Basic Concepts in Perl

Lesson 3

processing the data:

functions and modules

Core Perl Functions
The Standard Perl Library

- More than 250 built in functions instantly callable
- described in the perlfunc man page

- Much more functions in the Standard Perl Library
- Organized into Modules/Packages
- nearly all functions defined in the POSIX standard available with `use POSIX;`

- Additionally many modules installed from CPAN see `perldoc perllocal`

The CPAN

- Comprehensive Perl Archive network
- Overview of the 5000+ Modules e.g. At

  ftp://ftp.uni-hamburg.de:/pub/soft/lang/perl/CPAN/modules/00modlist.long.html
- Modules needed at DESY will be installed on request
- Commands `cpan` (UNIX) and `ppm` (NT) simplify installation
- **Use perl modules instead of calling system commands**
  - faster, the overhead of spawning new processes is big
  - parsing the command output is worse than using an API
  - huge amounts of code already written and debugged
  - modules are usually very well maintained

Subroutines

- Declaration/Definition: `sub name {Statements;}`
- Declaration (prior to Definition): `sub name;`
- Subroutines (functions) can be called with parameters and can return a scalar or a list
- `$retval = name(Parameter_list);`

  `@retlst = name Parameter_list;` # ***name*** is declared

  `&name;` or `name();` # if ***name*** not declared
- Number or Type of Calling Parameters normally not in Definition
  Correspondence of Call and Definition has to be ensured by Programmer
- Definition with Prototypes possible, not covered

Subroutine examples

```
sub callme{  print "Sub\n";  }
sub private {
  my ($par1, $par2, $par3) = @_;
  print "\$par1 = $par1\n";
  $par2 = 0; # The value in the calling program remains intact
  $_[0] = 1; # The value in the calling program gets overwritten
  return $par2;
}
callme; #defined subroutine without parameters
# equivalent call: &callme(); or callme(); or &callme;
$a=33;
$b = private $a, 44;
print "Value of \$a (par1 in private) after the call: $a\n";
print "Return value of subroutine private: $b\n";
```

Passing Parameters

- All Parameters passed in a single (Parameter) List
- Subroutines see the parameter list as the array `@_`
- The array `@_` gets propagated when calling a subroutine with the **`&callme;`** notation (no explicit parameters)
- Parameters get passed by Reference

  Changing elements of `@_` acts back to the calling program!
- Separately passed arrays get flatted out in `@_`

  pass references to arrays instead of the arrays to avoid it
- Return value is Value of the last assignment
- Can be given explicitly by

  `return $value;` or **`return @value;`**

File locking

- Two file locking mechanisms: `flock` and `fcntl`
- `fcntl` is the OS dependent system call
  - it usually does a better job on same architecture
  - not available on all platforms
  - may be incompatible between different architectures
- `flock` is always implemented
  - might use internally both the system flock or fcntl
  - might be to weak for a safe locking of files
- Better do not rely on a fool proof file locking

Exception Handling

- Simplest form of error handling is
  - Checking for return codes of programs and functions
  - reporting return codes (**`$?`**) and error messages (**`$!`**)
  - for handling abnormal situations use **`warn`** or **`die`** or **`use Carp;`** with the functions **`carp`** or **`croak`**
- Both run and compile time errors can be catched
  - compile time errors with **`eval`** ***`expr`***
  - run time errors with **`eval {`** ***`block`*** **`}`**

The eval function

- Argument of `eval` is regarded as perl code
  - `eval` ***`expr`*** Syntax check at run time, not possible at compile time, as ***`expr`*** may be built dynamically
  - `eval { `*`block`*` }` Syntax check at compile time
- `eval` returns values like in subroutines
- Errors during `eval` execution get trapped
  - then the return value is zero and
  - `$@` contains the run or compile time error message
  - otherwise `$@` is guaranteed to be empty
- Similar to try and catch from C++

Eval: An example

```
eval "This is not a Perl Program.";
print $@;
# dynamic program generation and execution
$myprog = 'print "3*7 yields ", 3*7, "\n"';
eval $myprog;
eval { 10/$b }; # Division by zero
if ( $@ ) {
  print $@; #or do something else
}
print "... and the Program goes on\n";
```

Access to System Information

- Group of functions that handles contents of UNIX specific information (/etc/passwd, /etc/group etc.)
- Some functions may be available on NT
- Naming convention **`getxxx, setxxx, endxxx`**
- Most important functions
  - **`getpwnam, getpwuid, getpwent # passwd info`**
  - **`gethostbyname, gethostbyaddr # DNS`**
- For NT specific tasks additional Modules available
  - **`Win32::AdminMisc`** (in Win32-AdminMisc) and
  - **`Win32::NetAdmin`** (in libwin32)

Extracting Account information

- The same construction as before to get only a selected number of return values from **`getpwnam`** assigned

```
 ($name, $uid, $shell)=(getpwnam("friebel"))[0,2,8];
print "User $name, uid=$uid has $shell\n";
```

Loops with map and grep

- Functions map and grep implicitly perform loops

  ```
  @sizes = map { -s $_ } @files;
  ```

  is equivalent to

  ```
  for ( @files ) { push @sizes, -s $_; }
  ```

- grep evaluates the Block and returns the elements of the array for which the expression was true

  ```
  @mylines = grep { /my/ } @lines;
  ```

  is equivalent to

  ```
  for (@lines) { push @mylines, $_ if /my/; }
  ```

Map and grep (2)

