Idiomatic Perl

Dave Cross
Outcome Technologies

Introduction

Larry Says

- "You can program in Perl Baby-Talk and we promise not to laugh"
- But eventually, it's nice to be able to use the strengths of the language

What is Idiomatic Perl?

- Using the strengths of the language
- Stop writing C or BASIC or sh
- Make you look like a guru!

What We Will Cover

- "use strict" and "use warnings"
- Variables (Package vs Lexical)
- Perl References
- Finding, Installing and Using Modules
- Creating Reuseable Code
- Sorting
- Cartoon Swearing
- "foreach" vs "grep" vs "map"
- Boolean operators as conditionals
- Statement modifiers
- Assignment Operators
- Quoting

use strict and use warnings

Be careful out there

- "use strict" and "use warnings" can catch many tricky bugs
- A very good habit to get into
- Programming with a safety net
- "BUGS: The -w switch is not mandatory." - Larry Wall

use strict

- refs - no symbolic references
- subs - no barewords
- vars - no undeclared variables
- turn them off carefully with "no strict"

use strict 'refs'

- Prevents symbolic references
- aka "using a variable as another variable's name"

```
$what = 'slayer';
$$what = 'Buffy';
# sets $slayer to 'Buffy'
```

- What if 'slayer' came from user input?
- Better to use a hash

```
$what = 'slayer';
$people{$what} = 'Buffy';
```

- Self contained namespace
- Less chance of clashes
- More information (e.g. all keys)

use strict 'subs'

- No barewords
- Bareword is a word with no other interpretation
- e.g. word without $, @, %, &
- Treated as a function call or a quoted string

```
$slayer = buffy;
```

- May clash with future reserved words

use strict 'vars'

- Forces predeclaration of variable names
- Prevents typos
- Less like BASIC - more like Ada
- Many ways to predeclare variables
- "my", "use vars", "our"
- Fully qualified names ($main::foo)
- Thinking about variable scope is good!

use warnings

- Warns against dubious programming habits
- use warnings available since 5.6.0
- use warnings is more flexible than -w

Some typical warnings

- Variables used only once
- Using undefined variables
- Writing to read-only file handles
- And many more...

Turning off warnings

- Need a very good reason!

```
{
  no warnings; # since 5.6.0
  # do dodgy stuff
}
```

- Previously, fiddle with $^W

```
{
  local $^W = 0;
  # do dodgy stuff
}
```

More info

- perldoc strict
- perldoc warnings
- perldoc perlrun (for -w)

Perl Variables

Package and Lexical Variables

- Perl variables are of two types
- Important to know the difference
- Package variables are created by default
- Lexical variables are created with "my"

Package Variables

- Live in the package's symbol table
- Can be refered to using a fully qualified name - $main::slayer or @Buffy::scoobies
- Can be seen from anywhere in the package (or anywhere at all when qualified)
- Can be predeclared with "use vars" or "our"

```
use vars qw($slayer @scoobies %powers);
our ($slayer, @scoobies, %powers);
```

Lexical Variables

- Created with "my"
- Live in a pad (associated with a block)
- Only visible within enclosing block
- "Lexical" because the scope is defined purely by the text

my vs local

- We now know what "my" does, but what about "local"?
- "local" is very badly named (it doesn't create a local variable)
- Saves value of package variable
- Previous value restored on exiting block

my vs local example

```
$x = $y = 'foo';
print "OUT: x = $x, y = $y\n";
print "OUT: x = $main::x, y = $main::y\n";
{
  local $x = 'bar';
  my $y = 'bar';
  print "IN:  x = $x, y = $y\n"; i
  print "IN:  x = $main::x, y = $main::y\n";
}
print "OUT: x = $x, y = $y\n";
print "OUT: x = $main::x, y = $main::y\n";
```

- Output:

```
OUT: x = foo, y = foo
OUT: x = foo, y = foo
IN:  x = bar, y = bar
IN:  x = bar, y = foo
OUT: x = foo, y = foo
OUT: x = foo, y = foo
```

- Within the block $main::y is different to $y

When to use local or my

- Easy answer - always use "my"
- Slightly more complex answer - always use "my" (except when it doesn't work)
- Full answer (by Mark-Jason Dominus) - http://perl.plover.com/FAQs/Namespaces.html http://perl.plover.com/local.html

