginner could have written this. Even someone who learned Perl that very morning could have written that.

Which of these techniques do you like the best? Look them over and think about it. Which do you like the least?

When I teach this in classes, someone often suggests their own favorite method of accomplishing this, which is often something like this:

```
return substr("00000$Integer", -5);
```

I'm not partial to that, but you may prefer it. What do you think?

Do Not Repeat Code

Now assuming that we have replaced `GetPadString()` with `sprintf()`, it is only one line long, and so it is a one-line function that is called from only one place, and we should eliminate it entirely, writing simply:

```
$NextOut = sprintf("%05d", FindLastOut('.') + 1);
```

The total net savings is 15 lines of code and two variables.

We're finished!

That was a lot of fuss, but look at the benefit! Our 71-line program has shrunk to 32 lines. Not be squeezing out the white space, or by writing everything on one line, but by genuinely simplifying the program, eliminating unnecessary code.

On Design

> Perfection is achieved not when there is nothing more to add, but when there is nothing more to take away.
>
> (Antoine de Saint-Exup?ry)

---

[1] The bug is on line 132, where the test is `$Integer >= 1000`, but it should have been `$Integer >= 10000`.

[2] Message-ID `<u1v5l1cf8qtv26@corp.supernews.com>`

[3] Message-ID: `<pqalftcopgu5addk8tkd0rlnfgujp1pef6@4ax.com>`

TOP