Hashes in Perl

hash | keys | value | associative | % | => | fat arrow | fat comma

Prev Next

In this article of the Perl Tutorial we are going to learn about **hashes**, one of the powerful parts of Perl.

Some times called associative arrays, dictionaries, or maps; hashes are one of the data structures available in Perl.

A hash is an un-ordered group of key-value pairs. The keys are unique strings. The values are scalar values. Each value can be either a number, a string, or a reference. We'll learn about references later.

Hashes, like other Perl variables, are declared using the `my` keyword. The variable name is preceded by the percentage (`%`) sign.

It's a little mnemonic trick to help you remind about the key-value structure.

Some people think that hashes are like arrays (the old name 'associative array' also indicates this, and in some other languages, such as PHP, there is no difference between arrays and hashes.), but there are two major differences between arrays and hashes. Arrays are ordered, and you access an element of an array using its numerical index. Hashes are un-ordered and you access a value using a key which is a string.

Each hash key is associated with a single **value** and the keys are all unique inside a single hash structure. That means no repetitive keys are allowed. (If you really, really want to have more than one values for a key, you'll will need to wait a bit till we reach the references.)

Let's see some code now:

Create an empty hash

```
1. my %color_of;
```

Insert a key-value pair into a hash

In this case 'apple' is the key and 'red' is the associated value.

```
1. $color_of{'apple'} = 'red';
```

You can also use a variable instead of the key and then you don't need to put the variable in quotes:

```
my $fruit = 'apple';
$color_of{$fruit} = 'red';
```

Actually, if the key is a simple string, you could leave out the quotes even when you use the string directly:

```
$color_of{apple} = 'red';
```

As you can see above, when accessing a specific key-value pair, we used the `$` sign (and not the % sign) because we are accessing a single value which is a **scalar**. The key is placed in curly braces.

Fetch an element of a hash

Quite similar to the way we inserted an element, we can also fetch the value of an element.

```
print $color_of{apple};
```

If the key does not exist, the hash will return an undef, and if `warnings` are enabled, as they should be, then we'll get a warning about uninitialized value.

```
print $color_of{orange};
```

Let's a few more key-value pairs to the hash:

```
$color_of{orange} = "orange";
$color_of{grape} = "purple";
```

Initialize a hash with values

We could have instantiated the variable with the key-value pairs simultaneous passing to the hash a list of key-value pairs:

```
my %color_of = (
      "apple"  => "red",
      "orange" => "orange",
      "grape"  => "purple",
);
```

`=>` is called the **fat arrow** or **fat comma**, and it is used to indicate pairs of elements. The first name, fat arrow, will be clear once we see the the other, thinner arrow (->) used in Perl. The name fat comma comes from the fact that these arrows are basically the same as commas. So we could have written this too:

```
my %color_of = (
      "apple",  "red",
      "orange", "orange",
      "grape",  "purple",
);
```

Actually, the fat comma allows you to leave out the quotes on the left-hand side making the code cleaner and more readable.

```
my %color_of = (
      apple  => "red",
      orange => "orange",
      grape  => "purple",
);
```

Assignment to a hash element

Let's see what happens when we assign another value to an existing key:

```
$color_of{apple} = "green";
print $color_of{apple};     # green
```

The assignment changed the value associated with the **apple** key. Remember, keys are unique and each key has a single value.

Iterating over hashes

In order to access a value in a hash you need to know the key. When the keys of a hash are not pre-defined values you can use the `keys` function to get the list of keys. Then you can iterate over those keys:

```
my @fruits = keys %color_of;
for my $fruit (@fruits) {
    print "The color of '$fruit' is $color_of{$fruit}\n";
}
```

You don't even need to use the temporary variable `@fruits`, you can iterate over directly the return values of the `keys` function:

```
for my $fruit (keys %color_of) {
    print "The color of '$fruit' is $color_of{$fruit}\n";
}
```

The size of a hash

When we say the size of a hash, usually we mean the number of key-value pairs. You can get it by placing the `keys` function in scalar context.

```
1. print scalar keys %hash;
```