Perl programming for the complete newbie

Introduction

I've seen too many questions on forums like: "where do I start" or "what should I learn first", those kinds of questions became sort of a replacement for the annoying question/request "teach me how to hack". Most usual answer those people get is: "learn programming", then the most usual response is "what programming language is the best", at this point people start a pissing contest, yapping about languages and nothing useful comes out of the whole thing.
My choice is Perl, Perl for beginners, Perl for C/C++, PASCAL and etc users. Why? I think Perl is easy to learn and its philosophy actually encouraged me to learn it. There are no bounds in Perl, integers, strings, arrays they can all be as big as you want.
This looked like the Holy Grail to me after I shoot my self in the foot few hundred times with C, seeing all those "Segmentation Faults".
A lot of people may and will say that Python is the best language for a beginner. I have only one thing to say to those people "(contents removed due vulgarities)".
Anyway enough of this lets get started.

Getting started with Perl

First thing you need to do is download Perl. If you use Linux you already have it, if you use windows go to http://www.activestate.com/ and download the latest version for windows. Installing is easy as any other software you have installed on your computer. Now after you installed Perl, you won't see a GUI executable program with easy drag and drop, like you maybe have seen in Delphi or Visual Basic/C++.
Perl can be called only from the command line (cmd.exe). Use a normal text editor like notepad to write programs in Perl, or if you want syntax highlighting use notepad2 from http://www.fios-freeware.ch/ its fast like notepad and it can make your programming easier.
Now that you are armed with Perl and text editor, you can write your first program. Let's see how the good ole Hello, World program looks in Perl.

```
print "Hello, World\n";
```

That's it. Save it as hello.pl. Its best to make a folder where you'll put your Perl programs, so make a new folder in C:\ and name it perlprogs.
To run "your" first program open up a console (run cmd.exe) and type cd C:\perlprogs, then type perl hello.pl. Hello to you two.
Now lets examine this code, first we have print, which I believe doesn't needs explanation, then we have "Hello, World\n"; We saw the Hello, World on our screen, but where is the \n? \n is called "the new line character" it means what it means go to new line.

Remove the \n from the above “program” and see what happens. We can get the same result if we wrote the program like this:

```
print “Hello, World”;
print “\n”;
```

or like this:

```
print “Hello”;
print “, World”;
print “\n”;
```

The output will be the same. Don’t get confused, I’m only trying to show you what is the purpose of \n. But \n is not the only character of this kind here is a “complete” list.

\n	New line
\r	Return
\t	Tab
\f	Form feed
\b	Backspace
\a	Bell
\e	Escape (ASCII escape character)
\007	Any octal ASCII value (here, 007 = bell)
\x7f	Any hex ASCII value (here, 7f = delete)
\cC	A "control" character (here, Ctrl-C)
\\	Backslash
\”	Double quote
\l	Lowercase next letter
\L	Lowercase all following letters until \E
\u	Uppercase next letter
\U	Uppercase all following letters until \E
\Q	Quote non-word characters by adding a backslash until \E
\E	Terminate \L, \U, or \Q

Fool around with these “special characters” see the output and try to understand them. Maybe you’ll never use some of these characters, but it won’t hurt knowing what they do.
And we also have ; at the end of the line, and it means end of line, our “command” has ended that’s it.

Scalar Data

Ok if you have experience with other programming language like C, this might sound odd, but in Perl integers, variables, strings, floating points and etc look the same, only what they hold is different. Let's not confuse beginners and explain, here is some code.

```
$string = "Hello, World";
$number = 69;
print "This is a string = $string This is a number = $number\n";
```

Save and run. See my point? Actually there are no integers, variables or floating points in Perl, only strings and numbers and they can hold any kind of data.
The above program is very constant, but we can make it more dynamic by getting the users input, there is more than one way of getting the users input and I'll try to explain the most common ones.

```
print "Enter a string: ";
$string = <STDIN>;
chomp $string;
print "Enter a number: ";
$number = <STDIN>;
chomp $number;
print "This is a string = $string This is a number = $number\n";
```

