5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246

joseph@5sigma.com • http://www.perltraining.com

Idiomatic Perl

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Idiomatic Perl in general

- Perl is a very high-level language
- C-like control structures and arithmetic operators
- Some shell-like constructs
- Other things unique to Perl, like scalar/list context and parentheses-less “list operator” syntax
- Mix all the ingredients together “effectively” and you get idiomatic Perl
- But what is idiomatic Perl like?
- In this session I’ll cover common Perl idioms and programming practices
- —Especially, some things that will confuse or seem strange to people new to Perl
- —You might consider this a lecture on “Perl programming style”
- We’ll start with the basics, then move to things that require more explanation ...

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

How do you organize a Perl program?

- Put your `use` directives at the top of the program
- Follow with declarations of any file-scoped `my` variables
- You can define other `my` variables within blocks
- If you like to use the "list operator" subroutine syntax (`foo $bar, $baz` vs. `foo($bar, $baz)`), define your subroutines in "Pascal order"—before you use them in the text of your program
- You don't have to define subroutines before you use them if you use `&foo()` or `foo()` syntax
- Consider embedding Perl pod (Plain Old Documentation) in your program
- Always `use strict` in a program of significant length, and test your programs with `-w`
- If you break your program down into modules, it's a good idea to use `h2xs` to develop the modules—a must if you plan to share or distribute them

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

C-like control structures

- Perl's control structures are mostly C-like—`foreach` is the most significant exception (covered below)
- Perl and C `while` and `for` are very similar, but braces are required

```
while ($condition) { do_something() }
for ($i = 0; $i < 10; $i++) { do_something_with($i) }
```

- Perl `if` is similar to C `if`, but again with required braces

```
if ($foo) {
  foo();
} else {
  if ($bar) {
    bar();
  }
}
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

C-like control structures (cont'd)

- So Perl also has `elsif`

```
if ($foo) {
  foo();
} elsif ($bar) {
  bar();
}
```

- Required braces make Perl parsing simpler (no "dangling else"—gets rid of those pesky shift-reduce conflicts) and also make program structure more obvious and possibly less error-prone
- There are also "inverses" like `unless` and `until`
- Some other C-like structures can be approximated (see below)
- C-like control structures are well-known and unsurprising to both novice and experienced programmers
- Prefer C-like control structures to other alternatives, all else being equal

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Other control structures: `foreach`

- The most commonly used control structure in Perl not derived from C is `foreach`

```
foreach $item ($foo, $bar, @baz) {
  do_something_with($item);
}
```

- `foreach` loops are an important Perl idiom, usually a programmer's first choice for looping
- Can be problematic at first for C programmers, because using a subscript with a `for` loop is decidedly less efficient and succinct

```
for ($i = 0; $i < @items; $i++) { fiddle_with($items[$i]) }
```
UGH!

```
foreach $item (@items) { fiddle_with($item) }
```
more Perl-like

- `foreach` can be spelled `for`, and the control variable defaults to `$_`

```
for (@items) { fiddle_with($item) }
```

- You can even (in 5.005+) use the statement modifier syntax

```
fiddle_with($item) for @items;
```
starting to look like Perl!

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Other control structures: statement modifiers

- You can follow an expression with a modifier: `if`, `while`, `unless`, `until`, `foreach`

```
print "A-OK\n" if $ok;
print "i = $i\n" until ++$i > 10;
warn $@ if $@;
```

- These are equivalent in efficiency to the C-like syntax, but often more readable or succinct because the braces and parentheses go away

```
if ($@) {
  warn $@;
}
```

looks even sillier if squeezed onto one line

- Statement modifiers illustrate perfectly the issues involved in writing idiomatic Perl—replicates functionality of a different syntax, so why use them?
- Why say “that’s cool” sometimes and “that is an elegant and unanticipated solution” others?
- Idiomatic Perl requires choosing the most appropriate of similar alternatives
- In general, use the shortest, simplest syntax

Quasi control structures: ||, &&, or, and

- The short-circuit logical operators || and && work mostly in Perl as they do in C, so you can use them as control structures