More info

- perldoc -f local
- perldoc -f my
- perldoc -f our
- perldoc -q lexical
- perldoc perlsub

Perl References

References

- A reference is a bit like a pointer in languages like C and Pascal (but better)
- A reference is a unique way to refer to a variable
- A reference can always fit into a scalar variable
- A reference looks like SCALAR(0x20026730)

Creating References

- Put \ in front of a variable name

```
$scalar_ref = \$scalar;
$array_ref = \@array;
$hash_ref = \%hash;
```

- Can now treat it just like any other scalar

```
$var = $scalar_ref;
$refs[0] = $array_ref;
$another_ref = $refs[0];
```

- It's also possible to create references to anonymous variables (similar to allocating memory using malloc in C)
- Create a reference to an anonymous array using [...]

```
$arr = [ 'an', 'anon', 'array' ];
```

- Create a reference to an anonymous hash using { ... }

```
$hash = { 1 => 'an',
          2 => 'anon',
          3 => 'hash' };
```

Creating References (cont)

```
@arr = (1, 2, 3, 4);
$aref1 = \@arr;
$aref2 = [@arr];
print "$aref1 $aref2";
```

- Output:

```
ARRAY(0x20026800)
ARRAY(0x2002bc00)
```

- Second method creates a **copy** of the array

Using References

- Use {$aref} to get back an array that you have a reference to

```
@array = @{$aref}; # original array
@rev = reverse @{$aref};
$elem = ${$aref}[0]; # one element
${$aref}[0] = 'foo';
```

- Use {$href} to get back a hash that you have a reference to

```
%hash = %{$href}; # original hash
@keys = keys %{$href};
$elem = ${$href}{key}; # one element
${$href}{key} = 'foo';
```

- You can find out what a reference is referring to using "ref"

```
$aref = [ 1, 2, 3 ];
print ref $aref; # prints ARRAY
$href = { 1 => 'one', 2 => 'two' };
print ref $href; # prints HASH
```

Why Use References?

- What does this do?

```
@arr1 = (1, 2, 3);
@arr2 = (4, 5, 6);
check_size(@arr1, @arr2);

sub check_size {
  my (@a1, @a2) = @_;
  print @a1 == @a2 ? 'Yes' : 'No';
}
```

- This doesn't work
- Why doesn't it work?
- Arrays are combined in @_
- All elements end up in @a1
- How do we fix it?
- Pass *references* to the arrays

Why Use References? (cont)

- Another attempt

```
@arr1 = (1, 2, 3);
@arr2 = (4, 5, 6);
check_size(\@arr1, \@arr2);

sub check_size {
  my ($a1, $a2) = @_;
  print @$a1 == @$a2 ? 'Yes' : 'No';
}
```

Complex Data Structures

- Another good use for references
- Try to create a 2D array

```
@arr_2d = ((1, 2, 3),
           (4, 5, 6),
           (7, 8, 9));
```

- This doesn't work
- @arr_2d contains (1, 2, 3, 4, 5, 6, 7, 8, 9)
- This is known a *array flattening*

Complex Data Structures (cont)

- 2D Array using references

```
@arr_2d = ([1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]);
```

- But how do you access individual elements?
- $arr_2d[1] is ref to array (4, 5, 6)
- $arr_2d[1]->[1] is element 5

Complex Data Structures (cont)

- Another 2D Array

```
$arr_2d = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]];
```

- $arr_2d->[1] is ref to array (4, 5, 6)
- $arr_2d->[1]->[1] is element 5
- Can omit intermediate arrows: $arr_2d->[1][1]

More Complex Data Structures

- Suppose you have a data file like this:

```
Summers,Buffy,Slayer
Giles,Rupert,Watcher
Rosenburg,Willow,Witch
Summers,Dawn,Key
```

- What's a good data structure?
- Array of hashes

Building More Complex Data Structures

- Building an array of hashes

```
my @records;
my @cols = qw(s_name f_name job);
while (<FILE>) {
  chomp;
  my %rec;
  @rec{@cols} = split /,/;
  push @records, \%rec;
}
```

