Multi dimensional arrays in Perl

Data::Dumper ARRAY

Prev Next

Technically speaking there are no multi-dimensional arrays in Perl, but you can use arrays in Perl to act as if they had more than one dimension.

In Perl each element of an array can be a **reference** to another array, but syntactically they would look like a two-dimensional array.

Creating a matrix in Perl

Let's see the following code:

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

5. my @matrix;

   $matrix[0][0] = 'zero-zero';
   $matrix[1][1] = 'one-one';
   $matrix[1][2] = 'one-two';
```

We just created an array called `@matrix`. It is a regular one-dimensional array in Perl, but we accessed it as if it was two dimensional.

What do you think the following line will do?

```
1. print "$matrix\n";
```

I know, it was a trick question. The program won't even compile. You will get the following error:

```
Global symbol "$matrix" requires explicit package name at ... line ..
Execution of ... aborted due to compilation errors.
```

If you read about the global symbol requires explicit package nameerror, you will see that it basically means you have not declared a variable. In this case `$matrix`. Indeed, if you read how to access array elements in Perl, you will see that you'd access the first element of the `@matrix` by using `$matrix[0]`. Notice the square brackets after the variable name!

There are 3 things here that can be a bit confusing:

`@matrix`, `$matrix[0]` and `$matrix`. The first two are related. The third is unrelated. The first one is an array. The second one is an element of an array and the third one is an **unrelated** scalar. If you declare an array such as `@matrix` you can automatically use `$matrix[0]` to access the first element, but if you'd also like to use `$matrix` you'd need to declare that separately.

That leads us to warning. While Perl is OK with you having the exact same variable name for an array and for a scalar, it is strongly recommended you don't have them in the same code. It can just confuse the reader.

After this short detour, let's go back to our example:

Matrix

Let's see what would the following print:

```
1. print "$matrix[0]\n";       # ARRAY(0x814dd90)
   print "$matrix[0][0]\n";    # zero-zero
   print "$matrix[1][1]\n";    # one-one
```

The first line prints **ARRAY(0x814dd90)**. As I mentioned, Perl does not have multi-dimensional arrays. What you see here is that the first element of the `@matrix` array is a **reference** to an internal, so-called anonymous array that holds the actual values. The **ARRAY(0x814dd90)**is the address of that internal address in the memory. You can't do much with this, except of knowing that you probably need to "de-reference" that address. In our case that de-referencing is done by the addition of another pair of square brackets.

That way you can get back the original values we put in the array.

Visualizing a multi-dimensional array

There is a module called Data::Dumper, which comes with Perl and that can provide a reasonably readable view of the matrix we created. In order to use it, you first need to load it to memory with the `use` statement. Then calling the `Dumper` function and passing a **reference**to it. The back-slash `\`, just before the `@matrix`, creates a reference to the array. The `Dumper` function serializes the data structure and returns a string, which is then printed by the `print` function.

```
1. use Data::Dumper qw(Dumper);
   print Dumper \@matrix;
```

The output will look like this:

```
$VAR1 = [
          [
            'zero-zero'
          ],
          [
            undef,
            'one-one',
            'one-two'
          ]
        ];
```

The `$VAR1` at the beginning is just a standard name Data::Dumper uses. You can disregard it for now. the rest of the output shows 3 pairs of square brackets. The outermost pair represents the main array we call `@matrix`. the first internal pair holds a single value (zero-zero). This represents the first row in the matrix. The second internal pair has 3 values. The first one is undef, this is the place of `$matrix[1][0]` where we have not assigned a value. The other two we assigned.

Two dimensional array or what?

As you can see this resembles a **two dimensional array**, but its shape is not rectangular, as you would expect from a matrix. The first row has only one element while the second row has 3. (even if one of them is undef).

In a similar way there could be elements in the `@matrix` array that have not other dimension. For example we could write:

```
1. $matrix[2] = 'two';
```

which would change the output of Data::Dumper to this:

```
$VAR1 = [
          [
            'zero-zero'
          ],
          [
            undef,
            'one-one',
            'one-two'
          ],
          'two'
        ];
```

Here the outer array has 3 elements. The first two are the internal arrays, and the 3rd one is a simple scalar.

So one of the "rows" in the "matrix" does not even have a dimension.

More than 2 dimensions?

What if we add the following code?

```
1. $matrix[1][3][0] = 130;
   $matrix[1][3][1] = 131;
```

The Dumper output will look like this:

```
$VAR1 = [
          [
            'zero-zero'
          ],
          [
            undef,
            'one-one',
            'one-two',
            [
              130,
              131
            ]
          ],
          'two'
        ];
```

Look, the second internal array now has a 4th element and that element itself, is an array (or rather a reference to an array).

Conclusion

An array in Perl can have any number of "dimensions" and it does not need to form a "regular" shape. Each element can have an internal array. And each element of the internal array can have its own internal array and so on.

Data::Dumper can help us figure out what is in such a data structure.