Perl Introduction

Joe Ammann
Version 1.1

26. September 2005

Slides generated with LaTeX

Table of contents

1. My first Perl program

Lesson Overview

- What is Perl?
- structure of each Perl program
- general rules

Lesson Goals

- learn something about Perl philosophy
- be able to run a Perl program

This first lesson will teach you how to write a first Perl script. We will look into the structure of each Perl script.

And we will learn some general guidelines how to write *clean* Perl scripts.

What is Perl? – 1.1

Perl is a language for getting your job done!

- Perl is intuitive! Most of the time, it does what you mean.
- TMTOWTDI - There’s more than one way to do it
- BUT! Perl can also be very cryptic.

Perl - Practical Extraction and Report Language

Perl - Pathetically Eclectic Rubish Lister

Perl is a very intuitive language! Perl programs can be written in a very readable way. Some examples:

```
open(LOG, ">/tmp/trace.log")
or die "can’t open file!";

print while (<INPUT>);
```

Also, there is *never* one single correct way to do something in Perl! Perl almost always offers many different ways to program the same thing. Every programmer may have her or his own style.

Perl scripts can be written in a very short way, using many “magical” shortcuts. But this can also render programs unreadable!

In this course, we will be rather on the “epic” side of Perl programming. This can help beginners to avoid common pitfalls. Experienced Perl programmers will probably think that this style is “very primitive” :-)

A taste of Perl – 1.2

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

use strict;

# print this to STDOUT
print "Hello world!\n";

exit(0);
```

- script named `first.pl`
- “hash-bang” or “shebang” line
- use strict checking

This tiny example shows some typical elements of each Perl script.

Perl scripts are normally given the file suffix `.pl` - this is just a convention and has no technical meaning to the system.

The first line of each Perl script identifies the Perl interpreter to be used. This is vital if you want to be able to start the Perl script.

The first line also passes the `-w` option to the Perl interpreter. This tells the Perl interpreter to treat warnings as fatal errors - the Perl interpreter will stop processing the script if it encounters a warning. This is defensive programming. Because Perl allows more lax coding than other programming languages, this is normally a good thing!

The last noteable thing is the `use strict;` pragma. This instructs the Perl interpreter to use strict variable checking. This means (among other things) that each variable that is used must be declared first. This urges the programmer to do cleaner coding.

My first Perl program 7

Run the program

```
prompt> perl first.pl

prompt> chmod +x first.pl
prompt> ./first.pl
prompt> first.pl        (if cwd is in PATH)

prompt> perl -cw first.pl
```

Exercise: Type in the program from the last slide and get it to run.

The first command assumes that the Perl interpreter is in your PATH. Use `type perl` to find out which Perl interpreter you are using.

When you make the Perl script executable, you can call it directly - provided the first line (“hash bang”) is correct.

A very helpful thing during development is the `-c` option of Perl. This allows you to do a syntax check. Combined with the `-w` option, this gives you clueful hints about problems in your Perl script.

General things

- comments begin with #
- each statement terminated by semicolon
- statements can span multiple lines
- blocks (surrounded with braces `{}`) group multiple statements

It is normally very helpful to use an editor that does syntax hilighting for Perl scripts, such as emacs or vim.

Comments in Perl begin with a # character and extend until the end of the line.

Generally, every statement must be terminated by a semicolon. This gives you the possibility of having a statement span multiple lines. Linebreaks normally don't matter for Perl. So the following is valid.

```
print "This will produce a long line, ",
      "include a value: ", $value,
      " and continue to a third line\n";
```

Important is the use of `{}` braces to group mutliple statements together. The use of braces is *mandatory* in most control structures, such as `if` statements or `while` loops.

Some Perl idioms – 1.3

- introduce some Perl idioms, to be able to show somewhat interesting examples
- will be explained in detail during the course

```
# read one line from stdin into variable input
$input = <STDIN>;

# print something on standard output
print("Please enter your name: ");  # no \n, so no newline
$name = <INPUT>;
print "Hello ", $name, "pleased to meet you.\n";

# execute a command, and assign output to a variable
$result = ‘date‘;
```

To be able to make somewhat meaningful examples and exercices, we introduce some typical idioms in Perl. They are often used, and will be explained later in the course in detail.

To read one line of input from STDIN, use the syntax as above. The Perl program waits until a full line is available, and then assigns this line to the variable `$input`.

The second example shows you how to write something to STDOUT. The `print()` function of Perl allows you to write anything to STDOUT. Just give the string (or multiple strings, separated with commas, and `print()` will show it. Note that if you want the cursor to go to the next line, you explicitly have to add a `\n` at the end of the string - otherwise the cursor stays on the same ilne.

Finally, an easy way to execute an external program and capture the output of the program is the backtick notation. This should be known from shell programming. You can then process the result of the command.

Note that once you get deeper into Perl, you’ll rarely need the backticks anymore - because Perl offers internal functions for almost anything!

Getting online help

- Perl has own documentation format: POD - plain old documentation
- conversion to other formats (pod2man, pod2html, pod2text)
- depending on you installation, `man perl` may or may not work
- `perldoc` command should always work

perldoc perl shows a list of doc chapters

perldoc -f func1 shows documentation on Perl function "func1"

perldoc -q keyword show FAQ entries with that keyword

Perl has its own, homegrown documentation format - POD. Every Perl distributions also has tools to convert the POD to other formats, such as man pages, HMTL or LaTeX.

So it depends totally on you installation, what formats of the Perl documentation are actually available online. It can be that `man perl` will work, and it probably will on most Linux distributions.

But Perl also comes with its own online documentation viewer, `perldoc`. `perldoc` takes advantage of the specific POD features, so when you are heavily doing Perl programming, you'll probably want to use `perldoc`. Some nice features of it:

`perldoc perl` shows a list of documentation chapters, which give you the overview on each aspect of Perl. `perldoc -f function` gives you the detailed description of the given Perl function.

`perldoc -q keyword` is a very helpful thing, when Perl does not behave as expected (which it will!). It gives you a list of items from the FAQ which contain this keyword.

Perl versions – 1.4

- developed by Larry Wall (with the help of many others)
- current version is 5.6.0
- commonly found are also 5.004 or 5.005
- available for most common platforms (Unix, Windows, Mac, etc)

```
prompt> perl -v

This is perl, version 5.005_03 built for i386-linux

......
```

The current version of the Perl interpreter is 5.6.0. Commonly found are also version 5.005 and 5.004. There has been a change in the version numbering scheme when moving to 5.6.0, following the old scheme that would have been 5.006.

You should generally avoid using the specific features of 5.6.0 (advanced things like threads and Unicode character set), because many environments still use 5.005.

2. Variables and Datatypes

Lesson Overview

- learn about Perl variables
- scalar variables
- arrays (lists)
- hashes (associative arrays)

Lesson Goals

- get to know the simple variable scheme
- be able to define and use variables

Perl variables come in 3 flavours: scalars, arrays and hashes. We will introduce them and show how to use them.

Scalars – 2.1

- most basic Perl variable type
- prefixed by `$`
- can hold strings and numbers
- no type checking (as in Java or Pascal)

```
$name = "Joe";
$age = 30;       # liar!

print "$name is $age years old\n";
```

Scalars are variables that hold one single value. They are prefixed by a `$` sign.

A scalar can hold a string or a number. Perl scalars are not typed! So the same variable can hold first a string, and then later a number.

Declaring variables

- declare with `my()` statement

```
my ($name, $age);
my $size = 1.7;
```

- introduce variable and “reserve” name
- optionally initialize it

Declaring variables with `my()` helps you getting a better overview on your Perl script. You can also just start to use a variable in the middle of a Perl script, without declaring it.

