

Mastering Perl

by brian d foy

Stonehenge Consulting Services, Inc.

version 1.0.5

March 29, 2008

http://www.pair.com/~comdog/Talks/mastering_perl_talk.pdf

Contents

Introduction

About this course	7
The path to mastery	8

Profiling

Profiling is better than benchmarking	10
A recursive subroutine	11
Calling a Profiler	12
Recursion profile	13
Iterate, not recurse	14
Iteration profile	15
Really big numbers	16
Memoize	17
What happened	18
More complex profiling	19
The basics	20
Record DBI queries	21
Database optimization	22
Profiling DBI Statements	23
Profiling DBI methods	24
Profiling test suites	25
Devel::Cover HTML report	26
Devel::Cover detail	27
Further reading	28

Benchmarking

Measuring Perl	30
Theory of measurement	31
Know where you are	32
Using benchmarks	33
Single points	34
Multiple points	35
All things being equal	36
Don't benchmark languages	37
Definitions of performance	38
Possible metrics	39
Devel::Peek	40
Memory use	41
About Benchmark.pm	42
Time a single bit of code	43
Compare several bits of code	44
Common misuse	45
Do these numbers make sense?	46
Report the situation	47
Do something useful	48
Now the results make sense	49
Verify with an experiment	50
Benchmarking summary	51
Further reading	52

Configuration

Configuration goals	54
---------------------	----

Configuration techniques	55	The :easy way	82
The wrong way	56	Logging levels	83
Slightly better (still bad)	57	Something more complex	84
Environment variables	58	Configuring Log4perl	85
Set defaults	59	Appenders handle the magic	86
Perl's Config	60	Logging to a database	87
Command-line switches	61	Changing configuration on-the-fly	88
perl's -s switch	62	Send to screen and file at once	89
Getopt::Std and getopt	63	Multiple loggers	90
Getopt::Std and getopts	64	Further reading	91
Getopt::Long	65		
More GetOpt::Long	66	Lightweight Persistence	
Extreme and odd cases	67	Persistence	93
Configuration files	68	Perl structures as text	94
ConfigReader::Simple	69	Using my own name	95
INI Files	70	Nicer output	96
Config::IniFiles	71	Reading Data::Dumper text	97
Config::Scoped	72	YAML Ain't Markup	98
AppConfig	73	YAML format	99
Using the program name	74	Reading in YAML	100
By operating system	75	Storable	101
Writing your own interface	76	Reading Storable files	102
Good method names	77	Freezing and thawing	103
Further reading	78	Storing multiple values	104
		Deep copies	105
		dbm files (old, trusty)	106
		A better DBM	107
		Further reading	108
Logging			
Log without changing the program	80		
Two major modules	81		

Conclusion

Main points	110
More information	111

Questions

Mastering Perl

by brian d foy

Stonehenge Consulting Services, Inc.

version 1.0.5

March 29, 2008

http://www.pair.com/~comdog/Talks/mastering_perl_talk.pdf

Introduction

About this course

- Selected topics for the working programmer based on *Mastering Perl*
- Mostly not about syntax or wizardly tricks
- Not for masters, but people who want to control Perl code
- Not necessarily the way to do it, just the way I've done it
- Create “professional”, robust programs other people can use
- We'll cover
 - * profiling
 - * benchmarking
 - * logging
 - * lightweight persistence

The path to mastery

- The guild system had a progression of skills
- Apprentices were the beginners and worked with supervision
- Journeymen were competent in their trade
- Masters taught journeymen
- Journeymen studied under different masters
 - * different masters teach different tricks and methods
 - * journeyman develop their own style
- A masterpiece showed that a journeyman mastered his trade

Profiling

Profiling is better than benchmarking

- Benchmarking is often pre-mature
- Profiling shows you the performance of your program
 - * speed
 - * memory
 - * whatever
- See what's taking up your resources
- Focus your efforts in the right places

A recursive subroutine

- A recursive subroutine runs itself many, many times

```
sub factorial
{
    return unless int( $_[0] ) == $_[0];
    return 1 if $_[0] == 1;
    return $_[0] * factorial( $_[0] - 1 );
}

print factorial($ARGV[0]), "\n";
```

Calling a Profiler

- Invoke a custom debugger with `-d`

```
perl -d:MyDebugger program.pl
```

- `MyDebugger` needs to be in the `Devel::*` namespace
- Uses special DB hooks for each statement
- Find several on CPAN
 - * `Devel::DProf`
 - * `Devel::SmallProf`
 - * `Devel::LineProfiler`

Recursion profile

- Runs several statements for each call

```
% perl -d:SmallProf factorial.pl 170
```

- Creates a file named *smallprof.out*

```
===== SmallProf version 1.15 =====
                        Profile of factorial.pl                        Page 1
=====
count wall tm   cpu time line
   0 0.000000 0.000000  1:#!/usr/bin/perl
   0 0.000000 0.000000  2:
 170 0.000000 0.000000  3:sub factorial {
 170 0.001451 0.000000  4: return unless int($_[0]) == $_[0];
 170 0.004367 0.000000  5: return 1 if $_[0] == 1;
 169 0.004371 0.000000  6: return $_[0] * factorial($_[0]-1);
   0 0.000000 0.000000  7: }
```

Iteration, not recursion

- Perl 5 doesn't optimize for tail recursion
- No need to run more statements than I need
- Use better algorithms
- No need to create more levels in the call stack

```
sub factorial {  
    return unless int( $_[0] ) == $_[0];  
  
    my $product = 1;  
  
    foreach ( 1 .. $_[0] ) { $product *= $_ }  
  
    $product;  
}  
  
print factorial( $ARGV[0] ), "\n";
```

Iteration profile

- Now I don't call needless statements

```
===== SmallProf version 2.02 =====  
Profile of factorial-iterate.pl Page 1  
=====
```

count	wall tm	cpu time	line
0	0.000000	0.000000	1:#!/usr/bin/perl
0	0.000000	0.000000	2:
0	0.000000	0.000000	3:sub factorial {
1	0.000001	0.000000	4: return unless int(\$_[0]) == \$_[0];
1	0.000000	0.000000	5: my \$f = 1;
170	0.000011	0.000000	6: foreach (2 .. \$_[0]) {\$f *= \$_};
1	0.000009	0.000000	7: \$f;
0	0.000000	0.000000	8: }

Really big numbers

- Now I want have a program that takes a long time
- My perl tops out at 170!, then returns `inf`
- The `bignum` package comes with Perl 5.8

```
use bignum;                                     get really large numbers

sub factorial {
    return unless int( $_[0] ) == $_[0];

    my $product = 1;

    foreach ( 1 .. $_[0] ) { $product *= $_ }

    $product;
}

print factorial( $ARGV[0] ), "\n";
```

- This still isn't that interesting because it's one shot

Memoize

- Remember previous results for future speed-ups

```
my @Memo      = (1);

sub factorial {
    my $number = shift;

    return unless int( $number ) == $number;
    return $Memo[$number] if $Memo[$number];

    foreach ( @Memo .. $number ) {
        $Memo[$_] = $Memo[$_ - 1] * $_;
    }

    $Memo[ $number ];
}

while(1) {
    print 'Enter a number> ';
    chomp( my $number = <STDIN> );
    exit unless defined $number;
    print factorial( $number ), "\n";
}
```

What happened

- One shot is not so bad
- Redoes a lot of work if called many times
- Memoizing is faster each time, but takes more memory

