1.1.2.8.8

Intermediate Perl – Session 8

- using the warnings system
- benchmarking code
- profiling code
- speeding up code
- style suggestions

Debugging

- what is the best way to debug?
  - the best way is the way which will discover the most bugs, and *all* the bad bugs, in the shortest time

- why are there bugs?
  - syntax errors
    - caught when the script is run
    - `print Dumber($a)` – you meant `Dumper`
  - sematic errors
    - hinted by unexpected behaviour
    - pernicious and sometimes not detected
    - `@a[0] = <FILE>` – you meant `$a[0]`
  - requirements changes
    - data domain expands beyond what was coded for
    - amount of data increases and script runs out of system resources

How to Code to Prevent Bugs

- code conservatively
  - code, run, code, run
- code carefully
  - if you're adventuring into new idioms, make sure you have the syntax right
  - do not pack too many piped operators (map/grep/sort) into a single line
  - keep in mind that toughest errors to spot may not be in complex parts of your code
    - do not take for granted that simple statements are not causing the error
    - `%x = {a=>1,b=>2,c=>3}`
- debug practically
  - spend time debugging in proportion to the need to debug
  - don't test everything
  - implement invariants
    - before a block/function/etc know what must be true for the code to function properly (what you are assuming)

`strict` and `Data::Dumper` and `Carp`

- we've already seen these

- use `strict` pragma to enforce variable declaration
  - `use vars qw($a %b)` allows for global variables `$a` and `%b`
  - consider using `our` (compare to `my`) for global variables in a package (perl 5.6)

- use `Data::Dumper` to visualize your data structures
  - see session 2

- use `Carp` to override warn and die and produce stack traces
  - see session 5

Preparing Your Script – employ `warnings`

- diagnostic mode (`-w`) displays verbose warnings
  - tedious warnings may be produced which you do not care about
  - develop with `-w` and remove it in production
  - locally scoped `^W=1` to toggle
  - `use diagnostics` for even more details

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

{
  local ^W=0; # warnings are off for this block
  ...
}
# warnings are automatically off
```

Preparing Your Script – employ `warnings`

```
#!/home/martink/bin/perl -w

use strict;

my @a;
my $x;
my $x;
print $x;
print @a[0];
warn;

"my" variable $x masks earlier declaration in same scope at myscript line 7.
Scalar value @a[0] better written as $a[0] at myscript line 10.
Use of uninitialized value in print at myscript line 9.
Use of uninitialized value in print at myscript line 10.
Warning: something's wrong at myscript line 12.
```

- there are many types of warning messages
  - see *Programming Perl*, ch 33 Diagnostic Messages
- not all messages are as critical as others

limit and scope your warnings

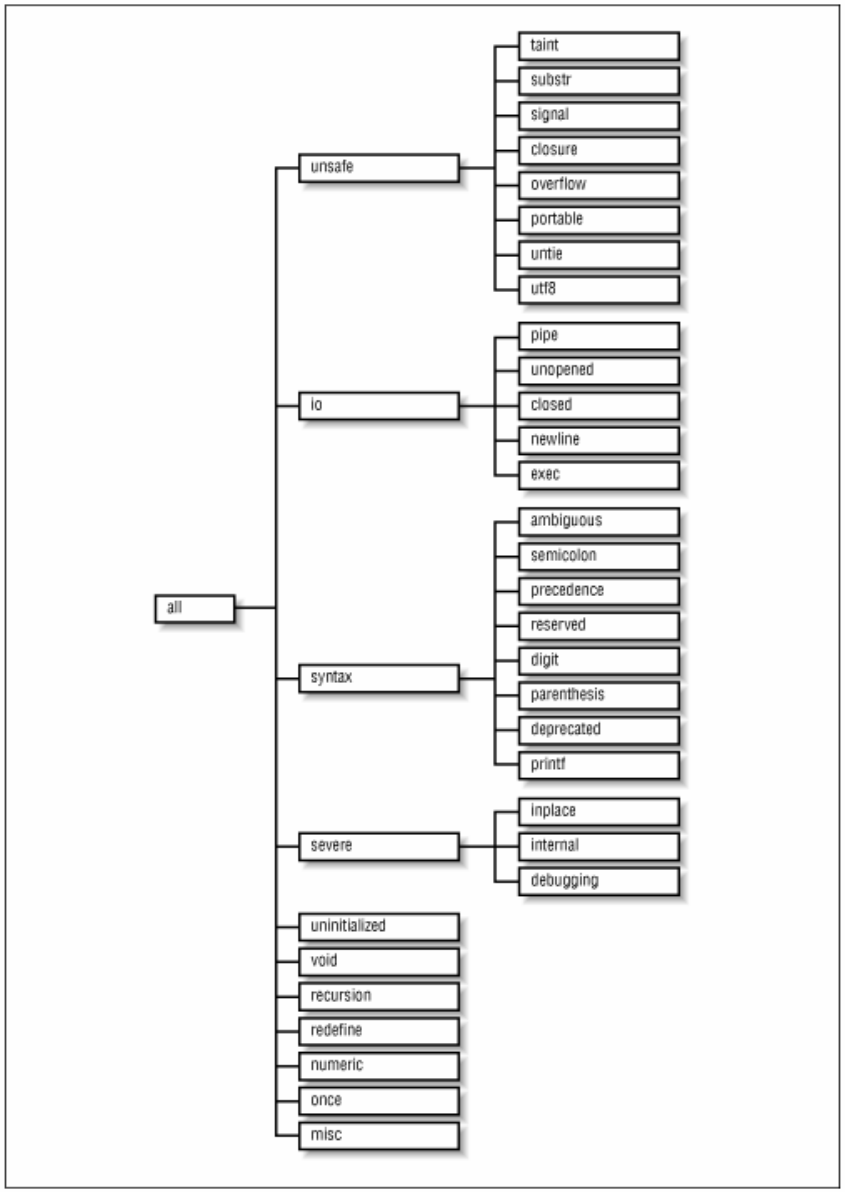
- because warnings can get spammy, it's useful to turn them off in production
- you may be receiving warnings from external modules
- the `warnings` pragma controls
  - where warnings can/cannot be triggered
  - what type of warnings are displayed