```
$age = 30;

# oops, a typo
print "I’m $rage years old.\n";
```

This will introduce the new scalar `$rage` with an undefined value. This is probably not what you intended. In a multipage Perl scripts, those errors can be hard to detect.

When you specify the `use strict;` command, Perl forces you to declare all variables before the first use. It is good style to use `use strict;` on all but the very simplest Perl script!

Literals – 2.2

- strings in single or double quotes
- double quotes with interpolation
- single quotes without interpolation

```
$s1 = "The value is: $value";
$s2 = 'It costs $100';
```

- number as integers or real numbers

```
$i = 100;
$pi = 3.1415;
```

Strings and numbers are the literals used in Perl. String literals are enclosed in either single or double quotes. The difference is that double quoted string literals are subject to interpolation. This means that variables are replaced by their values, and certain special characters (such as `\n` for newline and `\t` for tabstop) are replaced. Single quoted string literals are not subject to interpolation.

Numbers come either as integers or real numbers. They can be specified in many different ways

```
1000
1234.56
6.02E23         # scientific notation
0xff0e          # hexadecimal
0377            # octal, leading 0!
```

The magical $_ – 2.3

- `$_` is a very special, predefined variable
- comes into action when you don't specify a variable explicitly

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

while (<STDIN>) {
    print;
}
```

There is a very special magical Perl variable, named `$_`. This variable is extremly handy - it (almost) always comes into play when you do not specify another variable explicitly in an operation.

Again, Perl is intuitive - most of the time, it is intuitively clear when you can use `$_` and when you can't.

The use of `$_` can make programs much more readable!

When you use Perl more often, you'll see that almost all of the "non-word" characters in the ASCII set are used in Perl for special variables. `$$, $!, $<` etc. are all used for special meanings.

Arrays - aka Lists – 2.4

- very common construct in Perl
- prefixed by @
- can hold strings, numbers - just anything!

```
my(@firstarray);

@firstarray = ("a", 2, "b", 0);
```

An array is a well known construct in programming. In Perl, the use of arrays is very common.

Unlike in other programming languages, arrays in Perl are very dynamic. You never have to say how big an array should be (how many elements it has). Perl will make the arrays as big as it has to be. You can use them very flexibly, e.g. adding elements at the end, or also inserting other elements in the middle.

Because of this flexibility, arrays in Perl are often called lists.

Elements of an array

- access by index: `$array[index]`
- index of last element: `$#array`
- number of elements: `scalar(@firstarray)`
- accessing an element automatically resizes the array if needed
- grows as long as there is memory

```
print "Second element: $firstarray[1]\n";
print "Index of last element (not size!): $#firstarray\n";

# make the array big
$firstarray[10000] = 'big';
```

You access the elements of an array by passing the index of the desired elements. As usual, the index begins at 0.

Be alert: When accessing one single element of an array, the result is obviously a scalar value. So one does not write

```
@firstarray[1] # wrong!!
```

but rather

```
$firstarray[1]
```

The index of the last element in an array can be determined via the "special variable" `$#array`. This is the number of elements *minus one*!

The number of elements in an array can be determined directly by using the Perl function `scalar()`.

```
print "The array has " . scalar(@array) . " elements.\n";
print "The array has ", scalar(@array), " elements.\n";
    # equivalent
```

Program arguments

- arguments to Perl program passed as array
- `@ARGV`

Exercise: Write a little Perl script that prints out the number of arguments the you pass it, and each value!

One very common use of an array are the arguments that are passed to a Perl program. When you call a Perl script with additional arguments, those arguments are passed as the elements of an array called `@ARGV`.

Write a little Perl program, that prints out the number of arguments given, and the value of each argument.

```
prompt> perl argprint.pl first deuxieme drittes
Arguments: 3
first
deuxieme
drittes
```

To do this, you need to use a looping construct, which we haven't yet introduced. You can use the `foreach()` loop:

```
foreach $element (@ARRAY) {
     ....
}
```

Useful functions

- `sort()` returns a new, sorted array
- `reverse()` returns a new array in reversed order

```
@x = ('abc', 'def', 'x');

@y = sort(@x);
foreach (@y) {
    print $_, "\n";     # gives "abc  def  x"
}

@y = reverse(@x);
foreach (@y) {
    print $_, "\n";     # gives "x  def  abc"
}
```

`sort()` does - not surprisingly - sort the elements of an array. It returns a new array which contains the same elements as the original one, but sorted. The original array is untouched.

`reverse()` also simply does what it's name implies.

Of course, the actual output of the above `print()` statements would be 3 lines with one element each. The output in the comments has been (compressed) to fit on the slide.

Another very useful function is sometimes the `qw()` function. This functions saves you writing zillions of `", "` sequences when you have to specify a literal array. So instead of writing

```
@x = ('a', 'b', 'c');
```

breaking his fingers and most probably missing at least one quote or comma, a seasoned Perl programmer writes:

```
@x = qw(a b c);
```

The `qw()` function returns a list of "quoted words", hence the name. And yes, it is correct that there are *no* commas between the elements!

To be honest, this is not the full truth about the `qw()` function! You'll learn later in the chapter on Perl functions how this really works. But for now, you can use it like this.

Scalar vs. list context – 2.5

- “everything” is evaluated in a specific context
- 2 importants context: scalar and list
- can be confusing!!

```
@array = (1, 2, 3);
$scalar1 = @array;
$scalar1 = scalar(@array); # make it explicit

$scalar2 = (4, 5, 6);

($a, $b, $c) = (7, 8, 9);
```

Perl is very peculiar when it comes to evaluating “things” (terms, functions, etc). The result almost always depends on the context in which the thing is evaluated. There are 2 main context which are of common interest: scalar context and list context (the name “array context” is nowadays discouraged by the Perl developers)

Some examples above shall show the basic principles and differences of scalar and list context.

The first example shows (on the second line) how an array is assigned to a scalar variable. Obviously this can not work as such, so some conversion has to happen. The left side of the assignment operator decides on the context which is used for evaluation. In this case, the left side is a scalar variable, so the whole expression is evaluated in scalar context. The result is, that the right side is converted to a scalar value, and this will be the number of elements in the array.

The second example shows at first sight the very same case. But this is not true! In this case, the right side of the assignment is a sequence of instructions, namely: “Take number 1, then 2, then 3 and make a list out of them”. When this is evaluated in scalar context, the result is: “Take number one and throw it away, take number 2 and throw it away, take number 3 and return it back”. So in essence, the number 6 will be assigned to `$scalar2`.

The third example shows, how to assign mutliple elements of a list to multiple scalar variables.

Hashes – 2.6

- third important type of variables (aka “associative arrays”)
- index are key strings, not numbers
- `%hash`
- implemented as hashtables, hence the name

```
%hash = ();
$hash{’name’} = ’Ammann’;
$hash{’firstname’} = ’Joe’;
```

Hashes are a very powerful tool to store and manage values. Hashes are somehow like a special sort of array. The main difference is the way how elements are indexed (keyed, looked up, whatever). Whereas arrays are always sequential structures where the individual elements are indexed by number, hashes index their values with strings.

The prefix for a hash is a percent sign (%). To access the elements, the following syntax is used:

```
$hash{’name’} = ’Ammann’;
print "Name :\t$hash{’name’}\n";
$index = ’firstname’;
print "Firstname:\t$hash{$index}\n";
```

Useful functions

- getting a list of all keys

```
@indexes = keys(%hash);
```

- initialize hashes

```
%map = ( 'red'   => 0x0000ff,
         'green' => 0x00ff00,
         'blue'  => 0xff0000 );
```

- `%ENV` hash with all environment variables (e.g. `$ENV{'PATH'}`)

Exercise: Write a Perl script to print out the names and the values of all environment variables!

A common thing to do with hashes is to retrieve a list of all keys that are currently in the hash. The `key()` function does exactly this.

To initialize a hash, a special legible format has been introduced. It uses the `=>` operator.

Exercise: Enhance the script which prints out the arguments! It should now also print out all environment variables:

```
prompt> perl argprint.pl first deuxieme drittes
Arguments: 3
first
deuxieme
drittes

Environment: 25
PATH: /usr/bin:....
USER: joe
....
```

3. Operators

Lesson Overview

- give a quick overview
- introduce most important operators

Lesson Goals

- know important Perl operators
- avoid common pitfalls

Perl operators are very diverse and rich. They allow to perform the basic operations of each Perl program, such as comparison and assignment.

We can’t introduce all Perl operators, so we will concentrate on the most important ones.

Number and string operators – 3.1

- arithmethic operators (`+, -, *, /`)
- also modulus and exponentiation (`%, **`)
- string concatenation (`.`)
- “string multiplication” (`"abc" x 3`)
- autoincrement/decrement (`$a++, --$b`)

These are the most obvious and common operators. The 4 basic arithmetic operations, the modulus operations and exponentiation work on numbers. These operators also automatically convert between integer and real numbers where necessary.

