



A Crash Course in Perl5

Part 4: Flow control

Zeegee Software Inc.

<http://www.zeegee.com/>

Terms and Conditions

These slides are Copyright 2008 by Zeegee Software Inc. They have been placed online as a public service, with the following restrictions:

You may download and/or print these slides for your personal use only. Zeegee Software Inc. retains the sole right to distribute or publish these slides, or to present these slides in a public forum, whether in full or in part.

Under no circumstances are you authorized to cause this electronic file to be altered or copied in a manner which alters the text of, obscures the readability of, or omits entirely either (a) this release notice or (b) the authorship information.

Road map

- Basics
 - Introduction
 - Perl syntax
 - Basic data types
 - Basic operators
- Patterns
 - Introduction
 - String matching and modifying
 - Pattern variables
- Data structures
 - LISTS and arrays
 - Context
 - Hashes
- Flow control
 - Program structures
 - Subroutines
 - References
 - Error handling
- Data
 - Input and output
 - Binary data
 - Special variables
- Object-oriented programming
 - Modules
 - Objects
 - Inheritance
 - Tying



Flow control

Program structures

for/foreach

- The **for** loop is as you may have seen:

```
for ($i = 0; $i < $n; $i++) {  
    # ...do stuff with $i...  
}
```

- The **foreach** loop is intended for iterating through lists:

```
foreach $elem (@list) {  
    # ...do stuff with $elem...  
}
```

while/until

- The **while** loop is mostly as you may have seen:

```
$i = 0;
while ($i < $n) {
    # ...do stuff with $i: identical to the previous for loop!
}
continue { $i++; }
```

- The **until** keyword merely reverses the loop test, so *until EXPR* is the same as *while not EXPR*:

```
$i = 0;
until ($i >= 10) { ...
```

next/last

- The **next** statement forces a jump to the next iteration of a loop (executing any **continue** block first).

```
while ($i < 100) {  
    next if we_dont_like($i);  
    ...  
} continue { ++$i }
```

like continue
in C, but better!

- The **last** keyword breaks out of the loop:

```
for ($i = 0; $i < 100; $i++) {  
    last if $we_should_stop;  
    ...  
}
```

just like break
in C

if/unless...elsif...else

- No big mystery here, either...

```
if      ($cond1) { ... }  
elsif  ($cond2) { ... }  
else                   { ... }
```

- The **unless** keyword merely reverses the first test:

```
unless ($cond1) { ... }
```


Function calls

- Two major categories:
 - **List operators:** Take more than one argument (scalar arguments come before any list argument):

```
split /\s+/, $str, 3;  
join '::', @a, $b, @c;
```

- **Named unary operators:** Always have exactly one argument:

```
keys %somehash;
```

- Parentheses optional unless precedence requires them

Precedence in function calls

- If you use parens around function args, the simple rule is:
 - It looks like a function, so it *is* a function, and comma-precedence *doesn't* matter.
- If you *do not* use parens around function args:
 - Function is treated like a *list/unary operator*, and comma-precedence *does* matter:

```
print 1+2+3;           # prints 6
print(1+2) + 3;       # prints 3
print (1+2)+3;        # also prints 3!
print +(1+2)+3;       # very weird... but prints 6
print ((1+2)+3);      # prints 6
```

Flow control

Subroutines

What is a subroutine?

- A **subroutine** is Perl's equivalent to what C calls a “function”. We'll use the two terms interchangeably.

```
# Define a function to add two numbers:
sub add {
    my ($a, $b) = @_;    # get args
    return $a + $b;     # return result
}

# Call the function:
$sum = add(40, 2);
```

Flow control / Subroutines

my

- Use **my** to declare **lexically-scoped** "private variables" inside of subroutines (or, indeed, inside any blocks, or even at file scope):

```
sub stuff {
    my ($x, $y) = (10, 20);
    print "inside stuff, x = $x\n";    # $x = 10
}

my $x = 5;
print "before stuff, x = $x\n";      # $x = 5
stuff();
print "after stuff, x = $x\n";      # $x = 5
```

Flow control / Subroutines

my, my, my...