Ok we have a lot of new things here. You notice that we don't have new line character at the first line, you can insert a \n nothing wrong with that, but the users will have to give their input in a new line, so I guess this is more user friendly approach.
The second line sets the value of $string to <STDIN>, <STDIN> is a line input operator, meaning it reads what the users have entered, so we have $string holding the users data.
The third line is for removing the pesky new line. Since the users press enter after they gave us their input, a new line is stored in $string, so we get rid of it using chomp. Here is an example how to better understand chomp.

```
$string = "mumbo jumbo text\n";
chomp $string;
print "$string";
```

Save and run. See mumbo jumbo text gets printed but there is no new line. Remove chomp $string and see the difference.
Back to our previous program, the forth line is rather same as the first one, but we are using a different "string", but this string holds a number, unless users enter some random text, so we chomp $number and we can print $string and $number without worrying about the pesky new line.

Another way of getting the users input is @ARGV. Let me show you.

```
if (@ARGV != 2) {
        print "Usage: perl $0 string number\n";
        exit;
}
($string, $number) = @ARGV;
print "This is a string = $string This is a number = $number\n";
```

Now there are a lot of new things here to be explained. Although there is more than one way of using @ARGV, I guess this is the simplest..
@ARGV is an array, I'll explain what an array is later. @ARGV is no ordinary array. @ARGV holds the users arguments. If you are confused here is an explanation what an argument actually is

```
perl someprogram.pl ThisIsAnArgument ThisIsAnotherArgument
```

So these two arguments are stored in @ARGV. In our code we have
($string, $number) = @ARGV; this sets a value of $string and $number, the first argument goes in $string and the second goes in $number, and we can use them anywhere we want.
In the first line we have if (@ARGV != 2) { ... } this is called error handling. If the user didn't enter two arguments we display the usage of the program and we exit and nothing else gets executed by Perl, program ends. To understand better remove exit; and see what happens. Oh and yes I forgot about the brackets. After an if statement we must put the case between these kind of brackets "( )" and the stuff to do in that case between these kind of brackets "{ }". Best way to remember is this:

```
If (ThisStuff IsNotTheSame WithThisStuff) { Do This Things }
```

or

```
if (ThisStuff IsSame WithThisStuff) { Do This Things }
```

!= is maybe confusing to you now, it means not equal to, it's called a comparison operator, and if is a logical operator, here is a complete list of comparison operators, I have to mention that != is a numeric sign for not equal, there are string signs for these kind of operators and the string sign for != is ne meaning not equal.

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

Use the operator that you feel more comfortable with.
Now back to our program. I always like to use getopt in my programs, let's see how the same program would look like if we used getopt.

```
use Getopt::Std;
getopts (":s:n:", \%args);
if (defined $args{s}) {
        $string = $args{s};
}
if (defined $args{n}) {
        $number = $args{n};
}
if (!defined $args{s} or !defined $args{n}) {
        print "Usage: perl $0 -s string -n number\n";
        exit;
}
print "This is a string = $string this is a number = $number\n";
```

