
`if` and `else`

You probably learned about `if` and `else` the first day you learned Perl, and they seem simple. But it's surprising how often they are used badly. There are a lot of things that can go wrong.

Empty Blocks

You might think that nobody would ever leave an `if` or an `else` block empty, but it's not at all uncommon:

```
Subject: Turning on/off "use diagnostics;"
Message-Id: <71ln1f$d5m$1@nnrp1.dejanews.com>

$debug = 0;
foreach $argv (@ARGV){
  if($argv =~ /^-debug$/) {
    $debug = 1;
  } else {
  }
 }
```

This `else` block is not serving any purpose, so let's get rid of it:

```
$debug = 0;
foreach $argv (@ARGV){
  if($argv =~ /^-debug$/) {
    $debug = 1;
  }
 }
```

It's a little easier to understand when it's the `if` block that is empty:

```
Subject: Problem for uploading a file
Message-Id: <UCWz4.3319$Fk3.169728@weber.videotron.net>

if ($File_Handle =~ /([^\/\\]+)$/) { }
else { print p("Bad filename... check for forward or back slashes\n"); }
```

The default corection to apply here is to use `if (not ...)` or `unless`:

```
unless ($File_Handle =~ /([^\/\\]+)$/) {
  print p("Bad filename... check for forward or back slashes\n");
}
```

But sometimes the condition is already negated, as it is here, so we can write:

```
if ($File_Handle =~ /[\/\\]$/) {
  print p("Bad filename... check for forward or back slashes\n");
}
```

Or, cleaning up the **leaning toothpick syndrome** a bit:

```
if ($File_Handle =~ m{[/\\]$}) {
  print p("Bad filename... check for forward or back slashes\n");
}
```

When chaning `if` to `unless`, watch out for `^` in character classes and for `!` and `!~`operators.

Here's my favorite example ever of a program with empty `if` blocks:

```
Subject: Lame coding question
Message-Id: <NbPo2.3$_o.306113@news0.optus.net.au>

$choice = <STDIN>;
chomp $choice;

if ($choice eq "Y") {
        &mailatt;
}

if ($choice eq "y") {
        &mailatt;
}

if ($choice eq "yes") {
        &mailatt;
}

if ($choice eq "YES") {
        &mailatt;
}

if ($choice eq "N") {
}

if ($choice eq "n") {
}

if ($choice eq "no") {
}

if ($choice eq "NO") {
}
```

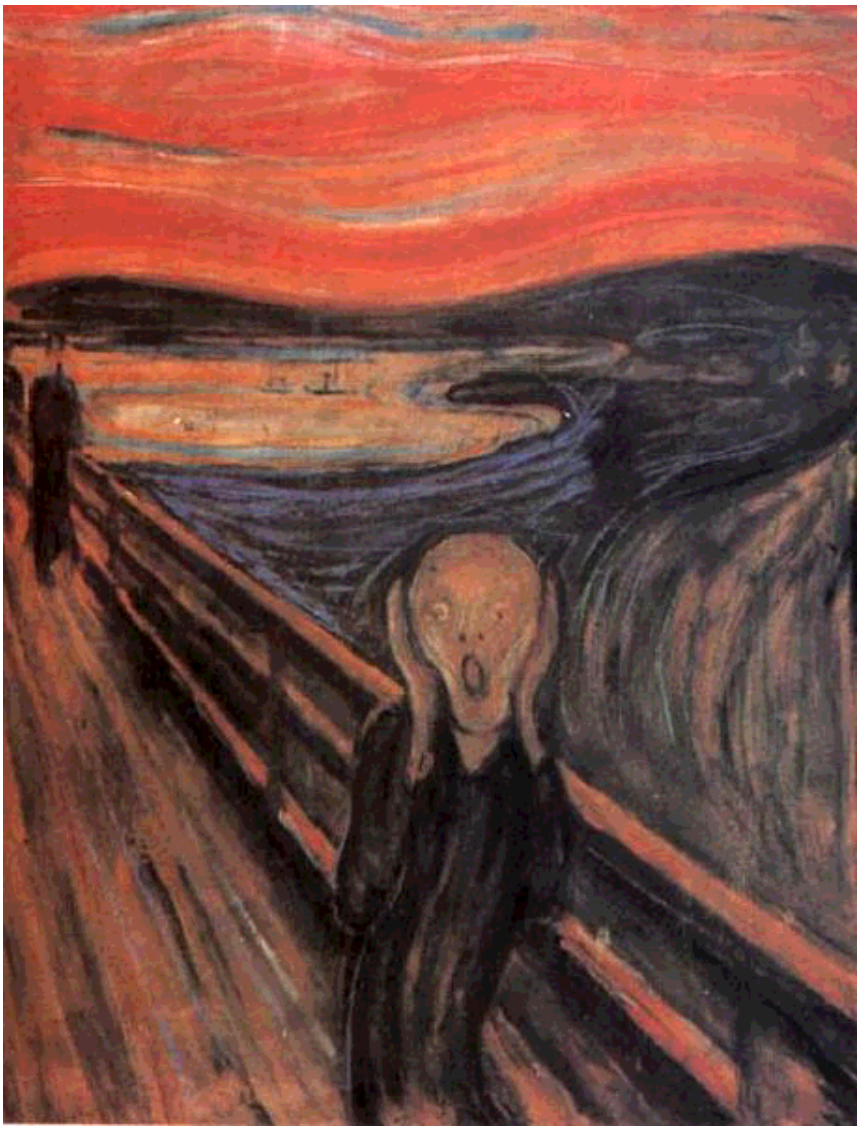

figure 7.1: `Scream`

This is why I have to use real examples instead of making up examples. I am not clever enough to make up anything as funny as this. Let's fix it quickly, before I barf:

```
$choice = <STDIN>;
chomp $choice;
mailatt() if $choice =~ /^y/i;
```