More complex profiling

- If `Devel::SmallProf` is too basic, try `Devel::DProf`

```
% perl -d:DProf journals
```

- Use `dprofpp` to make the report

```
$ dprofpp
Total Elapsed Time = 53.08383 Seconds
User+System Time = 0.943839 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
8.37 0.079 0.000 84 0.0009 0.0000 utf8::SWASHNEW
6.25 0.059 0.146 5 0.0118 0.0292 main::BEGIN
5.83 0.055 0.073 24 0.0023 0.0030 Text::Reform::form
5.09 0.048 0.067 2 0.0242 0.0334 HTTP::Cookies::BEGIN
4.24 0.040 0.040 10 0.0040 0.0040 LWP::UserAgent::BEGIN
4.24 0.040 0.049 9 0.0044 0.0054 Text::Autoformat::BEGIN
```

- In this example, most of the time is in the compilation

The basics

- Profiling counts something
- All the code runs through a central point, a recorder
- While recording, the program is slower
- At the end I get a report
- Use the report to make a decision

Record DBI queries

- Create a routine through which all queries flow
- Record the queries to gather the data

```
package My::Database;
```

```
my %Queries;
```

```
sub simple_query
```

```
{  
    my( $self, @args ) = @_;
```

```
    my $sql_statement = shift @args;
```

```
    $Queries{$sql_statement}++;
```

Profiling hook

```
    my $sth = $self->dbh->prepare( $sql_statement );  
    unless( ref $sth ) { warn $@; return }
```

```
    my $rc    = $sth->execute( @args );
```

```
    wantarray ? ( $sth, $rc ) : $rc;  
}
```

Database optimization

- Often, the database bits are the slowest part of my program
- Most of the work is not in my program because it's in the database server
- My program waits for the database response
- I usually talk to the database more than I need to
 - * Repeated `SELECT`s for the same, unchanging data
- My queries are too slow
 - * Optimize the slowest, most frequent ones

Profiling DBI Statements

- Profiling is built into DBI
- Uses the DBI_PROFILE environment variable
- Using !Statement orders by the query text
- `$ env DBI_PROFILE='!Statement' perl dbi-profile.pl`

```
DBI::Profile: 109.671362s 99.70% (1986 calls) dbi-profile.pl @  
2006-10-10 02:18:40
```

```
'CREATE TABLE names ( id INTEGER, name CHAR(64) )' => 0.004258s  
'DROP TABLE names' => 0.008017s  
'INSERT INTO names VALUES ( ?, ? )' =>  
3.229462s / 1002 = 0.003223s avg (first 0.001767s, min  
0.000037s, max 0.108636s)  
'SELECT name FROM names WHERE id = 1' =>  
1.204614s / 18 = 0.066923s avg (first 0.012831s, min  
0.010301s, max 0.274951s)  
'SELECT name FROM names WHERE id = 10' =>  
1.118565s / 9 = 0.124285s avg (first 0.027711s, min 0.027711s,  
max 0.341782s)
```

Profiling DBI methods

- Can also order by the DBI method name
- Set DBI_PROFILE to !MethodName

```
$ env DBI_PROFILE='!MethodName' perl dbi-profile2.pl
```

```
DBI::Profile: 2.168271s 72.28% (1015 calls) dbi-  
profile2.pl @ 2006-10-10 02:37:16
```

```
'DESTROY' =>
```

```
0.000141s / 2 = 0.000070s avg (first 0.000040s, min  
0.000040s, max 0.000101s)
```

```
'FETCH' => 0.000001s
```

```
'STORE' =>
```

```
0.000067s / 5 = 0.000013s avg (first 0.000022s, min  
0.000006s, max 0.000022s)
```

```
'do' =>
```

```
0.010498s / 2 = 0.005249s avg (first 0.006602s, min  
0.003896s, max 0.006602s)
```

```
'execute' =>
```

```
2.155318s / 1000 = 0.002155s avg (first 0.002481s, min  
0.001777s, max 0.007023s)
```

```
'prepare' => 0.001570s
```


Profiling test suites

- I can profile my test suite to see how much code it tests
- I want to test all code, but then there is reality
- Where should I spend my testing time to get maximum benefit?
- The `Devel::Cover` module does this for me

```
% cover -delete clear previous report
```

```
% HARNESS_PERL_SWITCHES=-MDevel::Cover make test
```

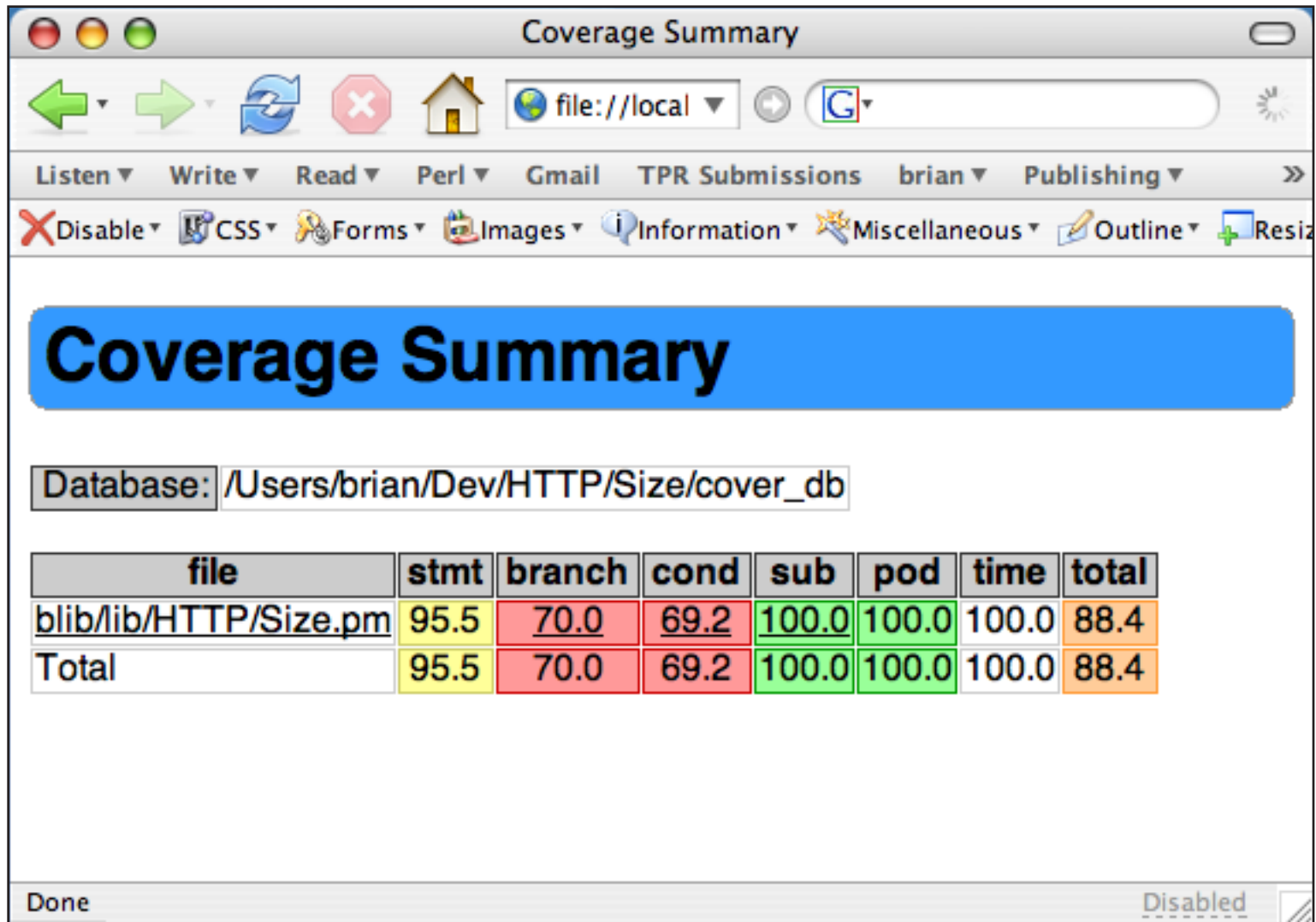
```
% ./Build testcover for Module::Build
```

```
% cover generates report from data
```

