



## Perl is Dead, Long Live Perl!

Renée Bäcker  
renee.baecker@smart-websolutions.de

I often hear that Perl is dead because a new version hasn't been released in a long time. Perl 5.8 came out in 2002, and although development of Perl 6 is moving along, we haven't given up on Perl 5.

The next release version, 5.10, is right around the corner and has many new features and improvements. Some of these new features come from the work on Perl 6, so you can use them in Perl 5 without waiting.

Rafael Garcia-Suarez, the pumpking for version 5.10 of Perl, is working on the experimental track with 5.9.x which will soon become the stable release, Perl 5.10. The features I present here are already in 5.9.4 and ready to use.

### New features

In order to be able to use the new features, the Perl developers added the `feature` pragma. All new features have a lexical scope so I can turn them on or off how as I please.

#### defined-or, //

I often see code to set values for a variable, making sure that it gets some value if it doesn't already have one. This uses the `||` short-circuit operator to find a value:

```
my $bar = $var || $vb;

# or

$foo ||= 10;
```

These idioms are usually mean that if the variable doesn't not already have a value, use a default value. This is a trap though, and one I've fallen into myself. The variables `$var` or `$foo` might already

have a value, but that value might be one of Perl's false values: `0`, `'0'`, `''`, or `undef`. Those can be valid values, and what I really want to do is use the default value only when the variable is `undef`.

Perl 5.10 adds the “defined or” operator, written as the double forward slash: `//`. It only short-circuits when the value on the left is defined, even though it might be false. This expression:

```
$var // $vb;
```

is the same as using the ternary operator:

```
defined $var ? $var : $vb
```

I can now modify my previous example to only replace undefined values:

```
my $bar = $var // $vb;
$foo ||= 10;
```

In Perl there are also equivalent operators with lower precedence for such operators. For `||` it is `or`, and for `//` it is `err`. Since `err` is a new keyword, I have to enable it with the `feature` pragma:

```
use feature qw(err);
my $bar = ($var err $another_var);
```

Since `err` has a low priority, I would need to use the parentheses in this case, just like I need them with `or`.

#### say

The `say` function saves me from the annoying missing newline problem. It adds a newline to my output, so it's effectively a wrapper around `print`:

```
sub say {
    print @_, "\n";
}
```