Cut-to-the-Chase Series - **Perl Programming**

Hashes

Perl Programming - Hashes

© Eric Matthews

visit us at www.anglesanddangles.com

Table of Contents

Hashes

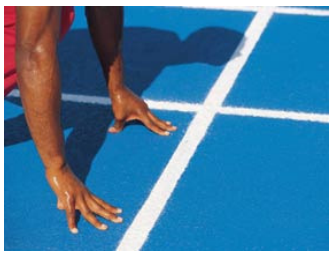
We see so little of anything at which we look that we are usually satisfied with simple surfaces, perhaps because the deeper view is often so terrifying in its complexity.

- Dean Koontz

ICON KEY
Toy Code
Hands-on
Important Info
Be Careful

This Module

It is assumed that you have already finished all of the foundation topics, or have a very solid understanding of Perl.



Introduction to Subject Matter

A hash is a primitive data structure in Perl. Hashes are sometimes referred to as associative arrays. They are nice tidy little structures that consist of a key/value pair. They also express their key as a natural value that makes them a bit less obtuse to use than arrays. Indexing into a hash on a key is a pretty fast and efficient operation.

Getting Started

This section consists of a review of hashes, and some advanced topics. The review is essentially the same materials as was covered in the 'Foundations' module and the 'References' module.

Review of Hash Fundamentals

This is a review of hashes. You can skip this section if you have already covered this material in the "Foundations" module.

A hash is an associative array. An associative array is an array where the index value is meaningful to a human being. Let us look at a conventional array and contrast it with an associative array (or hash as I will refer to it from here on).

Consider the following array.

```
@arr = (298, 114, 31, 12, 2, 67, 55);
```

The elements of this array are as follows.

```
@arr[0] = 298;
@arr[1] = 114;
@arr[2] = 31;
@arr[3] = 12;
@arr[4] = 2;
@arr[5] = 67;
@arr[6] = 55;
```

It is a stretch (to say the least) to know what item four in the array represents. When we are just using an array to stash a collection of like variables this is not a problem. But when the data we are storing is not self-described and when it becomes important to know what the data is about, an array can be limiting.

For example, suppose that the data above represents a baseball player's statistics. It now becomes more important to understand what location @arr[3] represents. Enter the hash.

```
my %batter_stats =
(
'Batting Average' = 298;
'Hits' = 114;
'Homeruns' = 31;
'Double' = 12;
'Triples' = 2;
'RBI' = 67;
'Runs' = 55;
)
```

A hash allows us to index values in an array in a meaningful way. This concept is similar to the concept of a column/field in database management systems, and how tables are created and stored in a catalog or data dictionary. If you think long and hard you will see how hashes can open up the world of creating and working with structured data in your Perl programs.

ⓘ If you are familiar with standard database terminology your senses will be somewhat offended by Perl's use of terms. A key is used to describe what in database terminology would be referred to as a field or column. I find the use of this terminology quite confusing, but in the interest of being consistent with the naming conventions of the language I will stick with them.

Creating hashes

As with most things in Perl there is more than one way to create a hash. These two ways are presented for your consideration.

🖫 HASH1.pl

```
# create an empty hash
my %hsh = ();

#add some elements
$hsh{'firstname'} = 'John';
$hsh{'middle_init'} = 'H.';
$hsh{'lastname'} = 'Smith';

print $hsh{lastname};
```

🖫 HASH2.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

print $hsh{lastname};
```

Adding an element to a hash

🖫 HASH3.pl

```
my %hsh =
```

```
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my $title = "Supreme Commander";

$hsh{title} = $title;

print $hsh{firstname};
print " ";
print $hsh{middle_init};
print " ";
print $hsh{lastname};
print ", ";
print $hsh{title};
```

Note that you can assign variables to a hash element. As you can see, adding a value to a hash is a pretty easy undertaking.

Deleting a hash element

HASH4.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

delete $hsh{middle_init};

if (exists($hsh{middle_init}))
{
 print "key exists";
}
else
{
 print "key does not exist";
}
```

Get hash keys

HASH5.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my $key="";
foreach $key (keys %hsh)
```

```
{
  print $key . "\n";
}
```

Get hash values

HASH6.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my $val="";
foreach $val (values %hsh)
{
  print $val . "\n";
}
```

Get both hash keys and values

HASH7.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my $key="";
my $val="";
while (($key, $val) = each(%hsh))
{
```

```
  print "$key => $val\n";
}
```

Finding a particular hash key

This is actually pretty neat. You can find a hash key without having to code a loop yourself.

HASH8.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

# exists is used to verify key
if (exists($hsh{middle_init}))
{
 print $hsh{middle_init};
 print "\n"
}
else
{
 print "key does not exist";
}

# delete hash key middle_init
delete $hsh{middle_init};

# is it deleted?
if (exists($hsh{middle_init}))
{
 print $hsh{middle_init};
 print "\n"
```

```
}
else
{
 print "key does not exist";
}

# re-add hash key middle_init
$hsh{middle_init} = "";

if (exists($hsh{middle_init}))
{
 print $hsh{middle_init};
 print "\n"
}
else
{
 print "key does not exist";
}
```

Reversing values and keys in a hash

This is quite an interesting little function that allows you to make your values the keys and you keys the values. This certainly could be used in defining meta data schemas.

