Multi-dimensional hashes in Perl

Every value in a hash in Perl can be a reference to another hash. If used correctly the data structure can behave as a two-dimensional or multi-dimensional hash.

Let's see the following example:

```perl
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper qw(Dumper);

my %grades;
$grades{"Foo Bar"}{Mathematics}   = 97;
$grades{"Foo Bar"}{Literature}    = 67;
$grades{"Peti Bar"}{Literature}   = 88;
$grades{"Peti Bar"}{Mathematics}  = 82;
$grades{"Peti Bar"}{Art}          = 99;

print Dumper \%grades;
print "----------------\n";

foreach my $name (sort keys %grades) {
    foreach my $subject (keys %{ $grades{$name} }) {
        print "$name, $subject: $grades{$name}{$subject}\n";
    }
}
```

Running the above script will generate the following output:

```
$VAR1 = {
          'Peti Bar' => {
                          'Mathematics' => 82,
                          'Art' => 99,
                          'Literature' => 88
                        },
          'Foo Bar' => {
                         'Mathematics' => 97,
                         'Literature' => 67
                       }
        };
----------------
Foo Bar, Mathematics: 97
Foo Bar, Literature: 67
Peti Bar, Mathematics: 82
Peti Bar, Art: 99
Peti Bar, Literature: 88
```

First we printed out all the content of the hash, and then, under the separation line we iterated over the data-structure ourselves to see how we can access the elements. If you are not familiar with the Dumper function of Data::Dumper yet, the $VAR1 at the beginning is just a place-holder variable. We don't need to pay attention to it now.

What is important is that the input of the Dumper function is a reference to a data structure and thus we put a back-slash \ in front of %grades.

The order of the keys is random as hashes do not keep them in any specific order.

Explanation

Let's see the details.

We create a hash called %grades. It is a simple, one dimensional hash that can hold key-value pairs. There is nothing special in it.

The next line: $grades{"Foo Bar"}{Mathematics} = 97;

This creates a key-value pair in the %grades hash where the key is Foo Bar and the value is a reference to another, internal hash. A hash that does not have a name. The only way to access that internal hash is through the reference in the %grades hash. In the internal hash the above line created a single key-value pair. The key is Mathematics and the value is 97.

Unlike in Python, the creation of the internal hash is automatic and it is generally referred to as autovivification.

If we used Data::Dumper to print out the content of %grades right after the first assignment we would see the following:

```
$VAR1 = {
          'Foo Bar' => {
```

```
                              'Mathematics' => 97
                            }
        };
```

In this output the outer pair of curly braces represent the %grades hash while the inner pair of curly braces represents the other hash.

Creating the third hash

Running the code further and printing out the content of the hash after 3 assignments:

```perl
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper qw(Dumper);

my %grades;
$grades{"Foo Bar"}{Mathematics}   = 97;
$grades{"Foo Bar"}{Literature}    = 67;
$grades{"Peti Bar"}{Literature}   = 88;

print Dumper \%grades;
```

we get the following:

```
$VAR1 = {
          'Peti Bar' => {
                           'Literature' => 88
                         },
          'Foo Bar' => {
                          'Literature' => 67,
                          'Mathematics' => 97
                        }
        };
```

Here we can see the outer pair of curly braces and two internal pairs representing a total of three separate hashes.

Here we can observe that the hash is not "balanced" or "symmetrical". Each hash has its own keys and their respective values. The same keys can appear in both hashes, but they don't have to.

Traversing the hashes manually

While Data::Dumper can be useful for debugging purposes, we would not want to use it to show the content of the data structure to users. So let's see how can we go over all the keys and values of this 2-dimensional hash.

keys %grades will return the keys of the %grades which are "Peti Bar" and "Foo Bar" in random order. sort keys %grades will return them sorted.

So in each iteration of the outer foreach loop, $name will contain either "Peti Bar" or "Foo Bar".

If we printed the respective values in %grades like this:

```
foreach my $name (sort keys %grades) {
    print "$grades{$name}\n";
}
```

we would get output like this:

```
HASH(0x7f8e42003468)

HASH(0x7f8e42802c20)
```

These are the "user friendly" representations of the references to the "internal" hashes.

Wrapping each such reference within %{ } will de-reference them. This expression represents the internal hash:

%{ $grades{$name} }

If we call the keys function passing this as an argument we will get back the keys of the internal hash. When $name contains "Peti Bar" this will be 'Mathematics', 'Art', and 'Literature'. For "Foo Bar" this will be 'Mathematics', and 'Literature' only.

