Basic Concepts in Perl

Lesson 4

Tuning, Debugging and Documenting your Code

Debugging

- Best strategy: write bug free programs
- Perl helps you with that by using
  - `perl -w` `#Bug in perl: -w is optional`
  - `use strict;` `#forces strong(er) typing`
  - Usage of CPAN Modules (usually well tested)
  - Usage of recipes from the literature (see Intro)
- If your program nevertheless seems to have bugs
  - print out intermediate results (poor mans debugger)
  - use a real debugger

Printing Debug Data

- Simply use `print $data if $debug;`
- use more advanced techniques for structured data

```
use Dumpvalue;
  my $dumper = new Dumpvalue;
  print "The variable \$client type ", ref
  ($client), " contains:\n";
  $dumper->dumpValue($client);
# we don't really want that:
# $dumper->dumpvars('main'); #all Variables
```

TMTOWTDI

- There is more than one way to do it:

```
use Data::Dumper;

  print "The variable \$client type ", ref
  ($client), " contains:\n", Dumper($client);
```

- Printing can be influenced by variables
- Output is valid perl code

Using Code development tools

- Call a program verifier (like lint, C language)
  - `perl -MO=Lint,all Ex5.pl`
- Produce a cross reference listing (very long!!)
  - `perl -MO=Xref Ex5.pl > Ex5_references.txt`
- Look for more tools based on perl code generators
  - see CPAN and e.g. the Camel book 3rd edition

Using the Perl debugger

- Perl comes with its own debugger
  - call with **`perl -d scriptname`**
  - important debugger commands (mostly 1 char)

    `h (help) n (next) s (step) t (trace mode)`

    `l (list next) v (prev) . (current line)`

    `b (set breakpoint) c (cont after break)`

    `p expr (print) x expr (extended print)`

    `q (quit)`
  - Execute perl statements in the debugger: `perl -de 0`

Graphical debuggers

- `ptkdb` is graphical frontend for `perl -d`
  - based on perl/Tk, runs on many platforms, slow
- `ddd` is a powerful debugger under UNIX
  - has also support for perl (is also a frontend)
- `emacs` has also some debug support
- Integrated development environments
  - Perlbuilder from Solutionsoft used in this course
  - colorful, with many gadgets, some minor bugs (2.0g)
  - also useable for program development

Use the correct algorithms

- An extremely short wrong program
- perl -e '("a"x2000) =~ /((a+?)+)/'
- Dumps core, why?

Unsafe Data

- Perl programs can get external data in many ways
  - user input from **`STDIN`**
  - Arguments passed from the command line
  - Output of external programs that gets processed
- Perl marks such data as tainted and offers checks
- Under Option **`-T`** perl does not transfer control to external processes that get passed tainted data
- SetUID programs will, CGI scripts should always run under taint (**`-T`**) control

Making tainted scripts safe

- Is not a guarantee for safety within the perl code

```
$line = <>;  #$line is tainted
`echo $line`;#insecure, run time error with -T
```

- The tainted status gets cleared by
  - usage of regex substrings (**`$1, $2, ...`**)
  - calling external programs with the full path or setting **`$ENV{PATH}`** to a known value
  - use of **`exec`** and **`system`** not using the shell (see above)
  - change an unsafe pipe into a "secure" exec

```
open IP, "-|" or exec 'echo', $tainted;
```

instead of **`open IP, "$tainted|";`**

Optimization in general

- Rule 1: don't optimize
  - Rule 2: don't optimize
    - Rule 3: (experts only) don't optimize yet
- Is optimization really required
  - If yes, time or space optimization
- Balance between optimization and readability
- Cost of optimization
  - Adding a CPU may be cheaper than manpower

Profiling

- Profile your program

  `perl -d:DProf script` writes profile data **`tmon.out`**

  `dprofpp` generates profiling information

  (many options to influence output)

  can be done in one go

  `dprofpp -p script`

- Do only optimize hotspots
  - Where most of the time is spent
  - Try to avoid excessive number of function calls

Benchmarking

- Benchmark your program

```
use Benchmark;
$count = 100;
$t = timethis($count, "CODE");
```

  Benchmark data contained in object $t, get also printed

- For finer grained resolution use calls to time() instead and

```
use Time::HiRes;
```

  it will replace the time calls (of 1 second resolution)