HASH9.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my %swap = reverse %hsh;

my $key="";
foreach $key (keys %swap)
```

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

Sorting a hash

A hash can be sorted by key or by value.

By Key

HASH10.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my $key="";
foreach $key (sort keys %hsh)
{
  print $key . "\n";
}
```

By Value

HASH11.pl

```
my %hsh =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my $val="";
foreach $val (sort values %hsh)
```

```
{
  print $val . "\n";
}
```

Merging two hashes

HASH12.pl

```
my %person =
(
firstname => 'John',
middle_init => 'H.',
lastname => 'Smith',
);

my %phone =
(
area_cde => '206',
exchange => '466',
extension => '1515',
);

my %person_phone = (%person, %phone);

my $key="";
my $val="";
while (($key, $val) = each(%person_phone))
{
  print "$key => $val\n";
}
```

Hash truth

An hash without any elements is false, otherwise if the hash contains at least one element it is true.

```
@a%a;                          # %a is false

%a=();                         # %a is false

%a=(1=>"one", 2=>"two");       # %a is true
```

And, in case you were wondering, scalar truth works the same from within a hash.

Hashes and References (materials from 'References' module)

These are the materials from the module on "References". If you have already covered this module you can use this section as a quick review. I have included these materials with the assumption that the module on references has already been covered. You could try to cover this material here without any prior understanding on references. If the audience is already well versed and savvy on the subject of pointers it will probably be okay. If the audience has little or no knowledge as to the subject of pointers then trying to cover this subject matter will likely be a nightmare.

References to Hashes

Hashes are an enormously efficient and useful way of storing data. First, they allow us to work with data in manner that is more in tune with our own set of linguistics. What I mean is that the key is not an esoteric number, it is in language that can be meaningful to us. Second, and highly related to first, we have a quick means of indexing and retrieval if we know exactly what we are looking for. This is unlike the array, where we need to loop until we find what we are looking for.

Hashes are flat structures by themselves. You get a key and a data element. If you are a business programmer you should already understand that quantifying data is more often than not a myriad of complex hierarchical and relational organization. Hashes by themselves do not readily help us to work in these problem spaces. But enter references and the world becomes our oyster.

DEREFERENCE HASHES

SYNTAX	EXAMPLE
%$<scalar >	%$myhsh
%{$<scalar >}	%{$myhsh}

Reference to a simple hash

This is pretty basic stuff if you already have an understanding of references. It shows the syntax of creating a hash reference and dereferencing a hash.

REFHSH1A.PL

```
my %phone = (area_cde  => "254",
```

```
              exchange  => "455",
             extension => "1223"
           );

my $phone = \%phone;
print $phone                 . "\n";
print ${$phone}{extension}   . "\n";

my @keys = keys(%{$phone});
print "@keys"                . "\n";

my @values = values(%{$phone});
print "@values"              . "\n";

my $key="";
my $val="";
while (($key, $val) = each(%{$phone}))
{
  print "$key => $val\n";
}
```

Print Results

$phone	HASH(0x368c60)
${$phone}{extension}	1223
$aarr[1]->[1]	1223
"@keys"	exchange area_cde extension
"@values"	455 254 1223
(Loop) "$key => $val\n"	exchange => 455
"$key => $val\n"	area_cde => 254
"$key => $val\n"	extension => 1223

Hash – Single key

The following is a simple example of an anonymous reference to a hash.

ANONREFHSH.PL

```
my $phone = {
             area_cde  => "425",
             exchange  => "277",
             extension => "1969"
           };

print $phone . "\n";
print $phone->{extension} , "\n";
print %$phone,"\n";  #key and values
# or, a with a little more control
while ( (my $key, my $val) = each (%$phone) )
{
  print "$key -> $val\n";
}
```

Print Results

`$phone`	HASH(0x1a7f120)
`$phone->{extension}`	1969
`%$phone`	area_cde425exchange277extension1969
`"$key -> $val\n"`	area_cde -> 425 exchange -> 277 extension -> 1969

In respect to general semantics we know that a phone number by itself is not in its most atomic form with respect to its data columns. In other words, a phone number is an ordered aggregation of three discrete fields; area code, exchange, and extension. We certainly could build this data structure by creating a hash named phone. We can also accomplish this feat using an anonymous hash named phone.

Dereferencing an anonymous hash

```
%$phone
```

area_cde425exchange277extension1969

Getting a value for an anonymous hash key

```
$phone->{extension}
```

```
1969
```

Looping through an anonymous hash

```
while ( (my $key, my $val) = each (%$phone) )
{
  print "$key -> $val\n";
}
```

```
area_cde -> 425
exchange -> 277
extension -> 1969
```

Hash – Multiple keys

It is possible to create even more complex data structures under the guise of an anonymous hash. Keeping in continuity with our previous example, let's extend it to include both work and home phone numbers.

ANONREFHSH1a.PL / ANONREFHSH1b.PL

Lets first look at our data structure. It is the same in both programs.

```
my $phn = {
          'home' => {
                   area_cde  => "254",
                   exchange  => "455",
                   extension => "1223"
               },
          'work'  => {
                   area_cde  => "312",
                   exchange  => "756",
                   extension => "7643"
               }
           };
```

As you already know $phn is a reference in and of itself.

```
$phn
```

HASH(0x1a755b0)

Lets dereference $phn and see what we get.

```
%$phn
```

homeHASH(0x1a7f120)workHASH(0x1a75538)

Hmmmmmm?