- Changing $_ in map and grep changes the content of the input array. In such cases a for loop is more readable

- map and grep tend to make the code more unreadable

- Typical uses of map and grep:
  - Use map to transform an input array into a new array
  - Use grep to extract elements with certain features from an array

Example: map and grep

```
# Construct the AFS home directory path of some users
@users = qw (leich nieprask fatima friebel);
$prefix = "/afs/ifh.de/user/";

# map array @users into array @homes
@homes = map { $prefix.substr($_,0,1)."/$_" } @users;
print join("\n", @homes), "\n";

# extract home directories containing the chars /f/
@dirs = grep { ($_ =~ m|/f/|) } @homes;
print "Users with initial letter f:\n", join("\n",
  @dirs), "\n";
```

Array Processing

- Functions **`shift,unshift,push,pop`** and **`splice`**
- **`shift`** &Co. are special cases of **`splice`**
- **`push/pop`** extend/truncate the array at the end
- **`shift/unshift`** extend/truncate array at the begin

  `splice` ***Array, Offset, Length, Values***

`shift @ARGV;`	`splice(@ARGV,0,1);`
`unshift @a,$val;`	`splice(@a,0,0,$val);`
`push @a,$val;`	`splice(@a,$#a+1,0,$val);`
`pop @a;`	`splice(@a,-1);`

Manipulating @ARGV

- Contains the list of arguments when script is called

```
@ARGV = qw( -a -bc2 file1 file2);
$par1 = shift;
print "Par1: $par1, further Arguments:@ARGV\n";
unshift @ARGV, $par1; # undo the shift
$lastarg = pop @ARGV;
print "Last Arg: $lastarg, further Args:@ARGV\n";
push @ARGV, $lastarg; # undo the pop
$file1 = splice @ARGV, 2, 1;
print "File Arg: $file1, further Arguments:@ARGV\n";
splice @ARGV, 2, 0, $file1; # undo the splice above
```

Processing Command Line Switches

- Do not code your own switch processing
- Getopt::Std and Getopt::Long come with Perl

```
use Getopt::Std;
getopts(’ab:c:d’) or Usage(); # -ab 3 -d
```

accept options abcd, bc require a value, sets variables: `$opt_a/d,` true/false, set `$opt_b/c` to a value

- Getopt::Long more flexible (corresponds to GNU standard)

```
use Getopt::Long;
GetOptions(Option_descriptions) or Usage();
```

Time and Date

- `time` returns the number of seconds since 1.1.1970
- `gmtime` and `localtime` convert seconds into

```
(sec, min, h, day, mon, year, weekday, yrday, isdst)
```

  - mon counts from 0, year counts from 1900 i.e. Dec=11, year 2001=101 (C library conventions)
  - weekday starts with 0 (Sunday)
- Don’t code date arithmetic yourself
  - powerful Modules `Date::Manip` and `TimeDate` in CPAN, installed at DESY

Submodules `Date::Format` and `Date::Parse`

Date Manipulations

```
Use Date::Manip;

#The Timezone processing in Windows does not work:
$ENV{TZ} = 'MET';
$date = ParseDate("3rd Tuesday in Jan 2001");
($yr, $mon, $day) = unpack("A4A2A2", $date);
print "The lesson took place at $day.$mon.$yr\n";
print "2000 was a leap year\n" if Date_LeapYear(2000);
```

Context of execution

- Subroutines (and Operations in general) act in a context
- Most important contexts are scalar and list contexts
- Context is usually defined by left hand side of an assignment
- Some Operations act differently depending on Context :

```
$a = @field #  left hand side is scalar,  $a => $#field
@a = @field #  LHS is an array,               @a => @field
$date=gmtime(); # Thu Jan 20 10:38:17 2000
@date=gmtime(); # (17,38,10,20,0,100,4,19,0)
```

- Default is List Context, Scalar Context can be enforced:
  `print scalar gmtime(),"\n";`

Sorting

do provide function byvalue

- `sort Function_or_Block List`
- `Function_or_Block` with 2 Arguments `$a` and `$b`
- Return Value -1 (a < b), 0 (a = b), 1 (a > b)
- Default is `{$a cmp $b}` if no function provided
  - alphabetic sort is achieved with `{$a cmp $b}`
  - numeric sort is done by `{$a <=> $b}`
- Compact sort expressions often seen in programs:

```
for (sort @array) { ... }
for (sort byvalue keys %hash) { ... }
```

The Schwartzian Transformation

- Sort function gets called proportional to n log n times
- For costly sort functions it is better to call the function for each element once and remember the values: “Schwartzian Transformation”

```
@sorted=map{$_->[1]}sort{$a->[0]<=>$b->[0]}
           map{[compute(),$_]} @unsorted
```

A Sort example

```
@unsortéd = qw( c=1 D=2 a=2 b=3);
#sort numerically descending, then
  alphabetically ascending
@sorted = map { $_->[0] }
        sort { $b->[1] <=> $a->[1]
                        ||
               $a->[2] cmp $b->[2]
        } map { [$_, /=(\d+)/, uc($_)] }
  @unsorted;
print "@sorted\n";
```