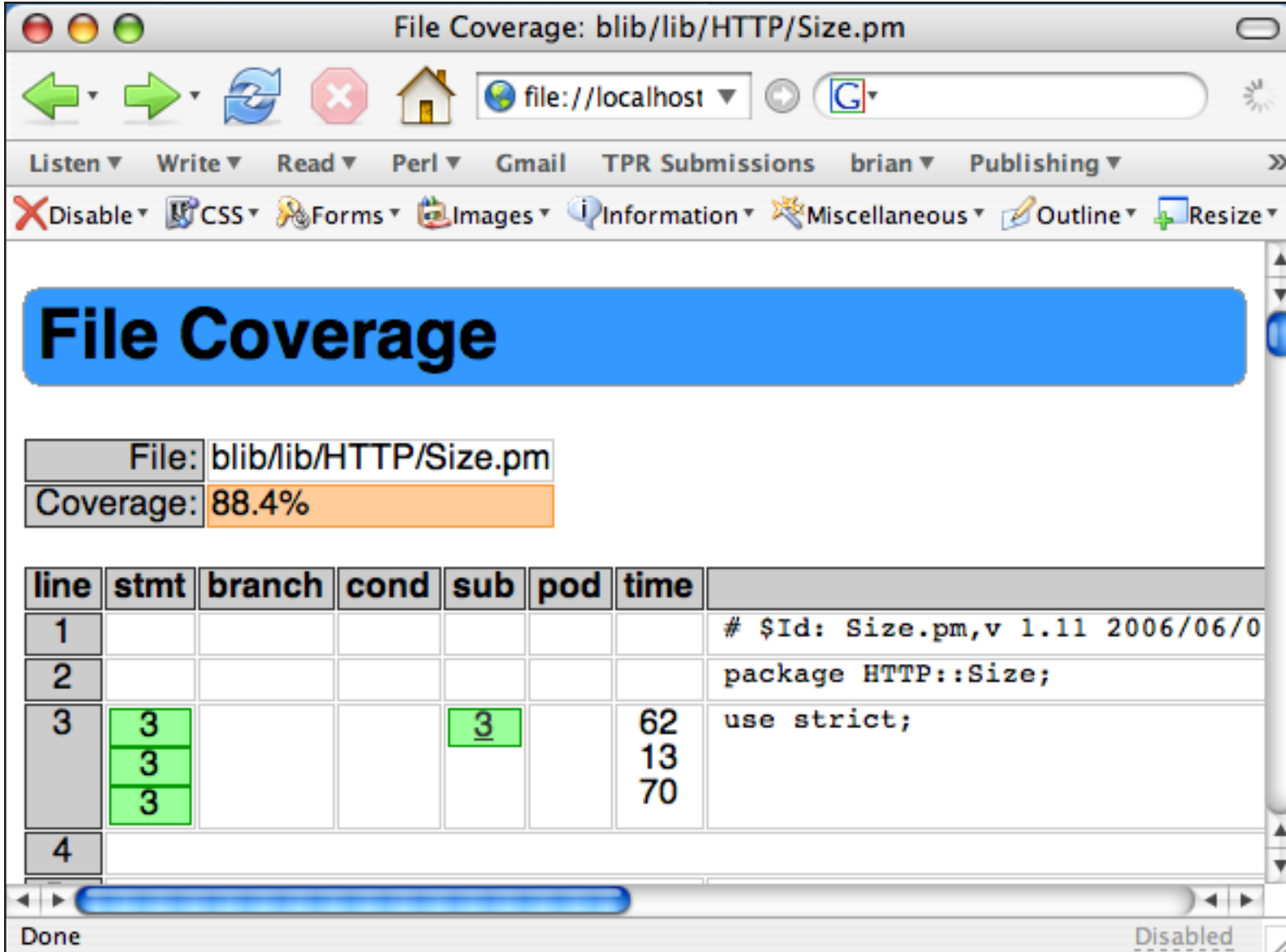
```
Reading database from Dev/HTTP/Size/cover_db
```

- Sends text report to standard output
- Also creates an HTML report

Devel::Cover HTML report



Devel::Cover detail



The screenshot shows a web browser window titled "File Coverage: blib/lib/HTTP/Size.pm". The browser's address bar shows "file:///localhost". Below the browser window, there is a blue header bar with the text "File Coverage". Below this, there is a table showing file and coverage information:

File:	blib/lib/HTTP/Size.pm
Coverage:	88.4%

Below the table, there is a table with columns: line, stmt, branch, cond, sub, pod, time. The data is as follows:

line	stmt	branch	cond	sub	pod	time	
1							# \$Id: Size.pm,v 1.11 2006/06/0
2							package HTTP::Size;
3	3			3		62	use strict;
	3					13	
	3					70	
4							

The browser window also shows a menu bar with items: Listen, Write, Read, Perl, Gmail, TPR Submissions, brian, Publishing. Below the menu bar, there are icons for Disable, CSS, Forms, Images, Information, Miscellaneous, Outline, and Resize. The browser window also shows a status bar with "Done" and "Disabled".

Further reading

- The *perldebugs* documentation explains custom debuggers
- “Creating a Perl Debugger” (<http://www.ddj.com/184404522>) and “Profiling in Perl” (<http://www.ddj.com/184404580>) by brian d foy
- “The Perl Profiler”, Chapter 20 of *Programming Perl, Third Edition*
- “Profiling Perl” (<http://www.perl.com/lpt/a/850>) by Simon Cozens
- “Debugging and Profiling mod_perl Applications” (http://www.perl.com/pub/a/2006/02/09/debug_mod_perl.html) by Frank Wiles
- “Speeding up Your Perl Programs” (<http://www.stonehenge.com/merlyn/UnixReview/col49.html>) and “Profiling in Template Toolkit via Overriding” (<http://www.stonehenge.com/merlyn/LinuxMag/col75.html>) by Randal Schwartz

Benchmarking

Measuring Perl

- Perl is just a programming language
- Measure Perl programs the same as other things
- Measure Perl programs against themselves
- Compare the results
- “Premature optimization is the root of all evil” — Tony Hoare

Theory of measurement

- Observation changes the universe
- Nothing is objective
- Tools have inherent uncertainties
- Precision is repeatability, not accuracy
- Accuracy is getting the right answer
- You want both precision and accuracy

Know where you are

“A benchmark is a point of reference for a measurement. The term originates from the chiseled horizontal marks that surveyors made into which an angle-iron could be placed to bracket (bench) a leveling rod, thus ensuring that the leveling rod can be repositioned in the exact same place in the future.”