What you may not realize at this point is that both 'work' and 'home' are references themselves. So we have a reference to other references.

Here are some basic techniques for dealing with these.

ANONREFHSH1b.PL

Return hash keys

```
my @keys = keys(%$phn)
```

```
home work
```

Accessing individual elements for a key

```
$phn->{'work'}->{'area_cde'}
```

```
312
```

Modifying data

```
$phn->{'work'}->{'extension'} = "4432"
```

Returning all hash keys and their elements/data

ANONREFHSH1a.PL

Since we are dealing with a hierarchical structure, we have a nested loop.

```
foreach $key (sort keys (%$phn))
{
   my $ref = $phn->{$key};
   print "$key => \n";
   foreach $key2 (keys (%$ref))
   {
      my $value = $ref->{$key2};
      print "$key2 => $value \n";
   }
}
```

```
home =>
area_cde => 254
exchange => 455
extension => 1223
work =>
```

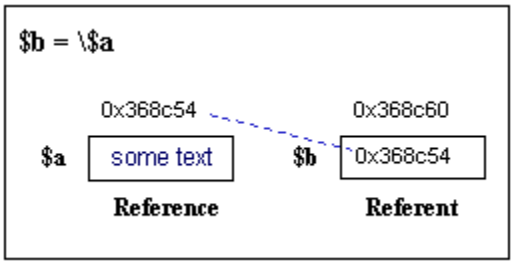
```
area_cde => 312

exchange => 756

extension => 7643
```

A Subtle Point regarding the syntax

With all this cryptic syntax flying around I want to belabor a point that can be quite troubling when working with references. By now it should be ingrained what a reference and a dereference are. We should take a look at a previous picture to quickly review the concept.



In this example, our reference is to a scalar. If we choose to obtain the value of $a through the referent $b we would code…

```
$$b
```

…thus allowing us to resolve our reference and see what is actually stored in this location.

With dereferencing we need to remain cognizant of the type of structure we are dereferencing. This may seem obvious but it is easy to get into trouble when dealing with complex structures. For example…

ANONREFHSH1c.PL

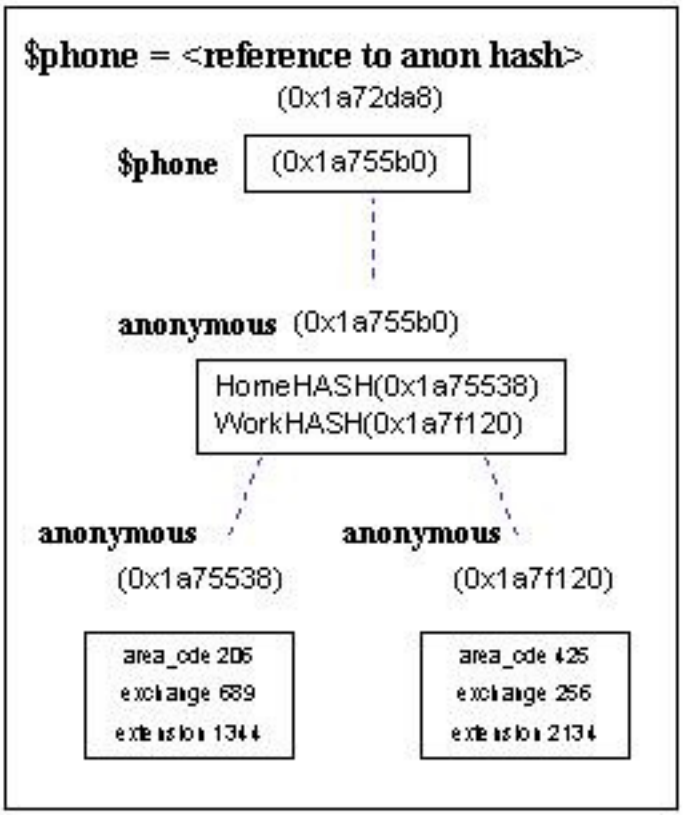
```
my $phone = {
            Work =>
                      { area_cde  => "425",
                        exchange  => "256",
                        extension => "2134"
                      } ,
            Home =>
                      { area_cde  => "206",
```

```
                        exchange  => "689",
                        extension => "1344"
                      }
          };

print my $a = \$phone . "\n";
print $phone . "\n";
print %$phone ;
print "\n";
print %$phone->{Work};
print "\n";
print %$phone->{Home};
print "\n";
print $$phone{Home};
print "\n";
print $$phone->{Home};
```

Please make note that our structure is the same as the ones in the previous examples.

We have so much indirection going on (even though this is a fairly simplistic example) I think it would be appropriate to examine graphically the data structure represented in our code.



First, our reference $phone holds a reference to an anonymous hash. Perl is quite nice (sophisticated) in that it is able to tell us what type of reference we have. In other words, not only does it contain the address, but the type of storage we have at that address. I think it is quite ironic that a language that does not offer typed data as part of the primitive language, would allow us to easily differentiate the actual data type stored in memory.