- Using an array of hashes

```
foreach (@records) {
  print "$_->{f_name} $_->{s_name} ";
  print "is a $_->{job}\n";
}
```

Even More Complex Data Structures

- Many more possibilities
- Hash of hashes
- Hash of arrays
- Multiple levels (array of hash of hash, etc.)
- Lots of examples in perldoc perldsc (the data structures cookbook)

More Information

- perldoc perlreftut
- perldoc perlref
- perldoc perllol
- perldoc perldsc

Finding, Installing and Using Modules

Modules

- A module is a reusuable 'chunk' of code
- Perl comes with over 100 modules (see perldoc perlmodlib for list)
- Perl has a repository of freely-available modules - the Comprehensive Perl Archive Network (CPAN)
- http://www.cpan.org/
- http://search.cpan.org/

Finding Modules

- http://search.cpan.org
- Search by module name, distribution name or Author name
- Note: CPAN also contains newer versions of standard modules

Installing Modules (the hard way)

- Download distribution file

```
MyModule-X.XX.tar.gz
```

- Unzip

```
gunzip MyModule-X.XX.tar.gz
```

- Untar

```
tar xvf MyModule-X.XX.tar
```

- Change directory

```
cd MyModule-X.XX
```

Installing Modules (the hard way) (cont)

- Create Makefile

```
perl Makefile.PL
```

- Build Module

```
make
```

- Test Build

```
make test
```

- Install Module

```
make install
```

Installing Modules (the hard way) (cont)

- **Note:** May need root permissions for make install
- You can have your own personal module library

```
perl Makefile.PL PREFIX=~/perl
```

- (need to adjust @INC)

Installing Modules (the easy way)

- Note: May not work (or may need some configuration) through a firewall
- CPAN.pm is included with standard Perl distribution
- Automatically carries out installation process
- Can also handle required modules
- Still need to be root

Installing Modules (the easy way) (cont)

- CPAN.pm

```
perl -MCPAN -eshell
CPAN> install My::Module
perl -MCPAN -e"install 'My::Module'"
```

- Also see CPANPLUS

Using Modules

- Two types of module
- Function vs Object
- Functional modules export new subroutines and variables into your program
- Object modules usually don't
- Difference not clear cut (e.g. CGI.pm)

Using Functional Modules

- Import defaults:

```
use My::Module;
```

- Import optional components:

```
use My::Module qw(my_sub @my_arr);
```

- Import defined sets of components:

```
use My:Module qw(:advanced);
```

- Use imported components:

```
$data = my_sub(@my_arr);
```

Using OO Modules

- Use the module:

```
use My::Object;
```

- Create an object:

```
$obj = My::Object->new;
```

- (Note: "new" is just a convention)
- Interact using object's methods:

```
$obj->set_name($name);
```

Useful Standard Modules

- constant - Creates constant values
- File::Copy - Copy files
- Time::Local - Convert times to epoch
- POSIX - Interface to POSIX
- Text::Parsewords - Parse words from text
- CGI - CGI applications

Useful Standard Modules (cont)

- Getopt::Std - Process command line options
- Carp - Better warn and die
- Cwd - Current working directory
- Benchmark - Timing code
- File::Basename - Break up filenames
- Data::Dumper - Dump data to text

Useful Non-Standard Modules

- Template - Insert data in boilerplate text
- DBI - Database access
- libnet - Various network protocols (e.g. Net::FTP)
- Time::Piece - Date/Time manipulation
- libwww - HTTP client library
- Number::Format - Formatting numbers

Useful Non-Standard Modules (cont)

- HTML::Parser - Parsing HTML
- XML::Parser - Parsing XML
- Text::CSV - Parse CSV data
- Regexp::Common - Common regular expressions
- MIME::Lite - Creating and sending MIME emails
- Memoize - Cache function return values

More Information

- perldoc perlmodlib
- perldoc perlmodinstall
- perldoc CPAN

Writing Reusable Code

Why write modules?

