

# Learn Perl by Example - Perl Handbook for Beginners - Basics of Perl Scripting Language

linuxreport.org

Copyright © 2006 - 2008 [www.linuxreport.org](http://www.linuxreport.org)

2008/01/29



This course is about Perl Programming Language. It is for beginners and it explain Perl basics in a easy to learn way. If you are a sysadmin and you learn Linux or UNIX this is what you need to be able to write Perl scripts, to know a language every sysadmin must know.

PERL is a powerful scripting language, very popular among UNIX/Linux admins. This tutorials will try to cover everything you need to know in order to program in Perl. Perl stands for Practical Extraction an Report Language, it was first used as text processor, it borrows features from C, shell scripting (UNIX sh), sed, awk, Lisp, Pascal. It can be used also for developing dyamic web applications as CGIs.

This tutorial was provided by <http://www.linuxreport.org>

You may freely distribute this document in any form without changing the text or removing copyright notice.

# Table of Contents

<b>1 Introduction</b> .....	<b>2</b>
<b>2 Perl Variables</b> .....	<b>3</b>
<b>3 Perl control structures</b> .....	<b>7</b>
<b>4 Defining and using subroutines</b> .....	<b>10</b>
<b>5 Using file parameters (positional parameters)</b> .....	<b>11</b>
<b>6 Perl Regular Expressions</b> .....	<b>11</b>
<b>7 About Tutorial</b> .....	<b>15</b>

## 1 Introduction

PERL is a powerful scripting language, very popular among UNIX/Linux admins. This tutorials will try to cover everything you need to know in order to program in Perl. Perl stands for Practical Extraction and Report Language, it was first used as text processor, it borrows features from C, shell scripting (UNIX sh), sed, awk, Lisp, Pascal. It can be used also for developing dynamic web applications as CGIs.

### 1.1 Few things to know before start programming in Perl

Perl code is portable. Most scripts are written for version 5.8 or higher. When start programming in Perl first you might want to find the path of Perl binary. On most UNIX/Linux systems you can do that with whereis command:

#### **whereis perl**

As a result of this command you might have: **/usr/bin/perl**. So your scripts must have first line of code with this value:

```
#!/usr/bin/perl  
# rest of code # will be used for comments.
```

On first line of sourcecode this line will help shell in finding what binary to use when running the script.

In order to properly run the script your source code must also have executable flag for the user you use to run the script. This can be achieved for example for file **progr1.pl** adding executable flag from command line:

**chown u+x progr1.pl**

So, to start your Perl script you will follow:

a) Create an empty file:

**touch program.pl**

b) Add executable flag to that file:

**chown u+x program.pl**

c) Find where your Perl binary is:

**whereis perl**

(you will get something like /usr/bin/perl).

c) Edit that file with your text editor, and add perl path with this syntax: **#!/usr/bin/perl** (not that # on first line of your code will not be seen as comment)

**edit program.pl**

and put there: **#!/usr/bin/perl**.

**Note:** For Linux you can use nano, pico or mcedit. Edit is your default text editor in FreeBSD. If you have installed Midnight Commander package, you can use mcedit, which is nice.

**Note: use strict;** put in perl sourcecode will force us to declare variables in a more safe (proper) way. All variables must be declared with my prefix.

**Note:** By adding **-w** to **#!/usr/bin/perl** it will activate perl warnings, very usefull for debugging.

**#!/usr/bin/perl -w**

## 2 Perl Variables

Perl has 3 types of variables:

- scalars;
- arrays;
- hashes;

### 2.1 Scalars

Example on how to define a scalar variable in Perl:

```
$var1 = "value"      # a scalar variable var1 is defined and a string
                    # "value" is assigned to that variable;
$var2 = 100         # a scalar variable var2 is defined, and an
                    #integer value is assigned.
```

Example: To **print a scalar** value we will use:

```
print "$var1";
```

### 2.2 Arrays

Example on how to **define an array** in Perl:

```
@array1 = ( "Value1", "Value2", Value3");
```

Example on how to **print an array**:

```
print "Our array variable contains: @array1\n";
```

In our example we've used `\n` escape char to insert a new line (escape chars can be used the same way are used in C language).

The previous example will display all values from **array1** array.

To display one element of the array:

```
print "First element of the array is: $array1[0]";
```

As you might notice we've defined array with @ but printed a single value of that array using \$. This is correct because we want to print a single value.