Looking at what data is actually stored in $phone…

```
$phone
```

```
HASH(0x1a755b0)
```

…we see that we have a reference to a hash. When we dereference $phone…

```
%$phone
```

```
HomeHASH(0x1a75538)WorkHASH(0x1a7f120)
```

…we get the data that is stored at location 0x1a755b0. Since, we already know our reference is to a hash we should expect to see at least one key/value pair. In this instance we return two **key**/**value** pairs.

```
Home  HASH(0x1a75538)

Work  HASH(0x1a7f120)
```

What starts to make this so abstract is that our key value pair is yet another reference. We have a key named 'Home' that is a hash reference and a key named 'Work' that is also a hash reference.

We could also access each value with the following syntax.

```
%$phone->{Work}

%$phone->{Home}
```

```
HASH(0x1a7f120)

HASH(0x1a75538
```

Please make note that we are using the infix operator (->) with the hash dereferencer. If we were to attempt to dereference $phone in the wrong context (in this case using the scalar dereferencer …

```
$$phone->{Work}
```

…we would receive the following runtime error.

```
Not a SCALAR reference at anonrefhsh1c.pl line 26.
```

A key point to remember here is that when you are using the infix operator, you must make sure you are applying the correct context.

The following code snippet uses the infix operator appropriately and is the equivalent of `%$phone->{Work}`

```
$phone->{Work}
```

…it returns…

```
HASH(0x1a75538)
```

Hopefully this has served to demystify all the indirection that is a natural byproduct of anonymous hashes. Some of you reading this may feel like the room is spinning. My goal is not to confuse or bewilder you (honest). But, if you ever hope to explore and utilize the myriad of

modules out there that are free for the taking then understanding all of this stuff will carry you a long way. This is certainly a very compelling and difficult subject for most programmers.

So we do not leave any loose ends regarding this topic, I am going to finish it by showing code used to access individual elements in our hashes.

ANONREFHSH1c2.PL

```
print $phone->{Work}->{extension} . "\n";
print $phone->{Home}->{extension} . "\n";

my $getphone = \$phone  ;
print ${$getphone}->{Work}->{extension};
```

The syntax for the first two print statements has been previously been discussed. Notice that we can create a reference to our anonymous hash.

```
my $getphone = \$phone  ;
```

And, we can dereference this scalar to get at the data.

```
print ${$getphone}->{Work}->{extension};
```

Array of Hash References

We have already discussed how to create an array reference to other arrays. It is also possible to create an array of hash references. For that matter, you can also have an array with references to any data type (subroutines, scalars, etc…).

ARRHSHREFS.PL

```
my %name = (firstname    => "Fred",
           middle_init  => "L",
           lastname     => "Flintstone"
           );

my %address = (street  => "123 Rubble Way",
               city    => "Seattle",
```

```
              state   => "WA",
              zip     => "98058"
             );

my %phone = (area_cde  => "254",
             exchange  => "455",
             extension => "1223"
            );

my @employee_record = (\%name, \%address, \%phone);

print %{$employee_record[1]} ; print "\n";
print ${$employee_record[0]}{lastname} ;
```

Finally, I want to end this topic by trying to put into context the practical merits of all of this discussion. The primary and principle goal of references is to allow us to abstract the programs we write. The primary reason we want to abstract programs is to accomplish a degree of reuse. The noble objective of reuse is that it minimizes the volume of code we need to write to accomplish a given task. Also reuse serves to help us save time and rework in the programs we write and deliver.

Each and every one of these examples could have easily been written to accomplish the same end output without using references. Granted, but the point of this discussion is to present in its most basic and stripped down format the mechanics of references so that you understand the syntax and techniques that are deployed in writing reusable code in Perl. Just venture out and take a look at the module code that is available for Perl. You will find much of it is written using references.

In perspective of the phone examples that have been presented, our goal in creating these may very well be to provide an agreed upon data structure to use for dealing with phone numbers. If this is the case then what the programmer reusing the structure needs to be concerned with is how to access the individual elements and how to refer to the structure as a whole. The technical details of how we design the structure really only needs to be understood by the architect.

I am writing this subject to address two distinct audiences: programmers that will be *creating* their own modules in Perl and programmers that will be *using* existing modules. I may certainly be a bit ambitious in trying to address both audiences in a single document.

If you are planning on writing your own modules and have done similar types of work in other languages then I do not need to tell you the importance of understanding all of this. If you are in the other camp you may be questioning why you need to understand all of this. This is certainly

a legitimate question. In a perfect world a programmer using a library or module or class (or whatever you want to call it) should not have to be concerned with the technical details of what the are using. But, this is not a perfect world. More often than not you will be faced with less than stellar documentation and it helps immeasurably if you understand the syntax and mechanics of the subject.

More on Hashes

It is going to be my goal here to address a number of subjects and issues on hashes. I will also try to provide you with tips and tricks as to how you might deal with certain coding scenarios.

Merging Hashes – some finer points

Back in the review we showed you how to merge two hashes. This worked fine as long as the keys were unique. Just take a look at "combinehash.pl" to see that this is the case. How would we merge the data from one hash into another hash?

Suppose I have two hashes.

