Perl Programming

A brief introduction to programming with Perl
(Variables, Flow Control, Regular Expressions).

GPN 2, Karlsruhe

Patrick Stahlberg <patrick@segv.de>

$Id: speech.mgp,v 1.7 2002/12/17 10:59:17 patrick Exp $

Structure of this talk

- About Perl
- Data Types
  - Scalars, Lists, Hashes
- Control Structures
  - if, while, for, foreach
- I/O Functions
  - Reding, Writing, Opening, Closing, Directory Access
- Regular Expressions
  - brief overview

Part One

About Perl

whatis perl

- 1987: Initial release to the public

```
$ whatis perl
perl (1)          - Practical Extraction and Report Language
$
```

- specialized for reading text files and generating reports
- often used for system administration tasks
- designed to be ‘practical’ (‘‘There’s more than one way to do it’’)

Part Two

Data Types

Scalar Data

- Simplest form of data in perl
- Numbers
  - 23, 5.678, -42,...
- Strings
  - "Hello World!\n"
  - 'Hello World!\n'
- Variable Names
  - $foo =! $Foo

Numbers and strings are automatically converted back and forth, if needed.

undef

A symbolic value that is assigned initially to every variable.

Indicates that the varaible hasn’t been asigned a value to.

Operands on Scalar Data

- Numbers:
  - ‘ordinary’ arithmetic operators
    - + - / *
  - exponentiation, modulus
    - ** %
  - comparison (numbers)
    - < <= == != >= >
- Strings:
  - concatenation, repetition
    - . x
  - comparison
    - lt le eq ne ge gt

Examples

5 + 4
- returns 9

23 % 5
- returns 3

5 <= 23
- returns true

"entropia is great" x 2
- returns "entropia is greatentropia is great"

"test" eq ’test’
- returns true

Functions for Scalar Data

chop $string;

- Takes a string and removes its last character.

chomp $string;

- Like chomp, but only removes newline.

print $string;

- Writes the contents of $string to STDOUT.

Examples

Assume that $str contains "hello world!\n".

```
chomp $str;
        # $str contains "hello world!"

chomp $str;
        # $str contains "hello world!"

chop $str;
        # $str contains "hello world"

print $string;
        # prints "hello world"
```

Assignment operators

=

- Assigns the right argument to the left argument.

+=

- Adds right argument to the left argument and stores the result in left argument

-= *= /= .=

- analogous

++ --

- Increments/Decrements its argument.

Examples

```
$a = $b = 42;
        # assigns 42 to both $a and $b


$a -= 19;
        # $a is now 23


$b--;
        # $b is now 41
```

Lists

Ordered list of scalar data.

("blue", "green", "red")

Variable that contains a list (‘array’):

@colors = ("blue", "green", "red");

- There are separate namespaces for scalar and list variables
- You don’t have to care about the size of an array. Perl will handle it automatically.

$colors[0] = "pink";

- Access certain elements of an array (Each element is identified by a number, starting with 0).

Array Operators

(5 .. 23)

List constructor operator. Generates a sorted list with each of the numbers between 5 and 23 in it.

=

Assignment operator

An easier way of entering lists

```
@array = qw(blue green red);
```

is the same as

```
@array = ("blue", "green", "red");
```

You may also write:

```
@array = qw(blue
            green
            red);
```

A special array

@ARGV

This array contains a list of command line arguments.

Examples

($a,$b,$c) = (1,2,3);

($e,@array) = @array;

Swap values of $a and $b using these operators:

($b, $a) = ($a, $b);

Functions for list data

pop/push

- Add/Delete elements at the end of a list

shift/unshift

- Add/Delete elements at the beginning of a list

reverse

- Reverse the order of a list

sort

- Sort a list

chomp

- chomp each element of a list

split/join

- split a string into array elements and put them together again

Examples

```
@array = qw(1 3 2 4);

push (@array, 2);
  # @array is now qw(1 3 2 4 2)
$a = pop @array;
  # $a gets 2, @array is qw(1 3 2 4)
unshift (@array, 8);
  # @array is now qw(8 1 3 2 4)
$a = shift @array;
  # $a gets 8, @array is now qw(1 3 2 4)
@array = sort @array;
  # @array is now qw(1 2 3 4)
@array = reverse @array;
```

Hashes

Unordered lists of scalar data where each element is not identified by a continuous number, but by an arbitrary scalar.

(key1 => value1, key2 => value2)

%hash

- Variable that contains a hash.

$hash{key1} = value1;

- Access a specific element of the hash.

Functions for hashes

keys

- get a list of a hash’s keys

values

- get a list containing the values

delete

- delete an element of a hash

Examples

```
%hash = ("router" => "10.23.0.1",
         "www" => "10.23.0.2");

print "$hash{’router’}\n";
  # prints "10.23.0.1", and newline

@some_list = keys %hash;
  # @some_list contains ("www", "router")

@another_list = values %hash;
  # @another_list contains ("10.23.0.2",
  #                         "10.23.0.1")

delete $hash{"www");
```

Part Three

Control Structures

if/unless

```
if (<cond>) {
    <statement1>;
    <statement2>;
} else {
    <statement3>;
    <statement4>;
}
```

Evaluate <statement1> and <statement2> if <cond> is true. Otherwise, evaluate <statement3> and <statement4>.