Two specific operators work on strings: The string concatentations (“dot operator”) and the (rarely used) “string multiplication”. The latter can be useful e.g. to create a string with 60 dashes (`$dashed_line = '-' x 60;`).

Perl is also “quite lax” when converting between numbers and strings. The following examples are all valid:

```
$s = 'abc';
print $s + 1; # prints out "1"
$s = 'abc';
print ++$s; # prints out "abd"
```

Assignment operators – 3.2

- single equals sign (=)
- can be prefixed with almost any operator

```
$x = 5 * 3;
$x += 15;

$s = "foo";
$s .= "bar";
```

The usual assignment operator can be prefixed with almost any operator. This means that this operator is applied to the variable which is left of the assignment operator.

Comparison operators – 3.3

- for numbers (`==, !=, <, >, <=, >=`)
- for strings (`eq, ne, lt, gt, le, ge`)

```
if ($index == 17)
....

if ($string eq "foo")
...
```

Nothing really surprising here.

The most important thing to remember for comparison operators is that Perl clearly differentiates between operators for numbers and for strings!

A very common mistake is

```
if ($string == "exit")
...
```

Logical operators – 3.4

- high precedence and low precedence operator
- high precedence (`&&, ||, !`)
- low precedence (`and, or, not`)

```
if ($a > 10 && $b < 100)
...

$s eq "bla" and print "hello\n";
```

Perl has two sets of logical operators. Both sets are are so called “short cut” operators. This means that evaluation of expressions stops, as soon as the outcome of the overall expression is determined.

The older, high precedence set, which follows the “normal way” of other programming languages.

The newer, low precedence operators are more “intuitive” in many situation. They can also make the code more legible.

There are no other difference between the 2 sets than precedence.

Exercise – 3.5

- Write a program that prompts for and reads 2 numbers, and prints out the results of the 4 basic mathematical operations of those 2 numbers.

```
prompt> perl basicops.pl
First number : 333
Second number: 3

Addition: 336
....
```

In the example, try what happens if you don’t specify number, but strings. What happens?

4. Control structues

Lesson Overview

- the most important control structures
- special Perl features

Lesson Goals

- know constructs to build loops etc.

Here we introduce the most common control structures used in Perl. Did we mention that Perl is normally very rich on syntactic constructs, and that control structures are no exception :-)

So we will again concentrate on the most important and often used features.

if/unless statements – 4.1

- `if (...) {  }`
- `if (...) {  } else {  }`
- `if (...) {  } elsif {  }`
- `unless (...) {  }`

```
if ($ARGV[0] eq "-d")
    $debug = 1;

print "now doing this" if $debug;
```

The well known `if ... else` statement is also used in Perl. Note that there is no explicit "then" element in this statement. Also note that if you use the common form "if ... elsif ...", the `elsif` part is written without the second e of else.

As shown in the example, you can also use numbers and strings directly (without explicit comparison operators) in if statements. The rules are simple:

- any number different from 0 is true
- any string other than `""` and `"0"` is true

while Loop – 4.2

```
while (some_expression) {
    statement1;
    statement2;
}
```

- loops as long as `some_expression` is true
- there is also an `until() {}` loop

The `while` loop is nothing surprising. Perl keeps looping over the statements inside the `while` block, as long as the statement return true.

Perl also knows about `until() {}` loops, although they are of course totally redundant (but sometimes helps you read through code as if it were english text).

A very typical `while` loop is:

```
while ($line = <STDIN>) {
    # do something with the next line
}
```

This the file (in this case, STDIN) line by line, and keeps on looping until there is no more data coming (the end of file has been reached).

foreach loop – 4.3

- an easy way to iterate over the elements of an array
- saves you a counter variable

```
foreach $element (@array) {
    ...
}

# or with use of $_
foreach (@ARGV) {
    print;
}
```

Since arrays are so commonly used in Perl, one of the most common operations in perl is iterating over all elements of an array.

Perl has a special looping construct for this, the `foreach()` loop. In the `foreach()` loop, you specify a variable as the so called “bind variable”, and you also must name the array over which the loop should go. `foreach()` will then iterate over each element of the array, setting the bind variable to the first element of the array during the first loop, to the second during the second loop, etc.

You can also use the magical `$_` variable again (which is very commonly done by many Perl programmers), so a typical loop over an array looks like:

```
foreach (@array) {
    # work with $_, which is used to iterate
    # on the elements of the array
    ....
}
```

for Loop – 4.4

- loop with explicit counter variable
- not very often used in Perl, because `foreach()` is easier in many cases

```
for (init counter; test counter; increase counter) {
    ...
}

for ($i = 0; $i <= $#ARGV; $i++) {
    print "$ARGV[$i]\n";
}
```

- `for` loop to do typical counting

The `for()` loop is also a very common concept. Actually, the `for()` loop is much more general than shown in the slide above. You can use a `for` loop for almost anything - in fact, you can code every `while` loop as a `for` loop - and vice versa!

```
while ($condition) {
}

for (;$condition;) {
}
```

In the opinion of the author (to which you may not necessarily agree :-) it is more readable in those cases to use a `while` loop. I recommend to use `for` loops only for strict counting loops.

A very common idiom is:

```
for (;;) {
}
```

which is an endless loop. It can also be written as:

```
while (true) {
}
```

OPTIONAL: Loop control, next and last – 4.5

- change loop control from inside the loop
- `next` starts the next iteration
- `last` exits the loop

```
foreach $arg (@ARGV) {
    next if ($arg eq "-d");
    print $arg;
}

while ($line = <STDIN>) {
    if ($line eq "quit\n") {
        last;
    }
}
```

Sometimes, it makes your program much easier if you can “abort” a loop while in the middle of one iteration. Programming language purists may not agree with that ...

Perl has the `next` and `last` statements for this (Please: Don’t confuse them with the `continue` statement as known from the C programming language. `continue` is used for something else in Perl!)

`next` makes a loop immediately start the next iteration. The statements between the `next` command the the closing brace of loop body are skipped.

`last` immediately exits the current loop. The iteration stops, and the statements between the `last` command the the closing brace of loop body are skipped. The Perl programm continues with the first statement after the closing braces of the loop body.

Exercise – 4.6

- Write a program that prints out all numbers from 1 to 20 and their squares.

```
prompt> perl squares.pl
1:   1
2:   4
3:   9
....
```

To make the exercise a bit more difficult, use a `while`, a `for` and a `foreach` loop to print out the squares 3 times.

Home Exercises

- Write a Perl program that prints out the lines from STDIN in reverse order.
- Second step: Try to make the program as short as possible!

Those exercise are not very easy! So if you have problems, you can ask me questions via Mail during the week. The mail address is `joe@pyx.ch`.

The first exercise should print out a file with the orders of the lines reversed. So for example:

```
prompt>cat file_to_be_reversed
first line
second line
.......
prompt> perl reverse.pl < file_to_be_reversed
.......
second line
first line
```

Home Exercises (2)

- Write a Perl program that counts how often each word occurs in a text file. At the end, the program should print out each word, and how many times it has occurred.

Use the `split()` function to split up one line of text into the individual words (separated with spaces).

```
while (<STDIN>) {
    # the line is now in $_
    @words = split;
    # the array words now contains a list of words
    ....
}
```

- Second step: Try to sort the output so that the words appear in alphabetical order.

For the second exercise, you need the `split()` function as describe above. You must use it in its simplest form, where `split()` automatically works on

The output of the program should look something like:

```
prompt> perl wordcount.pl < file_to_be_counted
word1: 1
word2: 17
word3: 4
```

The core of the program will be build around a hash (associative array). The keys for this hash are the words found in the text. The values of the hash are counters, indicating how often each words has already been found.

The pseudocode of the program would look something like:

```
while (there is another line)
    split the line in words
    foreach (word in line)
        increment the counter in the hash

foreach (key in the hash)
    print final counter value
```

5. Regular Expressions and Matching

Lesson Overview

- “revive” what is already known about regexp
- special Perl features
- match and replace operators

Lesson Goals

- recognize Perl’s strenghts in regular expressions

One of the most important uses of Perl is text processing. This is the area where Perl actually started its career. Perl has all the capabilities of common Unix utilities such as sh(1), grep(1), sed(1) and awk(1). And the advantage of Perl is that there are not arbitrary size limit (e.g. on the maximum line length).