Garden1

```
'Red'=> "Apple",
'Yellow'=> "Squash",
'Orange'=> "Carrot",
'Purple' => "Eggplant",
'Green' => "Broccoli"
```

Garden2

```
  'Green'=> "Beans,Spinach,Apple",
  'Red'=> "Pepper",
  'White'=>"Radish"
```

I want to merge Garden2 into Garden1. The data resulting from the merge should look something like…

Garden2 after the merge

```
'Red'=> "Apple,Pepper ",
'Yellow'=> "Squash",
'Orange'=> "Carrot",
'Purple' => "Eggplant",
'Green' => "Broccoli,Beans,Spinach,Apple "
'White'=>"Radish"
```

COMBINEHASH2.pl

```
my ($key, $val);
# desire to combine garden2 into garden1
while (($key, $val) = each %garden2)
{
```

```
  $garden1{$key} .= "," . $val;
}
# see what we got
while (($key, $val) = each %garden1)
{
 print "$key=>$val, \n"
};
```

The actual code needed to accomplish this feat is not all that spectacular. We need to loop through "garden2" and where the keys are identical between "garden1" and "garden2" we append the data to "garden1". If the keys are new, we will be adding it to "garden1"

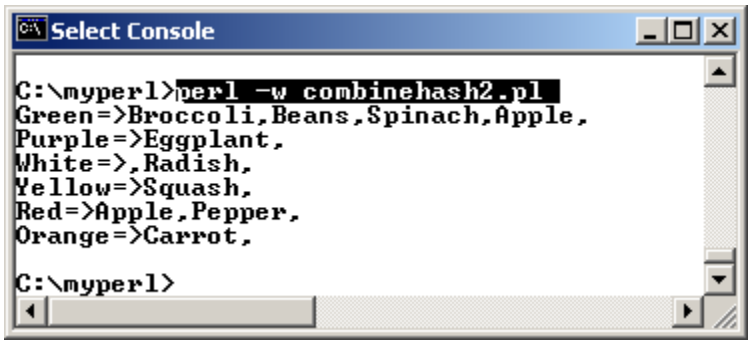
```
while (($key, $val) = each %garden2)

{

  $garden1{$key} .= "," . $val;

}
```

This code serves us well for the most part. But if we look at the output you can see that we have some formatting issues.



For the new stuff, we are appending a comma to the data, which is just not all that desirable.

Hands-on:

Rewrite the previous program above to handle the formatting problems that we encountered. And, after you have accomplished this how would you append "garden1", but NOT add new keys?

Using references

In the above solution we have decided to separate our data values with a comma. So what happens if the data we are storing contains commas? The problem with choosing a delimiter for our data is that we always run the risk of storing data that uses that character. If this happens we have "hosed" our storage. We certainly could check the data upon input to make sure this does not happen, but what would we change the character to? And, is it the role of the programmer to modify data? This all can get quite dicey and convoluted. Is there a better means of storing our data?

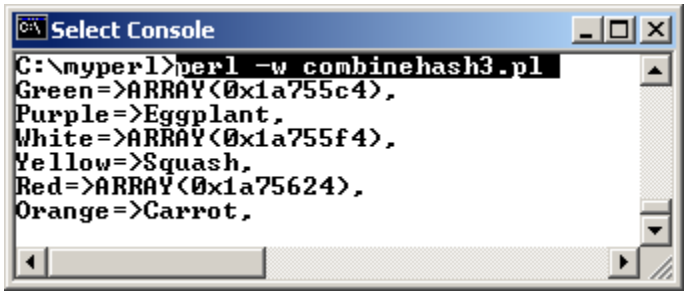
Yes. We can use references.

COMBINEHASH3.pl

```
# desire to combine garden2 into garden1 using a reference to an anon
array
while (($key, $val) = each %garden2)
{
  $garden1{$key} = [$val];
}
```

Putting brackets around $val creates an anonymous array. By doing this we preclude the need to have a character to separate our data when we store it. Now I want to stop here and disclaim that we still are using commas as the separator between the data we are storing in our individual hashes that we are combining. Albeit to say we could use the technique I just showed you to deal with each individual hash.

We still have a problem however. Our data is of a different type. See…



Hands-on

Rewrite the above program so that all the data is of the same type. Hint: make everything a reference.

Hash of an anonymous array – 2 examples and some semantics

In my estimation it is hard to separate technique from utilization, though too often this is what we do. When we decouple technique from practical application we must assume that someone will vision a use for a specific technique. Unfortunately we are not all wired to easily make these connections. Granted it is sometimes difficult to offer tiny examples in a way that can be seen on practical terms.

Bringing closure to before – But, practical?

The following example reads the text file "names.txt". This file contains comma delimited data.

```
Joe,D,Messina,12-17-1974
Kenny,L, Loggins,8-06-1952
Millie,Tony,Vanilli,9-14-1979
L,L,Bean,9-14-1979
```

Our goal is to take this data and stuff it into a hash containing the following keys.

```
$firstname,
$lastname ,
$middlename,
$unique_id,
$dob
```

To accomplish this we have two hashes. One named "%people" and one named "%person". The goal is to load each delimited line of our file into "%person" and then move the data from there to "%people". The code follows:

COMBINEHASH4a.pl / COMBINEHASH4b.pl

Loading %person

```
open ("F" ,"names.txt") or die $!;
while (<F>)
{
 if ($_ =~ /(.*),(.*),(.*),(.*)/)
```

```
 {
  $person{$firstname} = $firstname_data = $1;
  $person{$middlename} = $middlename_data = $2;
  $person{$lastname} = $lastname_data = $3;
  $person{$dob} = $dob_data = $4;
  $person{$unique_id} = ++$unique_id_data;
  add_person();
 }
}
```

Note that we are generating a unique identifier as we load the file. Also note that this code does not take into account how we would deal with generating this key in more than a "one shot" load scenario.

