Basic Concepts in Perl

Lesson 1

Reading and Writing your data

Input and Output in Perl

- Files are opened with `open HANDLE, string`
  - `STDIN`, `STDOUT` and `STDERR` already open
  - *`string`* defines **what** file gets opened **how**

`"file"`	read only	`"<file"`	also read only
`">file"`	write only	`">>file"`	append to file
`"+<file"`	open for reading and writing		

- Example:

```
$file = "/AUTOEXEC.BAT";
open FH, $file or die "Error opening $file: $!\n";
```

leaving out `\n` prints also the location of the error

Short-Circuiting the Expression Evaluation

no further evaluation, if truth value known

```
$a && print "1";   # print only if $a true
$a || print "2";   # print only if $a false
```

- Using these shortcuts is good practice, it helps often to avoid if/then/else constructs, the code gets more compact
- Do not overuse this feature, use it to increase readability
- **Readability is an important design criterion**

I/O with external Programs

`"cmd|"` read from cmd `"|cmd"` write to cmd

- simultaneous reading **and** writing in Pipe not possible, but:
  - under UNIX there are the perl functions `open2/open3`
  - functions accessible after `use IPC::Open2;`
  - used for interprocess communication (IPC)
  - Queries are sent to an output pipe (to a daemon)
  - Answers are read from input pipe (response from daemon)

Input Operations

- Reading from a file with `<HANDLE>`
  - in scalar context a line is returned
  - in list context the **whole file** is returned in array!!!
- To store a handle in a variable the globbing operator * has to be used: `$handle = *HANDLE;`
- `$/` (InputRecordSeparator) defines what is a line!

```
undef $/;     #whole file
$/="";        #a paragraph (up to empty line)
$/ = \num;    #a record of length <=num
```

  even multicharacter strings are allowed

Input Operations (2)

- Special handle `<>`
  - a.k.a. diamond operator
  - interpret `@ARGV` as file names and read from there
  - if no (more) files in `@ARGV` read from `STDIN`
- Special handle `DATA` reads text after `__DATA__` or `__END__` in the current (script) file
- Reading of single chars with `$c=getc HANDLE;`
  - if `HANDLE` omitted, read from `STDIN`
  - Close files after end of I/O : `close HANDLE;`

Example: File input

```
# DATA Stream is already open (lines after __DATA__)
{           # a new scope starts here
  local $/; # only valid in this block
   $/ = "";
  $headers = <DATA>; # read all mail headers in one go
  close DATA;
  print $headers;
}
open PROG, "Ex1.pl";
$magic = getc(PROG) . getc(PROG);
close PROG;
print "Ex1.pl is a script\n" if $magic eq "#!";
```

The readline function

- GNU readline is a library to support command line input
  - has command line editing, command line history
  - several input modes (raw, hidden, cooked)
- Not in the core of perl, but available on CPAN
  - Term::ReadKey and Term::ReadLine
  - installed at DESY (both UNIX and NT)

Programming a password dialog

```
use Term::ReadKey;
*IN = *STDIN;
ReadMode 2, IN; #hidden input
print "Your Password please: ";
my $password = ReadLine 0, IN;
print "\nYour Password was $password";
ReadMode 0, IN; #normal input
close IN;
```

Reading Directories

- works like reading of files:

```
opendir HANDLE, string;
while (<HANDLE>) {
  readdir HANDLE;  #$_ now contains Filename
  ...
}
#or without while: @files = readdir HANDLE;
closedir HANDLE;
```

- only filenames without path information returned
- all files (even . and ..) get returned by **`readdir`**

Output Operations

- Output with `print HANDLE list # no Comma!`
- Better control with `printf HANDLE format,list`
- If `HANDLE` omitted then output to `STDOUT`
- Alternatively data can be written by defining `FORMAT`s and use the function `write` (rarely used)
- When using pipes a flush of the buffers after each print/write can be necessary (”unbuffered”): `$|=1;`

Changing the print functionality

- `print` can be influenced by `$\`, `$,` and `$"`
- Output record separator is added after each record

```
{ local $\="\n"; # \n gets added now
  print "no linefeed required here"; }
```

