Array references in Perl

@ \@ $# $#$ @{ } ARRAY(0x703dcf2) Data::Dumper

Prev Next

In this part of the Perl Tutorial we are going to learn about **array references**.

There are two main uses of array references. One is to make it easy to pass more than one arrays to a subroutine, the other is to build arrays of arrays or other multi-dimensional data structures.

Creating a reference to a Perl array

If we have an array called `@names`, we can create a reference to the array using a back-slash `\` in-front of the variable: `my $names_ref = \@names;`.

We use the _ref extension so it will stand out for us that we expect to have a reference in that scalar. This is not a requirement and Perl does not care, but it might be useful while learning about them.

If we now call `print $names_ref;` we'll see the following:

```
ARRAY(0x703dcf2)
```

That is the address of the `@names` array in memory with the clear notion that it is the location of an ARRAY.

The only thing you can do with an array reference, is to get back the original array.

(If you are a C programmer, don't even think about pointer arithmetic. You can't do that in Perl.)

Basically, if you see such value printed somewhere, you know that the code is accessing a reference to an array and that you should probably change the code to access the content of that array.

Dereferencing an array

If you have a reference to an array and if you would like to access the content of the array you need to **dereference** the **array reference**. It is done by placing the `@` symbol (the sigil representing arrays) in-front of the reference.

This can be written either wrapped in curly braces: `@{$names_ref}` or without the curly braces: `@$names_ref`.

You could also put spaces around the name and write: `@{ $names_ref }`. This usually makes things nicer, and more readable.

You can then use this construct to access the array. For example: `print "@{ $names_ref }";`

A full script might look like this:

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

my @names = qw(Foo Bar Baz);

my $names_ref  = \@names;
print "$names_ref\n";          # ARRAY(0x703dcf2)

print "@$names_ref\n";         # Foo Bar Baz
print "@{ $names_ref }\n";     # Foo Bar Baz
```

Individual elements

If we have a reference to an array we can also easily access the individual elements.

If we have the array `@names` we access the first element using `$names[0]`. That is, we replace the `@`-sign with a `$`-sign and put the index in square brackets after the name.

With references we do the same. If we have the array reference `$names_ref` then the original array is represented by `@{$names_ref}`. Replace the `@` by the `$` and put the index after the thing in square brackets. `${$names_ref}[0]` or `$$names_ref[0]` if you like the brace less style.

Unfortunately both of these are a bit hard to read, but luckily Perl provides another, much clearer syntax for this: `$names_ref->[0]`. In this code, on one hand we eliminated the double `$` signs and on the other hand we represent the dereferencing by a simple arrow.

That's about it learning the basics of array references in Perl.

Array references cheat sheet

Given an array `@names` and an array reference called `$names_ref` that was created using `my $names_ref = \@names;` the following table shows how can we access the whole array, individual elements of an array and the length of an array in its normal array representation and the corresponding array reference representation:

```
                    Array                Array Reference
Whole array:        @names               @{ $names_ref }
Element of array:   $names[0]            ${ $names_ref }[0]
                                         $names_ref->[0]

Size of array:      scalar @names        scalar @$names_ref
Largest index:      $#names              $#$names_ref
```

But you really don't want to use `$#$names_ref` among people....

Data::Dumper for debugging

When you have an array or an array reference, the best way to visualize it during a debug-by-print session is by using one of the data dumper modules, for example the built-in Data::Dumper module.

```
1. use strict;
   use warnings;
   use Data::Dumper;

5. my @names = ('foo', 'bar', "moo\nand\nmoose");

   print Dumper \@names;
```

And the output is:

```
$VAR1 = [
          'foo',
          'bar',
          'moo
and
moose'
        ];
```

This clearly shows the individual elements even if some of the elements have spaces or newlines embedded in them.

Exactly the same could be done if we already had an array reference in our hand:

```
1. use strict;
   use warnings;
   use Data::Dumper;

5. my @names = ('foo', 'bar', "moo\nand\nmoose");
   my $names_ref = \@names;
```

```
print Dumper $names_ref;
```

Passing two arrays to a function

If you want to write a function that gets two or more arrays you have to use references. Let's say you'd like to write a function that adds the elements of two arrays, pair-wise.

If you call `add(@first, @second)`, on the receiving end the two arrays will be flattened together into `@_` and you won't be able to tell them apart.

Better to pass two reference like this: `add(\@first, \@second)` and then de-reference them inside the function:

```
1. sub add {
     my ($first_ref, $second_ref) = @_;
     ...
   }
```