The Perl
Data Structures Cookbook:
Lists of Lists

by Tom Christiansen
< tchrist@perl.com >

release 0.1
Sunday, 8 October 1995

Declaration and Access

The simplest thing to build is a list of lists (sometimes called an array of arrays). It's reasonably easy to understand, and almost everything that applies here will also be applicable later on with the fancier data structures.

A list of lists, or an array of an array if you would, is just a regular old array **@LoL** that you can get at with two subscripts, like **$LoL**[3][2]. Here's a declaration of the array:

```
    # assign to our array a list of list references
    @LoL = (
           [ "fred", "barney" ],
           [ "george", "jane", "elroy" ],
           [ "homer", "marge", "bart" ],
    );
    print $LoL[2][2];
  bart
```

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

```
    # assign a reference to list of list references
    $ref_to_LoL = [
        [ "fred", "barney", "pebbles", "bambam", "dino", ],
        [ "homer", "bart", "marge", "maggie", ],
        [ "george", "jane", "alroy", "judy", ],
    ];
    print $ref_to_LoL->[2][2];
```

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. **$ref_to_LoL** is a reference to an array, whereas **@LoL** is an array proper. Likewise, **$LoL**[2] is not an array, but an array ref. So how come you can write these:

```
$LoL[2][2]
$ref_to_LoL->[2][2]
```

instead of having to write these:

```
$LoL[2]->[2]
$ref_to_LoL->[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 array. But you need not do so for the very first one if it's a scalar containing a reference, which means that **$ref_to_LoL** 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 **@LoL** list containing all these, here's the right way to do that:

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

You might also have loaded that from a function:

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

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

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

It's very important that you make sure to use the `[]` list reference constructor. That's because this will be very wrong:

```
$LoL[$i] = @tmp;
```

You see, assigning a named list like that to a scalar just counts the number of elements in **@tmp**, which probably isn't what you want.

If you are running under `use strict`, you'll have to add some declarations to make it happy:

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

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

```
while (<>) {
    push @LoL, [ 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 (@LoL, $i, $line);
for $i ( 0 .. 10 )
    $line = <>;
    $LoL[$i] = [ split ' ', $line ];
}
```

or even just

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

You should in general be leary of using potential list functions in a scalar context without explicitly stating such. This would be clearer to the casual reader:

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

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

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

Actually, if you were using strict, you'd not only have to declare **$ref_to_LoL** as you had to declare **@LoL**, but you'd *also* having to initialize it to a reference to an empty list. (This was a bug in 5.001m that's been fixed for the 5.002 release.)

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

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

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        $LoL[$x][$y] = func($x, $y);
    }
}
for $x ( 3, 7, 9 ) {
    $LoL[$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 just wanted to append to a row, you'd have to do something a bit funnier looking:

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

Notice that I *couldn't* just say:

```
push $LoL[0], "wilma", "betty";  # WRONG!
```

In fact, that wouldn't even compile. How come? Because the argument to *push()* must be a real array, not just a reference to such.

Access and Printing

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

```
print $LoL[0][0];
```

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

```
print @LoL;         # WRONG
```

because you'll just get 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 $aref ( @LoL ) {
    print "\t [ @$aref ],\n";
}
```

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

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

or maybe even this. Notice the inner loop.

```
for $i ( 0 .. $#LoL ) {
    for $j ( 0 .. $#{$LoL[$i]} ) {
        print "elt $i $j is $LoL[$i][$j]\n";
    }
}
```

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 .. $#LoL ) {
    $aref = $LoL[$i];
    for $j ( 0 .. $#{$aref} ) {
        print "elt $i $j is $LoL[$i][$j]\n";
    }
}
```

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

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

Slices

If you want to get at a slide (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. (Remember, of course, that you can always write a loop to do a slice operation.)

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

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

That same loop could be replaced with a slice operation:

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

but as you might well imagine, this is 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? Hm... here's the simple way:

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

We can reduce some of the looping through slices

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

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

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

Although if your manager accused 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:

```
@newLoL = splice_2D( \@LoL, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;        # ref to list of list refs!
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;
    return map {
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

Passing Arguments

One place where a list of lists crops up is when you pass in several list references to a function. Consider:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );
sub popmany {
    my $aref;
    my @retlist = ();
    foreach $aref ( @_ ) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

This function was designed to pop off the last element from each of its arguments and return those in a list. In this function, you can think of @_ as a list of lists.

Just as a side note, what happens if the function is called with the "wrong" types of arguments? Normally nothing, but in the case of references, we can be a bit pickier. This isn't detectable at compile-time (yet--Larry does have a prototype prototype in the works for 5.002), but you could check it at run time using the *ref()* function.

```
use Carp;
for $i ( 0 .. $#_) {
    if (ref($_[$i]) ne 'ARRAY') {
        confess "popmany: arg $i not an array reference\n";
    }
}
```

However, that's not usually necessary unless you want to trap it. It's also dubious in that it would fail on a real array references blessed into its own class (an object). But since you're all going to be using `strict refs`, it would raise an exception anyway even without the die.

This will matter more to you later on when you start building up more complex data structures that all aren't woven of the same cloth, so to speak.

SEE ALSO

- perldsc(1)
- perlref(1)
- perldata(1)

AUTHOR

Tom Christiansen <tchrist@perl.com>
Last update: Sun Oct 8 23:18:56 MDT 1995