The ‘else’ part can be omitted.

‘unless’ works the same, but inverts the meaning of <cond>.

Example

These two statements do the same:

1:

```
unless (cond) {
    bla;
} else {
    blubb;
}
```

2:

```
if (cond) {
    blubb;
} else {
    bla;
```

while/until

```
while (<cond>) {
    <statement1>;
    <statement2>;
}
```

Check if <cond> is true. If it is, evaluate <statement1> and <statement2>. Repeat these steps until <cond> is not true.

With ‘while’ replaced by ‘until’: invert the meaning of <cond>

do ... while/until

```
do {
    <statement1>;
    <statement2>;
} while (<cond>);
```

Evaluate <statement1> and <statement2>. Check if <cond> is true. If it is, repeat these steps.

Note the difference to the while loop on the previous page.

‘do ... until’ will behave as expected

Examples

These two statements do the same:

1:

```
do {
    bla;
} while (cond);
```

2:

```
bla;
while (cond) {
    bla;
}
```

for

```
for (<init_expr>; <cond>; <re-init_expr>) {
    <statement1>;
    <statement2>;
}
```

is (almost) the same as:

```
<initial_expr>;
while (<cond>) {
    <statement1>;
    <statement2>;
    <re-init_expr>;
}
```

Example

This code adds 5 to each element of an array:

```
for ($i=0; $i<$#array; $i++) {
    $array[$i] += 5;
}
```

foreach

```
foreach $var (@list) {
    <statement1>;
    <statement2>;
}
```

Executes the statements for each element of the list. That means, in each iteration $var will be set to another element.

Example

```
%hash = ("router" => "10.23.0.1",
         "www" => "10.23.0.2");
foreach $key (keys %hash) {
 print "$key has IP address $hash{$key}\n";
}
```

prints:

```
www has IP address 10.23.0.2
router has IP address 10.23.0.1
```

Expression Modifiers (1)

- Conditionally evaluate <expr>:

<expr> if <cond>;

<expr> unless <cond>;

- Conditionally (and maybe repeatedly) evaluate <expr>:

<expr> while <cond>;

<expr> until <cond>;

Example:

```
die "Error: \$a is not equal to \$b!\n"
    unless ($a == $b);
```

Expression Modifiers (2)

<cond> ? <expr1> : <expr2>

If <cond> is true, evaluate <expr1>. Otherwise, evaluate <expr2>.

Example:

```
print "Time left: $time " .
      ($time == 1) ? "minute." : "minutes.";
```

Part Four

I/O Functions

<STDIN>

- In scalar context: get one line of data from standard input
- In list context: get complete data (up to end of file) from standard input as a list of lines

The operator ‘<>’ (diamond operator) works like ‘<STDIN>’ but gets the contents of files whose names are in @ARGV, if there are any.

Example

```
$i = 1;
while ($line = <STDIN>) {
    print "$i: $line";
    $i++;
}
```

This program prints the lines from standard input with line numbers.

```
while ($line = <>) {
    print $line;
}
```

This program works like the UNIX command 'cat'

Filehandles

A filehandle is a name that identifies an open file in a perl program.

It is convention to use UPPERCASE letters for filehandles.

Opening and Closing files

open (FH, "$filename");

Open file $filename. The filename string may be prepended by ‘<’ or ‘>’ to specify whether this file will be opened for writing or for reading.

close (FH);

Close the file. All opened file are automatically closed when the program exits.

Reading and Writing

print FH $string;

Write $string into the file.

$string = read (FH);

Read a line from the file and store it in $string.

Example

```
open(INFILE, "<$ARGV[0]")
    or die "open failed: $!";

while ($line = <INFILE>) {
    unshift (@array, $line);
}

close INFILE;


open(OUTFILE, ">$ARGV[1]")
    or die "open failed: $!";

foreach $line (@array) {
    print OUTFILE $line;
}
```

Accessing directories

opendir

- Open a directory

readdir

- Return a directory entry

closedir

- Close a directory

Part Four

Regular Expressions

man perlre

Sorry.

Part Five

Misc

Functions

```
<statement1>;
&function;
<statement2>;

...

sub function {
    <substatement1>;
    <substatement2>;
}
```

use strict;

Makes perl do some error checking. Perl will not allow you to run your program in some cases where it would run the program without ‘use strict’. It may also print more warnings during execution.

For example, you will have to declare each variable using ‘my $var;’ before using it.

Using this option is very much recommended!

Literature:

- perl(1)

- Randal Schwartz, Tom Christiansen & Larry Wall: Learning Perl, O’Reilly 1997

End. Questions?

You can download these slides at

http://segv.de/~patrick/papers/gpn2_perl/