- Use **my** to declare multiple variables in one go:

```
my ($scalar, @array, %hash);      # declare 3 vars
```

- Careful when you initialize! It is best to declare and init arrays/hashes separately:

```
my ($one, $two, @rest) = (1, 2, 3, 4, 5);
```

```
my ($one, $two) = (1, 2);  
my (@rest) = (3, 4, 5);
```

local

- If you *really need to*, use **local** to declare **dynamically-scoped** variables inside of subroutines (or any blocks)
- Same syntax as **my**, but variables so declared will be usable by any subroutines called, *without having to be passed in*
- Think of **local** as a "pass-by-name" mechanism



Unless you know what you're doing, use **my** instead of **local**: it's faster, safer, and probably what you *really* want anyway

Flow control / Subroutines

my vs. local

```
sub inner {
    print "inner: x=$x, y=$y\n";    # $x = 200, $y = 4
}

sub outer {
    local $x = 200;
    my    $y = 400;
    print "outer: x=$x, y=$y\n";    # $x = 200, $y = 400
    inner();
}

$x = 2;
$y = 4;
outer();
print "end: x=$x, y=$y\n";        # $x = 2, $y = 4
```


Calling subroutines

- All subroutines take a LIST of scalars as an argument.
- All subroutines return a LIST of values... even when a scalar is returned it's *really* returned as a one-elem list.
- A subroutine may be called in several ways...

`add (60, 4) ;`

Pass 2 arguments

`add 60, 4 ;`

Parens optional if predeclared/imported

`add () ;`

Passes *no* args to subroutine



`&add ;`

Passes *current value of @_* to subroutine!

`add (@x, @y) ;`

Remember list interpolation! This passes in all elements of `@x` and `@y`!

Getting the arguments

Arguments are passed into subroutines via a special array called `@_` ... that is, as `$_[0]`, `$_[1]`, ...)

You can use them as-is...

```
sub add {  
    return $_[0] + $_[1];  
}
```

You can grab them via list assignment...

```
sub add {  
    my ($a, $b) = @_;  
    return $a + $b;  
}
```

...or you can use `shift` (which works on `@_` by default)...

```
sub add {  
    my $a = shift @_;  
    my $b = shift;  
    return $a + $b;  
}
```

Beware the no-argument call!

- If your subroutine *doesn't use @_ at all*, you can safely use the "no-argument form" when calling it... in which case, the current value of @_ in the caller gets passed in:

```
&dostuff;    # passes in whatever @_ happens to be now
```



However, if your subroutine is extended in the future to *examine* its argument list for optional arguments, **existing code may break!**

- Get in the habit of using `()` for safety:

```
dostuff();    # passes in zero arguments, explicitly
```

Beware call-by-reference!



Altering the elements of the `@_` array will alter the caller's arguments!

```
sub swap {  
    my $tmp;  
    $tmp = $_[0]; $_[0] = $_[1]; $_[1] = $tmp;  
}  
  
$x = 40;  
$y = 2;  
swap($x, $y);           # now $x = 2, $y = 40!
```

- Get in the habit of using **my()** to grab the values... then you're doing the much-safer call-by-value.

“Named” parameters

- Since `@_` can be of the form *(key1, val1, key2, val2, ...)*, you can create subroutines that pass “named” args!

```
sub mailheader {
    my %params = @_;
    my $from = $params{From};
    my $to   = $params{To};
    my $subj = $params{Subject} || 'None';

    print "From: $from\n"      if $from;
    print "To: $to\n";        if $to;
    print "Subject: $subj\n\n";
}

mailheader(From    => 'me@myhost.com',
           To      => 'you@yourhost.com'
           Subject => "Hi!");
```

Returning values

- A subroutine returns the value(s) of its last statement:

```
sub answer {  
    40 + 2;  
}
```

- You can explicitly **return** from a subroutine if you like (usually for error handling):

```
sub add {  
    return undef unless @_; # no args!  
    $_[0] + $_[1];  
}
```

Flow control

References

What is a reference?

- **References** in Perl are like pointers in C. They are scalars which "point" to another Perl object...