One of the most important features in this area of text processing are regular expressions. Regular expressions are a vey powerful tool to recognize patterns in text files, such as log files or configuration files.

Perl has set new standards on the power and richness of regular expressions. And also the speed of the processing is unmatched.

Simple uses of Regexp – 5.1

- a regexp is a pattern - something to match against a string
- common in Unix

```
prompt> grep '^abc' somefile
prompt> sed -e 's/abc/xyz/' somefile
```

Or, in Perl

```
if (/^abc/) {
    print;
}
```

Regular expressions (regexp for short) are very common in Unix. They have been introduced long before Perl existed. But Perl has brought regexp to a new level of functionality and performance.

One disadvantage of regexp is that almost any Unix program that uses regexp has a slightly different subset of functionality. `sed`, `awk`, `grep` (with it's variants) have all a different set of regexp they can handle.

Perl is a superset of all of them and introduced many new features not known before.

The match operator

- place the regexp between 2 slashes: `/regexp/`
- very often applied to the magical `$_`
- also explicit variables

```
if ($text =~ /regexp/) {
    ....

if ($text =~ /reg\/exp\/with\/slashes/) {
    ....

if ($text =~ m|reg/exp/with/slashes|) {
    ....
```

The match operator in Perl are the tow slashes around the regexp. Very commonly, the match operator is applied to the magical `$_`, so you can just write:

```
if (/foo/) {
    ...
```

Of course, you can also match against explicitly named variables, just use the syntax as shown.

A special cases arises if the regexp that you are matching contains one or more slashes (commonly filenames). In this case, you can either quote the slashes with backslashes, but that quickly becomes ugly and unreadable.

In these cases, it is often easier to use the match operator in it's full beauty. Then, you can replace the 2 slashes by any other character that doesn't interfere with your specific regexp.

```
if ($x =~ m/bla/)

if ($x =~ m|/etc/passwd|)

if ($x =~ m#especially interesting#)
```

Patterns – 5.2

- literal characters

```
/a/          # matches an 'a'
```

- match any single character: `.`

```
/./          # matches any character
```

- character groups

```
/[abc]/      # matches an 'a', 'b' or 'c'
/[a-z0-9]/   # a lowercase char or a digit
```

- predefined character groups

```
/\d/         # a digit [0-9]
/\w/         # a "word" character [a-zA-Z0-9_]
/\s/         # a whitespace [ \r\t\n\f]
```

Regular expressions are made out of one or multiple patterns. The simplest group of these patterns are the single character patterns. They always match exactly one character (if the match at all, of course).

The easiest are literal characters. With brackets, they can be grouped. But they still match one single character out of the given selection.

There can also be negated character groups:

```
/[^abc]/     # every character which is not 'a', 'b' or 'c'
```

There are also predefined character groups. The most important ones are listed above. The negations of those predefined groups also exist, they are written with uppercase characters.

```
/\D/         # matches everything else than a digit
/\W/         # matches everything else than a word character
/\S/         # matches everything else than a whitespace
```

Grouping patterns

- sequence (ok, that easy!)

```
/abc/     # matches the sequence of 3 characters "abc"
```

- the “any number” multiplier: asterisk `*`

```
/ac*/     # matches an ’a’, followed by any number of ’c’
          # "a" (0 c’s !), "ac", "acccccccc"
```

- the “at least once” multiplier: `+`

```
/ac+/     # matches an ’a’, followed by at least one ’c’
          # "ac", "acccccccc", but not "a"!
```

There are 3 major grouping patterns. The simple sequence is self explanatory.

The “any number” multiplier `*` matches any number of the preceeding character. Where 0 is also a valid number!

Make always the clear distinction between this regexp `*` mutliplier, and the globbing character `*` of the shell! The pattern to say “match any string” in Perl is as follows:

```
if (/a*b/) {
    .....   # probably not what you wanted

if (/a.*b/) {
    .....   # matches "a<anything>b",
            # i.e. "ab", "abcbcbcb", "a123b"
```

The “at least one” multiplier allows you to match at least one, but as many as possible characters of the referenced character group.

Anchoring pattern

- beginning of line: `^`
- end of line: `$`
- word boundary: `\b`

```
/^joe:/    # match the pattern 'joe:' at beginning of line
           # e.g. useful in /etc/passwd
m|/usr/bin/bash$|
           # match at end of line
           # e.g. in /etc/passwd: all users with bash shell
```

Anchoring patterns define “rules” about the environment, in which another pattern has to occur to be a successful match. With anchoring patterns you can say things like: “match ’abc’ at the end of a line” or “match ’butter’ but only if it is a separated word (e.g. don’t match ’butterfly’)”.

There are three important anchoring pattern.

The caret defines “at the beginning of a line”.

The dollar sign defines “at the end of a line”.

`\b` defines that the match should only occur on a word boundary. Word boundaries are: beginning of line, end of line, whitespace characters, interpunction characters (comma, dot, exclamation mark, etc.)

Excursion: chop() and chomp() – 5.3

- often irritating: the trailing newline character at the end of a line

```
while (<STDIN>) {
    ... do some (maybe:-) clever regexp ....
    print "$result\n";        # may print 2 newlines!
}
```

- take away last character: `chop($string)`
- take away last character *only if it is a newline*: `chomp($string)`

WARNING! DON'T DO THE FOLLOWING

```
$string = chomp($string);
```

Sometimes you will hit situations where you are looping through the lines of a input file, and want to do some regular expression matching on the lines.

It can be that in some of those situation, thr trailing newline character that is part of each input line gets in your way! Of course, you could to all your patterns in a way that this newline character is correctly handled. But sometimes, it is easier to get rid of the newline before really starting to process the line.

```
while (<STDIN>) {
    chomp;          # operates on $_
```

This is a very common thing to do in Perl.

But don't ever do something like the last example on the slide! `chop()` and `chomp()` automatically work on the argument that is passed and change the variable. The return the number of the characters that have been removed.

Retrieving matches – 5.4

- often you'll want to fetch the matched sub-strings
- parentheses () to build substrings and matches

```
# not the most elegant way!
($username, $password) = /^([^:]+):([^:]+):.*/;

if ($line =~ /Name:\s*(\w+)\s*Firstname:\s(\w+)/) {
    print "Nom: ", $1, "\n";      # refers to the first match
    print "Prenom: ", $2, "\n";
}
```

Very often, you do not just want to know whether a string matches a certain pattern or not, but you also want to get the matched substrings. To do this, you have to enclose the desired pattern sequence in parentheses.

Perl will then assign the substrings to temporary variables. You can do 2 things with those. Either you can assign the matches to a list of variables (of course, you can also use an array variable):

```
($a, $b, $c) = /(a-pattern)foo(b-pattern)bar(c-pattern)/;
@matchlist = /(a-pattern)foo(b-pattern)bar(c-pattern)/;

# but NOT!!
$a = /foo(a-pattern)bar/;

# QUIZ: What is assigned to $a in this WRONG!!! example
```

The second way you can refer to the matches is via the temporary variables `$1` to `$9`.

OPTIONAL: Advanced features – 5.5

- matching special characters (such as backslashes or asterisks `*`)

```
/\n/        # matches a newline
/\\n/       # matches a backslash, followed by an 'n'
```

- using variables in matches

```
if ($line =~ /\b$whatever\b/) {
           # match if $line contains whatever is currently in
           # variable $whatever, as a single word
```

- parentheses as memory (back-reference previous matches)

```
/fred(.)barney\1/
```

When you want to explicitly match characters that have a special meaning in regular expressions (such as backslashes or dollar sign), you have to "escape them". Escaping means, prefix them with a backslash `\` to take away their special meaning.

Sometimes, you don't want to do matching on literal strings, but rather on values that are contained in a variable. You can easily interpolate variable into your regular expression.

Finally, an extremly powerful mechanism is to use parentheses (normally used to group patterns for later retrieval) as "memory". With this, you can do matches like "match 'fred', followed by any single character, and then 'barney', followed by the same character (whatever it was) that came after 'fred"'.

