Perl®

Notes for Professionals

100+ pages

of professional hints and tricks

Contents

About

Chapter 1: Getting started with Perl Language

Version	Release Notes	Release Date
1.000		1987-12-18
2.000		1988-06-05
3.000		1989-10-18
4.000		1991-03-21
5.000		1994-10-17
5.001		1995-05-13
5.002		1996-02-29
5.003		1996-06-25
5.004	perl5004delta	1997-05-15
5.005	perl5005delta	1998-07-22
5.6.0	perl56delta	2000-03-22
5.8.0	perl58delta	2002-07-18
5.8.8	perl581delta, perl582delta, perl583delta, perl584delta, perl585delta, perl586delta, perl587delta, perl588delta	2006-02-01
5.10.0	perl5100delta	2007-12-18
5.12.0	perl5120delta	2010-04-12
5.14.0	perl5140delta	2011-05-14
5.16.0	perl5160delta	2012-05-20
5.18.0	perl5180delta	2013-05-18
5.20.0	perl5200delta	2014-05-27
5.22.0	perl5220delta	2015-06-01
5.24.0	perl5240delta	2016-05-09
5.26.0	perl5260delta	2017-05-30

Section 1.1: Getting started with Perl

Perl tries to do what you mean:

```
print "Hello World\n";
```

The two tricky bits are the semicolon at the end of the line and the `\n`, which adds a newline (line feed). If you have a relatively new version of perl, you can use `say` instead of `print` to have the carriage return added automatically:

Version ≥ 5.10.0

```
use feature 'say';
say "Hello World";
```

The say feature is also enabled automatically with a `use v5.10` (or higher) declaration:

```
use v5.10;
say "Hello World";
```

It's pretty common to just use perl on the command line using the `-e` option:

```
$ perl -e 'print "Hello World\n"'
Hello World
```

Adding the `-l` option is one way to print newlines automatically:

```
$ perl -le 'print "Hello World"'
Hello World
```

Version ≥ 5.10.0

If you want to enable new features, use the `-E` option instead:

```
$ perl -E 'say "Hello World"'
Hello World
```

You can also, of course, save the script in a file. Just remove the `-e` command line option and use the filename of the script: `perl script.pl`. For programs longer than a line, it's wise to turn on a couple of options:

```
use strict;
use warnings;

print "Hello World\n";
```

There's no real disadvantage other than making the code slightly longer. In exchange, the strict pragma prevents you from using code that is potentially unsafe and warnings notifies you of many common errors.

Notice the line-ending semicolon is optional for the last line, but is a good idea in case you later add to the end of your code.

For more options how to run Perl, see perlrun or type `perldoc perlrun` at a command prompt. For a more detailed introduction to Perl, see perlintro or type `perldoc perlintro` at a command prompt. For a quirky interactive tutorial, Try Perl.

Chapter 2: Packages and modules

Section 2.1: Using a module

```
use Cwd;
```

This will import the `Cwd` module at compile time and import its default symbols, i.e. make some of the module's variables and functions available to the code using it. (See also: `perldoc -f use`.)

Generally this is will do the right thing. Sometimes, however, you will want to control which symbols are imported. Add a list of symbols after the module name to export:

```
use Cwd 'abs_path';
```

If you do this, only the symbols you specify will be imported (ie, the default set will not be imported).

When importing multiple symbols, it is idiomatic to use the `qw()` list-building construct:

```
use Cwd qw(abs_path realpath);
```

Some modules export a subset of their symbols, but can be told to export everything with `:all`:

```
use Benchmark ':all';
```

(Note that not all modules recognize or use the `:all` tag).

Section 2.2: Using a module inside a directory

```
use lib 'includes';
use MySuperCoolModule;
```

`use lib 'includes';` adds the relative directory `includes/` as another module search path in `@INC`. So assume that you have a module file `MySyperCoolModule.pm` inside `includes/`, which contains:

```
package MySuperCoolModule;
```

If you want, you can group as many modules of your own inside a single directory and make them findable with one `use lib` statement.

At this point, using the subroutines in the module will require prefixing the subroutine name with the package name:

```
MySuperCoolModule::SuperCoolSub_1("Super Cool String");
```

To be able to use the subroutines without the prefix, you need to export the subroutine names so that they are recognised by the program calling them. Exporting can be set up to be automatic, thus:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

Then in the file that **use**s the module, those subroutines will be automatically available:

```
use MySuperCoolModule;
SuperCoolSub_1("Super Cool String");
```

Or you can set up the module to conditionally export subroutines, thus:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT_OK = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

In which case, you need to explicitly request the desired subroutines to be exported in the script that `use`s the module:

```
use MySuperCoolModule 'SuperCoolSub_1';
SuperCoolSub_1("Super Cool String");
```

Section 2.3: Loading a module at runtime

```
require Exporter;
```

This will ensure that the `Exporter` module is loaded at runtime if it hasn't already been imported. (See also: `perldoc -f require`.)