Array

Scalar

Object

Hash

Subroutine

- Used for...
 - Nesting data structures (e.g., arrays of hashes of arrays...)
 - Bypassing call-by-value

Referencing and dereferencing

- Perl's **referencing operator** is `\ ...` it's like `&` in C:

```
$arrayref   = \@array;  
$hashref    = \%hash;  
$scalarref  = \ $scalar;  
$subrref    = \&subr;
```

- Perl's **dereferencing operators** are typed:

```
@array      equals @$arrayref;  
%hash       equals %$hashref;  
$scalar     equals $$scalarref;  
&subr       equals &$subrref;
```

Dereferencing

1. Anywhere you'd put an **identifier** as part of a var or function name, you can use a scalar that holds a reference:

```
$first = $array[0];  
$first = $$arrayref[0];
```

2. Anywhere you'd put an **identifier** as part of a var or function name, you can use a BLOCK that evaluates to a reference:

```
$first = ${ getArrayRef() }[0];
```

3. When accessing elems of arrays/hash refs, you can use `->`:

```
$first = $arrayref->[0];
```

Using references to arrays

- In each of these groups, compare the first form (no references) to the others (with references):

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

```
@sorted = sort @$arrayref;
```

```
$nelems = $#array + 1;
```

```
$nelems = $#{ $arrayref } + 1;
```

```
$first = $array[0];
```

```
$first = $$arrayref[0];
```

```
$first = ${ $arrayref }[0];
```

```
$first = $arrayref->[0];
```

Creating “anonymous” arrays

- If you just want a reference to an “anonymous” array, you can create one via lists bordered with [and] ...

```
$arrayref = ['A', 'B', 'C'];  
$capitalA = $arrayref->[0];
```

- Since reference are scalars, we can embed them in lists...

```
$arrayref = [[A, B, C], D, [E, F]];  
$capitalE = $arrayref->[2]->[0];
```



That’s how we nest data structures and make multi-dimensional arrays!

Using references to hashes

- In each of these groups, compare the first form (no references) to the others (with references):

```
@keys = keys %hash;
```

```
@keys = keys %$hashref;
```

```
$name = $hash{Name};
```

```
$name = $$hashref{Name};
```

```
$name = ${ $hashref }{Name};
```

```
$name = ${ gethashref() }{Name};
```

```
$name = $hashref->{Name};
```

Creating “anonymous” hashes

- If you just want a reference to an “anonymous” hash, you can create one via lists bordered with { and } ...

```
$worf = {  
    Name      => 'Worf',  
    Species  => 'Klingon',  
    Hobbies  => ['Model ship building',  
               'Meditating',  
               'Beating people up']  
};
```

```
$favehobby = $worf->{Hobbies}->[0];
```

Folding up -> chains

- You can write this:

```
# $screw is a ref to an array of hashes of arrays...  
$value = $screw->[$i]->{Stations}->[$j];
```

like this:

```
$value = $screw->[$i]{Stations}[$j];
```

 Any -> between a right and left brace (curly or square) can be eliminated... Perl knows what's going on!

Creating “anonymous” subs

- If you just want a reference to an “anonymous” subroutine, you can create one like this...

```
$subref = sub { ordinary subroutine code here };
```

Notice the ; at the end... we need it because the **sub{...}** is really just the RHS of an assignment statement

- Use it with & like this:

```
$result = &$subref(1, 2, 3);
```

- Anonymous subs act as *closures* with respect to **my** vars

Closures

- Closures allow you to "freeze" **my** variables inside anonymous subroutines:

Return a subroutine which will add \$n to its 1st argument:

```
sub make_adder {  
    my ($n) = @_;  
    return sub { $_[0] + $n };  
}
```

Create and use a subroutine which adds 40 to its 1st argument:

```
my $add40 = make_adder(40);  
print &$add40(2), "\n";    # prints 42
```

Flow control

Error handling

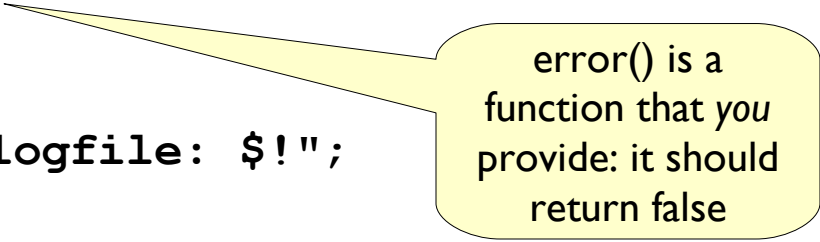
Error-handling wrappers

- It's standard in Perl for functions to return a true value on success, and a false value (0, undef, the empty list) on error
- Get in the habit of checking errors, like this:

```
# Non-fatal error, silent: return false from this function:  
open(LOG, $logfile)  
    or return undef;
```

```
# Non-fatal error, noisy: complain, and return false from this function:  
open(LOG, $logfile)  
    or return error("open $logfile: $!");
```

```
# Fatal error: exit program:  
open(LOG, $logfile)  
    or die "open $logfile: $!";
```



error() is a function that you provide: it should return false

Flow control / Error handling

\$! (\$OS_ERROR)