Substitution – 5.6

- powerful matching enhanced with powerful replacement!
- substitution operator: `s/match/replacement/`

```
$text = 'hello world';
$text =~ s/hello/byebye/;   # $text is now 'byebye world'

$_ = 'foo fool buffoon';    # work on magic $_
s/foo/bar/g;    # use 'g' modifier to replace all matches
                # $_ is now 'bar barl bufbarn'

$repl = 'goodbye';
$text =~ s/hello/$repl/;     # replace hello with goodbye

$text = 'this is a test';
$text =~ s/(\w+)/<\1>/g;     # <this> <is> <a> <test>
```

Of course, you can also do substitution on your variables. The substitution operator replaces whatever your match was, with the text in the replacement area.

So in the first part of the substitution operator, you can use everything that you learned so far in this chapter. The difference is that whatever part of your variable is matched with that pattern, that part will be replaced with the second string between the slashes.

The normal substitution operator just does one replacement. If you want to replace *all* occurences of a match, then you have do specify the `g` option.

Of course, you can interpolate variables also in substitutions.

And in the last example, you can see how to use the “memories” again, where you include parts of the match in parentheses, and then use these matches in the replacement part.

Exercises – 5.7

Construct regular expressions for:

- at least one `a` followed by any number of `b`'s
- any number of backslashes followed by any number of asterisks (`*`)
- 2 consecutive copies of whatever is in variable `$whatever`
- the same word written 2 times in a row, with some whitespaces in between

6. Perl Functions

Lesson Overview

- get to know some important Perl functions
- write your own subroutines
- functions vs. list operators

Lesson Goals

- learn the most important of Perl's rich set of functions
- learn the subtle difference between functions and list operators

Here we'll learn about some of the important functions that Perl provides. We will look into functions for:

- text handling
- array processing

And we will also learn how to write our own subroutines and use functions for structuring our Perl programs.

Text processing functions – 6.1

```
$text = "A simple text for simple functions\n";

$len = length($text);          # result is 35
$index = index($text, 'sim');  # result is 2
$index = index($text, 'sim', 10);  # second one, 18

$sub = substr($text, 10);      # "ext for simple functions\n"
$sub = substr($text, 10, 3);   # "ext"
```

These are very simple text processing functions.

There are many more in Perl!

Array processing functions – 6.2

- split up a string into a list of parts

```
@fields = split(/\s+/, "first second third");
    # ("first", "second", "third")
@fields = split(/:/, "user::UID:GID");
    # ("user", "", "UID", "GID")
```

- join an array into one string with a specific “glue”

```
@array = qw(abc def xyz);
$string = join(':', @array);
    # "abc:def:xyz"
```

`split()` allows you to split up a string into an array of substrings. You can specify any regular expression that should serve as the “separator”. The separator can be a simple character (e.g. such as a comma or a semicolon). But it can also be a full blown regular expression (such as “a comma, followed by any number of whitspace characters” - that would be `/,\s*/`).

A special case that we already have learned is to call `split()` without any arguments.

```
@words = split;     # same as split(/\s+/, $_)
```

`join()` is the opposite - it joins the elements of an array into one single string. Here, you specify the string that should be used as separator between the elements. This is now *not* a regular expression - because it doesn’t make sense! It’s just a simple string that will be inserted in between the elements of the array.

If you also want to add this separation string at the very beginning or the very end (that does not happen automatically - just *between* the array elements!) you can use the following trick:

```
$string = join('-', '', @array);
    # "-abc-def-xyz
```

push, pop, shift, unshift

- add/remove “at the end” of an array: `push()` and `pop()`

```
@x = qw(a b c);

push(@x, ’d’);     # @x is (a,b,c,d)

$elem = pop(@x);   # @x is (a,b,c), $elem is d
```

- add/remove “at the beginning” of array: `unshift()` and `shift()`

```
$elem = shift(@x);      # @x is (b,c), $elem is a

unshift(@x, ’0’, ’1’);  # @x is (0,1,b,c)
```

Those functions allow to add or remove elements from/to an array. `push()` and `pop()` work on the right side (the end) of an array.

You can also push more that one element at the same time onto the array. This is of course not true for `pop()`!

```
push(@x, ’e’, ’f’);
```

`shift()` and `unshift()` work on the left side (the beginning). Otherwise, they are equivalent.

OPTIONAL: grep, map

```
@x = qw(x abc def);

@y = grep(/.../, @x);
print(join(':', @y), "\n");     # gives "abc:def"

@y = map(length, @x);
print(join(':', @y), "\n");     # gives "1:3:3"
```

Some commonly used functions on an array.

`grep()` is very interesting: it does to a list what it's name giver (the Unix command grep(1)) does to the lines of a file: It "greps" out the lines that match a certain pattern. The resulting array contains a subset of the elements in the original one, namely the ones that matched for the given pattern. In the example above, the pattern consists of three characters, so 'abc' and 'def' match, but not 'x'.

`map()` finally is my favourite Perl function. It applies a given function to each element of an array. And it returns a new array, which contains the individual results of each application of the given function. In the example above, the `length()` function is applied to all strings in the original array. The resulting array is a list of all the length's of the strings.

Functions for Hashes – 6.3

- getting all keys from a hash: `keys()`

```
%x = ( 'Name' => 'Ammann',
       'Firstname' => 'Joe' );

@x = keys(%x);    # @x is (Firstname, Name)
       # or (Name, Firstname) ! Hashes are not ordered!!
```

- getting all values from a hash: `values()`

```
@x = values(%x);  # @x is (Joe, Ammann)
```

- fast iteration over keys and values: `each()`

```
while ( ($key, $value) = each(%x) ) {
    ....     # loop over each key/value pair
```

Typical operations on hashes are getting a list of all keys or getting a list of all values. Normally, the resulting lists will then bey used to iterate on every element of a hash. So a typical Perl code fragment is:

```
foreach (sort(keys(%ENV))) {
    print "$_  : $ENV{$_}\n";
}
```

When a hash is very big (has many elements), this operation can be slow. The reason is that Perl has to do the work many times: First it has to retrieve a list of all keys, and then for every single key, it has again to go into the hash and search for the value of that specific key.

So when you are iterating over a big hash, and you do not necessarily have to sort the keys, a loop with the `each()` function will be faster!

Managing hash elements

- check if an element exists: `exists($hash{'ELEMENT'})`
- check if element has a value: `defined($hash{'ELEMENT'})`
- removing elements from hash

```
if (defined($x{'Name'}) {
    ....     # check if there is a value for key 'Name'

delete($x{'Firstname'});       # delete from hash
```

Typical operations on hash elements are:

Check if a certain key exists in the hash.

Check if the key has actually a value. There is a slight difference between those 2 operations: The key might be in the has, but with an undefined value.

And sometimes you also want to remove an existing element from a hash.

```
delete($ENV{'PATH'});
```

Write your own Subroutines – 6.4

- structure your programs
- local variables declared with `my()`

```
sub func {
    my($value);
    ....
    return $value;
}
```

- `return` passes back something to the caller
- can also be an array!

```
$returnValue = func();
@returnArray = func_which_returns_array();
```

Perl allows you to write your own subroutines.

Perl also has the notion of “local variables”, that is variables that are just defined inside a subroutine. You can declare as many local variables as you want at the beginning of the subroutine.

Finally, Perl is very flexible when it comes to returning values from a function. Not only can you return a scalar value, but it is just as easy to return a whole array!

And if you ever need to return 2 or 3 different values from a subroutine, you don’t have to revert to “strange concepts” such as io/out parameter. Just do it!

```
sub func_with_2_return_values {
    ....
    return ($first, $second);
}

($firstreturn, $secondreturn) =
    func_with_2_return_values();
```

Argument passing

- argument passing via magic `@_`

```
sub addit {
   my($a, $b) = @_;
   return $a + $b;
}

$sum = addit(1, 2);
```

If you come from the world of Pascal or Java programming, you might wonder that there are no function prototypes. That is, you do not have to specify how many variables of what type a subroutine takes as parameter. Also, you don't have to specify what a subroutines will return.