- Output field separator is added after each element

```
{ local $,=","; #elements get separated by ,
  print "System is $^O", "did you know?\n";}
```

- List separator is added after each list element in a double quoted string

```
{ local $"="-"; local $\="\n"; local $,="+";
  @a=qw(1 2); print @a,"@a"; }# 1+2+1-2\n
```

Low Level I/O

- For better control and speed, but more difficult
- **`sysopen HANDLE, PATH, FLAGS, [MASK]`** opens files
- **`read`** , **`sysread`** read a number of bytes (like **`getc`**)

```
sysopen PROG, "Ex1.pl", O_RDONLY;
$magic = sysread PROG, 2, 0;
close PROG;
print "Ex1.pl is a script\n" if $magic eq "#!";
```

- **`syswrite`** write a number of bytes
- **`tell/telldir`** get position in file/directory
- **`seek/seekdir`** jump to position in file/directory
- **`truncate`** the contents of a file

Reading compressed files

- Compress::Zlib handles I/O with compressed files

```
# determine current directory
# for portable file name manipulation see also File::Spec
use Cwd;
use Compress::Zlib;
# read a gzipped file and count the lines in there
my $file = shift;
my $lines = 0;
my $gz = gzopen($file, "rb")
        or die "Cannot open $file: $gzerrno\n" ;
while ($gz->gzreadline($_) > 0) {
     $lines++;
}
die "Error reading from $file: $gzerrno". ($gzerrno+0) . "\n"
    if $gzerrno != Z_STREAM_END ;
$gz->gzclose() ;
print "File $file contains $lines lines\n";
```

Writing compressed files

- Compress a file with maximum compression
- Use the same module: **`Compress::Zlib`**
- lookup the documentation using perldoc
- Get more info with man zlib
- Finally find the real info in /usr/include/zlib.h (UNIX)
- See file **`01gzwrite.pl`** for a working script

Working with the Filesystem

- Many Function equivalents of UNIX/NT Commands
  - `chdir`(cd), `chmod`, `chown`(chown, chgrp)
  - `link` (ln), `symlink` (ln -s)
  - `mkdir`, `rmdir`, `unlink` (rm),
  - `utime`(acts on access and modify time, does not create files)
- Access to file information using `stat`, `lstat`, operators

```
$file = "Ex1.pl";
($mode, $size) = (stat $file)[2,7];
printf "File $file (length %d) has mode bits
  %o\n", $size, $mode;
```

Result wrapped into a list

Select Elements

Temporary files

- Big security hole if done improperly: The wrong way:

```
open (TMP, "/tmp/foo.$$") ...# seen in many places
```

- To be immune to security threats the file should
  - not reside in world writable directories
  - not have a predictable name
  - not already exist
- Hackers can place a hard or symlink in places where you are going to write. Instead of writing a temporary file you can find yourself overwriting important data

Temporary files (2)

- The correct way without File::Temp

```
use POSIX;
do { $name=tmpnam();
} until sysopen(TMP,$name,O_RDWR|O_CREAT|O_EXCL,0600);
# do something with TMP
close TMP;
unlink $name;
```

- With perl 5.6 and newer you can say

```
use File::Temp "tempfile";
(*TMP, $filename) = tempfile();
# do something with TMP
close TMP;
unlink $filename;
```

Where to read more

- Tutorial
  - perldoc perlopentut
- bidirectional communication
  - perldoc perlipc
- IO Layers
  - perldoc perlIO
- Function definitions
  - perldoc open, sysopen, opendir, readdir, flock, ...

Questions and Answers

- How can I read Characters from the Console without waiting for a <ENTER>?

```
use Term::ReadKey;
ReadMode 4; # Turn off controls keys
while(not defined($key=ReadKey(-1))) { #get key
}
print "Got key $key\n";
ReadMode 0;
```

- Are there other ways to mark an output stream for autoflushing? (Example uses **`STDERR`**)

```
$oldfh=select(STDERR); $|=1; select($oldfh);
```
or
```
use IO::Handle; STDERR->autoflush(1);
```