```
check_printer() || die("Printer hosed");
$debug && warn "Made it to here";
```

- However, they return the last evaluated argument rather than 1/0 for true/false, which makes them *much* more versatile

```
$size = param('size') || $defaults{'size'};
```

set $size *from* param('size') *if true*

- Shell programmers may think of || and && as a replacement for if/unless, but many people find this distracting (or just plain perverse), and precedence goofs are always possible

```
@args && process_args(@args);
valid($input) || do { close_temp_files(); die "invalid input: $input" };
```

- For clarity, use || and && when you need the result; otherwise use if/unless

```
process_args @args if @args;
unless (valid($input)) { close_temp_files(); die "invalid input: $input" }
```

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Quasi control structures: ||, &&, or, and (cont'd)

- But certain uses of || and && as control structures are entirely idiomatic

```
unless (open F, $file) {                                    acceptable but not the usual way
  die "couldn't open: $!";
}
open(F, $file) || die "couldn't open: $!";                  the usual way
!system("/usr/bin/cmp", "-s", "f1", "f2") || warn "compare failed";
system("/usr/bin/cmp", "-s", "f1", "f2") && warn "compare failed";
```

- The super low precedence versions or and and let you write these without parentheses

```
open F, $file or die "couldn't open: $!";                   probably the most readable
system "/usr/bin/cmp", "-s", "f1", "f2" and warn "compare failed";
```

Whither `switch`?

- There's no direct equivalent to C's `switch` in Perl—plain old `if`/`elsif`/`else` works fine

```
if ($x == 1) {
  print "case 1\n";
} elsif ($x == 2) {
  print "case 2\n";
} else {
  print "default\n";
}
```

- The `perlsyn` man page suggests many alternatives, but to no particular advantage
- Here's a trickier alternative, possibly better if you have many cases

```
&{{
  1 => sub { print "case 1\n" },                    case => anon sub
  2 => sub { print "case 2\n" },
}->{$x} || sub { print "default\n" }};              call "default" sub if not found in hash
```

Tricks with blocks

- Perl has a `goto` statement but you never need it (except for the special version used to hack the stack frame for `AUTOLOAD`)
- You can create any kind of gnarlified flow of control you need with blocks and loop control

```
{
  do_abc(), last if /^abc/;
  do_def(), last if /^def/;
  do_ghi(), last if /^ghi/;
}
```

another way of writing a "switch"

- The `redo` statement in a block is one of several idiomatic ways of writing infinite loops

```
{ do_something(); last if $finished; redo; }
while (1) { do_something(); last if $finished; }
for (;;) { do_something(); last if $finished; }
```

or just `while ()`

popular with some C programmers

- Did you know `last`/`next`/`redo` will even yank you out of a subroutine?—but, alas, `-w` slaps your hand

Control structure oddments

- Labels can be reused (perhaps this isn't all that weird)

```
LINE: while (<>) { last LINE if /^end of part 1/i }
LINE: while (<>) { last LINE if /^end of part 2/i }
```

- Labels are put on a stack at runtime, so you can have a loop labeled `LINE` inside another loop labeled `LINE`
- `if`, `do`, and `eval` blocks aren't loop blocks, but you can add a naked block inside or outside, which sometimes looks like "doubling braces"

```
do { last if /^exit/i; do_line() } while (<>);          will be a run-time error
{ do { last if /^exit/i; do_line() } while (<>) }       works now
do {{ next unless /\S/; do_line() }} while (<>);        for next, the block can go inside
eval {{
  last if $this;
  redo if $that;
}};
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Things to put in conditions

- Perl conditions often contain the same sorts of things you see in C conditions, especially assignments

```
process_data($x) while $x = slurp_data();
$x = slurp_data() or die "out of data";
```

