NAME

perldsc - Perl Data Structures Cookbook

DESCRIPTION

Perl lets us have complex data structures. You can write something like this and all of a sudden, you'd have an array with three dimensions!

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

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

How do you print it out? Why can't you say just `print @AoA`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is it 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 sections on each of the following:

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes
- more elaborate constructs

But for now, let's look at general issues common to all these types of 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 %HASHes are all internally one-dimensional. They can hold only 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 an 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 perlref. 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.) This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that 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.

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

Now, because the top level contains only 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:

```
my @AoA = ( [2, 3], [4, 5, 7], [0] );
print $AoA[1][2];
7
print @AoA;
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 my $i (1..10) {
my @array = somefunc($i);
$AoA[$i] = @array; # WRONG!
}
```

That's just the simple case of assigning an array 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 my $i (1..10) {
my @array = somefunc($i);
$counts[$i] = scalar @array;
}
```

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

```
# Either without strict or having an outer-scope my @array;
# declaration.

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

So, 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 `@AoA` refer to the *very same place*, and they will therefore all hold whatever was last in `@array`! 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:

```
# Either without strict or having an outer-scope my @array;
# declaration.

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

The square brackets make a reference to a new array with a *copy* of what's in `@array` 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:

```
# Either without strict or having an outer-scope my @array;
# declaration.
for my $i (1..10) {
@array = 0 .. $i;
@{$AoA[$i]} = @array;
}
```

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 `@{$AoA[$i]}` dereference on the left-hand-side of the assignment. It all depends on whether `$AoA[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated `@AoA` with references, as in

```
$AoA[3] = \@another_array;
```

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

```
@{$AoA[3]} = @array;
```

Of course, this *would* have the ''interesting'' effect of clobbering `@another_array`. (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 always to 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 my $i (1..10) {
my @array = somefunc($i);
$AoA[$i] = \@array;
}
```

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:

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

CAVEAT ON PRECEDENCE

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

```
$aref->[2][2] # clear
$$aref[2][2] # confusing
```

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, `$$aref[$i]` first does the deref of `$aref`, making it take `$aref` as a reference to an array, and then dereference that, and finally tell you the *i'th* value of the array pointed to by `$AoA`. If you wanted the C notion, you'd have to write `${$AoA[$i]}` to force the `$AoA[$i]` to get evaluated first before the leading `$` 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 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 $aref = [
[ "fred", "barney", "pebbles", "bambam", "dino", ],
[ "homer", "bart", "marge", "maggie", ],
[ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

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

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

DEBUGGING

You can use the debugger's `x` command to dump out complex data structures. For example, given the assignment to `$AoA` above, here's the debugger output:

```
DB<1> x $AoA
$AoA = 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 'elroy'
3 'judy'
```

CODE EXAMPLES

Presented with little comment (these will get their own manpages someday) here are short code examples illustrating access of various types of data structures.

ARRAYS OF ARRAYS

Declaration of an ARRAY OF ARRAYS

```
@AoA = (
[ "fred", "barney" ],
[ "george", "jane", "elroy" ],
[ "homer", "marge", "bart" ],
);
```

Generation of an ARRAY OF ARRAYS

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

# calling a function
for $i ( 1 .. 10 ) {
$AoA[$i] = [ somefunc($i) ];
}

# using temp vars
for $i ( 1 .. 10 ) {
@tmp = somefunc($i);
$AoA[$i] = [ @tmp ];
}

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

Access and Printing of an ARRAY OF ARRAYS

```
# one element
$AoA[0][0] = "Fred";

# another element
$AoA[1][1] =~ s/(\w)/\u$1/;
```

```
# print the whole thing with refs
for $aref ( @AoA ) {
print "\t [ @$aref ],\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoA ) {
print "\t [ @{$AoA[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoA ) {
for $j ( 0 .. $#{ $AoA[$i] } ) {
print "elt $i $j is $AoA[$i][$j]\n";
}
}
```

HASHES OF ARRAYS

Declaration of a HASH OF ARRAYS

```
%HoA = (
flintstones => [ "fred", "barney" ],
jetsons => [ "george", "jane", "elroy" ],
simpsons => [ "homer", "marge", "bart" ],
);
```