Loading %people

```
sub add_person
{
 while (($key, $val) = each %person)
 {
  push @{$people{$key}}, $person{$key};
 }
}
```

As you can see, we have created an anonymous array for each key. The array holds our data.

Getting a person from %people

```
while (($key, $val) = each %people)
{
 print "$key => @$val[2] \n";
}
```

The above code will return…

```
Millie,Tony,Vanilli,9-14-1979
```

…from the hash %people.

By the way, this example brings closure to the previous examples where I suggested a storage method that negated the use of a delimiter. Now, here is a graphic visualization of %people.

%person						
unique_id	qw(	1	2	3	4	)
firstname	qw(	Joe	Kenny	Millie	L	)
middlename	qw(	D	L	Tony	L	)
lastname	qw(	Messina	Loggins	Vanilli	Bean	)
dob	qw(	12/17/1974	8/6/1952	9/14/1979	9/14/1979	)

First, is this example all that practical? If retrieval by the array indices is our desired entry point then I would say yes. But, how would we lookup and retrieve Millie Vanilli by last name? You see our dilemma with this means of storage. Second, if we need to retrieve a specific key/value pair, then indeed this is a dynamite means of storage and retrieval (our second example forthcoming). But seriously, how often would someone really want to return a list of everyone's first name?

How about this for practical?

If our goal was to be able to lookup and retrieve a row by one of the keys (like last name), then our visual representation would look like this.

%person				
unique_id	firstname	middlename	lastname	dob
1	Joe	D	Messina	12/17/1974
2	Kenny	L	Loggins	8/6/1952
3	Millie	Tony	Vanilli	9/14/1979
4	L	L	Bean	9/14/1979

So what type of structure would we need to store to meet this condition? Now you might say that we could come up with a hash of hashes. This certainly could be a viable solution. So how would you define the key for what you are storing in %person to ensure that the name is unique?

A practical example using a Hash storing an anonymous array?

At the organization where I work we have a huge file named VT. VT stands for verification table. This file houses thousands of tables like…

SEX – Male, Female, Other

RELIGION – Catholic, Baptist, Morman, etc…

…and so forth and so on. These tables get hooked to our screens and allow the users to enter field data from a predefined list. If ever there were a practical implementation for using the technique of a hash that contains an anonymous array it is here (of course there are also many other applications as well). Instead of offering up an example, since you have already seen the technique, we will have an exercise here instead.

Hands-on

Write a Perl program that contains the following tables as hashes.

Religion	Emergency Disposition	Finance Flags
Description	Description	Description
Assembly of God	Discharged	Must See Financial Counselor
Baptist	Dis/Transferred to Inpatient Care	Must See Legal Dept
Buddhist	Skilled Nursing Facility	Must See Cashier
Catholic	Dis/Transferred to ICF	Medical Eligibility
Christian	Sent to Another Hospital	
Church of God		
Church of Christ		
Congregational		

Your program should be able to…

- Add new tables
- Return choices for a given table
- Check a table for a valid choice
- Delete a table
- Any additional functionality you care to provide

Here you have seen two very different examples that use the same technique. The last one shown has probably given you a great deal of food for thought. The first example is fraught with problems and not very practical (in my opinion).

Hash of Hashes

This was actually covered in the section "Hashes and References (materials from 'References' module)" under the heading "Hash – Multiple Keys". If you wish to see this technique go to that module.

Preserving hash order

Hashes get stored internally in a means that might not coincide with what we want the external order to be. Consider the following code:

```
my %hash;
my $k;
my $v;
$hash{'one'} = "hanna";
$hash{'two'} = "dool";
$hash{'three'} = "set";
$hash{'four'} = "net";
while (($k, $v) = each %hash) {print "$k=>$v, "};
```

When we look at the output, we see that it is in the following order.

```
one=>hanna, three=>set, two=>dool, four=>net,
```

We could probably assume from the context that we want to preserve the entry order. So how do we do this? By using *tie*.

```
my %hash;
use Tie::IxHash;
tie (%hash, 'Tie::IxHash');
my $k;
my $v;
$hash{'one'} = "hanna";
$hash{'two'} = "dool";
$hash{'three'} = "set";
$hash{'four'} = "net";
while (($k, $v) = each %hash) {print "$k=>$v, "};
```

By adding these two lines we preserve the entry order and our output is now...

```
one=>hanna, two=>dool, three=>set, four=>net,
```

There is much more to using "tie", unfortunately this is not the time nor place.

Retrieving environment variables

Here is a simple program that returns your environment variables. Since it is a hash, I have included it in this module.

```
foreach (sort keys %ENV) {
      #key    #value   ....pair
  print "$_  =  $ENV{$_}\n";
 }
```

Using Hashes to Create an Object

Brief introduction

Perl has a very open, simple, and flexible model for doing object oriented programming (OOP). One distinct difference between Perls view of OOP and other OOP languages (like Java), is that Perl does not truly encapsulate others from the source. Also, when doing object oriented programming it is essential to document your code and to document it well. Java has a terrific tool for doing this (Javadoc) in a programmatic fashion. Of course this is possible because of the explicit boundaries that it sets regarding its world vision of OOP. In Perl it is up to the diligence of the programmer to adequately document their code. They can either use POD (Plain Old Documentation) or some other means that is up to them to invent.