Now there are a lot of new things to be said here. The first line use Getopt::Std; means that we are using getopt. We are including special Perl module, this is kind of object oriented programming since we use a module.
The second line declares the options (flags) we'll be using, which are -s and –n, and we use a hash to store the flag's value. Again there is more than one way to use getopt, but this is my favorite one.
The third line says if (defined $args{s}) { ... } you should grabs this instantly, its simple if the user defined the flag –s do this stuff. $args is the hash name and {s} is the key, and this hash holds the users input.
The sixth line is the same as the third, we just do the same stuff for –n flag just as we did for –s flag.
The 9th line is error handling, !defined means not defined, so if –s or –n is not defined we display the usage info and exit the program. If they are both defined we continue our program to print "This is a string = $string this is a number = $number\n";.
This may be confusing to you, but you can just use <STDIN> (the first example of getting users input) in your programs for now. As you write more programs in Perl, you'll see that <STDIN> isn't always good enough.
I should have mentioned this earlier. Perl like any other programming language is learned by writing programs. No books or tutorials can help you, if you are too lazy to write your own programs and experiment with them.
I must mention that Perl can do any math you want. Of course you can use the common operations like "-, +, /, *" but you can also "do" imaginary numbers using the Math::Complex module. If you skipped school when imaginary numbers were being lectured here is a reminder what an imaginary number actually is i*i = -1, which is impossible, anyway using this module you can easily calculate this: (9846i + 75) / (10467i – 6), and not just this like I said you can do any math you want.
Check out the c:\perl\lib\math folder and open those .pm files scroll down, and you'll see explanations and the stuff you can do.
Since we are at math let's see what else Perl can offer. Here is some code.

```
print “Enter a number: “;
$number = <STDIN>;
chomp $number;
$number++;
print “$number\n”;
```

Now this should be familiar to you, except for line four. It says $number++ this just increases the number by one, or if we typed $number-- it will decrease the number, this is useful for loops (I’ll explain what a loop is later) and you’ll see this kind of code everywhere. If the users enter a letter for example “a”, Perl will not be confused and it will print “b”, cool ha?
There are a lot of things to be said, some things I know and some that I don’t, but I can’t tell you everything you can do with scalar data, that’s what books are for.
My goal is to make you competent for reading and writing Perl programs.

Arrays

Now arrays are different than scalar data. Arrays are “declared” with a @ sign in front. Notice the similarity $ - scalar and @ - array, $ = S and @ = A, like $calar and @rray, it’s easy to remember. Anyway in a program an array would look something like this

```
@array = (“one”, “two”, “three”, “four”);
print “@array\n”;
```

or to avoid typing all those quotes

```
@array = qw/ one two three four/;
print “@array\n”;
```

qw means quoted words, every word gets automatically quoted, less typing more fun. You may say so what’s the big deal when you can just use $words = “one two three four” and get the same result. The big deal is that you can access any item from the array using a scalar, let me show you

```
@array = qw/ one two three four/;
print “$array[3]\n”;
print “$array[0]\n”;
print “$array[1]\n”;
print “$array[2]\n”;
```

You must remember that the first item in the array is indexed with [0] not [1]. So one is in $array[0] and four is in $array[3], don’t let that confuse you.
This is very important and it will save you the headache.
So as you can see we can call any item from the array. This maybe looks like nothing special to you, but it is, you’ll see later when I explain loops.
Now let’s see what else we can do with arrays.

```
@array = qw/ one two three/;
$string = “four”;
@array2 = (@array, $string);
print “@array2\n;
```

This means we can place scalar data in arrays. We have one array holding one two three, a string that holds four and another array that contains the items from the first array plus the string. So the second array holds one two three four.
This means that arrays can hold another array and scalar data.
Let’s check out push and pop now.

```
@array = qw/ one two three four/;
$string = pop(@array);
print “ The array @array\n”;
print “The string $string\n”;
```

As you can see now $string got four and array is now one two three.
You must remember that pop and push will always place/remove the last item from the array. Lets see what push can do.

```
@array = qw/ one two three/;
$string = “four”;
push (@array, $string);
print “The array @array\n”;
print “The string $string\n”;
```

As you can see the array got four at the end, but $string still holds four.
You must remember that push and pop can only be used on arrays. You can also push arrays in to arrays.
Now lets check out shift and unshift, these are similar to push and pop, but they don’t place/remove items at the end of the array, but at the beginning, remember this: (push and pop at end) (shift and unshift at beginning).
Let’s see unshift in action

```
@array = qw/ two three four/;
$string = “one”;
unshift(@array, $string);
print “The array @array\n;
print “The string $string\n;
```