- You can also test a list assignment, which is false when an empty list is assigned

```
while (($k, $v) = each %hash) { print "$k: $v\n" }
```

- Scalar pattern matches and list assignments from matches are popular conditions

```
if (/^quit/i) { clean_up(); exit; }
if (($field, $value) = ($line =~ /^([\w-]+):\s+(\S+)/)) { ...
```

- You may want to test whether something is defined (not just true) ... *sometimes* this works

```
if (defined $foo) { ...
```
true if `$foo` *isn't* `undef`

```
if (%hash) { ...
```
true if hash contains key-value pairs; q.v. `keys %hash`

```
if (defined %hash) { ...
```
does NOT tell you whether `%hash` *is "defined"*

```
if (defined &$foo) { ...
```
true if subroutine named in `$foo` *is defined*

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

More things to put in conditions

- You will also see the popular and useful
  ```
  while (<STDIN>) { ...
  while (<>) { ...
  ```
- This is a shorthand for
  ```
  while (defined($_ = <STDIN>)) { ...
  while (defined($_ = <>)) { ...
  ```
- A recent patch also gives the same semantics to
  ```
  while ($_ = <STDIN>) { ...
  ```
- In the past, this would exit prematurely if the last line in a file was a 0 with no newline, but this was deemed confusing—silly, even—and the behavior was liberalized in 5.005-something
- The same thing applies to
  ```
  while (<*>) { print "$_\n" }
  ```
  works
  ```
  while ($_ = <*>) { print "$_\n" }
  ```
  formerly, stopped prematurely for a file named `0`

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

The default variable: `$_`

- `$_`, the "default variable," is another peculiar feature of Perl (mirrored in a few other languages)
- Used in many places as a default value if another isn't specified, most notably pattern matching and substitution, and in `while (<>)`
- `$_` is often best used when you don't see it at all

```
tr/A-Z/a-z/ for @list;
@text_files = grep -T, @files;
while (<>) { s/\bfoo\b/bar/g; print; }
```

same as `grep -T $_, @files`

- `$_` is easy to use but tricky to master—there is no rule for which things use `$_` and which don't, but whatever happens generally makes sense
- Using `$_` generally makes your programs shorter and easier to read (for all but complete beginners)
- If you modify `$_` in a subroutine, remember to localize it with `local`

5 Sigma Productions
P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Localizing variables

- In general, use `my` to declare local variables

```
my $this;
my (@that, %the_other_thing);
```

use parentheses for more than one thing

- The alternative is `local`, which saves/restores the value of an existing global variable

```
local $this;
local (@that, %the_other_thing);
```

- `my` variables are more efficient and work more like local variables from other languages
- Sometimes you need `local`, for example, to localize a special variable or a member of a hash

```
{ local $_ = shift; ... }
{ local $SIG{__WARN__} = sub {
      if ($warn++ > 10) { die "too many warnings" }
      else { warn shift }
  }; ...
{ local $/; $slurp = <FILE>; }
```

`my $_` *is illegal, mostly*

localize an element of the `%SIG` *hash*

clear the input record separator `$/` *inside this block*

Subroutine arguments

- Name your subroutine arguments with `my` variables

```
sub rosa { my $secret = shift; ... }                 usual way to name a single argument
sub ordinate { my ($toady, $sycophant) = @_; ... }   usual way to name more than one
sub terraneans { my ($bowie, $eno) = @_[0, 1]; ... }      or assign straight from @_
```

- Named arguments improve readability
- In most cases it is faster to use a copy of something in a `my` variable than a subscript like `$_[0]`
- But to change subroutine arguments in place, modify the elements of `@_`

```
sub uppercase_in_place {
  tr/a-z/A-Z/ for @_;
}
uppercase_in_place $a, $b, $c;
```

