NAME

perlreftut - Mark's very short tutorial about references

DESCRIPTION

One of the most important new features in Perl 5 was the capability to manage complicated data structures like multidimensional arrays and nested hashes. To enable these, Perl 5 introduced a feature called `references', and using references is the key to managing complicated, structured data in Perl. Unfortunately, there's a lot of funny syntax to learn, and the main manual page can be hard to follow. The manual is quite complete, and sometimes people find that a problem, because it can be hard to tell what is important and what isn't.

Fortunately, you only need to know 10% of what's in the main page to get 90% of the benefit. This page will show you that 10%.

Who Needs Complicated Data Structures?

One problem that came up all the time in Perl 4 was how to represent a hash whose values were lists. Perl 4 had hashes, of course, but the values had to be scalars; they couldn't be lists.

Why would you want a hash of lists? Let's take a simple example: You have a file of city and country names, like this:

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

and you want to produce an output like this, with each country mentioned once, and then an alphabetical list of the cities in that country:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA:  Chicago, New York, Washington.
```

The natural way to do this is to have a hash whose keys are country names. Associated with each country name key is a list of the cities in that country. Each time you read a line of input, split it into a country and a city, look up the list of cities already known to be in that country, and append the new city to the list. When you're done reading the input, iterate over the hash as usual, sorting each list of cities before you print it out.

If hash values can't be lists, you lose. In Perl 4, hash values can't be lists; they can only be strings. You lose. You'd probably have to combine all the cities into a single string somehow, and then when time came to write the output, you'd have to break the string into a list, sort the list, and turn it back into a string. This is messy and error-prone. And it's frustrating, because Perl already has perfectly good lists that would solve the problem if only you could use them.

The Solution

By the time Perl 5 rolled around, we were already stuck with this design: Hash values must be scalars. The solution to this is references.

A reference is a scalar value that *refers to* an entire array or an entire hash (or to just about anything else). Names are one kind of reference that you're already familiar with. Think of the President of the United States: a messy, inconvenient bag of blood and bones. But to talk about him, or to represent him in a computer program, all you need is the easy, convenient scalar string "George Bush".

References in Perl are like names for arrays and hashes. They're Perl's private, internal names, so

you can be sure they're unambiguous. Unlike "George Bush", a reference only refers to one thing, and you always know what it refers to. If you have a reference to an array, you can recover the entire array from it. If you have a reference to a hash, you can recover the entire hash. But the reference is still an easy, compact scalar value.

You can't have a hash whose values are arrays; hash values can only be scalars. We're stuck with that. But a single reference can refer to an entire array, and references are scalars, so you can have a hash of references to arrays, and it'll act a lot like a hash of arrays, and it'll be just as useful as a hash of arrays.

We'll come back to this city-country problem later, after we've seen some syntax for managing references.

Syntax

There are just two ways to make a reference, and just two ways to use it once you have it.

Making References

Make Rule 1

If you put a `\` in front of a variable, you get a reference to that variable.

```
$aref = \@array;         # $aref now holds a reference to @array
$href = \%hash;          # $href now holds a reference to %hash
$sref = \$scalar;        # $sref now holds a reference to $scalar
```

Once the reference is stored in a variable like $aref or $href, you can copy it or store it just the same as any other scalar value:

```
$xy = $aref;             # $xy now holds a reference to @array
$p[3] = $href;           # $p[3] now holds a reference to %hash
$z = $p[3];              # $z now holds a reference to %hash
```

These examples show how to make references to variables with names. Sometimes you want to make an array or a hash that doesn't have a name. This is analogous to the way you like to be able to use the string `"\n"` or the number 80 without having to store it in a named variable first.

Make Rule 2

`[ ITEMS ]` makes a new, anonymous array, and returns a reference to that array. `{ ITEMS }` makes a new, anonymous hash, and returns a reference to that hash.

