The Perl Data Structures Cookbook

by Tom Christiansen
< tchrist@perl.com >

release 0.1 (untested, may have typos)
Sunday, 1 October 1995

This is a cookbook of recipes for building up complex data structures in perl5. It has been extracted from a much larger and more expository document to be published in pod format and included with the standard perl distribution. The goal is to provide cookbook-like, cut-and-paste examples of the most often used data structures in perl. Think of the recipes as a quick reference

It currently has 6 parts:

- PDSC General Tips
- PDSC recipes: Lists of Lists
- PDSC recipes: Hashes of Lists
- PDSC recipes: Lists of Hashes
- PDSC recipes: Hashes of Hashes
- PDSC recipes: More Elaborate Structures

See also:

- Tom Christiansen's homepage at http://perl.com/.
- The Perl FAQ.

PDSC #0: General tips

The first thing you need to do is figure out how you want to access (such as via an assignment) just *one individual element* of your data structure just using lists and hashes. Use a list if you're thinking of an array, use a hash if you're thinking of a record or a lookup table.

- **$coordinates[$row][$col] = "empty"**
  This is a simple two-dimensionsal array indexed by integers. Each first level numeric index itself produces a list (reference). See the List of Lists document.

- **$flight_time{"denver"}[3] = "12:34"**
  This is an (associative) array of lists. Each first level string index itself produces a list (reference). See the Hash of Lists document.

- **$student[$i]{"age"} = 15**
  This is an array of records that include named fields. Each first level numeric index itself produces a hash (reference). See the List of Hashes document.

- **$tv_shows{"the simpsons"}{"start time"} = "Monday 20:00"**
  This is an lookup table of records where you lookup the show by the name, and then you look up the record field by the field name. Each first level string index itself produces a hash (reference). See the Hash of Hashes document.

- **$tv{"the simpsons"}{members}[0]{name} = "homer"**
  This is an elaborate data structure involving a mix of records that contain fields that are sometimes themselves other arrays and records. See the More Elaborate Structures document.

- **print {$rec->{FH}}**
  &{ $rec->{FUNC} } ($rec->{LIST}[0])
  This is a strange record that itself includes references to filehandles, functions, and other strings, lists, and hashes. We print to the filehandle referenced in **$rec**'s FH the result of calling the function in its FUNC field with an argument of the first element in the array which is its LIST field. See also the More Elaborate Structures document.

General Tips

Here are some further tips of general interest:

1. Always **use strict** and **-w**. The strict can be a pain, but it will save you from saying **$a[$i]** when you mean **$a->[$i]** and vice versa.

2. Things like **push()** require an @ sign, as in
   ```
   push @{ $a[3] }, @new_list
   ```
   You can't write
   ```
   push $a[3], @new_list
   ```

3. Things like **keys()** require a **%** sign, as in
   ```
   foreach $k (keys %{ $h{"key"} }) { ... }
   ```
   You can't write
   ```
   foreach $k (keys $h{"key"}) { ... }
   ```

4. Don't store pointers to existing data in a structure. Always create a new structure, eg. to build a 2D array indexed by line and by word number:

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

   This generally means never using the backslash to take a reference, but rather using the **[]** or **{}** constructors. This, for examples, is **wrong**!

```
while ( <> ) {
    @fields = split;
    push @a, \@fields;
}
```

An exception to this rule would be when you’re writing a recursive data structure or are creating multiple key indices for the same set of records.

5. Never write **$$a[$i]** when you mean **${$a[$i]}** or **@$a[$i]** when you mean **@{$a[$i]}**. Those won’t work at all.

6. Never write **$$a[$i]** even if you mean **$a->[$i]**. While it’ll work to do that, it will needlessly confuse C programmers, who will think that subscripting binds tighter than the prefix dereference operator. This is right in C but wrong in perl where it’s the other way around!

7. Remember that **$a[$i]** is the **i**’th elt of **@a**, but **$a->[$i]** is the **i**’th elt of the anon array pointed to by **$a**. use strict will help here.

8. Never write
   ```
   @ { $a[$i] } = @list
   ```
   instead of
   ```
   $a[$i] = [ @list ]
   ```
   It’ll work, but will confuse people.

9. Try to use pointer arrows and indirection bracketing whenever you feel the reader might be confused. Sometimes it’ll clear things up in your mind as well. Here are the five kinds of prefix dereferencers with disambiguating braces:

   - **print ${ $a[$i] }**
     Dereference a SCALAR reference.
   - **pop @{ $a[$i] }**
     Dereference an ARRAY reference.
   - **@k = keys %{ $a[$i] }**
     Dereference a HASH reference.
   - **$retval = &{ $a[$i] }( )**
     Dereference a CODE reference.
   - ***{ $a[$i] } = \&func**
     Dereference a GLOB reference; this is deep magic best left to the wizards.

10. The new perl5db (ftp to perl.com in /pub/perl/ext/) will help print out complex data structures using the **x** and **X** commands.

PDSC #1: Lists of Lists

Declaration of a LIST OF LISTS:

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

Generation of a LIST OF LISTS:

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

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

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

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

Access and Printing of a LIST OF LISTS:

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

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

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

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

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

PDSC #2: Hashes of Lists

Declaration of a HASH OF LISTS:

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

Generation of a HASH OF LISTS:

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

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

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

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

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

Access and Printing of a HASH OF LISTS:

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

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

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

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

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

PDSC #3: Lists of Hashes

Declaration of a LIST OF HASHES:

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

Generation of a LIST OF HASHES:

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

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

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

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

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

Access and Printing of a LIST OF HASHES:

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

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

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

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

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

PDSC #4: 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{$b}} } keys %HoH ) {
    print "$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{$b}} } keys %HoH ) {
    print "$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";
}
```

PDSC #5: More Elaborate Structures

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 = {
    STRING  => $string,
    LIST    => [ @old_values ],
    LOOKUP  => { %some_table },
    FUNC    => \&some_function,
    FANON   => sub { $_[0] ** $_[1] },
    FH      => \*STDOUT,
};

print $rec->{STRING};

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

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

$answer = &{ $rec->{FUNC} }($arg);
$answer = &{ $rec->{FANON} }($arg1, $arg2);

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

use FileHandle;
$rec->{FH}->autoflush(1);
```

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 datastructures 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 examples
# if you wanted a {kids} field that was an array reference
# to a list 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 list, but the list 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";
}
```