If you are approaching this section with some prior background in OOP you should be able to fill in the blanks. If you are brand new to OOP this material and the syntax presented might seem a bit strange to you. I am not planning on going into a long discussion on OOP. Instead, I will present a lean narrative as to the concepts we are deploying. My primary focus is to present the syntax and technique as pertaining to using hashes to create objects. I have chosen an example problem that everyone should be able to relate to.

When you whittle it down, OOP is focused on one principal and one principal only…CODE REUSE!

Since OOP is about the quantification of data and code in the form of an object, suppose I want to create a "Person" object. We can quantify the data for this object with the bare essentials. A person will have these attributes (for the most part):

```
Firstname (in most cultures)
Middlename (maybe)
Lastname
Some unique ID (like Social Security number in this country)
Date of birth
```

Of course this is a skeletal list but one that is enough to get the idea across. The high level goals of this object (and any object or code for that matter) will be to:

1. Enter data
2. Retrieve data
3. Manipulate data

One advantage of using an OOP approach is that data and code (what acts upon the data…often referred to in OOP jargon as methods) live side by side within the object. This

concept gives us the principal known as encapsulation (a fifty-cent term meaning code and data living together).

Now you may be thinking, "So what's the big deal? I can create a subroutine and use data structures within the subroutine". Indeed you can, and in some respect when you do this you are encapsulating your code to a degree but the key difference between this type of a solution and an OOP solution boils down to two distinct differences.

1) How you access the code;

2) How the code exists within your program.

You call a subroutine from another part of your program. In Perl this subroutine can be part of the caller's package or it could be in a different package scope. Of course this package could very well be part of a module. When the subroutine is called, it will provide some service or function then return to the caller. This service or function can be defined in the broadest terms as one of the three functions previously defined.

If the subroutine is encapsulated in the sense that its data structures are local to the subroutine then the persistence (unless otherwise forced) is for the duration of the call. Each caller essentially gets in line to access the routine. Of course it is possible to create your subroutine with a package or module to deal with issues of multiple users and persistence of storage. But, in doing so it is entirely up to you to define what this looks like and how it works.

To compare the conventional approach with an OOP approach we can state that the three functions we have described remain the same. What is very different is how we access the code. In an OOP solution when I need an object, I do not call a subroutine, instead, I create an instance of the object. This mechanism gives me an inherent separation between consumers of the code. In fact, consumers of an object do not even need to be aware of other consumers. Another nice by product of this approach is that the object remains in existence as long as the consumer has a reference to it. This inherent approach to code instantiation lessons my burden on these issues, allowing me to become more focused on solving my business problem.

Now before continuing on what may have been a commercial for the greatest thing since sliced bread, let me point out a few things. Since I only create an object when I need it in my code, the actual object creation happens (typically) at program runtime, not program compile time. This method can certainly introduce some latency into the performance of your program and should be given some consideration. Yet I digress.

Another somewhat sticky aspect to creating programs that use objects is that it causes us to really have to adopt a different approach (at least for most of us) as to how we go about solving the problem space within the program space. This raises a number of issues pertaining to design and test that are very different from using conventional programming methodologies. Your experience in this endeavor will attest to this to whatever degree is normal for you.

One thing that stands out is that I need to define the object before I use it. In our example we will now walk through creation of the "Person" object. For this example I have chosen to have the "Person" object and the consumer of this object in the same program, separated by a package. I mention this because the whole point of creating object is so you can resuse them. You would typically want to employ your object in the form of a module or at least a separate file that others could use.

Example

PERSON.pl

Constructor, blessing, and our hash

First, I quantify my data, and the access point we need to instantiate (create) the object.

```
package Person;
sub new
{
  my $self  = {};
  $self->{FIRSTNAME}     = " ";
  $self->{MIDDLENAME}    = " ";
  $self->{LASTNAME}      = " ";
  $self->{DOB}           = " ";
  $self->{SSN}           = " ";
  $self->{FULLNAME}      = " ";
  bless($self); # me into existence
  return $self;
}
```

I am going to do my very best now in keeping focused on explaining how to use hashes as objects, and not become pulled into a vortex that causes this to become a module on OOP using Perl.

First, examine the hash that I have created. Nothing is new here. Note that I did initialize the hash values to a space. I did this to eliminate warnings that could be issued by the compiler if a key is not initialized by the consumer. Also, since the goal of this package is to create an object I named the anonymous reference to the hash $self. This or $me or $this are all appropriate semantic names as they better reflect what we will be creating an instance of.

Next, we need to look at the stuff that uniquely transforms "person" into an object. Before we look at the details, here is some important verbage and information you need to assimilate.

- An object is created by creating an instance of a class.
- A class in Perl is a package that has been blessed.
- Methods are subroutines that have association to a class.

We use the bless( ) function to be able to (well) bless $self into existence when the time arises. Also, you will see for the sake of consistency that the name of the method we are going to use as our constructor is "new". If you are coming from another OOP language be aware that the name here is not confined to "new". We could call the method we are going to bless "skippy" if we really wanted to. My personal bias on this matter is that since a class has package scope (and hence containment of naming within the symbol table) that you should name your constructor "new" unless you have a compelling reason to do otherwise. Why? In doing so you set a standard for the consumer who wants to create an object. All they now have to be aware of is the name of the package (class).