```
$aref = [ 1, "foo", undef, 13 ];
# $aref now holds a reference to an array


$href = { APR => 4, AUG => 8 };
# $href now holds a reference to a hash
```

The references you get from rule 2 are the same kind of references that you get from rule 1:

```
# This:
$aref = [ 1, 2, 3 ];


# Does the same as this:
@array = (1, 2, 3);
$aref = \@array;
```

The first line is an abbreviation for the following two lines, except that it doesn't create the superfluous

array variable `@array`.

If you write just `[]`, you get a new, empty anonymous array. If you write just `{}`, you get a new, empty anonymous hash.

Using References

What can you do with a reference once you have it? It's a scalar value, and we've seen that you can store it as a scalar and get it back again just like any scalar. There are just two more ways to use it:

Use Rule 1

You can always use an array reference, in curly braces, in place of the name of an array. For example, `@{$aref}` instead of `@array`.

Here are some examples of that:

Arrays:

```
@a  @{$aref}  An array
reverse @a reverse @{$aref} Reverse the array
$a[3]  ${$aref}[3]  An element of the array
$a[3] = 17; ${$aref}[3] = 17 Assigning an element
```

On each line are two expressions that do the same thing. The left-hand versions operate on the array `@a`. The right-hand versions operate on the array that is referred to by `$aref`. Once they find the array they're operating on, both versions do the same things to the arrays.

Using a hash reference is *exactly* the same:

```
%h  %{$href}       A hash
keys %h  keys %{$href}       Get the keys from the hash
$h{'red'} ${$href}{'red'}       An element of the hash
$h{'red'} = 17 ${$href}{'red'} = 17  Assigning an element
```

Whatever you want to do with a reference, **Use Rule 1** tells you how to do it. You just write the Perl code that you would have written for doing the same thing to a regular array or hash, and then replace the array or hash name with `{$reference}`. "How do I loop over an array when all I have is a reference?" Well, to loop over an array, you would write

```
for my $element (@array) {
   ...
}
```

so replace the array name, `@array`, with the reference:

```
for my $element (@{$aref}) {
   ...
}
```

"How do I print out the contents of a hash when all I have is a reference?" First write the code for printing out a hash:

```
for my $key (keys %hash) {
  print "$key => $hash{$key}\n";
}
```

And then replace the hash name with the reference:

```
for my $key (keys %{$href}) {
```

```
    print "$key => ${$href}{$key}\n";
}
```

Use Rule 2

Use Rule 1 is all you really need, because it tells you how to do absolutely everything you ever need to do with references. But the most common thing to do with an array or a hash is to extract a single element, and the **Use Rule 1** notation is cumbersome. So there is an abbreviation.

`${$aref}[3]` is too hard to read, so you can write `$aref->[3]` instead.

`${$href}{red}` is too hard to read, so you can write `$href->{red}` instead.

If `$aref` holds a reference to an array, then `$aref->[3]` is the fourth element of the array. Don't confuse this with `$aref[3]`, which is the fourth element of a totally different array, one deceptively named `@aref`. `$aref` and `@aref` are unrelated the same way that `$item` and `@item` are.

Similarly, `$href->{'red'}` is part of the hash referred to by the scalar variable `$href`, perhaps even one with no name. `$href{'red'}` is part of the deceptively named `%href` hash. It's easy to forget to leave out the `->`, and if you do, you'll get bizarre results when your program gets array and hash elements out of totally unexpected hashes and arrays that weren't the ones you wanted to use.

An Example

Let's see a quick example of how all this is useful.

First, remember that `[1, 2, 3]` makes an anonymous array containing `(1, 2, 3)`, and gives you a reference to that array.

Now think about