- To change an array argument itself, use a reference or a prototype

```
sub pop2 { my $aref = shift; splice @$aref, -2 }      use like ($a, $b) = pop2 \@a
sub pop2 (\@) { my $aref = shift; splice @$aref, -2 }          ($a, $b) = pop2 @a
```

5 Sigma Productions
P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

More about localizing

- You can use `my` in the middle of files or blocks, which is good, but don't confuse yourself

```
{
  $a = 10; my $a = 20;
  print "a = $a, or is that $::a?\n";
}
```

prints `a = 20, or is that 10?`

- In fact you can use `my` just about anywhere!

```
for (my $i = 0; $i < 10; $i++) { print "i = $i\n" }
foreach my $i (1..10) { print "i = $i\n" }
```

- If `my` appears in a condition or loop clause, it lasts through the end of the loop block
- Of course, sometimes it "works"

```
until (my $done) { $done = "true, dammit!" }
```

infinite loop

```
$self->{Min} = 10; $value = 5;
```

`my` *takes effect* ***after*** *the current statement*

```
die "too small" if defined(my $min = $self->{Min}) and $value < $min;
```

doesn't die

Using `foreach`, `grep`, and `map`

- `foreach`, `grep`, and `map` perform basic operations on lists
- Use `foreach` to iterate over the contents of a list

```
foreach $item (@list) { do_something_with $item }
for ($a, $b, $c) { do_something_with($_) }
```

- Use `grep` to select or count elements according to a boolean expression

```
@existing_files = grep -e, @files;
$existing_count = grep -e, @files;
```

like `grep { -e $_ } @files`
in scalar context

- Use `map` to create a transformed copy of a list

```
@uppercase = map "\U$_", @strings;
```

- In all three constructs, `$_` is an alias for the current element—changing `$_` changes the element
- It is bad style to use void context `map` or `grep` to modify a list in place—use `foreach` instead

```
grep s/\bDr\./Doctor/, @strings;
s/\bDr\./Doctor/ for @strings;
```

UGH! void `grep`
`foreach` *loop—the right way*

Fun with `map`

- Sometimes you want a copy of a value in `$_` so that you can use it more than once conveniently
- You can use `foreach` if you don't need to use the result

```
$n[$i][$j] = 4;
printf "%d, %d, %d", $_, $_**2, $_**3 for $n[$i][$j];
```
prints `4, 16, 64`

- If you need the result, use `map`

```
@powers = map { $_, $_**2, $_**3 } $n[$i][$j];
```
`@powers = (4, 16, 64)`

- Use `map` to nest and slice

```
@xyz = map [$x[$_], $y[$_], $z[$_]], 0..$#x;
```
`[$x[0], $y[0], $z[0]], [$x[1], ...`

```
@x = map $_->[0], @xyz;
```
`@x = ($xyz[0][0], $xyz[1][0], $xyz[2][0], ...)`

- This leads to the "Schwartzian Transform"

```
@sorted_by_size = map { $_->[0] }
                  sort { $b->[1] <=> $a->[1] }
                  map { [$_, -s $_] }
                  <*>;
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

When does your program `require` use?

- `do` takes the contents of a file and string evals them (searches the include path)
  ```
  do $filename;
  $result = do $filename;
  ```
  same as result of `eval` *on the contents of the file*
- `do` is handy for loading configuration files, if they're written in Perl
- `require` is a smarter `do`—it doesn't include the same filename more than once, tests to see that the eval returns a true value, and automatically adds a `.pm` extension for `require` *bareword*
- `use` is a compile-time form of `require` adapted especially for loading modules—adds symbol importing, and the `require` is hoisted into a `BEGIN` block
- You can load modules with either `require` or `use`, but `use` gives you compile-time importing
  ```
  require File::Basename;  print File::Basename::basename('/foo/bar');
  use File::Basename;      print basename '/foo/bar';
  ```
- With `require`, names aren't imported, and you have to use `&` or `()` with subroutines
- Module writers often use `require` inside modules

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Can I quote you on that?

- Perl lets you quote strings many different ways
- Single-quoted strings without interpolation, double-quoted strings with
- There are generalized quote operators, `q` and `qq`, that let you use any punctuation character as a delimiter
- Matching delimiters are useful when quoting code