N.B.: Most users should `use` modules rather than `require` them. Unlike `use`, `require` does not call the module's import method and is executed at runtime, not during the compile.

This way of loading modules is useful if you can't decide what modules you need before runtime, such as with a plugin system:

```
package My::Module;
my @plugins = qw( One Two );
foreach my $plugin (@plugins) {
    my $module = __PACKAGE__ . "::Plugins::$plugin";
    $module =~ s!::!/!g;
    require "$module.pm";
}
```

This would try to load `My::Package::Plugins::One` and `My::Package::Plugins::Two`. `@plugins` should of course come from some user input or a config file for this to make sense. Note the substitution operator `s!::!/!g` that replaces each pair of colons with a slash. This is because you can load modules using the familiar module name syntax from `use` only if the module name is a bareword. If you pass a string or a variable, it must contain a file name.

Section 2.4: CPAN.pm

`CPAN.pm` is a Perl module which allows to query and install modules from CPAN sites.

It supports interactive mode invoked with

```
cpan
```

or

```
perl -MCPAN -e shell
```

Querying modules

By name:

```
cpan> m MooseX::YAML
```

By a regex against module name:

```
cpan> m /^XML::/
```

Note: to enable a pager or redirecting to a file use `|` or `>` shell redirection (spaces are mandatory around the `|` and `>`), e.g.: `m /^XML::/ | less`.

By distribution:

```
cpan> d LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Installing modules

By name:

```
cpan> install MooseX::YAML
```

By distribution:

```
cpan> install LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Section 2.5: List all installed modules

From command line:

```
cpan -l
```

From a Perl script:

```
use ExtUtils::Installed;
my $inst = ExtUtils::Installed->new();
my @modules = $inst->modules();
```

Section 2.6: Executing the contents of another file

```
do './config.pl';
```

This will read in the contents of the config.pl file and execute it. (See also: `perldoc -f do`.)

N.B.: Avoid `do` unless golfing or something as there is no error checking. For including library modules, use `require` or **use**.

Chapter 3: True and false

Section 3.1: List of true and false values

```
use feature qw( say );

# Numbers are true if they're not equal to 0.
say 0             ? 'true' : 'false'; # false
say 1             ? 'true' : 'false'; # true
say 2             ? 'true' : 'false'; # true
say -1            ? 'true' : 'false'; # true
say 1-1           ? 'true' : 'false'; # false
say 0e7           ? 'true' : 'false'; # false
say -0.00         ? 'true' : 'false'; # false

# Strings are true if they're not empty.
say 'a'           ? 'true' : 'false'; # true
say 'false'       ? 'true' : 'false'; # true
say ''            ? 'true' : 'false'; # false

# Even if a string would be treated as 0 in numeric context, it's true if nonempty.
# The only exception is the string "0", which is false.
# To force numeric context add 0 to the string
say '0'           ? 'true' : 'false'; # false
say '0.0'         ? 'true' : 'false'; # true
say '0e0'         ? 'true' : 'false'; # true
say '0 but true'  ? 'true' : 'false'; # true
say '0 whargarbl' ? 'true' : 'false'; # true
say 0+'0 argarbl' ? 'true' : 'false'; # false

# Things that become numbers in scalar context are treated as numbers.
my @c = ();
my @d = (0);
say @c            ? 'true' : 'false'; # false
say @d            ? 'true' : 'false'; # true

# Anything undefined is false.
say undef         ? 'true' : 'false'; # false

# References are always true, even if they point at something false
my @c = ();
my $d = 0;
say \@c            ? 'true' : 'false'; # true
say \$d            ? 'true' : 'false'; # true
say \0             ? 'true' : 'false'; # true
say \''            ? 'true' : 'false'; # true
```

Chapter 4: Subroutines

Section 4.1: Creating subroutines

Subroutines are created by using the keyword `sub` followed by an identifier and a code block enclosed in braces.