14 lines have become three. I bet you are thinking that this sort of example is unusual, and that I comb through my vast archive and carefully select the very worst of the worst. I don't. I wote a little Perl script to search for empty `if` blocks, and it quickly found 72 examples. This was the third one I looked at. I have not looked at the other 69, but odds are that some of them are even worse. Richard A. Posner, a federal judge, says:

> Truth really is stranger than fiction, because writers of fiction try to

be plausible, and reality has no aim.

`else` / `if`

A red flag is when the only thing in the `else` block is another `if`; you should usually prefer to rewrite it with `elsif`. For example, this:

```
Subject: Re: Out of Memory!!!
Message-Id: <3761700b.9617473@news.videotron.ca>


if (...)
{
  ...
}
else { if ($ExitCode == 259) { $alive=1; $nbalive++; }}
```

becomes this:

```
if (...)
{
  ...
}
elsif ($ExitCode == 259) { $alive=1; $nbalive++; }
```

If nothing else, this tends to keep code closer to the left-hand margin and to reduce punctuation.

Sometimes you see even more bizarre examples of this:

```
Subject: Retrieving an Entry Within a File
Message-Id: <MPG.139f24a68ea4317f989682@News.CIS.DFN.DE>

if ($device eq $_) {

$mail{To} = 'pager@domain.com'; } else {
if ($device ne $_) {$mail{To} = 'email@domain.com'; }
```

Replacing this with `elsif` would give us:

```
if ($device eq $_) {
  $mail{To} = 'pager@domain.com';
} elsif ($device ne $_) {
  $mail{To} = 'email@domain.com';
}
```

But since the two tests are exactly opposite, it should really be:

```
if ($device eq $_) {
  $mail{To} = 'pager@domain.com';
} else {
  $mail{To} = 'email@domain.com';
}
```

Let's **try it both ways**. Would it be better to write this:

```
$mail{To} = ($device eq $_) ? 'pager@domain.com'
                            : 'email@domain.com';
```

What about this:

```
$mail{To} = ($device eq $_ ? 'pager' : 'email')
            . '@domain.com';
```

Unbalanced `if-else` Blocks

Let's return to the the `each_file()` example of Chapter ???. When we last left it, it looked like this:

```
sub each_file {
  return unless -T $_;

  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";
  }
}
```

The `if` blocks are all quite top-heavy. The normal flow of control is nested 1-2 levels deep in the blocks; it was 2-3 levels before we banished **The Condition that Ate Michigan**. Normal flow should be close to the left-hand margin, or else it's hard to see which `else` goes with which `if`. We can rewrite this as:

```
sub each_file {
  return unless  -T $_ ;

  print "processing $File::Find::name\n";
  unless ( open F, "< $_" ) {
    print "Error opening file for read: $!\n";
    return;
  }

  my $s = <F>;
  close F;
  unless ( open F, "> $_" ) {
    print "Error opening file for write: $!\n";
    return;
  }

  $s =~ s/COLUMN1/COLUMN1/ig;
  print F $s;
  close F;
}
```

The control flow is now simple and linear, with the error-handling blocks as subsidiary branches, over on the right. It’s ovcious what the error handlers are doing, and they can all be understood independently of one another. The symmetry between the reading and writing operations has also become clearer.

This is a very common pattern, one of the most common in all programming:

```
if (some operation succeeds) {
  do the next operation;
} else {
  display an error message
}
```

But often it’s better to break out of the normal flow of control immediately:

```
unless (some operation successds) {
  display an error message
  next OR last OR return OR die
}
do the next operation
```

The normal case, which is the thing that the maintenance programmer needs to understand first, remains simple and linear, as in the previous example.

`unless-else`

Some people love `unless`, and some hate it, and I think I know how to tell the difference. Native English speakers tend to like it, and non-native English speakers tend not to. As a native English speaker, I can understand `unless ($x == $y)` intuitively and atomically. But Russian speakers in my class always complain that they have to mentally negate the condition to `if ($x != $y)`. `unless` is one of those great features of Perl that always makes me feel like an ass when I have to explain it to a class ful of Cantonese. (`next LINE` is another.)

I think that the usefulness of `unless` is closely bound up with its English meaning, and for that reason, it is best to avoid using `unless` with `else`:

```
unless (C) {
  A;
} else {
  B;
}
```

"Unless ... else ..." is just not a construction that we use in English. In Perl it is often better to write:

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

Sidebar: Finding the Minimum

This example comes from the book Programming Style: Examples and Counterexamples. The problem is to find the minimum of three numbers. The original code, which I have translated for you from the original FORTRAN, said:

```
if ($x < $y) {
  if ($x < $z) {
    $small = $x;
  } else {
    $small = $z;
  }
} else {
  if ($y < $z) {
    $small = $y;
  } else {
    $small = $z;
  }
```

Except that in 1974, FORTRAN had no `ELSE` so imagine this with six `GO TO`s instead. The improved version presented by the authors has:

```
$small = $x;
if ($y < $small) { $small = $y }
if ($z < $small) { $small = $z }
```

I don't know what to conclude from this, but it appeared to be worth mentioning.