- Code reuse
- Prevent reinventing the wheel
- Easier to share across projects
- Better design, more generic

Basic Module

```
use strict;
package MyModule;
use vars qw(@ISA @EXPORT);
use Exporter;
@ISA = ('Exporter');
@EXPORT = ('my_sub');

sub my_sub {
  print "This is my_sub\n";
}

1;
```

Using Your Module

```
use MyModule;

# my_sub is now available for use
# within your program

my_sub(); # Prints "This is my_sub()"
```

Exporting Symbol Names

- Most of MyModule.pm is concerned with *exporting* subroutine names
- Exporting names allows your caller to use the subroutine as my_sub() instead of MyModule::my_sub()

The import Subroutine

- The module Exporter.pm handles the export of subroutine (and variable) names
- Exporter.pm defines a subroutine called import
- import is automatically called whenever a module is used
- import puts references to our subroutines into our callers symbol table

Using @ISA

- How does MyModule use Exporter's import subroutine?
- We make use of *inheritance*
- Inheritance is defined using @ISA
- If we call a subroutine that doesn't exist in our module, then the modules in @ISA are also checked

```
@ISA = ('Exporter');
```

- Therefore Exporter::import is called

Using @EXPORT and @EXPORT_OK

- How does import know which subroutines to export?
- Exports are defined in @EXPORT or @EXPORT_OK
- Automatic exports are defined in @EXPORT
- Optional exports are defined in @EXPORT_OK

Using %EXPORT_TAGS

- You can define sets of exports in %EXPORT_TAGS
- Key is set name (without the colon)
- Value is reference to an array of names

```
%EXPORT_TAGS =
    (advanced => [qw(my_sub
                     my_other_sub)];
use MyModule qw(:advanced);
my_sub();
my_other_sub();
```

@EXPORT or @EXPORT_OK

- Why use @EXPORT_OK
- Give your users the choice of which subroutines to import
- Less chances of name clashes
- Use @EXPORT_OK in preference to @EXPORT
- Document the exported names and sets

Exporting Variables

- You can also export variables

```
@EXPORT_OK = qw($scalar
                @array
                %hash);
```

- Any variables you export must be package variables

Writing Modules The Easy Way

- Use h2xs to create a skeleton module

```
h2xs -A -X -n MyModule
```

- Creates six files
- MANIFEST, Makefile.PL, MyModule.pm, test.pl, README and CHANGES
- These are the standard module package files
- Follows the same conventions as CPAN modules

Writing Objects

- An Object is just a module that obeys certain extra rules
- A Class is a package
- An Object is a reference (usually to a hash)
- A Method is a subroutine
- "bless" tells a reference what kind of object it is

A Simple Object

```
package Slayer;
sub new {
  my $class = shift;
  my $name = shift;
  my $self = { name => $name };
  return bless $self, $class;
}


sub get_name {
  my $self = shift;
  return $self->{name};
}

sub set_name {
  my $self = shift;
  $self->{name} = shift;
}

1;
```

Using The Slayer Object

```
use Slayer;

my $obj = Slayer->new('Buffy');
print $obj->get_name; # prints 'Buffy'
$obj->set_name('Faith');
print $obj->get_name; # prints 'Faith'
```

More Information

- perldoc perlmod
- perldoc perlboot
- perldoc perltoot
- perldoc perlobj
- perldoc h2xs
- perldoc Exporter
- *Object Oriented Perl* - Damian Conway

Sorting

Basic Sorting

- Perl has a sort function that takes a list and sorts it

```
@sorted = sort @array;
```

- Note that it does not sort the list in place

```
@array = sort @array;
```

Default Sorts

- The default sort order is ASCII

```
@chars = sort 'e', 'b', 'a', 'd', 'c';
# @chars has ('a', 'b', 'c', 'd', 'e')
```

- This can sometimes give strange results

```
@chars = sort 'E', 'b', 'a', 'D', 'c';
# @chars has ('D', 'E', 'a', 'b', 'c')
@nums = sort 1 .. 10;
# @nums has (1, 10, 2, 3, 4, 5, 6, 7, 8, 9)
```

Defining Sort Criteria

- Can add a "sorting block" to customise sort order

```
@nums = sort { $a <=> $b } 1 .. 10;
```

- Perl puts two of the values from the list into $a and $b
- Block compares values and returns -1, 0 or 1
- <=> does this for numbers
- cmp does this for strings