```
use Benchmark;
timethese(500, {
  for => q{ for ($i = 0; $i < 10000; $i++) { $x++ } },
  foreach => q{ foreach $i (1..10000) { $x++ } },
});
```

- You can also use matching delimiters for block commenting (works for most things)

```
q{
   print "This code has been commented out.\n";
};
```

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Globs of fun

- Typeglobs are Perl's most idiosyncratic feature
- `*a` stands for all the things named a—$a, @a, %a, &a, the filehandle/dirhandle/formatname a
- In the past, typeglobs were a kind of stand-in for references, but you should always prefer references nowadays
- There are still some typeglob idioms though
- Assigning a typeglob to a typeglob (`*a = *b`) creates aliases (now $a is $b, %a is %b, etc.)—sometimes used to localize filehandles/dirhandles

```
sub print_it {
  local *FH = shift;
  print FH "The message is: ", shift;
}
print_it *STDOUT, "Hello, world!\n";
```

More globs of fun

- You can assign a reference to a typeglob, which just aliases that one part of the typeglob

```
*default = \&defaults;                    now, default is a synonym for defaults
sub default { ...
```

- This is useful for patching in new versions of subroutines at runtime—often used in AUTOLOAD subroutines

```
sub AUTOLOAD {
  (my $name) = ($AUTOLOAD =~ /(\w+)$/);
  print "autoloading $AUTOLOAD\n";          just to show you it's only autoloaded once
  *$AUTOLOAD = sub { print "Called $name!\n" };   create and patch in new subroutine
  goto &$AUTOLOAD;                                here's that funky kind of goto
}
foo(); bar(); foo();
```

- You can also extract a reference from a typeglob with the "typeglob subscript" syntax

```
$aref = *foo{ARRAY}; $ioref = *foo{IO};          IO is file/dirhandle part
```

Example 1: from CPAN.pm

- This snippet is from the `o debug` code in `CPAN.pm`
- It's trying to find a case-insensitive match between `$what` (the argument typed in after `o debug`) and the current keys of `%CPAN::DEBUG`, and when it does, it bitwise ors in the appropriate value from `%CPAN::DEBUG` into `$CPAN::DEBUG`

```
my($known) = 0;
for (keys %CPAN::DEBUG) {
  next unless lc($_) eq lc($what);
  $CPAN::DEBUG |= $CPAN::DEBUG{$_};
  $known = 1;
}
```

could also be written `my $known = 0;`
foreach spelled `for`*, using* `$_`
or, `next unless "\L$_" eq "\L$what"`
if found, or in the value

- Okay looking Perl, although he could have done the case-insensitive comparison without a loop if the keys of `%CPAN::DEBUG` were all upper or lower case (they're mixed), and it looks like he can do a `last` after setting `$known = 1`, because only one key will match
- But the code isn't mysterious

Example 2: from Cwd.pm

- This snippet is from the `fastcwd` subroutine in `Cwd.pm`
- Having already determined the current path and put it in `$path`, it verifies that the path is still valid by `chdir`-ing to it and checking that the device and inode numbers for the directory are the same as when the subroutine was called (stashed in `$orig_cdev` and `$orig_cino`)

