Table of Contents

1 Document Overview

Author: NETIKUS.NET ltd
Date: 19th Nov 2000
Revision: 1.1

Easy PERL Starter

Title	Easy PERL Starter
Summary	This is a crash course for those who want to get started in PERL in one evening. This is not a complete introduction but a short and efficient introduction.
Software	Perl 5
Skill Level	Beginners
Skills Required	- Basic understanding of Programming
Download	http://www.netikus.net/ (guides section)

2 What is Perl?

Perl and Activestate
PERL originates somewhere in the UNIX world and has been ported to the windows platform (WIN32) a while ago by a company now called ACTIVESTATE.

A Scripting Language
PERL is a scripting language and is known for its ability to process text very fast. Nowadays PERL can handle almost every task from network communication, database connections and so on.

Modules
This has been made possible by developing modules (mostly in C / C++) for PERL. Modules are separate files usually consisting of a .pm and/or a .pll file and available from CPAN (comprehensive Perl Archive network) and other sources (see links at the end for more).

"Active" Perl?
ActivePerl is the merge of the 2 previous available PERL distributions (CORE and ACTIVEWARE). You can download it at www.activestate.com for your platform.

What is the difference to other languages
Compared with Visual Basic and C I think PERL relates more to C. Longer scripts can easily look very confusing if you don't comment and format them properly. Perl is CASE SENSITIVE and hence not very tolerant with (typing) mistakes ;-(

What can I use Perl for?
Most people use PERL for either automating administrative tasks (write a .pl file and launch it with PERL.EXE) or in combination with a webserver to create dynamice web pages. Most UNIX based webservers have PERL support already integrated (like APACHE) whereas you have to install PERL on WIN32 platforms like Windows NT (IIS 4).

Installation
After downloading PERL just run the setup and by default you will find it in the c:\perl directory. PERL should now be associated with the .PL extension and you can run your first scripts by simply typing PERL myscript.pl.

3 Getting Started with Simple Examples

Create a new file in your favourite TEXT editor (I recommend Ultraedit) and create an empty text file myscript.pl (you can use another name if you are more creative...).

3.1 Output and static variables

```
1: $myname = "Heathcliff Hudginston";
2: $myageg = 15;
3:
4: print "My name is $myname and I am $myage years old.\n";
```

This will create an output like this:

```
My name is Heathcliff Hudginston and I am 15 years old
```

Pretty simple, hmmm?

*) Variables are declared with a $ and you print your output with print.
*) If you want to print a new line just write \n, for a tab write \t.

Now let's see how we can modify variables and read data from the command line:

3.2 Modifying variables

```
1: $firstname = "Heathcliff";
2: $lastname = "Hudginston";
3: # PROMPT FOR INPUT
4: print "Your age please $firstname $lastname: ";
5: $myage = <STDIN>;
6:
7: # REMOVE NEW LINE
8: chomp($myage);
9:
10:# INCREASE BY ONE
11:$myage++;
12:
13:# OUTPUT
14:print "\n\nHello $firstname,\n";
15:print "You will turn $myage on your next birthday.";
```

This should look like this:

```
Your age please Heathcliff Hudginston: 65
Hello Heathcliff,
You will turn 66 on your next birthday.
```

Allright, now what is <STDIN> and why do we use chomp ?

*) <STDIN> (line 8) prompts for input and stores the value in the according variable
*) chomp (line 11) removes the new line character that was appended to the variable $myage when we hit the ENTER button on the keyboard
*) $myage++ is the same as $myage = $myage + 1. You use this with other operators like - and so on as well.

3.3 Sophisticated variables: arrays

One of the most interesting and useful variables are arrays and hashes. The are single variables but contain numerous values - with hashes they can even have a relation to each other. Please note that when accessing arrays by number the first element in an array always starts with 0.
To initialize an array write:

```
@myarray = ("red","green","black"); or
@myarray = qw(red green black); or
$myarray[0] = "red";
$myarray[1] = "green";
$myarray[2] = "black";
```

To use this array I have a little example once again which could be used in a CGI environment:

```
1: @colors = qw(green red orange blue black);
2:
3: for (sort @colors)
4:    { print "<FONT COLOR=\"$_\">$_</FONT><BR>\n" }
5: print "<BR>There are $#colors colors available";
```

And this is what we see:

```
<FONT COLOR="black">black</FONT><BR>
<FONT COLOR="blue">blue</FONT><BR>
<FONT COLOR="green">green</FONT><BR>
<FONT COLOR="orange">orange</FONT><BR>
<FONT COLOR="red">red</FONT><BR>
<BR>There are 5 colors available
```

Line 6 just iterates through all entries in the @colors array and line 7 creates a print command that produces our desired output. \" is used instead of " since a single quotation mark would end our print command. Note the **sort** in line 3 which sorts the output alphabetically. If you remove the **sort**, the output would be in the original order (green, red ...)