[Actually, recent Perl version allow you to specify something like prototypes for subroutines, but almost nobody uses them :-]

Parameter (or argument) passing to subroutines is realized via the magic array `@_`. This is somewhat similar to the `$_`. But note the difference: one is an array, the other a scalar variable!

Theoretically, you can access the parameters inside subroutines also via something like this:

```
sub addit {
    return $_[0] + $_[1];
}
```

But this is not very readable!

Exercise – 6.5

- Write a subroutine to take a numeric value from 1 to 9 as an argument and return the English (or French, if you prefer :-) name of the number (such as `un, deux`).
- In the same file, write code to actually use this subroutine. So you should read input, call the subroutine and print out the return value.

OPTIONAL: Functions vs. (list) operators – 6.6

- many built-in Perl constructs can be used as either
- function: something to call with a sequence of arguments
- list operator: something to be applied to a list
- the big difference is precedence

```
# a typo in the file name
open(PW, "/etc/paswd") || die "can't open\n";

# open used as list operator
open PW, "/etc/paswd" || die "can't open\n";

# this now means !!!
open PW, ("/etc/paswd" || die "can't open\n");
```

Principle of least surprise: Use parentheses!

The Perl built-in functions can be either used "as functions" or as list operators. You can automatically use them as functions by putting parentheses around the arguments.

```
$result = function($arg1, $arg2, $arg3);
```

This is how you are used to call functions in programming languages that are less flexible than Perl! Or that have fewer boobietraps than Perl - however you like to put it.

But Perl gives you another way to call "functions", namely without parentheses:

```
$result = function $arg1, $arg2, $arg3;
```

This can have unexpected effects on precedence, if there are some other statements following (or preceding) the function call.

To keep it exact: There are not only list operators that have these "difficulties", but also so called unary operators, that work on a single argument. The reasoning is the same! And BTW: Effectively, there *aren't any functions* in Perl at all! With the parentheses, you just give a fixed precedence to the arguments passed to the operators - but this is programming language details!

7. File I/O

Lesson Overview

- accessing files, filehandles
- file test operators
- quickly touch other aspects of files (chown, unlink, etc.)

Lesson Goals

- learn how to open, read, write and close files
- know the file test operators
- know that there's much more to files!

We'll learn how to do basic I/O from and to files. For this we will officially introduce the concept of filehandles, and show how to use them.

File handles – 7.1

```
$input = <STDIN>;
```

- `STDIN` is a file handle
- pre-opened at start: `STDIN`, `STDOUT`, `STDERR`
- file handles are named in ALL UPPERCASE (by convention)

You have already learned how to use the pre-opened file handles. Here we now officially introduce file handles, and show you how to open your own files.

First, it is important to note that by convention, all file handles are written in all uppercase characters. This is not technically enforced, but everybody who ever has to read your Perl program (yourself included !) will hate you if you don't follow this rule!

Opening files

- opening a file assigns it to a file handle

```
open(FHANDLE, "/etc/passwd");      # opens for reading

open(FHANDLE, "</etc/passwd");     # same

open(FHANDLE, ">/tmp/log");
    # create or truncate the file, open for writing

open(FHANDLE, ">>/tmp/log");       # append to file

open(FHANDLE, '>', $file_name);    # safer version with
                                   # three arguments
                                   # (recent Perl)
```

Opening a file happens - surprise! - with the `open()` function. `open()` takes 2 arguments:

- the file handle to which the file is bound if it can be successfully opened
- the name of the file as a string

Actually the name of the file is normally preceeded by one of the “redirection characters” that you might know from shell programming. This defines how the file will be opened:

<**/some/file** open the file read-only

><**/some/file** open the file for writing. If the file does not yet exist, create it. If it exists, truncate the file (delete the old contents)

>><**/some/file** open the file for appending. If the file does not yet exist, create it. If it exists, leave the old contents and append at the end

Excursion: die() – 7.2

- always check the result of `open()`
- if you can't open, sometimes you want to terminate

```
if (! open(EP, "/etc/passwd") ) {
    print STDERR "$0: can't open file\n";
    exit(1);
}

# simpler!
open(EP, "/etc/passwd")
    or die "can't open file, $!";

open(LOG, ">>/tmp/log")
    or warn "$0: continue without logging\n";
```

You must always check the result of opening a file, whether it worked or not. Sometimes, depending on the logic of your program, you'll want to terminate the program if you can't open a file, sometimes you just want to continue with a warning message.

To make this very common idiom easier to both write and read, Perl has the `die()` and `warn()` functions. Actually, there are even more differentiated functions than those.

`die()` just prints out the give message, and terminates the program with an exit code of -1. If the string given to `die()` is terminated with a `\n`, then this will be just printed. If there is no terminating newline, Perl will add the script name and line number where the `die()` was called.

Another very helpful thing when using `die()` is the `$!` variable. This contains the operating system specific error code of the last error that happened. So the `die()` example above would produce something like:

```
can't open file, No such file or directory at check.pl line 1.
```

Using Filehandles – 7.3

- read with diamond operator `<FILEHANDLE>`
- write by passing handle to `print()`

```
open(EP, "/etc/passwd") or die "...";
while (<EP>) {
    ...
}

open(LOG, "/tmp/log");
print LOG ("The result is: ", 5*3);

# or maybe more readable
print LOG "The result is: ", 5*3;     # no comma after LOG !!!

close(LOG);
```

You can use your own filehandles just as you would use the pre-opened ones STDIN, STDOUT and STDERR.

So you read from those files with the diamond operator. The diamond operator read the file line by line. But be cautious! If you use the diamond operator in list context (you remember :-), it will read in the whole file in one, return an array of the individual lines.

```
$line = <EP>;
@lines = <EP>;    # slurp in all lines at once!
```

To send output to a file, you normally use the `print()` function. When you want to send output to another file than `STDOUT`, you have to pass the filehandle to print.

This shows you again clearly that there actually are *no Perl functions at all*. Everything effectively is an operator.

File test operators – 7.4

- operators to check for existence, readability, etc. of a file
- modelled after the `test(1)` command in Unix

```
if ( -e $file ) {
    ....
}
```

-r file or directory is readable

-w file or directory is writable

-x file or directory is executable

The `-e` operator above check for the existence of a file. Other operators are `-r`, `-w` and `-x`. There are many more!

-z file exists and has zero size

-s file or directory exists and has non-zero size (actually returns the size in bytes

-f entry is a plain file

-d entry is a directory

-l entry is a symbolic link

```
foreach (@list_filenames) {
    print "$_ is readable\n" if -r;
}
```

OPTIONAL: Some functions on files – 7.5

- many well-known Unix commands as built-in's
- can be *much* faster than the comparable shell script

```
chown($uid, $gid, $f1, f2, ...);   # -1 for $gid to keep group
chmod(0750, $f1, $f2, ...);

unlink($f1, $f2, ...);
link($oldfile, $newfile);
symlink($oldfile, $newfile);
rename($oldfile, $newfile);
```

Perl also provides the most common operations on files as internal Perl functions. The list above shows you the most important ones.

Note that a Perl script that operates with these functions on a large number of files can be *much* faster than the equivalent Shell script. This is because Perl implements these operations as internal function, whereas the Shell has to create a new process for each file operation!

Exercise – 7.6

- Modify the `reverse.pl` script so that you can give the name of the file to be reversed on the command line.
- Try to use the `reverse.pl` script from above to files which do not exist (or for which you don’t have read permission. Modify the script so that it behaves correctly with meaningful error messages.

8. Perl Modules

Lesson Overview

- What is a Perl module?
- using modules
- getting/updating modules from CPAN

Lesson Goals

- know how to use Perl modules
- know where to look for existing modules

With Perl modules, you can extend the functionality of Perl. A Perl module can provide additional functions to be used from a Perl script.

There are hundreds (if not thousands) of Perl modules freely available. There’s probably a module for everything!

The CPAN (Comprehenisive Perl Archive Network) collects all these modules and makes them available.

Using a module – 8.1

- Perl distribution comes with many standard modules
- files ending with `.pm`
- in directory `/usr/lib/perl5/<VERSION>`

```
#! /usr/bin/perl

use File::Copy;    # /usr/lib/perl5/.../File/Copy.pm

copy("/etc/passwd", "/tmp/passwd");
```

The most commoly used Perl modules are normally delivered with the Perl distribution, so you should already have them installed on the system. Perl modules come as files with the ending `.pm`.