```
# At this point $path may be tainted (if tainting) and chdir would fail.
# To be more useful we untaint it then check that we landed where we started.
$path = $1 if $path =~ /^(.*)$/;    # untaint
CORE::chdir($path) || return undef;
($cdev, $cino) = stat('.');
die "Unstable directory path, current directory changed unexpectedly"
      if $cdev != $orig_cdev || $cino != $orig_cino;
```

`Cwd` *also defines its own* `chdir`

- Very helpful comments—it might take a bit for a reader to figure out why `fastcwd` is doing a `chdir`, and (if unfamiliar with tainting) to understand the regular expression
- Typical use of Perl control structures, logical operators, list assignments, etc.

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

So, what is good Perl style?

- Good style is simple, succinct, and unsurprising
- Don’t invent syntax—Perl gives you plenty of alternatives to begin with
- Use familiar constructs—Perl will let you do just about anything, but tend toward the plain
- Don’t write C (or shell) in Perl, but feel free to borrow
- Write for a moderately experienced audience, not beginners
- The complexity of your code should parallel the complexity of the problem
- Don’t use surprising constructs to solve simple problems
- Code plainly, or at least decipherably—leave clues
- Format and indent your code consistently
- Perl “baby talk” is officially sanctioned, but Perl has its own complex nature that takes time to discover
- Look in the core modules (`/usr/local/lib/perl5` or wherever) and in the CPAN for examples

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Ten ways Perl programmers mess up their lives

- **(10) They keep using Perl 4**
- Perl 4 is dead. Long live Perl 5. If you are thinking of buying a CGI programming book, make sure it uses Perl 5. You don't want one that uses `cgi-lib.pl`, calls all its subroutines with the `&func()` syntax, and doesn't tell you anything about all the wonderful Perl modules that have been written in the past half decade.
- **(9) They confuse list and scalar context**
- Remember that operators work differently, sometimes very differently, depending on the context they're used in.
- For example, "Boolean" contexts like the condition of a while loop are scalar. `while (1..10) {…}` means something entirely different than `foreach (1..10) {…}`
- Or the difference between `my ($foo) = @bar` and `my $foo = @bar`

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Ten ways Perl programmers mess up their lives (cont'd)

- **(8) They don't understand regular expressions**
- You can't write Perl effectively without understanding and using regular expressions
- **(7) They omit (too much) punctuation**
- Leaving out semicolons, commas, closing quotes, and/or parentheses can produce very strange error messages, or sometimes silent misbehavior
- For example, don't forget the semicolon after an `eval` block or anonymous `sub`
- Don't omit parentheses around arguments of `||` (better, use `or` instead)
- **(6) They don't read their error messages**
- When a program dies, always look at the *first* error message and/or warning issued
- When you return to the program source code, go to the line number mentioned in the error
- The error in the code is either at or before that line—not after

Ten ways Perl programmers mess up their lives (cont'd)

- **(5) They write fragile programs**
- Always check return status from `open`, `opendir`, and other system calls
- Avoid using relative paths and shell metacharacters in backticks, `system`/`exec`, and pipe opens
- Avoid using non-portable code when a module exists to do the same thing portably (`File::Basename`, `Cwd`, etc.)
- **(4) They bite off too much at once**
- Learn to write two- or three-line programs to test features
- Or use `perl -e '`*insert Perl code here*`'`, or use the debugger as a Perl shell
- **(3) They reinvent the wheel**
- Have an ordinary CGI or network programming task? It's in the CPAN
- Perl modules have been tested, used, and (usually) thoughtfully designed, too
- You can get a lot done in a half dozen lines of Perl, especially if they just run someone else's code

5 Sigma Productions

P.O. Box 6250 • Chandler AZ 85246
joseph@5sigma.com • http://www.perltraining.com

Ten ways Perl programmers mess up their lives (cont'd)

- **(2) They don't read the documentation**
- Read the Perl man pages, especially `perlfunc`
- Use perldoc to read stuff your man command can't see
- Read any (good) Perl books you can find
- **(1) They don't `use strict` or `-w`**
- There is very little sympathy for a Perl programmer who has a problem that `-w` would have caught
- `-w` is a pain sometimes, but it will catch many, many, many careless errors
- `use strict` helps you write better-designed programs
- Okay if you don't bother in programs a few lines long, but always use them both otherwise—especially when embarking on a big new project (definition of "big" varies according to skill level :-)