*) A preceeding backslash introduces a special character to PERL, like " $
*) The variable $_ holds the current entry in the array which is also true for other loops
*) **$#array** returns the number of elements in an array

3.4 Sophisticated variables: hashes

Hashes are a little more complex but can be more useful if needed. A Hash consists not only of one but two values each - a KEY and a VALUE.
To initialize a hash write:

```
%myhash = (
            "Name" => "Ingmar Köcher",
            "Hobby" => "Nothing and everything",
            "shoe size" => 10
       );
```

or

```
$myhash{'Name'} = "Ingmar Köcher";
$myhash{'Hobby'} = "Nothing and everything";
$myhash{'shoe size'} = 10;
```

and now an example of how to make use of this:

```
1: %myhash = ("Name" => "Ingmar Köcher","Hobby" => "Nothing and
everything", "shoe size" => 10);
2:
3: foreach $key (keys %myhash)
4: {
5:     print $key.": ".$myhash{$key}."\n";
6: }
```

This will show up like:

```
Name: Ingmar KÖCHER
Hobby: Nothing and everything
shoe size: 10
```

Here we make use of the keys feature by iterating through all items in the hash. After the key is assigned to $key, we can then retrieve the value by printing $myhash{$key}.

4 Operators

Now you know enough about PERL to make little useless programs, but we hopefully want more to give our life a sense of automatism. See what PERL has to offer about operators:

4.1 Equality Operators

Strings: To compare string values, you would either use **eq** or **ne**. Example:

```
if ($password eq "Quicksie") { print "PWD OK" }
if ($password ne "Quicksie") { print "ACCESS DENIED" }
```

Of course it would make more sense to use else here, but for the sake of examples be happy with this solution.

Numbers: To compare numbers, use <, <=, >, >=, <> and ==.

Example:

```
if ($number > 10) { print "Wow ..." }
if ($number == 3) { print "You are correct" }
```

4.2 If...then...else...elsif

If we want to extend that previous example a little bit by defining multiple conditions with elsif, we can use them here:

```
$number = 15;
if ($number < 10) { print "10" }
elsif ($number < 20) { print "20" }
else { print "Thats ok" }

20
```

Now that was easy - and we already learned about the if-then-else statement. There is really not much to add here, except that we can make it a little shorter and easier too if it's not too complex:

```
$pwd = "dontlikeperl";
print "PWD WRONG YOU LOOSER" if $pwd ne "WeLovePerl!";
```

Makes sense, right ? But to make it complete, let's print another message if the password is correct !

```
$pwd eq "Superstar!" ? $output="YOU GOT IT MAN" : $output="PWD WRONG YOU
LOOSER";
print $output;
```

Looks complicated ? Not at all. The evaluation is marked blue and if it is true, the code after the question marked will be executed. If not, the code ofter the colon will be executed. So if your if block only needs to do one statement, use "? :"

4.3 For loop

To loop through values, use the for loop. Here a small example that explains it all:

```
for ($counter = 0; $counter <= 10; $counter++)
    { print $counter.","  }

0,1,2,3,4,5,6,7,8,9,10,
```

Makes sense. **$counter++** means, that **$counter** will be increased one by one and is short for $counter = $counter + 1. You can also use it with a minus, making it **$counter- -** . But you can do a little bit more while you are in a loop. Place any of the commands into the loop and you con do this:

last Exit the loop immediately
redo Repeat the last run in the loop
next Proceed to the next run, don't run any command after this in the loop

The main use for those is, if you have to change the standard loop behaviour under special circumstances or rules, when you want to abort the loop and so on.

5 Launching Applications from PERL

Oh yes, people wrote applications before and sometimes it does not make sense to do things twice. Unless, of course, you don't trust other people. I assume we trust people and their applications and we will no take look in how we can launch those 3rd party tools from our awesome perl script.

