NAME

perllol - Manipulating Arrays of Arrays in Perl

DESCRIPTION

Declaration and Access of Arrays of Arrays

The simplest two-level data structure to build in Perl is an array of arrays, sometimes casually called a list of lists. It's reasonably easy to understand, and almost everything that applies here will also be applicable later on with the fancier data structures.

An array of an array is just a regular old array @AoA that you can get at with two subscripts, like `$AoA[3][2]`. Here's a declaration of the array:

```
    use 5.010;  # so we can use say()

    # assign to our array, an array of array references
    @AoA = (
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "george", "jane", "elroy", "judy", ],
    [ "homer", "bart", "marge", "maggie", ],
    );
    say $AoA[2][1];
  bart
```

Now you should be very careful that the outer bracket type is a round one, that is, a parenthesis. That's because you're assigning to an @array, so you need parentheses. If you wanted there *not* to be an @AoA, but rather just a reference to it, you could do something more like this:

```
    # assign a reference to array of array references
    $ref_to_AoA = [
 [ "fred", "barney", "pebbles", "bambam", "dino", ],
 [ "george", "jane", "elroy", "judy", ],
 [ "homer", "bart", "marge", "maggie", ],
    ];
    say $ref_to_AoA->[2][1];
  bart
```

Notice that the outer bracket type has changed, and so our access syntax has also changed. That's because unlike C, in perl you can't freely interchange arrays and references thereto. $ref_to_AoA is a reference to an array, whereas @AoA is an array proper. Likewise, `$AoA[2]` is not an array, but an array ref. So how come you can write these:

```
    $AoA[2][2]
    $ref_to_AoA->[2][2]
```

instead of having to write these:

```
    $AoA[2]->[2]
    $ref_to_AoA->[2]->[2]
```

Well, that's because the rule is that on adjacent brackets only (whether square or curly), you are free to omit the pointer dereferencing arrow. But you cannot do so for the very first one if it's a scalar containing a reference, which means that $ref_to_AoA always needs it.

Growing Your Own

That's all well and good for declaration of a fixed data structure, but what if you wanted to add new elements on the fly, or build it up entirely from scratch?

First, let's look at reading it in from a file. This is something like adding a row at a time. We'll assume that there's a flat file in which each line is a row and each word an element. If you're trying to develop an @AoA array containing all these, here's the right way to do that:

```
    while (<>) {
@tmp = split;
push @AoA, [ @tmp ];
    }
```

You might also have loaded that from a function:

```
    for $i ( 1 .. 10 ) {
$AoA[$i] = [ somefunc($i) ];
    }
```

Or you might have had a temporary variable sitting around with the array in it.

```
    for $i ( 1 .. 10 ) {
@tmp = somefunc($i);
$AoA[$i] = [ @tmp ];
    }
```

It's important you make sure to use the `[ ]` array reference constructor. That's because this wouldn't work:

```
    $AoA[$i] = @tmp;   # WRONG!
```

The reason that doesn't do what you want is because assigning a named array like that to a scalar is taking an array in scalar context, which means just counts the number of elements in @tmp.

If you are running under `use strict` (and if you aren't, why in the world aren't you?), you'll have to add some declarations to make it happy:

```
    use strict;
    my(@AoA, @tmp);
    while (<>) {
@tmp = split;
push @AoA, [ @tmp ];
    }
```

Of course, you don't need the temporary array to have a name at all:

```
    while (<>) {
push @AoA, [ split ];
    }
```

You also don't have to use push(). You could just make a direct assignment if you knew where you wanted to put it:

```
    my (@AoA, $i, $line);
    for $i ( 0 .. 10 ) {
$line = <>;
$AoA[$i] = [ split " ", $line ];
```

```
    }
```

or even just

```
    my (@AoA, $i);
    for $i ( 0 .. 10 ) {
$AoA[$i] = [ split " ", <> ];
    }
```

You should in general be leery of using functions that could potentially return lists in scalar context without explicitly stating such. This would be clearer to the casual reader:

```
    my (@AoA, $i);
    for $i ( 0 .. 10 ) {
$AoA[$i] = [ split " ", scalar(<>) ];
    }
```

If you wanted to have a $ref_to_AoA variable as a reference to an array, you'd have to do something like this:

```
    while (<>) {
push @$ref_to_AoA, [ split ];
    }
```

Now you can add new rows. What about adding new columns? If you're dealing with just matrices, it's often easiest to use simple assignment:

```
    for $x (1 .. 10) {
for $y (1 .. 10) {
    $AoA[$x][$y] = func($x, $y);
}
    }

    for $x ( 3, 7, 9 ) {
$AoA[$x][20] += func2($x);
    }
```

It doesn't matter whether those elements are already there or not: it'll gladly create them for you, setting intervening elements to `undef` as need be.

If you wanted just to append to a row, you'd have to do something a bit funnier looking:

```
    # add new columns to an existing row
    push @{ $AoA[0] }, "wilma", "betty";   # explicit deref
```

Prior to Perl 5.14, this wouldn't even compile:

```
    push $AoA[0], "wilma", "betty";        # implicit deref
```

How come? Because once upon a time, the argument to push() had to be a real array, not just a reference to one. That's no longer true. In fact, the line marked "implicit deref" above works just fine--in this instance--to do what the one that says explicit deref did.