It is also possible to print multiple values from an array:

```
print "Our array contains: @array1[0..2]";
```

Previous example will print elements from 0 to element nr.2 from array1.

You can also print **multiple distinct elements from array**:

```
print "Our array contains: @array1[0,4,7]";
```

The previous example will print only values for element 0, 4 and 7. Note that in perl first value of an array is number 0.

That is fine but how do we find the **number of elements of an array**?

```
print "Number of elements of an array: $#array1";
```

Note that \$#array1 in our example is number of elements, but because elements from an array in Perl starts with value 0, the real number of elements of an array is \$#array + 1.

There is another method to define an array:

```
@array2 = qw(Value1 Value2 Value3 Value4);
```

Perl functions for working with arrays:

- **pop** - remove last element of an array;
- **push** - add an element to the end of array;
- **shift** - removes first element of an array;
- **unshift** - add an element to the beginning of array;

- **sort** - sort an array.

Let's see some examples:

**Pop Function** (remove last element of an array):

```
#!/usr/bin/perl -w

@array1 = ("Data1", "Data2", "Data3");

print "Array1 values: @array1[0..$#array1]\n";

pop @array1;

print "Array1 after applying pop function: @array1[0..$#array1]\n";
```

**Push Function** (add an element to the end of array):

```
#!/usr/bin/perl -w

@array1 = ("Data1", "Data2", "Data3");

print "Array1 values: @array1[0..$#array1]\n";

push @array1, "Data4";

print "Array1 after applying push function: @array1[0..$#array1]\n";
```

**Shift Function** (removes first element of an array):

```
#!/usr/bin/perl -w

@array1 = ("Data1", "Data2", "Data3");

print "Array1 values: @array1[0..$#array1]\n";

shift @array1;

print "Array1 after applying shift function: @array1[0..$#array1]\n";
```

The same principle apply for unshift and sort functions. Sort functions works best with strings.

## 2.3 Hashes

Hashes are types of variables defined as key - value pair.

Example of defining a hash variable:

```
%name_email = ("John", "john@example.com" , "George", "george@example.com");
```

Another way to define a has variable:

```
%name_email = (  
    John => "john@example.com",  
    George => "george@example.com",  
);
```

Example of using hash variables:

```
#!/usr/bin/perl -w  
  
%name_email = ( "John", "john@example.com", "George", "george@example.com");  
  
print $name_email{"John"};
```

**Note:** Note: We've used escape character to preserver @. Also note that printing a hash variable means to print a scalar with value key between braces {}.

## 3 Perl control structures

### 3.1 Conditionals

For testing conditionals within Perl **if** it is used. To better illustrates, see the following example:

```
#!/usr/bin/perl -w  
  
$var1 = 100;
```

```
$var2 = 200;

if ($var1 < $var2) {
    print "$var1 < $var2\n";
}
```

**Note:** When evaluating expressions if variables are numbers we will use mathematical operators ( < > = <= >= ==). When we use string variables we use string evaluation operators like gt (greater then) eq (equal) and so on.

**Note:** When we evaluate two numbers to be identical, we use == operator (not = which is used for assigning values).

Another example follows:

```
#!/usr/bin/perl -w

$var1 = 400;

$var2 = 200;

if ($var1 < $var2) {
    print "$var1 < $var2\n";
}
elsif ($var1 > var2) {
    print "$var1 > $var2\n";
}
```

**elsif** function as a nested if.

The inverse test of if is **unless** function:

```
unless ($var1 == $var2) {
    print "$var1";
}
```

## **3.2 Loops**

### **3.2.1 For Loops**

In Perl sometimes are many way to solve a problem. We will show 3 ways to construct a loop using **for**.

#### **Example 1: For loop using C style:**

```
#!/usr/bin/perl -w

# for loop example 1

for ($i = 1; $i < 100; $i++) {
    print "$i\n";
}
```

#### **Example 2: for loops using ranges:**

```
#!/usr/bin/perl -w

# for loop example 2
$var1 = 1;
$var2 = 100;
$i = 1;

for ($var1..$var2) {
    print "$i\n";
    $i+=1;
}
```

#### **Example 3: loop using foreach:**

```
#!/usr/bin/perl -w

# for loop example 3
@array1 = ( "Val1", "Val2", "Val3", "Val4", "Val5");

foreach (@array1) {
    print "$_\n";
}
```

**Note:** `$_` will print the current value of an array.

### 3.2.2 While Loops

An example is presented next:

```
#!/usr/bin/perl -w

$var1 = 1;
$var2 = 8;

while ($var1 < $var2) {
    print "$var1\n";
    $var1 += 1;
}
```

### 3.2.3 Until Loops

Until is negation of while. Here is an example:

```
#!/usr/bin/perl -w

$var1 = 1;
$var2 = 8;

until ($var2 < $var1) {
    print "$var2\n";
    $var2 -= 1;
}
```

## 4 Defining and using subroutines

Subroutines allow us to better structure our code, organize it and reuse it.