- In a **numeric context**, yields current value of errno. In a **string context**, yields current error string:

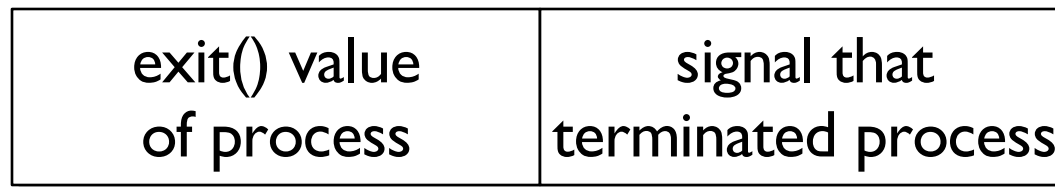
```
open(LOG, "personal.log") or  
die "errno ", int($!), ": $!";
```

- Don't depend on it being defined unless a condition arises which indicates a system error.
- **Mnemonic:** what just went bang?

Flow control / Error handling

\$? (\$CHILD_ERROR)

- The status returned by the last pipe close, backtick (``) command, or system() operator.
- Actual exit value of process is ($\$? \gg 8$).
- **Mnemonic:** similar to sh/ksh



high 8 bits

low 8 bits

\$?

Flow control / Error handling

warn(*LIST*) / carp(*LIST*)

- The **warn** function prints *LIST* to STDERR as a warning message. It is the standard way to issue a warning.

```
warn "She canna take the strain, Jim!"  
if ($warp > 9);
```

- If you "**use Carp**", you can use the alternative, **carp**:

```
carp "I canna change the laws of physics, Jim!"  
if ($restart && ($intermix_temp < 1200));
```

- You can replace **warn**'s output handler with your own:

```
$SIG{__WARN__} = sub {  
    print STDERR "AHOY! ", $_[0];  
};
```

die(LIST) / croak(LIST) / confess(LIST)

- The **die** function prints LIST to STDERR (like **warn**), and exits. It is the standard way to deal with a fatal error/exception.

```
die "warp core breach: field collapsed"  
    if ($containment_field_strength < 0.15);
```

- If you "use **Carp**", you can also use **croak** or **confess**:

```
croak "auto-destruct triggered";  
    if ($autodestruct_countdown == 0);
```

- You can hook into **die**, and do stuff just before a death:

```
$_SIG{__DIE__} = sub {  
    abandon_ship("EMERGENCY! $_[0]");  
};
```

Catching fatal errors

- Sometimes, you will have to call a function that can cause the program to die...
 - It might call die() in a panic response to some errors
 - It might contain a patch of bad code that coredumps

...but you want to *prevent it* from actually killing the program.
Wrap the call in an **eval**, and check **\$@** to see if a fatal error was caught:

```
eval { risky_business() };  
$@ and warn "fatal error caught: $@";
```


Flow control / Error handling

`$@` (`$EVAL_ERROR`)

- The Perl syntax/execution error from the last **`eval()`** command.
- If a null string, indicates that last eval parsed and executed okay (although *non-fatal* errors may have occurred!).
- If *not* null, it contains the fatal error message:

```
eval '$x = '; warn $@ if $@;
```


- **Mnemonic:** where was the syntax/fatal error "at"?

Detecting *possible* problems

- The recommended way to invoke Perl in your scripts is:

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

-W Warns about identifiers mentioned only once, scalars that are used before being set, redefined subroutines, attempts to use undefined filehandles, numeric use of things that don't look like numbers... etc... etc...

 -T Turns on "taint checking". Perl will detect if you attempt to perform certain unsafe operations, like running a system command (e.g., "rm") where part of the command line came from outside this script (e.g., from user input).