Examples of Sort Blocks

- Other simple sort examples

```
sort { $b cmp $a } @words
sort { lc $a cmp lc $b } @words
sort { substr($a, 4)
         cmp
       substr($b, 4) } @lines
```

Using a Sort Subroutine

- Can also use a subroutine name in place of a code block

```
sub dictionary {
  my ($A, B) = ($a, $b);
  $A =~ s/\W+//g;
  $B =~ s/\W+//g;
  $A cmp $B;
}
my @dict = sort dictionary @words;
```

More Complex Sorts

```
my @names = ('Buffy Summers',
             'Rupert Giles',
             'Willow Rosenberg',
             'Dawn Summers');

@names = sort sort_names @names;
```

- Need to write "sort_names" so that it sorts on surname and then forename.

```
sub sort_names {
  my @a = split /\s/, $a;
  my @b = split /\s/, $b;
  return $a[1] cmp $b[1]
    || $a[0] cmp $b[0];
}
```

- This can inefficient on large amounts of data
- Multiple splits on the same data

More Efficient Complex Sorts

- Split each row only once

```
@split = map { [ split ] } @names;
```

- Do the comparision

```
@sort = sort { $a->[1] cmp $b->[1]
               || $a->[0] cmp $b->[0] } @split;
```

- Join the data together

```
@names = map { join ' ', @$_ } @sort;
```

Putting It All Together

- Can rewrite this as

```
@names =
    map  { join ' ', @$_ }
    sort { $a->[1] cmp $b->[1]
          || $a->[0] cmp $b->[0] }
    map  { [ split ] } @names;
```

- All functions work on the output from the previous function in the chain

The Schwartzian Transform

```
@data_out =
    map  { $_->[1] }
    sort { $a->[0] cmp $a->[0] }
    map  { [func($_), $_] } @data_in;
```

More Information

- perldoc -f sort

Cartoon Swearing

Special Variables

- Perl has a number of special built-in variables that we can use
- Often named with punctuation characters ($_) or control characters ($^W)
- Always in main package
- Can't create with "my" - always use "local"
- Only alter them localised in a block

```
{
  local $^W = 0;
  # do stuff...
}
```

The Default Variable - $_

- $_ is the default argument or operand for many functions and operators
- Functions (print, lc, chr)
- Operators (m//, s///, tr///)
- When $_ is being used you often don't see it

```
while (<FILE>){
  next unless /^http/;
  print;
}
```

Input Record Separator - $/

- We know that <FILE> reads one record from the filehandle FILE
- But what defines a 'record'?
- <FILE> actually reads up to and including the next occurance of $/
- Default value is "\n"

The fortune Program

- The Unix "fortune" program reads files in this format

```
'A witty quote'
              -- Oscar Wilde
%%
'Something deep and meaningful'
              -- William Shakespeare
```

- Quotes are separated by "\n%%\n"

```
my @quotes;
{
  local $/ = "\n%%\n";
  @quotes = <FORTUNE>;
  chomp @quotes;
}

print $quotes[rand @quotes];
```

- Note: $/ also effects "chomp"

Special Values For $/

```
$/ = undef;
```

- Puts Perl into 'slurp' mode.
- Next call to <FILE> will read in all remaining data

```
$/ = ''; # empty string
```

- Puts Perl into 'paragraph' mode.
- Records are defined by one or more blank lines

```
$/ = \1024; # or \$number
```

- Reads up to 1024 bytes

Current Record Number - $.

- No need to keep a count of the current record number

```
while (<FILE>) {
  print "$.: $_";
}
```

- **Note:** contains current record number from most recently read filehandle
- **Note:** reset to zero on close

Output Control Variables

- $\ is the output record separator
- Appended to the output to each print call. Default value is the empty string
- $| controls autoflushing on the current default output filehandle
- Default is 0 which buffers output if it's going to a file

More Output Control Variables

- $, is the output field separator
- Controls what is output between the arguments to "print"

```
@nums = 1 .. 3;
print @nums; # 123
$, = ' - ';
print @nums; # 1 - 2 - 3
```

