The WTFish side of using Perl

Lech Baczyński

http://perl.baczynski.com

Thanks, Hurra

I would like to thank Hurra Communications,
the company that sponsored my trip here,
probably the most Perl-ish company in Poland
www.hurra.com

What do I mean by WTF

Strange unexpected results of Perl code:
- developer's inexperience, or
- Perl's strange behaviour

(both seem to be the same after a while :-))

Examples: foreach variable localization, “zero but true”, dangers of omitting brackets, lazy **print**...

Some are for beginners (not-beginners – have patience!), some may puzzle intermediate programmers.

Lazy print

Let's say you would like to see progress of a long computing...

```
foreach my $element (@array) {
        # ..... long time operations ...
        $i++;
        if ( $i % 100 == 0) {  print ".";  };
```

Will it print dot every 100th iteration?
No. It will print all of them at once at the end.

Lazy print - continued

$|++

WTF is $|++ ? It is incrementing one of Perl's strange variables (magic punctuation variables).

$| If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel.

Using $|++ is not perl best practice. Solution:

use English; # '-no_match_vars';
then use the $OUTPUT_AUTOFLUSH variable

Who needs brackets

Omitting brackets – both convenient and dangerous

```
$a="Hello";
$b="world";

print $a . " " . $b . "\n";

print "Found ". scalar @arr ." elements\n";
```

Who needs brackets - continued

$a = 'Hello';
print (length ($a));
print length $a;
output: 5 - ok

print length $a . " letters\n" ;
output: 14 – WTF??

Length was calculated of concatenated strings $a and "letters\n", thus length returned 14. So beware - sometimes you may ignore brackets, sometimes you may not.

Regexps: {n,m}

Let's say you want to match three to five small letters :

/[a-z]{3,5}/

And now any number not less than three

/[a-z]{3,}/

And now, analogically, any number not more than five

/[a-z]{,5}/ # WRONG

It works only one way - {n,}, not {,n} - the explanation is easy: you can easily write 0, but it not so easy to write infinity

Regexps:
Minus (-) sign in character classes [ ]

Beware of matching "-" in regexps. If you want to match letters a,b,c and minus sign:

Good: [-abc]

Good: [abc-]

Bad: [a-bc]

Good: [a\-bc]

Regexps: delimiters

Only if you use // then m is optional. Some need to be closed other way than opened.

/abc/ - ok
|abc| - wrong
m|abc| - will work
m/abc/ - will work, m is optional
m#abc# - will work, not a comment
m(abc) m{abc} m[abc] - ok

Perl Bast Practices: Don't use any delimiters other than // or m{}

Foreach var localization

```
my @array = (1,2,3);
my $var = 'foo';

foreach $var (@array) {  # note - no "my $var"
                         print $var . "\n"; }
print 'Value after the loop: ' . $var . "\n";
```

Value of $var after loop is “foo”. It is the same as before loop! WTF?

Foreach var localization - continued

The same example but with $_

```
$_ = 'foo';
print;
foreach (@array) {
        print;
}
print;
```

And again, the $_ is the same as before loop!

This behavior can be observed in "foreach" loops, but not in "while" loops. So be careful.

So, you would like to create two dimensional array?

```
@array = ( (1,2,3),
           (4,5,6),
           (7,8,9));
```

Wrong. It makes @aray = (1,2,3,4,5,6,7,8,9);.
This is caled array flattening.

```
@array = ( [1,2,3],
           [4,5,6],
           [7,8,9]);
```

You do not get exactly array of arrays - you get the array of references to array.

Autodefining var

```
my $var;   # $var is not defined
if (defined $var) { warn "yes"; } else {  warn "no"; }
# warns: "no"
if (defined $var->{'foo'}) {  warn "yes"; } else
{ warn "no"; }
# well, it is not defined. But
  "if (defined $var->{'foo'})" made our $var defined!
if (defined $var) {  warn "yes"; } else { warn "no"; }
# warns: "yes"!
print ref $var; # HASH!
```

Analogically, if ($var->[0]) makes an empty arrayref.

Fun with counting months, years and weekdays

($sec,$min,$hour,$mday,$mon,$year,$wday, $yday,$isdst) = localtime();

Month day: from **1** to **31**
Month: **0** to **11.** WTF?

Explanation: This makes it easy to get a month name from a list: my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
print "$abbr[$mon] $mday";

Fun with counting months, years and weekdays

$year is the number of years since 1900, not just the last two digits of the year. That is, **$year** is 123 in year 2023.

We have seen using last two digits for year, and Y2K problems it made. We have seen counting time from 1970. We have seen counting time from begining of A.D. (like: 2009). So why 1900? It is strange, but it is the same in C and Java. Perl inherited it from C libs. It neither have the possiblility to store year in two digits, nor the possiblility not force programmers to count year in special way.

$year += 1900;

Fun with counting months, years and weekdays

$wday is the day of the week, with 0 indicating Sunday and 3 indicating Wednesday.

Monday is 1st, Saturday is 5th, Sunday is zeroth. Beware that Sunday is not 7th, nor 1st, as most people would thought. This solution is good for both groups of people - those that think that Monday is first day of weeks (as it is 1st) and those that Sunday is at the beginning of the week (as it is zeroth) ;-)

Just be aware of those little traps in localtime.

Three ways of calling subroutine

sub my_subroutine {...

- **my_subroutine;** The subroutine need to be declared earier
- **my_subroutine();** No need to declare earlier
- **&my_subroutine;** No need to declare earlier, too, but it passes the content of @_ to called subroutine! If you call subroutine with the "&" at beginning, you get an implicit argument list passed that you probably did not intend.

last in function

Imagine you have a loop and call a function (sub) in it – your, or from some module. And someone by mistake left there **last**. It terminates your loop.

```
for … {
something…;
function();
something… that would not be executed…
};
```

For some people it is a WTF, for some it is very logical way, that it should work like.

The truth is out there

What is false?

- 0
- '' (empty string)
- '0' (string zero)
- undef
- () empty list

what about "0.0"? And "0E0"?
"0.0", "0E0", "0 but true","false","foo" are all true.
0.0 is false (number, not string)

The truth is out there

"0 but true" - self documenting WTF :)

It is true, but when treated as a number it is zero.

```
print "0 but true"  ? "true\n" : "false\n";
print "0 but true" + 0 ? "true\n" : "false\n";
```

First is true, second is false.

You can add it to number:

```
print "0 but true" + 7;
```

As most other strings:

```
print "foo" + 7;
```

Beware of strings starting with inf... nad nan...

Comparing apples to oranges

if ("apple" == "orange") { ....

True!

Beware of "==" and "eq" difference.
"==" is for numbers, "eq" for strings.

perl 5.10 and later: $scalar ~~ $scalar;
If both look like numbers, do "==", otherwise do "eq"

That's all folks!

Thank you.