You can access the arguments by using the special variable `@_`, which contains all arguments as an array.

```
sub function_name {
    my ($arg1, $arg2, @more_args) = @_;
    # ...
}
```

Since the function `shift` defaults to shifting `@_` when used inside a subroutine, it's a common pattern to extract the arguments sequentially into local variables at the beginning of a subroutine:

```
sub function_name {
    my $arg1 = shift;
    my $arg2 = shift;
    my @more_args = @_;
    # ...
}

# emulate named parameters (instead of positional)
sub function_name {
    my %args = (arg1 => 'default', @_);
    my $arg1 = delete $args{arg1};
    my $arg2 = delete $args{arg2};
    # ...
}

sub {
    my $arg1 = shift;
    # ...
}->($arg);
```

Version ≥ 5.20.0

Alternatively, the experimental feature `"signatures"` can be used to unpack parameters, which are passed by value (*not* by reference).

```
use feature "signatures";

sub function_name($arg1, $arg2, @more_args) {
    # ...
}
```

Default values can be used for the parameters.

```
use feature "signatures";

sub function_name($arg1=1, $arg2=2) {
    # ...
}
```

You can use any expression to give a default value to a parameter – including other parameters.

```
sub function_name($arg1=1, $arg2=$arg1+1) {
```

```
    # ...
}
```

Note that you can't reference parameters which are defined after the current parameter – hence the following code doesn't work quite as expected.

```
sub function_name($arg1=$arg2, $arg2=1) {
    print $arg1;  # => <nothing>
    print $arg2;  # => 1
}
```

Section 4.2: Subroutines

Subroutines hold code. Unless specified otherwise, they are globally defined.

```
# Functions do not (have to) specify their argument list
sub returns_one {
  # Functions return the value of the last expression by default
  # The return keyword here is unnecessary, but helps readability.
  return 1;
}

# Its arguments are available in @_, however
sub sum {
  my $ret = 0;
  for my $value (@_) {
    $ret += $value
  }
  return $ret;
}

# Perl makes an effort to make parens around argument list optional
say sum 1..3;     # 6

# If you treat functions as variables, the & sigil is mandatory.
say defined &sum; # 1
```

Some builtins such as `print` or `say` are keywords, not functions, so e.g. `&say` is undefined. It also does mean that you can define them, but you will have to specify the package name to actually call them

```
# This defines the function under the default package, 'main'
sub say {
  # This is instead the say keyword
  say "I say, @_";
}

# ...so you can call it like this:
main::say('wow'); # I say, wow.
```

Version ≥ 5.18.0

Since Perl 5.18, you can also have non-global functions:

```
use feature 'lexical_subs';
my $value;
{
  # Nasty code ahead
  my sub prod {
    my $ret = 1;
    $ret *= $_ for @_;
```

```
    $ret;
  }
  $value = prod 1..6; # 720
  say defined &prod; # 1
}
say defined &prod; # 0
```

Version ≥ 5.20.0

Since 5.20, you can also have named parameters.

```
use feature 'signatures';
sub greet($name) {
  say "Hello, $name";
}
```

This should *not* be confused with prototypes, a facility Perl has to let you define functions that behave like built-ins. Function prototypes must be visible at compile time and its effects can be ignored by specifying the `&` sigil. Prototypes are generally considered to be an advanced feature that is best used with great care.

```
# This prototype makes it a compilation error to call this function with anything
# that isn't an array. Additionally, arrays are automatically turned into arrayrefs
sub receives_arrayrefs(\@\@) {
   my $x = shift;
   my $y = shift;
}

my @a = (1..3);
my @b = (1..4);
receives_arrayrefs(@a, @b);    # okay,    $x = \@a, $y = \@b, @_ = ();
receives_arrayrefs(\@a, \@b);  # compilation error, "Type … must be array …"
BEGIN { receives_arrayrefs(\@a, \@b); }

# Specify the sigil to ignore the prototypes.
&receives_arrayrefs(\@a, \@b); # okay,    $x = \@a, $y = \@b, @_ = ();
&receives_arrayrefs(@a, @b);   # ok, but $x = 1,   $y = 2,   @_ = (3,1,2,3,4);
```

Section 4.3: Subroutine arguments are passed by reference (except those in signatures)

Subroutine arguments in Perl are passed by reference, unless they are in the signature. This means that the members of the `@_` array inside the sub are just *aliases* to the actual arguments. In the following example, `$text` in the main program is left modified after the subroutine call because `$_[0]` inside the sub is actually just a different name for the same variable. The second invocation throws an error because a string literal is not a variable and therefore can't be modified.

```
use feature 'say';

sub edit {
    $_[0] =~ s/world/sub/;
}

my $text = "Hello, world!";
edit($text);
say $text;      # Hello, sub!

edit("Hello, world!"); # Error: Modification of a read-only value attempted
```

To avoid clobbering your caller's variables it is therefore important to copy `@_` to locally scoped variables (`my ...`) as

described under "Creating subroutines".