- Default value is the empty string
- $" is the list separator
- Controls what is put between the elements of an array (or array slice) when interpolated in a double quoted string

```
@nums = 1 .. 3;
print "@nums"; # 1 2 3
$" = '|';
print "@nums"; # 1|2|3
```

- Default is a space

$, vs $"

- $, and $" do very similar things
- Actually they do completely different things
- Easy to get them confused

```
@nums = 1 .. 3;
$, = '+';
$" = '-';
print @list; # prints 1+2+3
print "@list"; # prints 1-2-3
```

- Think about what is actually being passed to "print"
- $" really has nothing to do with "print"

```
$" = '-';
@nums = 1 .. 3;
$nums = "@nums"; 1-2-3
```

Error Variables

- $! contains the error from the last operating system call

```
open FILE, 'file' or die $!;
```

- $? contains the error from the last child process

```
@file = `ls`;
die $? if $?;
```

- $@ contains the error from the last "eval" call

```
eval { require Module };
die $@ if $@;
```

Process Variables

- $0 contains the name of the current program name
- $$ contains the process ID
- $<, $(contain the real user and group IDs
- $>, $) contain the effective user and group IDs

Useful Environment Information

- $^O contains the operating system
- $] and $^V contain different representations of the Perl version

```
print $]; # 5.006001
printf '%vd', $^V; # 5.6.1
```

- %ENV contains all of the environment variables

Regular Expression Variables

- $1, $2, etc contain any captured subpatterns

```
while (<MAIL>) {
  if (/^Subject:\s*(.*)/) {
    print "The subject was: $1\n";
  }
}
```

- $` contains the part of the target string that came before the last regex match
- $& contains the part of the target string that matched the last regex
- $' contains the part of the target string that came after the last regex match
- Useful for debugging regular expressions

Regex Debugging

```
$_ = 'This is a string';
if (/is (\w+) (\w+)/) {
  print "$1 : $2\n";
}
```

- Prints "is : a"
- To see what went wrong try this

```
if (/is (\w+) (\w+)/) {
  print "$`<$&>$'\n";
}
```

- "Th<is is a> string"

More Information

- perldoc perlvar

Context

Context

- Be aware of context
- Code can often do completely different things in different contexts

```
print localtime;
print scalar localtime;
```

- Differences can be hard to spot

```
print localtime , "\n";
print localtime . "\n";
```

Contextual Differences

- There is no way to work out what an operator or function will do in a given context

```
@a = reverse(1, 2, 3) # (3, 2, 1)
$s = reverse('123')   # '321'
print reverse('123')  # '123'
```

Knowing Your Context

- You can find out what context your function has been called in by using the (badly named) "wantarray"

```
sub what {
  if (defined wantarray) {
    if (wantarray) { print "list\n" }
    else { print "scalar\n"; }
  } else {
    print "void\n";
  }
}

what();
$sca = what();
@arr = what();
```

More Information

- perldoc -f wantarray
- perldoc -f localtime
- perldoc -f *many other functions and operators*

foreach vs grep vs map

foreach vs grep vs map

- foreach, map and grep do very similar things.
- Important to choose the most appropriate
- grep and map are often better than a foreach loop - but not in void context

Using grep

```
@odds = grep { $_ % 2 } @ints;
```

- grep applies block to each element in the input list
- $_ is aliased to the element
- If block evaluates to true, $_ is added to the output list

Using map

```
@squares = map { $_ * $_} @ints;
```

- map applies block to each element in the input list
- $_ is aliased to the element
- Whatever is returned by the block is added to the output list

More with map

- The output list doesn't need to be the same length as the input list

```
%squares = map { $_, $_ * $_ } @ints;
```

- or

```
%squares = map { $_ => $_ * $_ } @ints;
```

Bad Uses of map and grep

- Don't use map or grep in a void context

```
map { $_ *= $_ } @ints;
```

- Use foreach instead

```
foreach (@ints) { $_ *= $_ };
```

- or

```
$_ *= $_ foreach @ints;
```

- This restriction is being worked on in latest Perls

map & grep Caveats

```
if (grep { /some_pattern/ } @long) {
  ...
}
```

- is inefficient as it always checks every element of @long
- A better solution

```
sub at_least_one_match {
  my $pattern = shift;
  foreach (@_) {
    return 1 if /$pattern/
  }
  return; # No match
}

if (at_least_one_match('pattern', @long)) {
  ...
}
```

- But

```
@matches = grep  {/$pattern/ } @list;
```

- Is better than

```
foreach (@list) {
  push @matches, $item if /$pattern/;
}
```

More Information

- perldoc perlsyn (foreach)
- perldoc -f map
- perldoc -f grep

Boolean Expressions as Conditionals

Boolean Expressions as Conditionals

- In Perl Boolean expressions are *short-circuiting*
- Perl only does as much work as necessary
- (Laziness is a virtue!)