Optimization tips

- Avoid calling external programs (pipes from/to commands, backticks, the `system` function
  - look on CPAN for perl modules that do the job
  - examples: UNIX commands **`du, df, ls, ps`**
  - write your own modules, use the XS interface for speed (calling C Code from perl)
  - or use the module `Inline.pm` (inline C code in perl, similar to the XS interface, easier to handle) Installed at DESY (UNIX)
    - Majority of modules is using XS, not inline

Inlining C Code with Inline.pm

```
use Inline C => <<'END_C';
void greet () { /* define C function */
  printf("Hello, World\n");
}
END_C
greet; # call C function from perl
```

- C function gets compiled on first invocation or if code has changed. (good for testing, less suitable for stable production code)
- Subsequent script calls use compiled code

Calling Perl from C (Inline::CPR)

```
/usr/local/bin/cpr
int main(void) {
  printf("Hello World, I'm running Perl %s\n",
         CPR_eval("use Config; $Config{version}")
         );
  return 0;
}
```

- Execution of script with **`cpr scriptfile`**
- Gets compiled on first invocation or after changes
- Subsequent script calls use compiled code
- No longer supported at DESY (not used)

Compiling Perl Code

- Compiler included in distribution
- Not as useful as one could think
  - Not yet production quality
  - Mainly parsing step saved (faster startup)
  - Remaining code not much faster
  - No longer platform independent
  - If e.g. generating C resulting code hardly readable

Using Threads

- Starting with 5.8 new thread model "ithreads"
  - Called Interpreter threads
  - Was available internally already in 5.6
- Perl binary and modules containing XS get compiled differently if threads are used
- Threaded perl now standard, runs slightly slower
- In threads no data are shared by default
  - Most pure perl modules therefore thread safe
- threaded code harder to write, harder to debug
  - timing problems, deadlocks, shared access to data

Threads (2)

- Threads covered in a tutorial: `perldoc perlthrtut`
- Startup penalty for threads, therefore
  - Using few long living threads is advantageous
  - Benchmark to see if the threaded program is worth the effort
- Most rules of parallel programming apply
  - Several models to use threads (e.g. master/slave)
  - Synchronization between threads (wait for the others)

Documenting your code

- Inline documentation is the preferred way
  - POD (plain old documentation) format is used
- Documentation is started with an empty line followed by a pod directive, e.g.

  `=pod`

  `=head1`

  followed by more directives and ordinary text

  and ended with a blank line followed by

  `=cut`

  followed by another blank line

A pod example

```
=head1 NAME

sudo - execute a command as another user

=head1 SYNOPSIS

B<sudo> B<-V> | B<-h> | B<-l> | B<-L> | B<-v> | B<-k> | B<-K> |
[ B<-H> ] [B<-P> ] [B<-S> ] [ B<-b> ] | [ B<-p> I<prompt> ]
[ B<-c> I<class>|I<-> ] [ B<-a> I<auth_type> ]
[ B<-u> I<username>|I<#uid> ] I<command>

=head1 DESCRIPTION

B<sudo> allows a permitted user to execute a I<command> as the
superuser or another user, as specified in the I<sudoers> file.
```

Converting pod documentation

- Common commands for displaying and converting pod

```
perldoc sudo.pod                          # ASCII
pod2man sudo.pod | groff -man -Tps # PS
pod2html sudo.pod                         # HTML
```

- For more format conversions and options see the man pages of the above commands and

```
pod2latex
pod2usage
pod2text
```

Suggestions for own Experiments

- Rewrite one of the Example Scripts that it gets taint safe, i.e. it runs under `perl -T`
- Generate a list of all Variables using `dumpvars`
- Check the speed of the `stat` function on different platforms using the `Benchmark` module
- Document your own perl scripts

Check your knowledge of perl

- I prepared 20 multiple choice questions (see attached document)
  - Try to answer them using manuals, perl, this tutorial, ...
  - Some of the choices represent frequent coding flaws
- Answers to these questions can be checked by contacting "kursserver" using the client program `/afs/ifh.de.user/f/friebel/public/kursclient`
  - HTML form also available (see link to these slides)
- Answers can be given as a list as e.g. in `1a,2b,3c`
- There can be several valid answers to a question