The Perl Data Structures Cookbook: General Tips

by Tom Christiansen
< tchrist@perl.com >

release 0.1
Sunday, 8 October 1995

The single feature most sorely lacking in the Perl programming language prior to its 5.0 release was complex data structures. Even without direct language support, some valiant programmers did manage to emulate them, but it was hard work and not for the faint of heart. You could occasionally get away with the `$m{$a,$b}` notation borrowed from *awk* in which the keys are actually more like a single concatenated string `"$a$b"`, but traversal and sorting were difficult. More desperate programmers even hacked Perl's internal symbol table directly, a strategy that proved hard to develop and maintain--to put it mildly.

The 5.0 release of Perl let us have complex data structures. You may now write something like this and all of a sudden, you'd have a array with three dimensions!

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $z (1 .. 10) {
            $LoL[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct that meets the eye!

How to you print it out? Why can't you just say `print @LoL`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is is an object? Can you save it to disk to read back later? How do you access whole rows or columns of that matrix? Do all the values have to be numeric?

As you see, it's quite easy to become confused. While some small portion of the blame for this can be attributed to the reference-based implementation, it's really more due to a lack of existing documentation with examples designed for the beginner.

This document is meant to be a detailed but understandable treatment of the many different sorts of data structures you might want to develop. It should also serve as a cookbook of examples. That way, when you need to create one of these complex data structures, you can just pinch, pilfer, or purloin a drop-in example from here.

Let's look at each of these possible constructs in detail. There are separate documents on

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes
- more elaborate constructs
- recursive and self-referential data structures
- objects

But for now, let's look at some of the general issues common to all.of these types a data structures.

References

The most important thing to understand about all data structures in Perl -- including multidimensional arrays--is that even though they might appear otherwise, Perl `@ARRAYS` and `%HASH`es are all internally one-dimensional. They can only hold scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain *references* to other arrays or hashes.

You can't use a reference to a array or hash in quite the same way that you would a real array or hash. For C or C++ programmers unused to distinguishing between arrays and pointers to the same, this can be confusing. If so, just think of it as the difference between a structure and a pointer to a structure.

You can (and should) read more about references in the perlref(1) manpage. Briefly, references are rather like pointers that know what they point to. (Objects are also a kind of reference, but we won't be needing them right away--if ever.) That means that when you have something that looks to you like an access to two-or-more-dimensional array and/or hash, that what's really going on is that in all these cases, the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can *use* it as though it were a two-dimensional one. This is actually the way almost all C multidimensional arrays work as well.

```
    $list[7][12]                        # array of arrays
    $list[7]{string}                    # array of hashes
    $hash{string}[7]                    # hash of arrays
    $hash{string}{'another string'}     # hash of hashes
```

Now, because the top level only contains references, if you try to print out your array in with a simple *print()* function, you'll get something that doesn't look very nice, like this:

```
    @LoL = ( [2, 3], [4, 5, 7], [0] );
    print $LoL[1][2];
  7
    print @LoL;
  ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

That's because Perl doesn't (ever) implicitly dereference your variables. If you want to get at the thing a reference is referring to, then you have to do this yourself using either prefix typing indicators, like `${$blah}`, `@{$blah}`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

Common Mistakes

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly. Here's the case where you just get the count instead of a nested array:

```
for $i (1..10) {
    @list = somefunc($i);
    $LoL[$i] = @list;       # WRONG!
}
```

That's just the simple case of assigning a list to a scalar and getting its element count. If that's what you really and truly want, then you might do well to consider being a tad more explicit about it, like this:

```
for $i (1..10) {
    @list = somefunc($i);
    $counts[$i] = scalar @list;
}
```

Here's the case of taking a reference to the same memory location again and again:

```
for $i (1..10) {
    @list = somefunc($i);
    $LoL[$i] = \@list;      # WRONG!
}
```

So, just what's the big problem with that? It looks right, doesn't it? After all, I just told you that you need an array of references, so by golly, you've made me one!

Unfortunately, while this is true, it's still broken. All the references in `@LoL` refer to the *very same place*, and they will therefore all hold whatever was last in `@list`! It's similar to the problem demonstrated in the following C program:

```
#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
    rp = getpwnam("root");
    dp = getpwnam("daemon");
    printf("daemon name is %s\nroot name is %s\n",
            dp->pw_name, rp->pw_name);
}
```

Which will print

```
daemon name is daemon
root name is daemon
```

The problem is that both `rp` and `dp` are pointers to the same location in memory! In C, you'd have to remember to *malloc()* yourself some new memory. In Perl, you'll want to use the array constructor **[]** or the hash constructor **{}** instead. Here's the right way to do the preceding broken code fragments

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

The square brackets make a reference to a new array with a *copy* of what's in `@list` at the time of the assignment. This is what you want.

Note that this will produce something similar, but it's much harder to read:

```
for $i (1..10) {
    @list = 0 .. $i;
    @{ $LoL[$i] } = @list;
}
```

Is it the same? Well, maybe so--and maybe not. The subtle difference is that when you assign something in square brackets, you know for sure it's always a brand new reference with a new *copy* of the data. Something else could be going on in this new case with the `@{$LoL[$i]}}` dereference on the left-hand-side of the assignment. It all depends on whether `$LoL[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated `@LoL` with references, as in

```
$LoL[3] = \@another_list;
```

Then the assignment with the indirection on the left-hand-side would use the existing reference that was already there:

```
@{ $LoL[3] } = @list;
```

Of course, this *would* have the ‘‘interesting’’ effect of clobbering `@another_list`. (Have you ever noticed how when a programmer says something is ‘‘interesting’’, that rather than meaning ‘‘intriguing’’, they're disturbingly more apt to mean that it's ‘‘annoying’’, ‘‘difficult’’, or both? :-)

So just remember to always use the array or hash constructors with **[]** or **{}**, and you'll be fine, although it's not always optimally efficient.

Surprisingly, the following dangerous-looking construct will actually work out fine:

```
for $i (1..10) {
    my @list = somefunc($i);
    $LoL[$i] = \@list;
```

```
    }
```

That's because `my()` is more of a run-time statement than it is a compile-time declaration *per se*. This means that the `my()` variable is remade afresh each time through the loop. So even though it *looks* as though you stored the same variable reference each time, you actually did not! This is a subtle distinction that can produce more efficient code at the risk of misleading all but the most experienced of programmers. So I usually advise against teaching it to beginners. In fact, except for passing arguments to functions, I seldom like to see the gimme-a-reference operator (backslash) used much at all in code. Instead, I advise beginners that they (and most of the rest of us) should try to use the much more easily understood constructors **[]** and **{}** instead of relying upon lexical (or dynamic) scoping and hidden reference-counting to do the right thing behind the scenes.

In summary:

```
    $LoL[$i] = [ @list ];       # usually best
    $LoL[$i] = \@list;          # perilous; just how my() was that list?
    @{ $LoL[$i] } = @list;      # way too tricky for most programmers
```

Caveat on Precedence

Speaking of things like `@{$LoL[$i]}`, the following are actually the same thing:

```
    $listref->[2][2]
    $$listref[2][2]
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: **`$ @ * % &`**) make them bind more tightly than the postfix subscripting brackets or braces! This will no doubt come as a great shock to the C or C++ programmer, who is quite accustomed to using `*a[i]` to mean what's pointed to by the *i'th* element of `a`. That is, they first take the subscript, and only then dereference the thing at that subscript. That's fine in C, but this isn't C.

The seemingly equivalent construct in Perl, `$$listref[$i]` first does the deref of `$listref`, making it take `$listref` as a reference to an array, and then dereference that, and finally tell you the *i'th* value of the array pointed to by `$LoL`. If you wanted the C notion, you'd have to write `${$LoL[$i]}` to force the `$LoL[$i]` to get evaluated first before the leading `$` dereferencer.

Why You Should Always `use strict`

If this is starting to sound scarier than it's worth, relax. Perl has some features to help you out avoid its most common pitfalls. The best way to avoid getting confused is to start every program like this:

```
    #!/usr/bin/perl -w
    use strict;
```

This way, you'll be forced to declare all your variables with `my()` and also disallow accidental ‘‘symbolic dereferencing’’. Therefore if you'd done this:

```
my $listref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "alroy", "judy", ],
];
print $listref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@listref`, an undeclared variable, and it would thereby remind you to instead write:

```
print $listref->[2][2]
```

Debugging

The standard Perl debugger in 5.001 doesn't do a very nice job of printing out complex data structures. However, the perl5db that Ilya Zakharevich <ilya@math.ohio-state.edu> wrote, which is accessible at ftp::/ftp.perl.com/pub/perl/ext/perl5db-kit-0.9.tar.gz.

The version for the debugger has several important new features, including command line editing as well as the `x` and `X` commands to dump out complex data structures. For example, given the assignment to `$LoL` above, here's the debugger output:

```
DB<1> X $LoL
$LoL = ARRAY(0x13b5a0)
   0  ARRAY(0x1f0a24)
      0  'fred'
      1  'barney'
      2  'pebbles'
      3  'bambam'
      4  'dino'
   1  ARRAY(0x13b558)
      0  'homer'
      1  'bart'
      2  'marge'
      3  'maggie'
   2  ARRAY(0x13b540)
      0  'george'
      1  'jane'
      2  'alroy'
      3  'judy'
```

There's also a lower-case `x` command which is nearly the same.

SEE ALSO

perlref(1), perldata(1)

AUTHOR

Tom Christiansen <tchrist@perl.com>
Last update: Sat Oct 7 22:41:09 MDT 1995