<http://en.wikipedia.org/wiki/Benchmark>

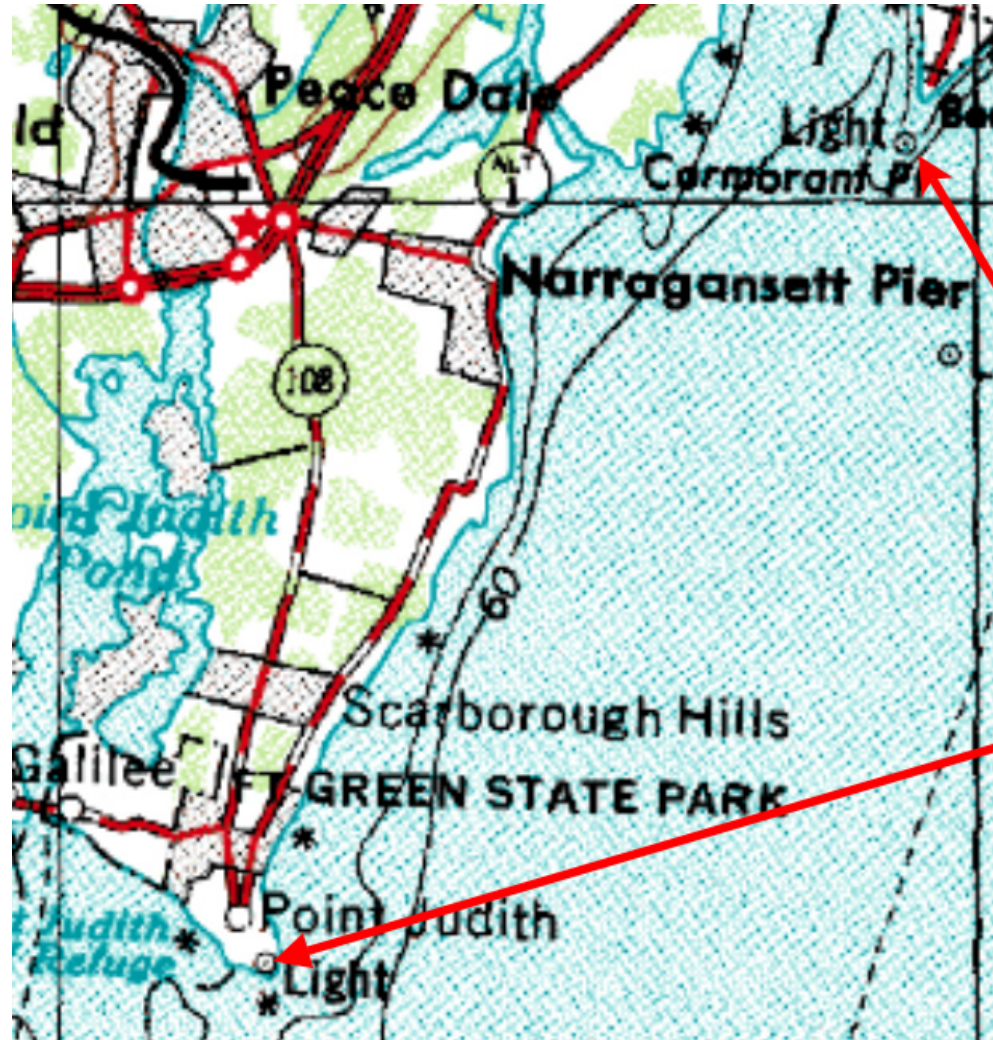
Using benchmarks

- Find the bad parts
- Profile the application first
- Find out who's taking all the...
 - * time
 - * memory
 - * network
- Compare situations
- Fix the worst situations first

Single points



Multiple points



All things being equal

- There are lies, damned lies, and benchmarks
- Everyone has an agenda
- You don't run testbeds as production
- Skepticism wins the day

Don't benchmark languages

“How can we benchmark a programming language? We can't—we benchmark programming language implementations. How can we benchmark language implementations? We can't—we measure particular programs.”

<http://shootout.alioth.debian.org/>

Definitions of performance

- A major factor in determining the overall productivity of a system, performance is primarily tied to availability, throughput and response time (<http://www.comptia.org/sections/ssg/glossary.aspx>).
- A performance comprises an event in which generally one group of people behave in a particular way for another group of people (<http://en.wikipedia.org/wiki/Performance>)
- Your investment's activity over time. Past performance does not guarantee future results (my accountant)

Possible metrics

- Speed isn't the only metric
- Speed might not even be the most important one
 - * power, speed, use of use—pick any two
 - * disk use, concurrent users, CPU time, completion time, memory use, uptime, bandwidth use, network lag, responsiveness, binary size
- What about programmer time?

Devel::Peek

- `Devel::Peek` lets you look at the perl data structure

```
use Devel::Peek;
```

```
my $a = '';  
Dump( $a );  
$a = "Hello World!\n";  
Dump( $a );
```

- See all of the gory bits. An empty scalar still takes up space

```
SV = PV(0x801060) at 0x800c24  
  REFCNT = 1  
  FLAGS = (PADBUSY,PADMY,POK,pPOK)  
  PV = 0x207740 ""\0  
  CUR = 0  
  LEN = 4  
SV = PV(0x801060) at 0x800c24  
  REFCNT = 1  
  FLAGS = (PADBUSY,PADMY,POK,pPOK)  
  PV = 0x207740 "Hello World!\n"\0  
  CUR = 13  
  LEN = 16
```


Memory use

- `Devel::Size` can measure the byte size of a data structure

```
use Devel::Size qw(size total_size);
```

```
my $size = size( "A string" );
```

size of scalar

```
my @foo = ( 1, 2, 3, 4, 5 );
```

```
my $other_size = size( \@foo );
```

just array size, not elements

```
my $foo = {  
    a => [ 1, 2, 3 ],  
    b => { a => [1, 3, 4] }  
};
```

```
my $total_size = total_size( $foo );
```

array and element sizes

- `Size` is more than just the data, it's the perl SV, et cetera

```
print size( my $a );
```

12 bytes on perl 5.8.8

About Benchmark.pm

- Benchmark with Perl
- Often used incorrectly and without thought
- Only measures speed
- Uses a null loop as a control
 - * `sub { }`
 - * It's just a timer
 - * Subtracts the null loop time
 - * Introduces an error of about 7%
- Only measures time on the local CPU

Time a single bit of code

- Time a single bit of code with `timethis`

```
timethis( $count, 'code string' );  
timethis( $count, sub { ... } );
```

- Time several bits of code with `timethese`

```
timethese( $count, {  
    'Name1' => sub { ...code1... },  
    'Name2' => sub { ...code2... },  
});
```

- If positive, `$count` is a number of iterations
- If negative, `$count` is the minimum number of CPU seconds

Compare several bits of code

- Compare several bits of code with `cmpthese`
- Runs `timethese` then prints a comparison report
- Be careful what you compare
 - * they should do the same thing
 - * compare all as code strings, or all as code refs

Common misuse

- Taken from http://www.perlmonks.org/index.pl?node_id=536503

```
use Benchmark 'cmpthese';
my @long = ('a' .. 'z', '');
my $iter = shift || -1;
cmpthese(
    $iter, {
        long_block_ne => q{grep {$_ ne ''} @long},
        long_block_len => q{grep {length} @long},
        long_bare_ne   => q{grep $_ ne '', @long},
        long_bare_len  => q{grep length, @long},
    }
);
```

- Do these numbers make sense?