Generation of a HASH OF ARRAYS

```
# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
next unless s/^(.*?):\s*//;
$HoA{$1} = [ split ];
}

# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
($who, $rest) = split /:\s*/, $line, 2;
@fields = split ' ', $rest;
$HoA{$who} = [ @fields ];
}

# calling a function that returns a list
for $group ( "simpsons", "jetsons", "flintstones" ) {
$HoA{$group} = [ get_family($group) ];
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
@members = get_family($group);
$HoA{$group} = [ @members ];
}

# append new members to an existing family
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

Access and Printing of a HASH OF ARRAYS

```
# one element
$HoA{flintstones}[0] = "Fred";

# another element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoA ) {
print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoA ) {
print "family: ";
foreach $i ( 0 .. $#{ $HoA{$family} } ) {
print " $i = $HoA{$family}[$i]";
}
print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing sorted by number of members and name
foreach $family ( sort {
@{$HoA{$b}} <=> @{$HoA{$a}}
||
$a cmp $b
} keys %HoA )
{
print "$family: ", join(", ", sort @{ $HoA{$family} }), "\n";
}
```

ARRAYS OF HASHES

Declaration of an ARRAY OF HASHES

```
@AoH = (
{
Lead => "fred",
Friend => "barney",
},
{
Lead => "george",
Wife => "jane",
Son => "elroy",
},
{
Lead => "homer",
Wife => "marge",
Son => "bart",
}
);
```

Generation of an ARRAY OF HASHES

```
# reading from file
# format: LEAD=fred FRIEND=barney
while ( <> ) {
$rec = {};
for $field ( split ) {
($key, $value) = split /=/, $field;
$rec->{$key} = $value;
}
push @AoH, $rec;
}


# reading from file
# format: LEAD=fred FRIEND=barney
# no temp
while ( <> ) {
push @AoH, { split /[\s+=]/ };
}

# calling a function that returns a key/value pair list, like
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
push @AoH, { %fields };
}

# likewise, but using no temp vars
while (<>) {
push @AoH, { parsepairs($_) };
}

# add key/value to an element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";
```

Access and Printing of an ARRAY OF HASHES

```
# one element
$AoH[0]{lead} = "fred";

# another element
$AoH[1]{lead} =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @AoH ) {
print "{ ";
for $role ( keys %$href ) {
print "$role=$href->{$role} ";
}
print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoH ) {
print "$i is { ";
for $role ( keys %{ $AoH[$i] } ) {
```

```
print "$role=$AoH[$i]{$role} ";
}
print "}\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoH ) {
for $role ( keys %{ $AoH[$i] } ) {
print "elt $i $role is $AoH[$i]{$role}\n";
}
}
```

HASHES OF HASHES

Declaration of a HASH OF HASHES

```
%HoH = (
flintstones => {
lead => "fred",
pal => "barney",
},
jetsons => {
lead => "george",
wife => "jane",
"his boy" => "elroy",
},
simpsons => {
lead => "homer",
wife => "marge",
kid => "bart",
},
);
```

Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
next unless s/^(.*?):\s*//;
$who = $1;
for $field ( split ) {
($key, $value) = split /=/, $field;
$HoH{$who}{$key} = $value;
}


# reading from file; more temps
while ( <> ) {
next unless s/^(.*?):\s*//;
$who = $1;
$rec = {};
$HoH{$who} = $rec;
for $field ( split ) {
($key, $value) = split /=/, $field;
$rec->{$key} = $value;
}
}

# calling a function that returns a key,value hash
```

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
$HoH{$group} = { get_family($group) };
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
%members = get_family($group);
$HoH{$group} = { %members };
}

# append new members to an existing family
%new_folks = (
wife => "wilma",
pet => "dino",
);

for $what (keys %new_folks) {
$HoH{flintstones}{$what} = $new_folks{$what};
}
```

Access and Printing of a HASH OF HASHES

```
# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
print "$family: { ";
for $role ( keys %{ $HoH{$family} } ) {
print "$role=$HoH{$family}{$role} ";
}
print "}\n";
}

# print the whole thing somewhat sorted
foreach $family ( sort keys %HoH ) {
print "$family: { ";
for $role ( sort keys %{ $HoH{$family} } ) {
print "$role=$HoH{$family}{$role} ";
}
print "}\n";
}