5.1 Open - close

The most common way is to open a pipe to the shell. This is especially useful if you want to type/send multiple lines to the shell. In this example we will launch the UNIX application sendmail to send an insulting email to a big company:

```
01: $sendmail_location = '/usr/sbin/sendmail';
02: # NOTE THAT THIS VARIES FROM SYSTEM TO SYSTEM
03:
04: $subject = 'I DO NOT LIKE HFC !!!!';
05: $sender = 'PERL BEGINNER <perlstarter\@netikus.net>';
06: $recipient = 'COCA COLA <office\@cocacola.com';
07: # PERL REQUIRES A PRECEEDING BACKSLASH BEFORE THE @ CHARACTER !!
08:
09: open (MAIL,"|$sendmail_location -t");
10: # YOU CAN WRITE ANYTHING INSTEAD OF MAIL, THIS IS THE SO CALLED
FILEHANDLE
11: # -t IS AN OPTION OF SENDMAIL
12: print MAIL "To: $recipient\n";
13: print MAIL "From: $sender\n";
14: print MAIL "Subject: $subject\n\n";
15: print MAIL "After I heared that Coca Cola is using HFC in their cooling
machines \n";
16: print MAIL "worldwide even though there is a GREEN alternative, I
stopped drinking all \n";
17: print MAIL "beverages from the CocaCola company. Thank you for taking
part in the \n";
18: print MAIL "destruction of the OZON layer.\n";
19:
20: close(MAIL);
21: # THE FILEHANDLE IS CLOSED AND WE ARE DONE ! BRAVO !
```

5.2 System

An easier way to launch a simple command that does not require multiple inputs, is probably the system call

```
01: $e_string = 'ls -l /root/';
02:
03: $output = system($e_string);
04: print "Output of \"$e_string\": $output\n";
```

5.3 Backticks"

The backticks don't really need much explanation, they work exactly like the system function. The above example would look like this with the backticks:

```
01: $e_string = 'ls -l /root/';
```

```
02:
03: $output = `$e_string`;
04: print "Output of \"$e_string\": $output\n";
```

6 Reading and Writing Files

Finally we are there, we will learn how to read and write files from PERL. PERL is known for it's great speed when processing files, so we'll see how easy it actually is.

6.1 Reading files

To read a file, we use the open command again, just a little different than before

```
01: open (WEBCOUNTER,"<webcounter.txt") || die "Cannot open file";
02: $webcounts = <WEBCOUNTER>;
03: close(WEBCOUNTER);
```

Now this works fine if we have a file with only one line. Please note that you can assign any handle here, instead of WEBCOUNTER you could simply use FILE of anything else. Just make sure it's the same whenever you the filehandle (line 2!). Now let's see how we read a file with multiple lines:

```
01: open (WEBCOUNTER,"<webcounter.txt") || die "Cannot open file";
02: for (<WEBCOUNTER>)
03: {
04:      print $_;
05: }
03: close(WEBCOUNTER);
```

Quite easy too I would say. But would if would like to store the content in an array? Look at this:

```
01: open (WEBCOUNTER,"<webcounter.txt") || die "Cannot open file";
02: @webcounter = <WEBCOUNTER>;
03: close(WEBCOUNTER);
```

And that's this. Now let's write to a file, you have already seen how that works by the way ...

6.2 Writing to files

To write to a file, we use the open command again, this time with a >

```
01: open (LOGFILE,">logfile.txt") || die "Cannot open file";
02: print LOGFILE "$user at $current_time\n";
03: close(LOGFILE);
```

Please note the \n at the end of the line, otherwise there will be no line breaks in the file. This will create a new file from scratch if it doesn't exist yet. To append to an exisiting file you use:

```
01: open (LOGFILE,">>logfile.txt") || die "Cannot open file";
```

instead. That's it! In our next example we will make a simple webcounter, one that I am actually using myself on my website.

6.3 Both

Here we will read the content of a file (a number), increase the number, and write the new number to the same file:

```
01: open (WEBCOUNTER,"<webcounter.txt") || die "Cannot open file";
```

```
02: $webcounts = <WEBCOUNTER>;
03: close(WEBCOUNTER);
04:
05: $webcounts++;
06:
07: open (WEBCOUNTER,">webcounter.txt") || die "Cannot open file";
08: print WEBCOUNTER $webcounts;
09: close(WEBCOUNTER);
10:
11: print "$webcounts since July 1903\n";
```

Yes, it is as simple as that. The variable $webcounts now holds the current web counts. Please note the differences in lines 1 (<), where we read from the file, and line 7, where we write to the file.

7 Regular Expressions

Regular Expressions are used in many programming languages and PERL is no exception here. Regular Expressions are well supported and can be useful in a lot of ways. So what are they good for? With Regular Expressions you can easily find and manipulate information in strings. Let's start with the easier one, finding information. Before I start digging in I'll list a table with some major commands you will be needing. Don't worry about their meaning – just refer to it as you need it.