```
#!/usr/bin/perl

# all warnings will be used
use warnings;
# only warnings related to IO or syntax will be shown
use warnings qw(io syntax);

...

# now IO warnings will not be shown - calls to pragma are cumulative!
no warnings qw(io);
```



warnings categories
Programming Perl, Figure 31.1

also see
perldoc.perl.org/perllexwarn.html

limit and scope your warnings

- you can escalate a warning to become a fatal error

```
#!/usr/bin/perl

# all warnings will be used
use warnings;
# no uninitialized warings please
no warnings qw(uninitialized);
# syntax warnings are now fatal
use warnings FATAL => qw(syntax);

...

# syntax warnings are gone
no warnings FATAL => qw(syntax)
# or, downgrade to nonfatal
use warnings NONFATAL => qw(syntax);
```

- this is useful if you consider some warnings more important and wish to be forced to deal with them

limit and scope your warnings

- if you are writing modules, you can produce warnings from your modules in a manner that is respectful of the caller
  - the caller can toggle the warnings

```
##########################################
# MyPackage.pm
package MyPackage;
use warnings::register;
sub func {
  warnings::warnif("some warning");
}
```

```
use strict;
use MyPackage;
use warnings;

MyPackage::func();
some warning at ./myscript
line 10
```

```
use strict;
use MyPackage;
use warnings;
no warnings
qw(MyPackage)

MyPackage::func();
# no warning produced
```

```
use strict;
use MyPackage;
use warnings;
use warnings FATAL=>qw(MyPackage)

MyPackage::func();
# now warnings from MyPackage are
fatal
```

Benchmarking using Benchmark

- understand how long it takes to run your code using Benchmark
  - pass the number of iterations and code references
  - Benchmark can also create chart data and count iterations of code in a given time

```
use Benchmark qw(:all);

my $iterations = 1e7;
my $x = 2;
# also try cmpthese()
timethese($iterations,
              { func1=>sub{$x*$x},
                func2=>sub{$x**$x} });

Benchmark: timing 10000000 iterations of func1, func2...
     func1:  0 wallclock secs ( 0.79 usr + -0.04 sys =  0.75 CPU) @ 13333333.33/s (n=10000000)
     func2:  0 wallclock secs ( 1.18 usr +  0.00 sys =  1.18 CPU) @ 8474576.27/s (n=10000000)