As you can see the array got the item one at the beginning unlike push.
Let’s see shift now

```
@array = qw/ one two three four/;
$string = shift(@array);
print “The array @array\n”;
print “The string $string\n”;
```

Now $string got one and array has only two three four. Just like pop, but pop got four. You can also use pop and shift to remove elements from the array. If we only type shift(@array) or pop(@array) the array will lose the first or the last item.
If you want to print out an e-mail in Perl you must place a \ left from the @, or Perl will think that you are trying to print an array, or if you want to print something like "I have $20" you must add the \ on the left, like this \$20.
Let's see what happens in both cases.

```
$ThisIsWrong = "ludakot@gmail.com";
$ThisIsRight = "ludakot\@gmail.com";
print "The wrong way $ThisIsWrong\n";
print "The right way $ThisIsRight\n";
```

See first it prints only ludakot.com and then it print ludakot@gmail.com, this is because we don't have array called @gmail, but we can bypass this, but there is no reason for it, either way here it is

```
@gmail = qw/ @gmail/;
$ThisIsWrong = "ludakot@gmail.com";
$ThisIsRight = "ludakot\@gmail.com":
Print "The wrong way $ThisIsWrong\n";
Print "The right way $ThisIsRight\n";
```

This is absolute nonsense. Since we create an array for our email might as well place the whole god damn thing in the array.

```
@gmail = qw/ ludakot@gmail.com/;
print "@gmail\n";
```

Either way its best to just use a \. That's all I'm going to say about arrays at this time, will get back to them when I'll explain loops. Next, subroutines.

Subroutines

Subroutines are actually functions, that can be placed anywhere in your code. They are very useful, especially if you have a lot of if statements in if statements and so on. Subroutines wrap your code and it becomes more clean and readable.
Let's see how a subroutine is declared and called.

```
sub mysubroutine {
        print "You have called the subroutine\n";
}
print "Your subroutine will be called in 3 seconds\n";
sleep(3);
mysubroutine;
```

As you can see the subroutine "waits" until you call it, it doesn't get executed unless you say so.

Don't let sleep(3) confuse you, that's just a syscall to wait for 3 seconds and then continue reading the code.
As you can see subroutines are easy to make just type
sub YourNameOfTheSubrouteHere { code } and to call the subroutine just type its name and it gets executed.
There are a lot of things to be mentioned about subroutines, like there are no subroutines that don't return a value unlike C, self calling subroutines and so on. But that's all I'm going to say about them, I don't want you to get confused (maybe I already did confuse you?).

Loops

Now loops, I believe are very important for every programming language, so pay special attention in this part. Let's see how a loop looks like.

```
for ($i=0; $i <= 10; $i++) {
        print "$i\n";
}
```

Don't bump your head, just read it in English. We set a value on $i, which is zero, we tell when our loop will end, which is when $i is lesser or equal to 10 and we increase $i for one. The loop will end after 11 hops and you'll see the numbers from zero to ten. A more "perlish" way of doing the above is this

```
for (0..10) {
        print "$_\n";
}
```

Now this is self explanatory, except for $_, this is the Perl's default variable. When something is true, all the stuff that comes out of it is stored in $_. That's the best way of remembering $_. Now lets take a look at a while loop.

```
$i = -1;
while($i <= 10) {
        $i++;
        print "$i\n";
}
```

Now this loop will do the same thing as the above two.
Notice that we set a value on $i which is -1. Why -1 you may ask. Since the first operation in the loop is $i++ at the first hop $i will have a value 0, when it reaches 10 the loop exits, meaning $i is now lesser or equal to 10, so the condition is true and the loop exits. Now let's check out another kind of loop

```
@array = 0..10;
foreach $i (@array) {
        print "$i\n";
}
```