`EXPR1 or EXPR2`

- Only need to evaluate EXPR2 if EXPR1 is false

`EXPR1 and EXPR2`

- Only need to evaluate EXPR2 if EXPR1 is true

The open Idiom

- Probably the most common Perl idiom

```
open FILE, $file
  or die "Can't open $file: $!";
```

- If "open" returns true then the "die" call isn't evaluated
- If "open" returns false then the "die" call must be evaluated

Precedence Issues

- This doesn't do what you want

```
open FILE, $file || die $!;
```

- Perl parses this as

```
open FILE, ($file || die $!);
```

- Use lower precendence "or"

```
open FILE, $file or die $!;
```

- Or use parentheses

```
open(FILE, $file) || die $!;
```

Some Other Examples

```
/match/ and process($_)
$DEBUG and print 'debug message'
@ARGV == 2 or &usage
```

More Information

- perldoc perlop

Statement Modifiers

Statement Modifiers

- Perl allows you to put a statement modifier at the end of each statement
- This is a conditional (if or unless) or a loop (foreach, while or until)
- This simplifies some code

```
process($_) if /match/
print 'debug message' if $DEBUG
&usage unless @ARGV == 2
print "$. : $_" while <FILE>
$_ *= $_ foreach @ints
```

Statement Modifiers & Boolean Conditionals

- Note the similarities and differences between these

```
/match/ and process($_)
process($_) if /match/
```

- There's more than one way to do it
- Choose the one that makes the code more readable

More Information

- perldoc perlsyn

Assignment Operators

Assignment Operators

- Perl has a number of assignment operators that it shares with other languages
- +=, -=, *=, /=
- But Perl extends the list greatly
- %=, .=, x=, ||=
- Look for an assignment operator version of just about any binary operator

Using Assignment Operators

```
$year %= 100;
$line .= "\n";
$indent x= 4;
$value ||= $default;
```

Problems With ||=

- There's one small problem with ||=
- It checks truth, not defined-ness
- Can't use it if 0 or '' are valid values

```
$value = 0;
$value ||= 2; # $value is now 2
```

- Perl 6 will introduce the //= operator

```
$value = 0;
$value //= 2; # $value is still 0
```

More Information

- perldoc perlop

Quoting

Quoting

- Perl has two types of quotes
- Single quotes contain literal strings
- Double quotes interpolate variables and escape sequences (e.g. "\n")
- Escape interpolated characters with a backslash

```
print "\$var is $var"
```

Overusing Backslashes

- Have you ever done this?

```
print "<img src=\"$file\"
       width=\"100\" height=\"50\">";
```

- Nasty isn't it?
- This is much nicer

```
print qq(<img src="$file"
         width="100" height="50">);
```

- qq// allows you to choose your own quote characters

qq// Examples

- Use *any* character

```
qq[some $text]
qq<some $text>
qq{some $text}
qq!some $text!
qq/some $text/
qq#some $text#
qq asome $texta (please forget this at once)
```

Other Quotes

- The same thing works for single quotes using q//

```
q[some text]
q!some text!
```

- Also works for s///, m// and tr///

```
s(something)[something else]
m|/a/directory/name|
tr=A-Z=a-z=
```

Quoting to Build Lists

- The qw// operator splits a string on whitespace and single quotes each element

```
@days = ('Sun', 'Mon', 'Tue', 'Wed',
         'Thu', 'Fri', 'Sat');
@days = qw(Sun Mon Tue Wed
           Thu Fri Sat);
```

More Information

- perldoc perlop

The End

- That's All Folks
- Any Questions?