	Rate	bare_ne	block_len	block_ne	bare_len
long_bare_ne	3635361/s	--	-6%	-6%	-8%
long_block_len	3869054/s	6%	--	-0%	-2%
long_block_ne	3872708/s	7%	0%	--	-2%
long_bare_len	3963159/s	9%	2%	2%	--

Do these numbers make sense?

- Don't get excited about the percentages

	Rate	bare_len	bare_ne	block_ne	block_len
long_bare_len	2805822/s	--	-0%	-1%	-3%
long_bare_ne	2805822/s	0%	--	-1%	-3%
long_block_ne	2840569/s	1%	1%	--	-2%
long_block_len	2885232/s	3%	3%	2%	--

- Also need to report the platform
 - * Mac OS X.4.5
 - * 15" G4 Powerbook
 - * perl5.8.4

Report the situation

```
This is perl, v5.8.4 built for darwin-2level
Summary of my perl5 (revision 5 version 8 subversion 4)
configuration:
Platform:
  osname=darwin, osvers=7.3.1, archname=darwin-2level
  uname='darwin albook.local 7.3.1 darwin kernel version 7.3.1:
mon mar 22
21:48:41 pst 2004; root:xnuxnu-517.4.12.obj~2release_ppc power
macintosh powerpc `
  config_args=''
  hint=recommended, useposix=true, d_sigaction=define
  usethreads=undef use5005threads=undef useithreads=undef
usemultiplicity=undef
  useperlio=define d_sfio=undef uselargefiles=define
usesocks=undef
  use64bitint=undef use64bitall=undef uselongdouble=undef
  usemymalloc=n, bincompat5005=undef
Compiler:
  cc='cc', ccflags ='-pipe -fno-common -DPERL_DARWIN -no-cpp-
precomp -fno-strict-aliasing',
  ...
```

Do something useful

- Assign to an array so Perl does something

```
use Benchmark 'cmpthese';
```

```
our @long = ('a' .. 'z', '');
```

```
my $iter = shift || -1;
```

```
cmpthese(  
    $iter, {  
        long_block_ne    => q{my @a = grep {$_ ne ''} @long},  
        long_block_len  => q{my @a = grep {length} @long},  
        long_bare_ne     => q{my @a = grep $_ ne '', @long},  
        long_bare_len    => q{my @a = grep length, @long},  
    }  
);
```


Now the results make sense

- Thousands per second is much more believable

	Rate	block_ne	block_len	bare_ne	bare_len
long_block_ne	31210/s	--	-3%	-3%	-5%
long_block_len	32119/s	3%	--	-0%	-2%
long_bare_ne	32237/s	3%	0%	--	-2%
long_bare_len	32755/s	5%	2%	2%	--

Verify with an experiment

- It should take longer to do more

```
use Benchmark 'cmpthese';
our @long = ('a' .. 'z', 0 .. 10_000, '');
my $iter = shift || -1;
cmpthese(
    $iter, {
        long_block_ne => q{my @a = grep {$_ ne ''} @long},
        long_block_len => q{my @a = grep {length} @long},
        long_bare_ne   => q{my @a = grep $_ ne '', @long},
        long_bare_len  => q{my @a = grep length, @long},
    }
);
```

- Output shows that it takes longer to do more

	Rate	bare_ne	block_ne	block_len	bare_len
long_bare_ne	59.8/s	--	-1%	-2%	-3%
long_block_ne	60.4/s	1%	--	-1%	-3%
long_block_len	60.9/s	2%	1%	--	-2%
long_bare_len	61.9/s	4%	3%	2%	--

Benchmarking summary

- Decide what is important to you
- Realize you have bias
- Report the situation
- Don't turn off your brain
- Make predictions that you can verify
- Find better algorithms, not different syntax

Further reading

- “Benchmarking”, The Perl Journal #11, http://www.pair.com/~comdog/Articles/benchmark.1_4.txt
- “Wasting Time Thinking About Wasted Time”, http://www.perlmonks.org/?node_id=393128
- “Profiling in Perl”, <http://www.ddj.com/documents/s=1498/ddj0104pl/>
- “Benchmarking Perl”, a presentation by brian d foy (Perlcast: <http://perlcast.com/2007/04/08/brian-d-foy-on-benchmarking/>, slides: http://www.slideshare.net/brian_d_foy/benchmarking-perl/)

Configuration

Configuration goals

- Don't make people bother you
- Change behavior without editing code
- Same program can work for different people
- Configurable programs are flexible programs
- The wrong way is any way that creates more work
- Too much configuration may be a design smell

Configuration techniques

- Change the code every time (wrong, but common)
- Read Perl's own configuration
- Set environment variables
- Use command-line switches
 - * the `-s` switch
 - * fancy modules
- Use a configuration file
- Combine them

The wrong way

- The easiest thing is to put configuration in the code

```
#!/usr/bin/perl  
use strict;  
use warnings;
```

```
my $Debug    = 0;  
my $Verbose  = 1;  
my $Email    = 'alice@example.com';  
my $DB       = 'DBI:mysql';
```

```
#### DON'T EDIT BEYOND THIS LINE !!! ####
```

- Editing the configuration may break the program

Slightly better (still bad)

- Put the configuration in a separate file

```
# config.pl
use vars qw( $Debug $Verbose $Email $DB );

$Debug    = 0;
$Verbose  = 1;
$Email    = 'alice@example.com';
$DB       = 'DBI:mysql';
```

- Then, in my program, I require the file

```
#!/usr/bin/perl
use strict;
use warnings;

BEGIN { require "config.pl"; }
```

- A syntax errors still kills the program
- People still need to know Perl

Environment variables

- Environment variables are easy to set

```
% export DEBUG=1
```

```
% DEBUG=1 perl program.pl
```

- Look in %ENV for the values

```
use warnings;
```

```
my $Debug    = $ENV{DEBUG};  
my $Verbose  = $ENV{VERBOSE};
```

```
...
```

```
print "Starting processing\n" if $Verbose;
```

```
...
```

```
warn "Stopping program unexpectedly" if $Debug;
```

- Fine for command-line lovers

Set defaults

- No “use of uninitialized value” warnings
- Checking truth won’t work. What is VERBOSE should be off?

```
my $Debug    = $ENV{DEBUG}    || 0;  
my $Verbose  = $ENV{VERBOSE}  || 1;
```

- Check for defined-ness. Before Perl 5.10:

```
my $Debug    = defined $ENV{DEBUG}    ? $ENV{DEBUG}    : 0;  
my $Verbose  = defined $ENV{VERBOSE}  ? $ENV{VERBOSE}  : 1;
```

- Use the defined-or operator in Perl 5.10

```
my $Verbose  = $ENV{VERBOSE} // 1;
```

- Set defaults first, then override with the environment

```
my %config;  
my %defaults = ( ... );  
@config{ keys %defaults } = values %defaults;  
@config{ keys %ENV       } = values %ENV;
```

Perl's Config

- Perl has its own configuration
- Mostly information discovered by `Configure`
- It's in the `Config` module
- Automatically imports a tied hash, `%Config`

```
use Config;

if ($Config{usethreads}) {
    print "has thread support\n"
}
else {
    die "You need threads for this program!\n";
}
```