```
@a = ( [1, 2, 3],
       [4, 5, 6],
  [7, 8, 9]
     );
```

@a is an array with three elements, and each one is a reference to another array.

`$a[1]` is one of these references. It refers to an array, the array containing `(4, 5, 6)`, and because it is a reference to an array, **Use Rule 2** says that we can write `$a[1]->[2]` to get the third element from that array. `$a[1]->[2]` is the 6. Similarly, `$a[0]->[1]` is the 2. What we have here is like a two-dimensional array; you can write `$a[ROW]->[COLUMN]` to get or set the element in any row and any column of the array.

The notation still looks a little cumbersome, so there's one more abbreviation:

Arrow Rule

In between two **subscripts**, the arrow is optional.

Instead of `$a[1]->[2]`, we can write `$a[1][2]`; it means the same thing. Instead of `$a[0]->[1] = 23`, we can write `$a[0][1] = 23`; it means the same thing.

Now it really looks like two-dimensional arrays!

You can see why the arrows are important. Without them, we would have had to write `${$a[1]}[2]` instead of `$a[1][2]`. For three-dimensional arrays, they let us write `$x[2][3][5]` instead of the unreadable `${${$x[2]}[3]}[5]`.

Solution

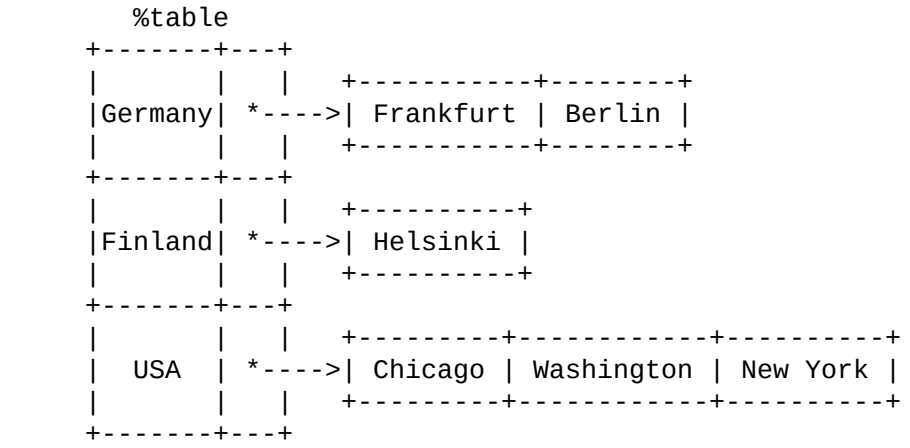
Here's the answer to the problem I posed earlier, of reformatting a file of city and country names.

```
 1   my %table;

 2   while (<>) {
 3    chomp;
 4     my ($city, $country) = split /, /;
 5     $table{$country} = [] unless exists $table{$country};
 6     push @{$table{$country}}, $city;
 7   }

 8   foreach $country (sort keys %table) {
 9     print "$country: ";
10     my @cities = @{$table{$country}};
11     print join ', ', sort @cities;
12     print ".\n";
13 }
```

The program has two pieces: Lines 2--7 read the input and build a data structure, and lines 8-13 analyze the data and print out the report. We're going to have a hash, `%table`, whose keys are country names, and whose values are references to arrays of city names. The data structure will look like this:



We'll look at output first. Supposing we already have this structure, how do we print it out?

```
 8   foreach $country (sort keys %table) {
 9     print "$country: ";
10     my @cities = @{$table{$country}};
11     print join ', ', sort @cities;
12     print ".\n";
13 }
```

`%table` is an ordinary hash, and we get a list of keys from it, sort the keys, and loop over the keys as usual. The only use of references is in line 10. `$table{$country}` looks up the key `$country` in the hash and gets the value, which is a reference to an array of cities in that country. **Use Rule 1** says that we can recover the array by saying `@{$table{$country}}`. Line 10 is just like

```
@cities = @array;
```

except that the name `array` has been replaced by the reference `{$table{$country}}`. The `@` tells Perl to get the entire array. Having gotten the list of cities, we sort it, join it, and print it out as usual.

Lines 2-7 are responsible for building the structure in the first place. Here they are again:

```
2   while (<>) {
3    chomp;
4     my ($city, $country) = split /, /;
5     $table{$country} = [] unless exists $table{$country};
6     push @{$table{$country}}, $city;
7   }
```

Lines 2-4 acquire a city and country name. Line 5 looks to see if the country is already present as a key in the hash. If it's not, the program uses the `[]` notation (**Make Rule 2**) to manufacture a new, empty anonymous array of cities, and installs a reference to it into the hash under the appropriate key.

Line 6 installs the city name into the appropriate array. `$table{$country}` now holds a reference to the array of cities seen in that country so far. Line 6 is exactly like

```
push @array, $city;
```

except that the name `array` has been replaced by the reference `{$table{$country}}`. The `push` adds a city name to the end of the referred-to array.

There's one fine point I skipped. Line 5 is unnecessary, and we can get rid of it.