The reason I said "in this instance" is because that *only* works because `$AoA[0]` already held an array reference. If you try that on an undefined variable, you'll take an exception. That's because the implicit derefererence will never autovivify an undefined variable the way `@{ }` always will:

```
    my $aref = undef;
    push $aref,  qw(some more values);  # WRONG!
    push @$aref, qw(a few more);        # ok
```

If you want to take advantage of this new implicit dereferencing behavior, go right ahead: it makes code easier on the eye and wrist. Just understand that older releases will choke on it during compilation. Whenever you make use of something that works only in some given release of Perl and later, but not earlier, you should place a prominent

```
    use v5.14;   # needed for implicit deref of array refs by array ops
```

directive at the top of the file that needs it. That way when somebody tries to run the new code under an old perl, rather than getting an error like

```
    Type of arg 1 to push must be array (not array element) at /tmp/a line
8, near ""betty";"
    Execution of /tmp/a aborted due to compilation errors.
```

they'll be politely informed that

```
    Perl v5.14.0 required--this is only v5.12.3, stopped at /tmp/a line 1.
    BEGIN failed--compilation aborted at /tmp/a line 1.
```

Access and Printing

Now it's time to print your data structure out. How are you going to do that? Well, if you want only one of the elements, it's trivial:

```
    print $AoA[0][0];
```

If you want to print the whole thing, though, you can't say

```
    print @AoA;  # WRONG
```

because you'll get just references listed, and perl will never automatically dereference things for you. Instead, you have to roll yourself a loop or two. This prints the whole structure, using the shell-style for() construct to loop across the outer set of subscripts.

```
    for $aref ( @AoA ) {
 say "\t [ @$aref ],";
    }
```

If you wanted to keep track of subscripts, you might do this:

```
    for $i ( 0 .. $#AoA ) {
 say "\t elt $i is [ @{$AoA[$i]} ],";
    }
```

or maybe even this. Notice the inner loop.

```
    for $i ( 0 .. $#AoA ) {
 for $j ( 0 .. $#{$AoA[$i]} ) {
     say "elt $i $j is $AoA[$i][$j]";
 }
    }
```

As you can see, it's getting a bit complicated. That's why sometimes is easier to take a temporary on your way through:

```
    for $i ( 0 .. $#AoA ) {
 $aref = $AoA[$i];
 for $j ( 0 .. $#{$aref} ) {
     say "elt $i $j is $AoA[$i][$j]";
 }
    }
```

Hmm... that's still a bit ugly. How about this:

```
    for $i ( 0 .. $#AoA ) {
 $aref = $AoA[$i];
 $n = @$aref - 1;
 for $j ( 0 .. $n ) {
     say "elt $i $j is $AoA[$i][$j]";
 }
    }
```

When you get tired of writing a custom print for your data structures, you might look at the standard *Dumpvalue* or *Data::Dumper* modules. The former is what the Perl debugger uses, while the latter generates parsable Perl code. For example:

```
    use v5.14;     # using the + prototype, new to v5.14

    sub show(+) {
 require Dumpvalue;
 state $prettily = new Dumpvalue::
       tick        => q("),
       compactDump => 1,  # comment these two lines out
       veryCompact => 1,  # if you want a bigger dump
   ;
 dumpValue $prettily @_;
    }

    # Assign a list of array references to an array.
    my @AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
    );
    push $AoA[0], "wilma", "betty";
    show @AoA;
```

will print out:

```
    0  0..3  "fred" "barney" "wilma" "betty"
    1  0..2  "george" "jane" "elroy"
    2  0..2  "homer" "marge" "bart"
```

Whereas if you comment out the two lines I said you might wish to, then it shows it to you this way instead:

```
    0  ARRAY(0x8031d0)
       0  "fred"
```

```
        1  "barney"
        2  "wilma"
        3  "betty"
    1   ARRAY(0x803d40)
        0  "george"
        1  "jane"
        2  "elroy"
    2   ARRAY(0x803e10)
        0  "homer"
        1  "marge"
        2  "bart"
```

Slices

If you want to get at a slice (part of a row) in a multidimensional array, you're going to have to do some fancy subscripting. That's because while we have a nice synonym for single elements via the pointer arrow for dereferencing, no such convenience exists for slices.

Here's how to do one operation using a loop. We'll assume an @AoA variable as before.

```
    @part = ();
    $x = 4;
    for ($y = 7; $y < 13; $y++) {
 push @part, $AoA[$x][$y];
    }
```

That same loop could be replaced with a slice operation:

```
    @part = @{$AoA[4]}[7..12];
```

or spaced out a bit:

```
    @part = @{ $AoA[4] } [ 7..12 ];
```

But as you might well imagine, this can get pretty rough on the reader.

Ah, but what if you wanted a *two-dimensional slice*, such as having $x run from 4..8 and $y run from 7 to 12? Hmm... here's the simple way:

```
    @newAoA = ();
    for ($startx = $x = 4; $x <= 8; $x++) {
 for ($starty = $y = 7; $y <= 12; $y++) {
     $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
 }
    }
```

We can reduce some of the looping through slices

```
    for ($x = 4; $x <= 8; $x++) {
 push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
    }
```

If you were into Schwartzian Transforms, you would probably have selected map for that

```
    @newAoA = map { [ @{ $AoA[$_] } [ 7..12 ] ] } 4 .. 8;
```

Although if your manager accused you of seeking job security (or rapid insecurity) through inscrutable

code, it would be hard to argue. :-) If I were you, I'd put that in a function:

```
    @newAoA = splice_2D( \@AoA, 4 => 8, 7 => 12 );
    sub splice_2D {
 my $lrr = shift;  # ref to array of array refs!
 my ($x_lo, $x_hi,
     $y_lo, $y_hi) = @_;

 return map {
     [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
 } $x_lo .. $x_hi;
    }
```

SEE ALSO

perldata, *perlref*, *perldsc*

AUTHOR

Tom Christiansen <*tchrist@perl.com*>

Last update: Tue Apr 26 18:30:55 MDT 2011