# print the whole thing sorted by number of members
foreach $family ( sort { keys %{$HoH{$b}} <=> keys %{$HoH{$a}} }
keys %HoH )
{
print "$family: { ";
for $role ( sort keys %{ $HoH{$family} } ) {
print "$role=$HoH{$family}{$role} ";
}
print "}\n";
```

```
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } }
keys %HoH )
{
print "$family: { ";
# and print these according to rank order
for $role ( sort { $rank{$a} <=> $rank{$b} }
keys %{ $HoH{$family} } )
{
print "$role=$HoH{$family}{$role} ";
}
print "}\n";
}
```

MORE ELABORATE RECORDS

Declaration of MORE ELABORATE RECORDS

Here’s a sample showing how to create and use a record whose fields are of many different sorts:

```
$rec = {
TEXT => $string,
SEQUENCE => [ @old_values ],
LOOKUP => { %some_table },
THATCODE => \&some_function,
THISCODE => sub { $_[0] ** $_[1] },
HANDLE => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");
```

Declaration of a HASH OF COMPLEX RECORDS

```
%TV = (
flintstones => {
series => "flintstones",
nights => [ qw(monday thursday friday) ],
members => [
{ name => "fred", role => "lead", age => 36, },
{ name => "wilma", role => "wife", age => 31, },
{ name => "pebbles", role => "kid", age => 4, },
],
},

jetsons => {
series => "jetsons",
nights => [ qw(wednesday saturday) ],
members => [
{ name => "george", role => "lead", age => 41, },
{ name => "jane", role => "wife", age => 39, },
{ name => "elroy", role => "kid", age => 9, },
],
},

simpsons => {
series => "simpsons",
nights => [ qw(monday) ],
members => [
{ name => "homer", role => "lead", age => 34, },
{ name => "marge", role => "wife", age => 37, },
{ name => "bart", role => "kid", age => 11, },
],
},
);
```

Generation of a HASH OF COMPLEX RECORDS

```
# reading from file
# this is most easily done by having the file itself be
# in the raw data format as shown above. perl is happy
# to parse complex data structures if declared as data, so
# sometimes it's easiest to do that

# here's a piece by piece build up
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# assume this file in field=value syntax
while (<>) {
%fields = split /[\s=]+/;
push @members, { %fields };
}
$rec->{members} = [ @members ];

# now remember the whole thing
$TV{ $rec->{series} } = $rec;
```

```
############################################################
# now, you might want to make interesting extra fields that
# include pointers back into the same data structure so if
# change one piece, it changes everywhere, like for example
# if you wanted a {kids} field that was a reference
# to an array of the kids' records without having duplicate
# records and thus update problems.
############################################################
foreach $family (keys %TV) {
$rec = $TV{$family}; # temp pointer
@kids = ();
for $person ( @{ $rec->{members} } ) {
if ($person->{role} =~ /kid|son|daughter/) {
push @kids, $person;
}
}
# REMEMBER: $rec and $TV{$family} point to same data!!
$rec->{kids} = [ @kids ];
}

# you copied the array, but the array itself contains pointers
# to uncopied objects. this means that if you make bart get
# older via

$TV{simpsons}{kids}[0]{age}++;

# then this would also change in
print $TV{simpsons}{members}[2]{age};

# because $TV{simpsons}{kids}[0] and $TV{simpsons}{members}[2]
# both point to the same underlying anonymous hash table

# print the whole thing
foreach $family ( keys %TV ) {
print "the $family";
print " is on during @{ $TV{$family}{nights} }\n";
print "its members are:\n";
for $who ( @{ $TV{$family}{members} } ) {
print " $who->{name} ($who->{role}), age $who->{age}\n";
}
print "it turns out that $TV{$family}{lead} has ";
print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
print "\n";
}
```

Database Ties

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does partially attempt to address this need is the MLDBM module. Check your nearest CPAN site as described in perlmodlib(1) for source code to MLDBM.

SEE ALSO

perlref(1), perllol(1), perldata(1), perlobj(1)

AUTHOR

Tom Christiansen <*tchrist@perl.com*>