The default Perl modules are normally installed in `/usr/lib/perl5/<VERSION>` (this could also be `/usr/local/lib/perl5/<VERSION>`). There is normally also a directory where one can install the site specific Perl modules, typically `/usr/lib/perl5/site_perl`.

To use one of the existing Perl modules, just find out which modules you want to use, and then say

```
use File::Copy;
```

This will load the appropriate module, and make the functions defined in this module available.

You can get documentation about the modules with the command `perldoc <module>`.

Some CPAN modules

Getopt::Std parse command-line options (-d, etc.) to your Perl program

File::Copy copy and move files

File::Find traverse a tree of files with many options

POSIX access to almost all system calls of a POSIX operating system

There are so many modules, that it is hard to pick from. Have a look at

```
http://www.cpan.org/modules/01modules.index.html
```

to get an impression. Some of the modules distributed with almost all Perl installations:

`Getopt::Std` gives you the functions `getopt()` and `getopts()`, to parse optionals command-line arguments to your Perl script.

`File::Copy` provides you with an easy and fast `copy()` function

The `POSIX` module gives you access to all POSIX routines of your operating system. This is one of the cases, where you probably do not want *all* functions defined by this module, but just some of them. To do this:

```
use POSIX ('setsid');

setsid();     # call POSIX setsid()
```

Installing a non-standard module

- most CPAN modules not in distribution
- getting a newer version of a module
- standard way of installing

1. Download and unpack (tar.gz files)
2. `perl Makefile.pl`
3. `make`
4. `make test`
5. `make install` (with correct privileges)

CPAN modules are grouped into categories. There is a wide variety of available modules. They all share a common method of installing.

Exercise: Go to CPAN and find a module called Devel::Trace. Download it and install it with the steps given above.

Make sure you execute the 'make install' step with root privilege.

Certain modules depend on other CPAN modules, which may also not be installed on your system. This sometimes gets you into a boring chain of downloading and installing several modules.

CPAN shell

- standard module CPAN.pm with access to CPAN
- initial configuration
- install modules with dependencies

```
# perl -MCPAN -e shell
.... initial first-time questions

cpan> install File::Slurp
..... download, build and install File::Slurp.pm
      with all dependencies
```

The `CPAN.pm` module is an easy way to download, build and install Perl modules in one step and with all required dependencies.

It is just a first-time configuration that is a bit hard to do :-)

Exercise: Let's install the File::Slurp module together.

If you don't have the most recent version of Perl installed, a the CPAN module will prompt you to upgrade Perl itsself "on the fly". In most cases, you should not do this and abort as soon as possible. This has been an ongoing source of frustration with CPAN module users and will probably be changed in the future.

If you don't execute the CPAN shell as root, it will obviously fail during the install step. To avoid this, set the location in the CPAN.pm options where Perl modules will be installed.

```
o conf                  # shows current CPAN.pm options

o conf makepl\_arg LIB=/path/to/your/perllib
```

Pragmas or "Where is module strict.pm" ? – 8.2

- `use strict` is a compiler pragma, not a module!
- a (hopefully unsuccessful :-) attempt to confuse programmers
- other pragmas
  - `use integer;`
  - `use vars qw($var1 $var2);`

One might wonder where the module 'strict.pm' has gone. The 'use' keyword of Perl has 2 functions. One is to load modules as we have seen. The other are pragmas, instructions for the Perl compiler.

The most commonly used pragma is `use strict`, to enforce strict checking of the use of variables, functions and references. There are other pragmas like `use integer`, which tells the compiler to use integer arithmetic instead of the default floating point arithmetic.

Another useful pragma is `use vars`. This is used in modules that don't declare all the variables that they are exporting. A common example are the `Getopt::` modules, where one is using variables like `$opt_d`, etc. and not all are exported.

Write your own modules – 8.3

- Perl script with .pm suffix
- define variables, functions
- module is mostly a namespace
- avoid naming conflicts

```
package My::Module;

use strict;
....

sub func1 { ... }

1;
```

A Perl module is really not much more than a regular Perl script that just defines some functions and/or variables. It must have a .pm suffix.

A Perl module does not have a '#!' line. It normally starts with the definition of the package it represents. This opens a new namespace, and all functions or variables that are defined afterwards are in that namespace. This is mainly done to avoid name collisions.

Other than that, the only noteable thing in a Perl module is the habit of writing the last line as follows:

```
1;
```

The reason for this is that when a Perl module is used (loaded) by another Perl script, it is basically evaluated with 'eval'. So the module must terminate with a true value, otherwise the eval fails and the Perl script terminates.

The real story behind this is a bit more complicated, but this is correct for 99% of the cases.

Using your module

- place your .pm file in a library directory
- 'use' it like a regular module

```
#! /usr/bin/perl

use strict;
use My::Module;

My::Module::func1();
```

To use your own module just like any regular module, you must place it into one of the directories where Perl will look for it. The list of these directories is determined when Perl is compiled. It is stored in the internal array `@INC`. The actual value can be queried by running

```
perl -V
```

If you don't have the permission to save your module into one of these directories, you can use either of the following methods in your Perl script that uses the module.

```
# method 1, use lib pragma
use lib "/path/to/dir";

# method 2, put dir at beginning of module path
BEGIN { unshift(@INC, "/path/to/dir"); }

# method 3, put dir at end of module path
BEGIN { push(@INC, "/path/to/dir"); }

# method 4, set PERL5LIB environment variable
# before calling perl, do something like
export PERL5LIB=/path/to/dir
```

Module initialization code

- sometimes needed
- e.g. load config file
- BEGIN block executed during load of module
- END block also possible

```
BEGIN { .......; }
```

Initialization code is a handy feature for some modules.

Of course, this can also be used with regular Perl script (as shown on the slide before when pushing/unshifting some directory onto the module path).

Using the Exporter – 8.4

- export some functions or variables into global namespace
- call `func1` instead of `My::Module::func1`

```
package My::Module2;

use strict;
use Exporter;

our($VERSION, @ISA, @EXPORT);

$VERSION = 2.00;
@ISA    = qw(Exporter);
@EXPORT = qw(func2 $var2);
```

The Exporter is a tool (it is also a module) to put certain functions and/or variables from the module namespace into the global namespace `MAIN::`. This allows a Perl script that uses this module to access the functions/variables without the package prefix.

Exercise: Write a module that exports one function and one variable. Use this module in a script.

9. Perl Debugger

Lesson Overview

- how to debug Perl programs
- built-in debugger
- other debuggers (Tk, DDD)

Lesson Goals

- interactively debug a Perl program
- know the options of other debuggers

The Perl interpreter has a built-in debugger which allows you to interactively step through your program.

There are also other debuggers that you can use with Perl, e.g. there is a graphical debugger using the Tk Perl module.

The built-in debugger – 9.1

- just start Perl with `-d`

```
prompt> perl -d xxx.pl

Loading DB routines from perl5db.pl version 1.0402
Emacs support available.

Enter h or ‘h h’ for help.

main::(xxx.pl:3):       print "Hello\n";
  DB<1> _
```

When you start Perl with the `-d` option, it will load the script, show you the first statement of the script and then wait for your input. The Perl debugger waits for you to give it a command.

Debugger commands

n execute next statement, do not go into a subroutine, rather step over it

s single step, e.g. execute next statement or dive into the subroutine.

l list the next 10 lines of the script. An optional argument can be a number (how many line) or a range (20-40, meaning lines 20 to 40)

p print, give it an expression, and the Debugger prints the result (e.g. `p $value`)

b set a breakpoint. You give it either a number (the line number in the script) or the name of a subroutine.

c continue, run the script until a breakpoint is hit

r return, finish the current subroutine that the script is currently executing and then stop

Exercise: Run one of your scripts statement by statement through the debugger.

Debugging Tools – 9.2

- Perl provides a whole debugger framework
- hooks for `Devel::....` modules
- look on CPAN (profiling, stack traces etc.)

```
perl -d:Trace xxx.pl
```

Exercise: Run one of your scripts with Devel::Trace.

BTW: Modules normally have a man page: `man Devel::Trace`

The Perl debugging environment is more than just a simple text debugger. It is a whole framework that allows modules to use several hooks to get internal information about Perl workings.