Command-line switches

- Everyone seems to want their own command-line syntax

% `foo -i -t -r` *single char, unbundled, no values*

% `foo -i -t -d/usr/local` *single char, unbundled, values*

% `foo -i -t -d=/usr/local`

% `foo -i -t -d /usr/local`

% `foo -itr` *single char, bundled*

% `foo -debug -verbose=1` *multiple char, single dash, with values*

- Some people try to mix them

% `foo --debug=1 -i -t` *double dash multiple char, single dash single char*

% `foo --debug=1 -it`

perl's -s switch

- Perl has built-in command-line switch parsing
 - * single dash, multiple character
 - * no bundling
 - * boolean or values
- Use it on the shebang line

```
#!/usr/bin/perl -sw  
use strict;
```

```
use vars qw( $a $abc );
```

must be package vars

```
print "The value of the -a switch is [$a]\n";  
print "The value of the -abc switch is [$abc]\n";
```

- Use it on the command line

```
% perl -s ./perl-s-abc.pl -abc=fred -a  
The value of the -a switch is [1]  
The value of the -abc switch is [fred]
```

Getopt::Std and getopt

- `Getopt::Std` with Perl and handles most simple cases
 - * single character, single dash
 - * bundled

- Call `getopt` with a hash reference

```
use Getopt::Std;
```

```
getopt('dog', \ my %opts );
```

declare and take ref in one step

```
print <<"HERE";
```

```
The value of
```

```
  d $opts{d}
```

```
  o $opts{o}
```

```
  g $opts{g}
```

```
HERE
```

- Must call with values, or nothing set

```
% perl options.pl -d 1
```

```
% perl options.pl -d
```

sets \$opts{d} to 1
WRONG! *nothing set*

Getopt::Std and getopt

- getopt allows boolean and values
- Call getopt as before
- A colon (:) means it takes a value, otherwise boolean

```
use Getopt::Std;
```

```
getopt('dog:', \ my %opts );
```

g: takes a value

```
print <<"HERE";
```

```
The value of
```

```
  d $opts{d}
```

```
  o $opts{o}
```

```
  g $opts{g}
```

```
HERE
```

- Mix boolean and value switches

```
% perl options.pl -d -g Fido
```

sets \$opts{d} to 1, \$opts{g} to Fido

```
% perl options.pl -d
```

sets \$opts{d} to 1

Getopt::Long

- Getopt::Long with Perl
 - * single character switches, with bundling, using a single dash
 - * multiple character switches, using a double dash
 - * aliasing
- Call GetOptions and bind to individual variables

```
use Getopt::Long;
```

```
my $result = GetOptions(  
    'debug|d'    => \ my $debug,  
    'verbose|v' => \ my $verbose,  
    );
```

--debug and -d the same thing

```
print <<"HERE";  
The value of  
    debug      $debug  
    verbose    $verbose  
HERE
```

More GetOpt::Long

- Can validate some simple data types

```
use Getopt::Long;
```

```
my $config = "config.ini";  
my $number = 24;  
my $debug  = 0;
```

```
$result = GetOptions (  
    "number=i" => \ $number,  
    "config=s" => \ $config,  
    "debug"    => \ $debug,  
);
```

*numeric type
string value
boolean*

- Can also handle switches used more than once

```
GetOptions( "lib=s" => \@libfiles );
```

```
% perl options.pl --lib jpeg --lib png
```

- Can take hash arguments

```
GetOptions( "define=s" => \%defines );  
% perl options.pl --define one=1 --define two=2
```

Extreme and odd cases

- There are about 90 option processing modules on CPAN
- There's probably one that meets your needs
- Choosing something odd confuses users
- Too much configuration might mean no one can use it

Configuration files

- Store configuration so normal people can edit it
- Changes don't affect the code
- The program can spot configuration errors
- If there is a format, there is probably a module for it

ConfigReader::Simple

- Handles line-oriented configuration
- Flexible syntax, including continuation lines

```
# configreader-simple.txt
file=foo.dat
line=453
field value
field2 = value2
long_continued_field This is a long \
    line spanning two lines
```

- Access through an object

```
use ConfigReader::Simple;

my $config = ConfigReader::Simple->new( "config.txt" );
die "Could not read config! $ConfigReader::Simple::ERROR\n"
    unless ref $config;

print "The line number is ", $config->get( "line" ), "\n";
```

INI Files

- Handles the Windows-style files
- Has sections and field names

```
[Debugging]
;ComplainNeedlessly=1
ShowPodErrors=1
```

```
[Network]
email=brian.d.foy@gmail.com
```

```
[Book]
title=Mastering Perl
publisher=O'Reilly Media
author=brian d foy
```

Config::IniFiles

- Access by section and field name

```
use Config::IniFiles;

my $file = "mastering_perl.ini";

my $ini = Config::IniFiles->new(
    -file => $file
) or die "Could not open $file!";

my $email  = $ini->val( 'Network', 'email' );
my $author = $ini->val( 'Book', 'author' );

print "Kindly send complaints to $author ($email)\n";
```

Config::Scoped

- Scoped configuration, as Perl code

```
book {  
    author = {  
        name="brian d foy";  
        email="brian.d.foy@gmail.com";  
    };  
    title="Mastering Perl";  
    publisher="O'Reilly Media";  
}
```

- Looks almost like Perl
- Get it as a Perl hash

```
use Config::Scoped;  
  
my $config = Config::Scoped->new(  
    file => 'config-scoped.txt' )->parse;  
die "Could not read config!\n" unless ref $config;  
  
print "The author is ",  
    $config->{book}{author}{name}, "\n";
```


AppConfig

- Integrates all configuration, including command-line switches, files, and anything else

```
#!/usr/bin/perl
# appconfig-args.pl

use AppConfig;

my $config = AppConfig->new;

$config->define( 'network_email=s'    );
$config->define( 'book_author=s'      );
$config->define( 'book_title=s'       );

$config->file( 'config.ini' );

$config->args();

my $email  = $config->get( 'network_email' );
my $author = $config->get( 'book_author' );

print "Kindly send complaints to $author ($email)\n";
```

Using the program name

- An older trick uses the program name, \$0 (zero)
- It's the same program, called differently

```
% ln -s program.pl foo.pl  
% ln -s program.pl bar.pl
```

- Switch based on \$0

```
    if( $0 eq 'foo.pl' ) { ... }  
elseif( $0 eq 'bar.pl' ) { ... }  
else      { ... default }
```

By operating system

- Configure based on \$O (capital O)
- File::Spec works differently on different platforms

```
package File::Spec;

my %module = (MacOS      => 'Mac',
              MSWin32    => 'Win32',
              os2        => 'OS2',
              VMS        => 'VMS',
              epoc       => 'Epoc',
              NetWare    => 'Win32',
              dos        => 'OS2',
              cygwin     => 'Cygwin');

my $module = $module{$^O} || 'Unix';

require "File/Spec/$module.pm";
@ISA = ("File::Spec::$module");

1;
```

Writing your own interface

- Don't use any of these directly in your big applications
- Create a façade to hide the details
- You can change the details later without changing the application
- The interface just answers questions
- Your configuration object might be a singleton

```
my $config = Local::Config->new;      always gets the same reference
```

Good method names

- Your configuration answers task-oriented questions

```
$config->am_debugging
```

```
$config->am_verbose
```

```
$config->use_foo
```

- You don't care how it gets the answer, you just want it

Further reading

- The *perlrun* documentation details the `-s` switch
- The *perlport* documentation discusses differences in platforms and how to distinguish them inside a program.
- Teodor Zlatanov wrote a series of articles on AppConfig for IBM developerWorks, “Application Configuration with Perl” (<http://www-128.ibm.com/developerworks/linux/library/l-perl3/index.html>), “Application Configuration with Perl, Part 2”, (<http://www-128.ibm.com/developerworks/linux/library/l-appcon2.html>), and “Complex Layered Configurations with AppConfig” (<http://www-128.ibm.com/developerworks/opensource/library/l-cpappconf.html>)
- Randal Schwartz talks about `Config::Scoped` in his *Unix Review* column for July 2005, (<http://www.stonehenge.com/merlyn/UnixReview/col59.html>).

