Perl Arrays

@ array arrays length size foreach Data::Dumper scalar push pop shift $#

Prev Next

In this episode of the Perl Tutorial we are going to learn about **arrays in Perl**. This is an overview of how arrays work in Perl. We'll see more detailed explanations later.

Variable names of arrays in Perl start with the at mark: `@`.

Due to our insistence on using `strict` you have to declare these variables using the `my` keyword before the first usage.

Remember, all the examples below assume your file starts with

```
use strict;
use warnings;
use 5.010;
```

Declare an array:

```
my @names;
```

Declare and assign values:

```
my @names = ("Foo", "Bar", "Baz");
```

Debugging of an array

```
use Data::Dumper qw(Dumper);

my @names = ("Foo", "Bar", "Baz");
say Dumper \@names;
```

The output is:

```
$VAR1 = [
          'Foo',
          'Bar',
          'Baz'
        ];
```

foreach loop and perl arrays

```
my @names = ("Foo", "Bar", "Baz");
foreach my $n (@names) {
  say $n;
}
```

will print:

```
Foo
Bar
Baz
```

Accessing an element of an array

```
my @names = ("Foo", "Bar", "Baz");
say $names[0];
```

Note, when accessing a single element of an array the leading sigil changes from `@` to `$`. This might cause confusion to some people, but if you think about it, it is quite obvious why.

`@` marks plural and `$` marks singular. When accessing a single element of an array it behaves just as a regular scalar variable.

Indexing array

The indexes of an array start from 0. The largest index is always in the variable called `$#name_of_the_array`. So

```
my @names = ("Foo", "Bar", "Baz");
say $#names;
```

Will print 2 because the indexes are 0,1 and 2.

Length or size of an array

In Perl there is no special function to fetch the size of an array, but there are several ways to obtain that value. For one, the size of the array is one more than the largest index. In the above case `$#names+1` is the **size** or **length** of the array.

In addition the `scalar` function can be used to to obtain the size of an array:

```
my @names = ("Foo", "Bar", "Baz");
say scalar @names;
```

Will print 3.

The scalar function is sort of a casting function that - among other things - converts an array to a scalar. Due to an arbitrary, but clever decision this conversion yields the size of the array.

Loop on the indexes of an array

There are cases when looping over the values of an array is not enough. We might need both the value and the index of that value. In that case we need to loop over the indexes, and obtain the values using the indexes:

```
my @names = ("Foo", "Bar", "Baz");
foreach my $i (0 .. $#names) {
  say "$i - $names[$i]";
}
```

prints:

```
0 - Foo
1 - Bar
2 - Baz
```

Push on Perl array

`push` appends a new value to the end of the array, extending it:

```
my @names = ("Foo", "Bar", "Baz");
push @names, 'Moo';

say Dumper \@names;
```

The result is:

```
$VAR1 = [
        'Foo',
        'Bar',
        'Baz',
        'Moo'
      ];
```

Pop from Perl array

`pop` fetches the last element from the array:

```perl
my @names = ("Foo", "Bar", "Baz");
my $last_value = pop @names;
say "Last: $last_value";
say Dumper \@names;
```

The result is:

```
Last: Baz
$VAR1 = [
        'Foo',
        'Bar',
      ];
```

shift the Perl array

`shift` will return the left most element of an array and move all the other elements to the left.

```perl
my @names = ("Foo", "Bar", "Baz");

my $first_value = shift @names;
say "First: $first_value";
say Dumper \@names;
```

The result is:

```
First: Foo
$VAR1 = [
        'Bar',
        'Baz',
      ];
```