\s || \S	space || no space
\w || \W	word || no word (word also means a combination of letters (without space))
\d || \D	digit || no digit
\b || \B	word border || no word border
^	beginning of line
$	end of line
[12468]	quantity (iteration)
[3-7]	quantity (scope)
[^456]	excluded quantity (iteration)
[^3-9]	excluded quantity (scope)
.	any character (exept new line)
*	**no** or **any** occurence
?	**no** or **one** occurence
+	**one** or **any** occurence
a|b|c	alternatives

7.1 Finding

Let's look at some examples again to find a string in another string!

```
1: $text = "Hey, my name is Ingmar and I am 25 years old!";
2:
3: if ($text =~ /my name is/) {
4: print "Somebody has a name";
5: }
```

This of course would yield the following output:

```
Somebody has a name
```

You probably already saw how a regular expression should look like:

```
$test =~ /something/;
```

Please make sure you have **no** space after the **=** and that there **is** a space **after** the **~**. Now what if we want to know if somebody is screaming? We do:

```
1: $text = "Hey, my name is Ingmar and I am 25 years old!";
2:
3: if ($text =~ /!$/) {
4: print "Somebody is screaming ...";
5: }
```

which also would yield

```
Somebody is screaming ...
```

That's because the **$** in regular expressions refers to the end of the line. One more example to find the age!

```
1: $text = "Hey, my name is Ingmar and I am 25 years old!";
2:
3: if ($text =~ /\d\d/) {
4: print "Somebody has a two digit age!";
5: }
```

which also would yield

```
Somebody has a two digit age!
```

You can build your own examples here if you refer to table above. I would actually recommend this since practicing seems to be the only way to really understand (more complex) regular expressions. And that's why we move on to the next chapter.

7.2 Modifying

This will be more fun than the previous one on finding, we'll manipulate sentences without turning crazy. Let's be real funny and modify the age:

```
1: $text = "Hey, my name is Ingmar and I am 25 years old";
2:
3: $text =~ s/\d\d/45/;
```

```
4:
5: print $text;
```

We would then see:

```
Hey, my name is Ingmar and I am 45 years old
```

What changed? Now we have a **s** before the first slash and we have an additional slash too. Still not too complicated. **s** stands for substitute, the pattern after the first slash tells PERL what to look for, the pattern after the second slash tells PERL with what he should exchange it. So:

$text_to_change =~ s/one/two/;

But what happens when we have two occurrences of the same pattern? Let's assume this example:

```
1: $text = "Hey, my name is Ingmar, I am 25 years old and I used to own 10
fish";
2:
3: $text =~ s/\d\d/45/;
4:
5: print $text;
```

We would then see:

```
Hey, my name is Ingmar, I am 45 years old and I used to own 10 fish.
```

Quite allright still, we see that PERL only changes the first occurrence. To make PERL change all occurrence of our search pattern, we would change line three to:

```
3: $text =~ s/\d\d/45/g;
```

and we would get

```
Hey, my name is Ingmar, I am 45 years old and I used to own 45 fish.
```

Neither am I 45 years old nor do I ever have more than 10 fish, but the **greedy** operator at the end makes on believe so. To really make sure we only change the years, we just modify line three to

```
3: $text =~ s/\d\d\syears/45 years/g;
```

which means we only look for **two digits** followed by a **space** and the word **years**. This occurrence will then be replaced with **45 years**. So the fish are not being manipulated anymore, and if I announce my age twice in a string, we make sure we change it everytime. Now what if want to change several words to one same word? Look at this real life example:

```
1: $text = "I love Coke and like to have 5 spoons of sugar in a glass.";
2:
3: $text =~ s/love|like/hate/g;
4:
5: print $text;
```

this would inevitably show this on the screen:

```
I hate Coke and hate to have 5 spoons of sugar in a glass.
```

Here we simply used the **|** (=or) operator to change multiple patterns to one generic pattern. Quite useful, isn't it? Now we'll extract some strings.

7.3 Confusing

Everything we did so far was not even remotely confusing, but I'll try to come up with some weird examples that make you at least sweat a little bit. I have to admit that I'm no master at regular expressions, but let's just see. In this case we simply want to extract the age out of the string.

```
1: $text = "Ingmar is 25 years old";
2:
3: ($age) = $text =~ /is\s(\d\d)\syears/;
4:
5: print "Age: $age";
```

Here we make sure that **is** and **years** surround the age. The **$age** has to be in brackets, as does **(\d\d)**. However, this only works with a two digit age. What about our younger friends? We just get two digit numbers here, but we want any kind of number. We change line 3 to

```
3: ($age) = $text =~ /is\s(\d+)\syears/;
```

and we catch any age. The **+** means that it's previous pattern should be matched either one time or any times!

8 Under Construction

This document is under construction and not yet finished yet. But then again, it might never be complete.