Logging

Log without changing the program

- I don't want to change the program to
 - * get extra information
 - * change information destination
 - * turn off some output
- I want to log different sorts of messages
 - * error messages
 - * debugging messages
 - * progress information
 - * extra information

Two major modules

- There are many ways to do this
- Everyone seems to reinvent their own way
- There are two major Perl modules
 - * `Log::Dispatch`
 - * `Log::Log4perl`
- I'll use `Log::Log4perl` since it can use `Log::Dispatch`

The :easy way

- `Log::Log4perl` is Perl's version of Log4java
- It's easy to use with few dependencies
- The `:easy` import gives me usable defaults

```
use Log::Log4perl qw(:easy);
```

```
Log::Log4perl->easy_init( $ERROR );
```

`$ERROR` exported

```
ERROR( "I've got something to say!" );
```

- The message is formatted with a timestamp

```
2006/10/22 19:26:20 I've got something to say!
```

- I can change the format (more later)

Logging levels

- Log4perl has five different levels

```
DEBUG( "The value of x is [$x]" );  
INFO(  "Processing record $number" );  
WARN(  "Record has bad format" );  
ERROR( "Mail server is down" );  
FATAL( "Cannot connect to database: quitting" );
```

- Each level has a method of that name
- The method only outputs its message if it is at the right level (or higher)
 - * The DEBUG level outputs all messages
 - * The ERROR level only outputs ERROR and FATAL
- Don't need conditionals or logic
- Can be changed with configuration

Something more complex

- I want to send different levels to different destinations
- It's still simple with the `:easy` setup

```
use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init(
{
    file    => ">> error_log",
    level   => $ERROR,
},

{
    file    => "STDERR",
    level   => $DEBUG,
}
);

ERROR( "I've got something to say!" );

DEBUG( "Hey! What's going on in there?" );
```

Configuring Log4perl

- I don't want to change the code
- I can use a configuration file

```
use Log::Log4perl;

Log::Log4perl::init( 'root-logger.conf' );

my $logger = Log::Log4perl->get_logger;

$logger->error( "I've got something to say!" );
```

- The configuration file has the logging details

```
log4perl.rootLogger          = ERROR, myFILE

log4perl.appender.myFILE     = Log::Log4perl::Appender::File
log4perl.appender.myFILE.filename = error_log
log4perl.appender.myFILE.mode   = append
log4perl.appender.myFILE.layout = Log::Log4perl::Layout::Pattern
    Layout
log4perl.appender.myFILE.layout.ConversionPattern = [%p] (%F line
    %L) %m%n
```

Appenders handle the magic

- An appender is something that gets a message and send it somewhere
- You can send it just about anywhere you like

```
Log::Log4perl::Appender::Screen  
Log::Log4perl::Appender::ScreenColoredLevels  
Log::Log4perl::Appender::File  
Log::Log4perl::Appender::Socket  
Log::Log4perl::Appender::DBI  
Log::Log4perl::Appender::Synchronized  
Log::Log4perl::Appender::RRDs
```

- Use the right appender with its specialized configuration
- Can also use `Log::Dispatch` appenders

Logging to a database

- Use the DBI appender with the right data source and insert statement

```
log4perl.category = WARN, CSV
```

```
log4perl.appender.CSV = Log::Log4perl::Appender::DBI
log4perl.appender.CSV.datasource = DBI:CSV:f_dir=.
log4perl.appender.CSV.username = sub { $ENV{CSV_USERNAME} }
log4perl.appender.CSV.password = sub { $ENV{CSV_PASSWORD} }
log4perl.appender.CSV.sql = \
    insert into csvdb \
    (pid, level, file, line, message) values (?, ?, ?, ?, ?)
```

```
log4perl.appender.CSV.params.1 = %P
log4perl.appender.CSV.params.2 = %p
log4perl.appender.CSV.params.3 = %F
log4perl.appender.CSV.params.4 = %L
log4perl.appender.CSV.usePreparedStmt = 1
```

```
log4perl.appender.CSV.layout = Log::Log4perl::Layout::NoopLayout
log4perl.appender.CSV.warp_message = 0
```

Changing configuration on-the-fly

- Log4perl can reload the configuration file on the fly
- Check the configuration file every 30 seconds

```
Log::Log4perl::init_and_watch( 'logger.conf', 30 );
```
- Change the log level to get more (or less) information
- Change the appender to send the messages to a different place

Send to screen and file at once

- To send to multiple destinations, just add an appender
- This configuration uses myFile and Screen

```
log4perl.rootLogger          = ERROR, myFILE, Screen

log4perl.appender.myFILE     = Log::Log4perl::Appender::File
log4perl.appender.myFILE.filename = error_log
log4perl.appender.myFILE.mode   = append
log4perl.appender.myFILE.layout = Log::Log4perl::Layout::Pattern
    Layout
log4perl.appender.myFILE.layout.ConversionPattern = [%p] (%F line
    %L) %m%n

log4perl.appender.Screen     = Log::Log4perl::Appender::Screen
log4perl.appender.Screen.stderr = 0
log4perl.appender.Screen.layout = Log::Log4perl::Layout::SimpleLayout
```

- Appenders can have different configuration and layouts

Multiple loggers

- Define multiple loggers inside your configuration file
- Use a “category”

```
log4perl.rootLogger          = ERROR, myFILE, Screen
log4perl.category.Foo        = DEBUG, myFile
log4perl.category.Foo.Bar    = FATAL, Screen
```

- In the code, create new logger instances for what you need

```
my $logger = Log::Log4perl->new('Foo');
my $logger = Log::Log4perl->new('Foo.Bar');
```

- Categories are inheritable, so Foo.Bar inherits from Foo in the configuration
 - * can extend
 - * can override
 - * can turn off features

Further reading

- The Log4perl project at Sourceforge, (<http://log4perl.sourceforge.net/>), has Log4Perl FAQs, tutorials, and other support resources for the package. Most of the basic questions about using the module, such as “How do I rotate log files automatically”
- Michael Schilli wrote about Log4perl on Perl.com, “Retire Your Debugger, Log Smartly with Log::Log4perl!”, (<http://www.perl.com/pub/a/2002/09/11/log4perl.html>).
- Log4Perl is closely related to Log4j (<http://logging.apache.org/log4j/docs/>), the Java logging library, so you do things the same way in each. You can find good tutorials and documentation for Log4j that you might be able to apply to Log4perl too.