On CPAN, there is a number of `Devel::....` modules that use these hooks. Examples are Profilers (module that show you which of your Perl functions use a lot of CPU), Regexp debuggers, etc.

We downloaded one of these modules, `Devel::Trace`. This module shows you every line of Perl code that is executed, much like `sh -x xx.sh` does for Perl scripts.

Graphical debuggers – 9.3

- several graphical IDE for Perl (many commercial)
- one very useful free variant: GNU DDD - Data Display Debugger

Exercise: If not yet installed, install the Debian package ddd (with `apt-get`) and run one of your Perl scripts through it:

```
ddd myscript.pl
```

- graphical Tk Debugger `Devel::ptkdb`

```
prompt> perl -d:ptkdb myscript.pl
   ... goes to X environment and displays window
```

10. References

Lesson Overview

- What is a reference
- using references
- Arrays of Hashes, etc.

Lesson Goals

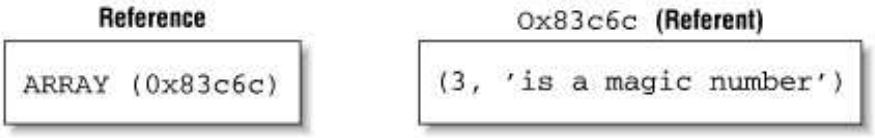
- learn why reference are necessary
- know the basics of using references

If you know C or C++, you know about pointers. Sometimes, pointers are very useful constructs.

Perl also has a similar construct, namely references. You can take the reference of a variable, an array or a function and you get back ’a pointer’ to it.

Why references? – 10.1

- use complex data structures (records, arrays of arrays)
- pass arrays/hashes as function arguments




```
sub takeargs {
    my(@array, $scalar) = @_;

    print "Array: @array, Scalar: $scalar\n";
}

takeargs( (1, 2, 3), "abc" );
```

References are vital in Perl to use complex data structures beyond scalars, arrays and hashes.

But also for passing arrays (or hashes) as arguments into functions, references are very helpful!

Trivia: What will the code snippet above display ?

```
a) Array: 1 2 3, Scalar: abc
b) Array: 1 2 3 abc, Scalar:
c) Array: 1, Scalar: 2 3 abc
```

Taking a reference – 10.2

- backslash operator \
- reference of scalars, arrays, hashes

```
$scalar_ref = \$myvariable;
$array_ref  = \@mylist;
$hash_ref   = \%myhash;

$$scalar_ref = 5;
print $myvariable;    # prints ’5’
```

The backslash operator is used to take a reference of an existing variable. One can take a reference of just about anything that exists in Perl. Scalars, arrays and hashes are the most commonly used.

You can also take a reference of a constant:

```
$pi = \3.14159;

$$pi = 4;     # runtime error
```

Using references

- commonly called 'dereferencing'
- rule: prefix the reference variable with the character ($, @, %) of its type

```
print $$scalar_ref;

foreach $elem (@$array_ref) {
  ....

keys(%$hash_ref);
```

Dereferencing a reference as a whole is done by prefixing it with the character that identifies its type. It can then be used again as a scalar, an array, or a hash.

If you think it is more readable, you can surround the reference variable with curly braces.

```
print ${$scalar_ref};

foreach $elem (@{$array_ref}) {
  ....

keys(%{$hash_ref});
```

Using elements of array or hash references

- as always is Perl: TMTOWTDI :-)
- for abstract thinking people - this is possible:

```
print "Second element: ${$array_ref}[1] \n";
print "Key element: ${$hash_ref}{'key'} \n";
```

- for the rest of us - this may (?) be more readable:

```
print "Second element: $array_ref->[1] \n";
print "Key element: $hash_ref->{'key'} \n";
```

It is common to access individual elements of array or hash references. As a very abstract language, Perl of course allows the notation as shown in the first example.

It is very logical:

`@something` is an array

`$something[1]` is the second element of this array

For most people, the pointer-arrow notation to access individual elements is more readable. It is in fact the very same thing, just written differently.

Anonymous data

- commonly used for arrays and hashes
- avoid assigning to a variable first

```
$aref = [ 3, 4, 5 ];   # anonymous array
$href = { "How" => "Now", "Brown" => "Cow" };
```

This technique is commonly used. It avoids the necessity to give names to arrays before handling references to them.

Exercise: Write a Perl script with a function, and pass into this function one array, one hash and 2 scalar variables.

Complex data structures – 10.3

- limited data types in Perl
- references allow arbitrarily complex structures
- hashes of arrays
- arrays of arrays
- etc.etc.

While Perl is basically limited to very few datatypes, references allow to build very complex data structures.

For example, when you want to use an hash, but you need to store more than one value per key, a hash of arrays comes in handy.

Let’s assume you want to go through a list of image files, and depending on the suffix of the filename, store the list of GIFs, JPGs and PNGs in a hash.

```
%images = ();
...
if ($filename =~ /.*\.gif$/) {
   push(@{ $images{’gifs’} }, $filename);
} elsif ($filename =~ /.*\.je?pg$/) {
   push(@{ $images{’jpgs’} }, $filename);
}
...
```

Records

- no real record data structure in Perl
- best simulated with reference to hash
- hash key treated like a record member

```
$Joe = { "Name"     => "Joe Ammann",
         "Address"  => ".......",
         ....
       };
```

A reference to a hash can be treated like a record in other programming languages (struct in C/C++).

Storable.pm module

- useful module for persisting complex data structures
- `store()` writes to file (binary)
- `nstore()` writes to file (portable)
- `retrieve()` reads from file

```
use Storable;

store(\%hash, "filename");

$href = retrieve("filename");
```

The Storable.pm module allows to persist complex data structures to files.

Copying of data structures is also made easy with the `dclone()` function.

11. Perl command line options

Lesson Overview

- important command line options
- invoking perl directly

Lesson Goals

- learn to use Perl on the command line

Normally, Perl is used for executing scripts that are first typed in an editor, and then executed.

But Perl can also be invoked directly from the command line, without first writing a Perl script. Here, we show you some important options.

Command line options – 11.1

- man perlrun
- do tasks normally done with awk, sed, grep and the like
- `-e` to execute Perl code directly

```
$ perl -e 'while (<>) { print if /^root/; }' /etc/passwd
```

Write little Perl scripts directly on the command line. This is often very useful as a replacement for sed, awk, grep or a combination thereof.

The advantages of using Perl instead of the 'usual suspects' for these tasks are:

- no arbitrary limitation on line length, file size, etc.
- normally a lot faster on larger files
- avoid long, slow pipes (`sed ... | awk ... | grep ... | sed`)

-p and -n Options

- in conjunction with -e
- build automatic per-line-loops around the Perl code
- -n just builds the loop
- -p also prints $_ at the end

```
$ perl -n -e 'print if /^root/;' /etc/passwd

$ perl -p -e 's/:/-/g;' /etc/passwd
```

Options -n and -p are used together with -e. They make Perl behave very much like sed, one line after the other of the input is processed.

-n builds the following loop around the code specified with -e:

```
LINE:
  while (<>) {
     ...             # your program goes here
  }
```

-p builds the following loop around the code specified with -e:

```
LINE:
  while (<>) {
     ...             # your program goes here
  } continue {
     print or die "-p destination: $!\n";
  }
```

-i Option

- normally used together with `-pe`
- modify files ’in line’
- avoid creating temp file and moving it back
- optionally create backup file

```
$ perl -pie ’s/foo/bar/;’ file1 file2

$ perl -pe -i.orig ’s/foo/bar/;’ file1 file2
```

The -i option is commonly used to modify the contents of files ’inline’. This means without first creating a temporary file and then moving the file back.

A similar command line with sed would look like:

```
sed -e ’s/foo/bar/’ < file1 > file1.tmp
[ $? -eq 0 ] && mv file1.tmp file1
```

-l Option

- normally used together with `-ne`
- `chomp()` input lines

```
find /tmp -atime +7 | perl -nle unlink
```

Option -l can be used to have input lines chomp()'ed, that means the end of line character is automatically removed.

This is very useful if the input lines are file names, which should be processed by some Perl function.

The example above removes all files in /tmp that haven't been accessed for a week.