See we can use a loop to access all the items in the array. With each hop $i gets the value of the next item in the array. We can do the same the good old fashion way

```
@array = 0..10;
for ($i=0; $i < scalar@array; $i++) {
        print "$array[$i]\n";
}
```

scalar@array gives the number of the items in the array, so we have $i =0 $i is lesser than the number of the items in the array and we increase $i by one, then we print the items in the array. Remember $array[0] ... $array[3], well $i is used just to call the element that is indexed under that number.

```
while(1)  {
        $i++;
        print "$i\n";
}
```

This is an endless loop. Don't panic, just press Ctrl+C and the program will exit. This loop will "never" end because we are not comparing two values, so we are not making condition when will the loop end. The 1 doesn't mean anything. You can put any number you want there, as long as it's a real number and if it's not zero.
You'll use these kinds of loops in your programs, if you are planning on writing a daemon server.
That's all the loops I could think of, if there are more please notify me.
Now I'll show you just a few Perl "functions" that will come in handy in your programs.

Getting practical with Perl

If you want to run another program with Perl just use the system() function

```
system("notepad.exe");
```

You can also add arguments

```
system("perldoc –f fork");
```

You can open text files and place them in an array, and then use a foreach loop to read the lines.

```
$text = "mytext.txt"
open(file, $text) or die "Can't open file: $!\n";
@array = <file>;
close file;
foreach $line (@array) {
        print "$line\n";
}
```

The or die “..” is just error handling, if Perl fails to open the file it will print out Can’t open file : the reason why it couldn’t open the file is stored in $!.
Socket programming is a peace of cake

```
use IO::Socket;
$socket = IO::Socket::INET->new(proto=>'tcp',PeerAddr=>”127.0.0.1”, PeerPort=>”23”)
or die “Error connecting: $!\n”;
print $socket “Hello There\r\n”;
close $socket;
```

Ok I admit, Perl’s objects can be a bitch sometimes, but if you use them more often they do make sense.
Its simple actually we make a new socket, its protocol is tcp, the address is 127.0.0.1 and the port is 23,
Of course you can use $variables instead of constant numbers, which I’ll show you in the next example.
If we want to send something to the socket we have created we just print stuff on $socket. \r is return and \n is new line, you remember the table.
Since we are at sockets lets make a port scanner

```
use IO::Socket;
usage() unless (@ARGV == 2);
($host, $port) = @ARGV;
for($i=0;$i<=$port;$i++) {
	$socket = IO::Socket::INET->new(Proto=>"tcp", PeerAddr=>$host, PeerPort=>$i);
	if ($socket != 0) {
		print “Opened -> $i\n”;
		close $socket;
	}
}
sub usage {
	print “perl $0 host ports\n”;
	exit;
}
```

Try to understand this program by yourself. All the stuff I said earlier is here. Take a nice long look and try to find logic.

Final words

I’ll update this tutorial as time goes by. A lot of things are not mentioned. But then I guess I’m too ambitious.
There are a lot of books for Perl, huge books, and they don’t mention everything. Don’t stop reading about Perl here. There is an ocean ahead of you.
If you are serious about Perl I strongly suggest getting O’Reilly’s Perl Cookbook 2nd edition and Learning Perl 3rd edition. Programming Perl is of course a must have reference book.

There is free online documentation on Perl, but those books worth every cent. Perl is a whole universe, and it's still expanding. You, me and everyone else has a lot of catching up to do.
Get those books, read them, write/rewrite/experiment/modify programs, you won't regret it.
Remember: If you can't do it in Perl, it's not worth doing it.
If you haven't already noticed you can't copy stuff from this file. It's not like I don't want things to be copied from here, it's for your own good. I did that because I want you to write the programs, not just copy and paste them. It's easier to learn and remember when you actually type the code.
I hope this file was helpful and understandable. Any comments, critics and questions are welcomed. Feel free to e-mail me on ludakot@gmail.com and thank you for reading.