Oh sorry, some of you may be wondering what a constructor is. A constructor is a special type of method that is automatically invoked when there is a reference to our class (package) from a consumer. It is the method (subroutine) that is going to allow our object to spring into existence.

Class methods, using a class, creating an object

Now that we have discussed our constructor, our hash that defines "person", and how we use bless within a constructor we can look at the other methods in our class.

```
sub firstname
{
  my $self = shift;
  if (@_) { $self->{FIRSTNAME} = shift }
  return $self->{FIRSTNAME};
}
sub middlename
{
  my $self = shift;
  if (@_) { $self->{MIDDLENAME} = shift }
  return $self->{MIDDLENAME};
}
sub lastname
{
```

```
  my $self = shift;
  if (@_) { $self->{LASTNAME} = shift }
  return $self->{LASTNAME};
}
sub fullname
{
  my $self = shift;
  #print $self;
  $self->{FULLNAME} = %$self->{FIRSTNAME} . " " . %$self->{LASTNAME};
  #print %$self->{FULLNAME};
  #print "\n";
  if (@_) { $self->{FULLNAME} = shift }
  return $self->{FULLNAME};
}
sub dob
{
  my $self = shift;
  if (@_) { $self->{DOB} = shift }
  return $self->{DOB};
}
sub ssn
{
  my $self = shift;
  if (@_) { $self->{SSN} = shift }
  return $self->{SSN};
}1;
```

Before we look at our methods, it is time to see how we spring an object into existence.

```
my $me  = Person->new();
$me->firstname("Joseph");
$me->lastname("Momma");
$me->ssn("245-46-9088");
$me->dob("10-16-1947");
```

The following line is used to create an instance of "Person". We create a reference, in this case $me, and assign it to the name of our class (package) referencing our constructor. Hopefully, you can see why I suggest you always name you constructor "new" unless you have a compelling reason to do otherwise.

```
my $me  = Person->new();
```

Once we have created an instance of "Person" we are able to access any of the methods and data members. As a side note for people that already have background in OOP I want to point out that Perl does not really offer true encapsulation in the strictest sense of the term. What I mean by this is that the consumer has full access to any of the data and methods (remember this is just a fancy OOP term for a subroutine). There is no real concept of public and private and all of that other jazz. Perl's goals have never been about hiding or restriction and probably never will be (at least as long as Larry Wall is breathing).

Here is an example of how we access the "lastname" method.

```
$me->lastname("Momma");
```

Lets take a gander at the code in this method and discuss some of the finer points.

```
sub lastname
{
  my $self = shift;
  if (@_) { $self->{LASTNAME} = shift }
  return $self->{LASTNAME};
}
```

First, we create a reference to our anonymous hash. If we were to print out the value of $self we would get something like…

```
Person=HASH(0x1b9ef7c)
```

Make sure you note that we are declaring a new scalar in this method. We are NOT referring to the one we created in the method named "sub"! I have opted to assign this variable to shift. This is a terse form that really means…

```
Shift @_
```

@_ actually contains two elements. Element 0 is the name of the class (package), and element 1 is the reference to the hash we created in "sub". This is why when we hit the statement…

```
if (@_) { $self->{LASTNAME} = shift }
```

…it will be true and we will be shifting $_[1], the array reference to our assignment statement which is the dereferencer that gets us the value for the lastname key. Don't you just love all this indirection going on? ☺ Finally, our method returns the reference. The entity that created the object could print the reference with a statement like…

```
print "Name: " . $you->lastname
```

This certainly may seem like an awful lot of work and abstraction that could be more easily accomplished using conventional practices. I would say that my agreement or disagreement to that statement would depend on a number of factors. For one, consider economy of scale. Suppose "person" code was needed in a very large scale, highly integrated product. I would say the object oriented solution (of course taking performance also into account) would be a more elegant and manageable solution. At the other end of the spectrum, if this was a one time deal or to be used in a small application in a small shop, this type of solution might be akin to taking your 747 down to the convenience store for a pack of smokes.

I have two more items to finish this topic. First, you can see by the code that we are able to create as many discrete "person" objects as we need.

```
my $me  = Person->new();

my $you  = Person->new();
```

Also as long as we have a reference to the object it will exist. Once we are done with the object Perl will handle all the management and other cleanup that needs to occur. For this reason we do not need to create a destructor like we typically would in other OOP languages. Of course we do not need to create objects to leverage this benefit. Remember from the "References" module that this is a benefit inherent in using references.

Finally, we are able to create and use as many different methods as we need to act upon our object. For example, I have a subroutine name "fullname" that aggregates the other names and could be used for display purposes.

```
sub fullname
```

```
{
  my $self = shift;
  #print $self;
  $self->{FULLNAME} = %$self->{FIRSTNAME} . " " . %$self->{LASTNAME};
  #print %$self->{FULLNAME};
  #print "\n";
  if (@_) { $self->{FULLNAME} = shift }
  return $self->{FULLNAME};
}
```

Hands-on

The method "fullname" does not presently deal with middlename. Enhance this method to deal with displaying fullname based on whether or not middlename exists.