# calling cmpthese() instead of timethese() produces a chart
            Rate func2 func1
func2  8695652/s    --  -39%
func1 14285714/s   64%    --
```

Time::HiRes – benchmark yourself

- get current time (hi-resolution) and calculate intervals

```
use Time::HiRes qw(gettimeofday tv_interval);

my $iterations = 1e7;
my $x = 2;
my $t = [gettimeofday];
# multiply a lot with map
map { $x*$x } (1..$iterations);
printf("it took %.2f s to run the code",tv_interval($t));

it took 2.87 s to run the code
```

Time::HiRes – benchmark yourself

- set an alarm

```
use strict;
use Time::HiRes qw(alarm);

my $counter = 0;

# define what happens when ALRM signal is received
$SIG{ALRM} = sub { print $counter,"\n" ; exit};
# set alarm to go off in 0.1 seconds - at which time ALRM signal is sent
alarm(0.1);

# now run code
while(1) {
    $counter++;
}
```

Profile code with Devel::Dprof

- to find out how much time your code is spending in functions, profile it
  - a file tmon.out will be created
  - dprofpp tmon.out displays a profile table

```
#!/usr/bin/perl -d:DProf

use strict;

my $iterations = 1e5;
for my $i (1..$iterations) {
    call_square( call_random() );
    call_root( call_random() );
}

sub call_random {
    return rand();
}
sub call_square {
    return $_[0] ** 2;
}
sub call_root {
    return sqrt($_[0]);
}
```

```
> dprofpp tmon.out
Total Elapsed Time = -4.07070 Seconds
  User+System Time =        0 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 0.00   0.470  0.470 200000   0.0000 0.0000  main::call_random
 0.00   0.220  0.220 100000   0.0000 0.0000  main::call_square
 0.00   0.150  0.150 100000   0.0000 0.0000  main::call_root
 0.00       - -0.000      1        -      -  strict::bits
 0.00       - -0.000      1        -      -  strict::import
 0.00       - -0.000      1        -      -  main::BEGIN

# large number of calls to tiny functions may suffer from
# round-off errors in report and yield strange values
#
# your code should run for enough time to reduce effect
# of system housekeeping and code overhead (e.g. importing
# modules)
```

speeding up code with Memoize

- Memoize is an auto-magical caching module
  - it registers your functions and sets up a lookup table
  - when your function is called, it stores the output for a given set of arguments
  - if same arguments are seen again, function is bypassed and lookup value is returned

```
use strict;
use Memoize;

memoize("complex_function");

# function is called
complex_function(1)
# memoized cache is used
complex_function(1);

sub complex_function {
  my $arg = shift;
}
```

www.slashcode.com/docs/slashstyle.html

Don't use single-character variables, except as iterator variables.

Don't use two-character variables just to spite us over the above rule.

Constants are in all caps; these are variables whose value will never change during the course of the program.

```
$Minimum = 10;          # wrong
$MAXIMUM = 50;          # right
```

Other variables are lowercase, with underscores separating the words. They words used should, in general, form a noun (usually singular), unless the variable is a flag used to denote some action that should be taken, in which case they should be verbs (or gerunds, as appropriate) describing that action.

```
$thisVar      = 'foo';  # wrong
$this_var     = 'foo';  # right
$work_hard    = 1;      # right, verb, boolean flag
$running_fast = 0;      # right, gerund, boolean flag
```

Arrays and hashes should be plural nouns, whether as regular arrays and hashes or array and hash references. Do not name references with ref or the data type in the name.

```
@stories     = (1, 2, 3);      # right
$comment_ref = [4, 5, 6];      # wrong
$comments    = [4, 5, 6];      # right
$comment     = $comments->[0]; # right
```

Make the name descriptive. Don't use variables like $sc when you could call it $story_count.

other style resources

- man perlstyle
- www.extremeperl.org/bk/coding-style
- http://perltidy.sourceforge.net/tutorial.html
- *Perl Best Practices* by Damian Conway (O'Reilly)

1.1.2.8.8

Introduction to Perl – Session 8

- use the warning system
- benchmark with Benchmark or Time::HiRes
- profile with -d:DProf and speed up with Memoize
- keep style clean and consistent