The $subject variable from the internal foreach loop will get these values and we arrive to the familiar construct $grades{$name}{$subject} to fetch the actual grades of these two students.

Less than two dimensions

In this example we saw a 2-dimensional hash but Perl has no requirement and restrictions. We could have another entry in the %grades hash that does not have a second dimension like this:

$grades{Joe} = 'absent';

Here the Joe key does not have a second dimension. This would work almost flawlessly:

The output would look like this:

```
$VAR1 = {
          'Peti Bar' => {
                          'Literature' => 88,
                          'Mathematics' => 82,
                          'Art' => 99
                        },
          'Foo Bar' => {
                         'Mathematics' => 97,
                         'Literature' => 67
                       },
          'Joe' => 'absent'
        };
----------------
HASH(0x7fabf8803468)
Foo Bar, Mathematics: 97
Foo Bar, Literature: 67
absent
Can't use string ("absent") as a HASH ref while "strict refs" in use a
t files/hash.pl line 20.
```

Data::Dumper could show the data structure properly (Joe does not have curly braces as he does not have an internal hash), but when we traversed the code we got an exception. use strict has stopped us from using a string ('absent') as a symbolic reference. Which is a good thing.

More than two dimensions

We could also have more than two dimensions. For example 'Foo Bar' might also have learned 'Art' and she got grades for several sub-subjects:

```
$grades{"Foo Bar"}{Art}{drawing}    = 34;
$grades{"Foo Bar"}{Art}{theory}     = 47;
$grades{"Foo Bar"}{Art}{sculpture}  = 68;
```

For now we have removed the entry of Joe, so that won't disturb us. The output looks like this:

```
VAR1 = {
          'Peti Bar' => {
                          'Mathematics' => 82,
                          'Art' => 99,
                          'Literature' => 88
                        },
          'Foo Bar' => {
                         'Art' => {
                                    'sculpture' => 68,
                                    'theory' => 47,
                                    'drawing' => 34
                                  },
                         'Mathematics' => 97,
                         'Literature' => 67
                       }
        };
----------------
Foo Bar, Art: HASH(0x7fbe9a027d40)
Foo Bar, Mathematics: 97
Foo Bar, Literature: 67
Peti Bar, Mathematics: 82
Peti Bar, Art: 99

Peti Bar, Literature: 88
```

Data::Dumper still works well and shows the extra internal hash as the value of Art.

Unfortunately our manual traversing does not work well again, but at least this time it does not throw an exception. It "only" prints the reference to the internal data structure of Art of "Foo Bar".

As we can see here, there are places where the data structure has 2 dimension and places where it has 3 dimensions.

We can have even more mixed dimensions.

Mixed dimensions

What if 'Foo Bar' might have another subject called 'Programming' where she has grades for each exercise. Here the grades don't have names, they are numbered 0, 1, 2, 3 etc. We could an internal hash to represent these and we would have 0, 1, 2, 3, etc. as keys but probably it is better to hold them in an array:

```
$grades{"Foo Bar"}{Programming}[0]  = 90;
$grades{"Foo Bar"}{Programming}[1]  = 91;
$grades{"Foo Bar"}{Programming}[2]  = 92;
```

In the output Data::Dumper will represent the array using square brackets:

```
$VAR1 = {

          'Foo Bar' => {

                         'Literature' => 67,

                         'Programming' => [

                                            90,

                                            91,

                                            92

                                          ],

                         'Mathematics' => 97,

                         'Art' => {

                                    'sculpture' => 68,

                                    'theory' => 47,
                                               'drawing' => 34
                                             }
                        },
          'Peti Bar' => {
                            'Mathematics' => 82,
                            'Art' => 99,
                            'Literature' => 88
                          }
        };
----------------
Foo Bar, Literature: 67
Foo Bar, Programming: ARRAY(0x7fc409028508)
Foo Bar, Mathematics: 97
Foo Bar, Art: HASH(0x7fc409027d40)
Peti Bar, Mathematics: 82
Peti Bar, Art: 99
Peti Bar, Literature: 88
```

And our own traversing cannot handle this either, but this time, instead of HASH(0x7fc409027d40) it prints ARRAY(0x7fc409028508) as this is a reference to an array and not to a hash.

Checking if a key exists

Given a multi-dimensional hash like %grades one can check the existance of a key using the existskeyword:

```
if (exists $grades{"Foo Bar"}) {
    if (exists $grades{"Foo Bar"}{Programming}) {
        ...
    }
}
```

One should also be careful using the second-level construct without trying the first-level first as that might trigger unwanted autovivification.

```perl
if (exists $grades{"Foo Bar"}{Programming}) {
}
```