```
2   while (<>) {
3    chomp;
4     my ($city, $country) = split /, /;
5   ####  $table{$country} = [] unless exists $table{$country};
6     push @{$table{$country}}, $city;
7   }
```

If there's already an entry in `%table` for the current `$country`, then nothing is different. Line 6 will locate the value in `$table{$country}`, which is a reference to an array, and push `$city` into the array. But what does it do when `$country` holds a key, say `Greece`, that is not yet in `%table`?

This is Perl, so it does the exact right thing. It sees that you want to push `Athens` onto an array that doesn't exist, so it helpfully makes a new, empty, anonymous array for you, installs it into `%table`, and then pushes `Athens` onto it. This is called `autovivification'--bringing things to life automatically. Perl saw that they key wasn't in the hash, so it created a new hash entry automatically. Perl saw that you wanted to use the hash value as an array, so it created a new empty array and installed a reference to it in the hash automatically. And as usual, Perl made the array one element longer to hold the new city name.

The Rest

I promised to give you 90% of the benefit with 10% of the details, and that means I left out 90% of the details. Now that you have an overview of the important parts, it should be easier to read the *perlref* manual page, which discusses 100% of the details.

Some of the highlights of *perlref*:

- You can make references to anything, including scalars, functions, and other references.
- In **Use Rule 1**, you can omit the curly brackets whenever the thing inside them is an atomic scalar variable like `$aref`. For example, `@$aref` is the same as `@{$aref}`, and `$$aref[1]` is the same as `${$aref}[1]`. If you're just starting out, you may want to adopt the habit of always including the curly brackets.
- This doesn't copy the underlying array:

```
$aref2 = $aref1;
```

You get two references to the same array. If you modify `$aref1->[23]` and then look at `$aref2->[23]` you'll see the change.

To copy the array, use

```
$aref2 = [@{$aref1}];
```

This uses `[...]` notation to create a new anonymous array, and `$aref2` is assigned a reference to the new array. The new array is initialized with the contents of the array referred to by `$aref1`.

Similarly, to copy an anonymous hash, you can use

```
$href2 = {%{$href1}};
```

- To see if a variable contains a reference, use the `ref` function. It returns true if its argument is a reference. Actually it's a little better than that: It returns `HASH` for hash references and `ARRAY` for array references.
- If you try to use a reference like a string, you get strings like

  `ARRAY(0x80f5dec)` or `HASH(0x826afc0)`

  If you ever see a string that looks like this, you'll know you printed out a reference by mistake.

  A side effect of this representation is that you can use `eq` to see if two references refer to the same thing. (But you should usually use == instead because it's much faster.)
- You can use a string as if it were a reference. If you use the string `"foo"` as an array reference, it's taken to be a reference to the array `@foo`. This is called a *soft reference* or *symbolic reference*. The declaration `use strict 'refs'` disables this feature, which can cause all sorts of trouble if you use it by accident.

You might prefer to go on to *perllol* instead of *perlref*; it discusses lists of lists and multidimensional arrays in detail. After that, you should move on to *perldsc*; it's a Data Structure Cookbook that shows recipes for using and printing out arrays of hashes, hashes of arrays, and other kinds of data.

Summary

Everyone needs compound data structures, and in Perl the way you get them is with references. There are four important rules for managing references: Two for making references and two for using them. Once you know these rules you can do most of the important things you need to do with references.

Credits

Author: Mark Jason Dominus, Plover Systems (`mjd-perl-ref+@plover.com`)

This article originally appeared in *The Perl Journal* (http://www.tpj.com/) volume 3, #2. Reprinted with permission.

The original title was *Understand References Today*.

Distribution Conditions