A subroutine will start with keyword `sub`. The following example shows how to define a subroutine which calculates sum of two numbers:

## *Learn Perl by Example - Perl Handbook for Beginners - Basics of Perl Scripting Language*

```
#!/usr/bin/perl -w

$var1 = 100;
$var2 = 200;
$result = 0;

$result = my_sum();
print "$result\n";

sub my_sum {
    $tmp = $var1 + $var2;
    return $tmp;
}
```

**Note:** Subroutines might have parameters. When passing parameters to subroutines, it will be stored in `@_` array.

Do not confuse it with `$_` which stores elements of an array in a loop.

## **5 Using file parameters (positional parameters)**

Sometimes we need to transmit parameters to our script files.

`@ARGV` is an array reserved for parameters transmitted to files (default value of number of arguments is set -1 if no parameters are transmitted).

```
#!/usr/bin/perl -w

if ($#ARGV < 2) {
    print "You must have at least 3 parameters.\n";
}
else {
    print "Your parameters are: @ARGV[0..$#ARGV]\n";
}
```

## 6 Perl Regular Expressions

Perl Regular Expressions are a strong point of perl. You can ease your sysadmin job by learning and using Perl Regex.

### 6.1 Searching for a string

The following example will search for **This** string in expression \$exp.

```
#!/usr/bin/perl -w

$exp = "This is a string";

if ($exp =~ /This/) {
    print ("String Matches!\n");
}
```

### 6.2 Searching for a string using case insensitive

The next example will search for string **this** in expression \$exp using case insensitive (case will be ignored in search).

```
#!/usr/bin/perl -w

$exp = "This is a string";

if ($exp =~ /tHis/i) {
    print ("String Matches!\n");
}
```

### 6.3 Searching for a digit

The next example shows how to search for a digit in a string.

```
#!/usr/bin/perl -w

$exp = "This is 8 string";

if ($exp =~ /\d/) {
    print ("String Matches!\n");
}
```

## **6.4 Searching for 2 digits**

The next example shows how to search for two digits in a string.

```
#!/usr/bin/perl -w

$exp = "This is 88 string";

if ($exp =~ /\d\d/) {
    print ("String Matches!\n");
}
```

## **6.5 Searching for whitespaces**

The next example shows you how to use regular expressions to search for whitespaces in a string.

```
#!/usr/bin/perl -w

$exp = "This is string";

if ($exp =~ /\s/) {
    print ("String Matches!\n");
}
```

## **6.6 Searching for a string that begins with a pattern**

The following example shows you how to use regular expressions to check if a string begins with a keyword/string.

```
#!/usr/bin/perl -w

$exp = "This is string";

if ($exp =~ /^This/) {
    print ("String Matches!\n");
}
```

## **6.7 Searching for a string that ends with a pattern**

The following example shows you how to use regular expressions to check if a string ends with a keyword/string.

```
#!/usr/bin/perl -w

$exp = "This is string";

if ($exp =~ /string$/) {
    print ("String Matches!\n");
}
```

## **6.8 Search for a digit with white space in front and after it**

The next example shows how to use perl regex to search for a digit with white space in front and after it.

```
#!/usr/bin/perl -w

$exp = "This 1 is string";

if ($exp =~ /\s\d\s/) {
    print ("String Matches!\n");
}
```

## **6.9 Search for a blank line**

The next example shows how to use Perl regex to search for a blank line.

```
#!/usr/bin/perl -w

$exp = "";

if ($exp =~ /^$/) {
    print ("String Matches!\n");
}
```

## **6.10 Replace a pattern**

The next example shows you how to use Perl regex to replace a text with a pattern.

```
#!/usr/bin/perl -w

$exp = "This is a string";
```

*Learn Perl by Example - Perl Handbook for Beginners - Basics of Perl Scripting Language*

```
if ($exp =~ s/This is/Test/) {  
    print ("$exp\n");  
}
```

## **7 About Tutorial**

This tutorial was provided by <http://www.linuxreport.org>