Lightweight Persistence

Persistence

- Data persists so it sticks around between program runs
- Pick up where you left off last time
- Share data with another program
- I'm thinking about anything too small for DBI
 - * SQLite is nice, but you just use DBI

Perl structures as text

- The `Data::Dumper` module outputs Perl data as text

```
use Data::Dumper;

my %hash = qw(
    Fred  Flintstone
    Barney Rubble
);
my @array = qw(Fred Barney Betty Wilma);

print Dumper( \%hash, \@array );
```

- The output is Perl code

```
$VAR1 = {
    'Barney' => 'Rubble',
    'Fred'  => 'Flintstone'
};
$VAR2 = [
    'Fred',
    'Barney',
    'Betty',
    'Wilma'
];
```

Using my own name

- I don't want the \$VAR1 and \$VAR2 style names
- I can choose my own names

```
use Data::Dumper qw(Dumper);

my %hash = qw(
    Fred    Flintstone
    Barney Rubble
);

my @array = qw(Fred Barney Betty Wilma);

my $dd = Data::Dumper->new(
    [ \%hash, \@array ],
    [ qw(hash array) ]
);

print $dd->Dump;
```

Nicer output

- Now I can see what names go with what data

```
$hash = {  
    'Barney' => 'Rubble',  
    'Fred' => 'Flintstone'  
};  
$array = [  
    'Fred',  
    'Barney',  
    'Betty',  
    'Wilma'  
];
```


Reading Data::Dumper text

- I read in the text then `eval` it in the current lexical context

```
my $data = do {  
    if( open my $fh, '<', 'data-dumped.txt' ) {  
        local $/; <$fh> }  
    else { undef }  
};
```

```
my $hash;  
my $array;
```

comes back as a reference

```
eval $data;
```

```
print "Fred's last name is $hash{Fred}\n";
```

YAML Ain't Markup

- The YAML module acts like Data::Dumper
- The output is prettier and easier to hand-edit
- All the cool kids are doing it

```
use Business::ISBN;
use YAML qw(Dump);

my %hash = qw(
    Fred Flintstone
    Barney Rubble
);

my @array = qw(Fred Barney Betty Wilma);

my $isbn = Business::ISBN->new( '0596102062' );

open my($fh), ">", 'dump.yml'
    or die "Could not write to file: $!\n";
print $fh Dump( \%hash, \@array, $isbn );
```

YAML format

- The YAML format is nicer than Data::Dumper

```
---
Barney: Rubble
Fred: Flintstone
---
- Fred
- Barney
- Betty
- Wilma
--- !perl/Business::ISBN
article_code: 10206
checksum: 2
country: English
country_code: 0
isbn: 0596102062
positions:
  - 9
  - 4
  - 1
publisher_code: 596
valid: 1
```

Reading in YAML

- Loading the YAML is slightly easier, too

```
use Business::ISBN;
use YAML;

my $data = do {
    if( open my $fh, '<', 'dump.yml' ) {
        local $/; <$fh> }
    else { undef }
};

my( $hash, $array, $isbn ) = Load( $data );

print "The ISBN is ", $isbn->as_string, "\n";
```

- Doesn't depend on lexical scope, but I have to remember variable order

Storable

- Storable makes a binary, packed file that it can read later

```
use Business::ISBN;  
use Storable qw(nstore);
```

```
my $isbn = Business::ISBN->new( '0596102062' );
```

```
my $result = eval {  
    nstore( $isbn, 'isbn-stored.dat' ) };           needs a reference
```

```
if( $@ )  
    { warn "Serious error from Storable: $@" }  
elsif( not defined $result )  
    { warn "I/O error from Storable: $!" }
```

- Use `nstore` to avoid endianness issues
- I can also store to a filehandle

```
open my $fh, ">", $file  
    or die "Could not open $file: $!";  
my $result = eval{ nstore_fd $isbn, $fh };
```

Reading Storable files

- Use `retrieve` to unpack the data

```
my $isbn = eval { retrieve($filename) };
```

- Use `fd_retrieve` to read from a filehandle

```
my $isbn = eval { fd_retrieve(\*SOCKET) };
```

- There's no `nretrieve` because Storable figures it out

Freezing and thawing

- I don't need a file or filehandle
- With `nfreeze`, I can get the packed data back as a string

```
use Business::ISBN;  
use Data::Dumper;  
use Storable qw(nfreeze thaw);  
  
my $isbn = Business::ISBN->new( '0596102062' );  
  
my $frozen = eval { nfreeze( $isbn ) };  
  
if( $@ ) { warn "Serious error from Storable: $@" }
```

- To turn the packed data back into Perl, I use `thaw`

```
my $other_isbn = thaw( $frozen );  
  
print "The ISBN is ", $other_isbn->as_string, "\n";
```

Storing multiple values

- To store multiple values, I need to make a single reference

```
my $array = [ $foo, $bar ];  
my $result = eval { nstore( $array, 'foo.dat' ) };
```

- I have to remember the structure I used

```
my $array_ref = retrieve( 'foo.dat' );  
my( $foo, $bar ) = @$array_ref;
```


Deep copies

- When I copy a reference, I get a *shallow copy*
- Any internal references point to the same data as the source
- Storable can make a *deep copy*, so the copy is completely independent
- A freeze followed by a thaw will do it

```
my $frozen = eval { nfreeze( $isbn ) };  
my $other_isbn = thaw( $frozen );
```

independent of \$isbn

- I can also use `dclone`

```
use Storable qw(dclone);  
my $deep_copy = dclone $isbn;
```

independent of \$isbn, again

dbm files (old, trusty)

- DBM files are like hashes that live on a disk
- They retain their values between program invocations
- There are many implementations, each with different limitations; simple key and value, no deep structure
- Perl uses a tied hash to connect to the file

```
dbmopen %DBM_HASH, "/path/to/db", 0644;  
$DBM_HASH{ 'foo' } = 'bar';  
dbmclose %DBM_HASH;
```

sync all changes

- Often used for large hashes, so be careful with memory

```
my @keys = keys %DBM_HASH;  
foreach ( @keys ) { ... }
```

now in memory!

- Use while with each instead

```
while( my( $k, $v ) = each %DBM_HASH )  
{ ... }
```

one pair at a time

A better DBM

- The DBM::Deep module lets me use any structure
- The value can be a reference

```
use DBM::Deep;
```

```
my $isbns = DBM::Deep->new(  
    file      => "isbn.db"  
    locking   => 1,  
    autoflush => 1,  
);
```

```
if( $isbns->error ) {  
    warn "Could not create db: " . $isbns->error . "\n";  
}
```

```
$isbns->{'0596102062'} = 'Intermediate Perl';
```

```
my $title = $isbns->{'0596102062'};
```

- Treat it like a normal Perl reference. Persistence is free

Further reading

- *Advanced Perl Programming, Second Edition*, by Simon Cozens: Chapter 4, “Objects, Databases, and Applications”.
- *Programming Perl, Third Edition*, discusses the various implementations of DBM files.
- Alberto Simões wrote “Data::Dumper and Data::Dump::Streamer” for *The Perl Review* 3.1 (Winter 2006).
- Vladi Belperchinov-Shabanski shows an example of `Storable` in “Implementing Flood Control” for Perl.com: (<http://www.perl.com/pub/a/2004/11/11/floodcontrol.html>).
- Randal Schwartz has some articles on persistent data: “Persistent Data”, (<http://www.stonehenge.com/merlyn/UnixReview/col24.html>); “Persistent Storage for Data”, (<http://www.stonehenge.com/merlyn/LinuxMag/col48.html>); and “Lightweight Persistent Data”, (<http://www.stonehenge.com/merlyn/UnixReview/col53.html>)

Conclusion

Main points

- Profile your application before you try to improve it
- Be very careful and sceptical with benchmarks
- Make your program flexible through configuration
- Use Log4perl to watch program progress, report errors, or debug
- Use lightweight persistence when you don't need a full database server

More information

- Stonehenge: *www.stonehenge.com*
- Feel free to email me: *brian@stonehenge.com*
- See all of my talks, *<http://www.pair.com/~comdog/>*
- Also on SlideShare, *http://www.slideshare.net/brian_d_foy*
- Often on Perlcast, *<http://www.perlcast.com>*

Questions