Chapter 5: Debug Output

Section 5.1: Dumping with Style

Sometimes Data::Dumper is not enough. Got a Moose object you want to inspect? Huge numbers of the same structure? Want stuff sorted? Colored? Data::Printer is your friend.

```
use Data::Printer;

p $data_structure;
```

Data::Printer writes to STDERR, like `warn`. That makes it easier to find the output. By default, it sorts hash keys and looks at objects.

```
use Data::Printer;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
p $ua;
```

It will look at all the methods of the object, and also list the internals.

```
LWP::UserAgent  {
    Parents       LWP::MemberMixin
    public methods (45) : add_handler, agent, clone, conn_cache, cookie_jar, credentials,
default_header, default_headers, delete, env_proxy, from, get, get_basic_credentials,
get_my_handler, handlers, head, is_online, is_protocol_supported, local_address, max_redirect,
max_size, mirror, new, no_proxy, parse_head, post, prepare_request, progress, protocols_allowed,
protocols_forbidden, proxy, put, redirect_ok, remove_handler, request, requests_redirectable,
run_handlers, send_request, set_my_handler, show_progress, simple_request, ssl_opts, timeout,
use_alarm, use_eval
    private methods (4) : _agent, _need_proxy, _new_response, _process_colonic_headers
    internals: {
        def_headers             HTTP::Headers,
        handlers                {
            response_header   HTTP::Config
        },
        local_address           undef,
        max_redirect            7,
        max_size                undef,
        no_proxy                [],
        protocols_allowed       undef,
        protocols_forbidden     undef,
        proxy                   {},
        requests_redirectable   [
            [0] "GET",
            [1] "HEAD"
        ],
        show_progress           undef,
        ssl_opts                {
            verify_hostname   1
        },
        timeout                 180,
        use_eval                1
```

```
    }
}
```

You can configure it further, so it serializes certain objects in a certain way, or to include objects up to an arbitrary depth. The full configuration is available in the documentation.

Unfortunately Data::Printer does not ship with Perl, so you need to install it from CPAN or through your package management system.

Section 5.2: Dumping data-structures

```
use Data::Dumper;

my $data_structure = { foo => 'bar' };
print Dumper $data_structure;
```

Using Data::Dumper is an easy way to look at data structures or variable content at run time. It ships with Perl and you can load it easily. The `Dumper` function returns the data structure serialized in a way that looks like Perl code.

```
$VAR1 = {
        'foo' => 'bar',
}
```

That makes it very useful to quickly look at some values in your code. It's one of the most handy tools you have in your arsenal. Read the full documentation on metacpan.

Section 5.3: Data::Show

The function `show` is automatically exported when `use Data::Show;` is executed. This function takes a variable as its sole argument and it outputs:

1. the name of the variable
2. the contents of that variable (in a readable format)
3. the line of the file that `show` is run from
4. the file `show` is run from

Assuming that the following is code from the file `example.pl`:

```
use strict;
use warnings;
use Data::Show;

my @array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

my %hash  = ( foo => 1, bar => { baz => 10, qux => 20 } );

my $href = \%hash;

show @array;
show %hash;
show $href;
```

`perl example.pl` gives the following output:

```
======(  @array  )=======================[ 'example.pl', line 11 ]======
```

```
    [1 .. 10]

======(  %hash  )=======================[ 'example.pl', line 12 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }

======(  $href  )=======================[ 'example.pl', line 13 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }
```

See the documentation for `Data::Show`.

Section 5.4: Dumping array list

```
my @data_array = (123, 456, 789, 'poi', 'uyt', "rew", "qas");
print Dumper @data_array;
```

Using Data::Dumper gives an easy access to fetch list values. The `Dumper` returns the list values serialized in a way that looks like Perl code.

Output:

```
$VAR1 = 123;
$VAR2 = 456;
$VAR3 = 789;
$VAR4 = 'poi';
$VAR5 = 'uyt';
$VAR6 = 'rew';
$VAR7 = 'qas';
```

As suggested by user @dgw When dumping arrays or hashes it is better to use an array reference or a hash reference, those will be shown better fitting to the input.

```
$ref_data = [23,45,67,'mnb','vcx'];
print Dumper $ref_data;
```

Output:

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

You can also reference the array when printing.

```
my @data_array = (23,45,67,'mnb','vcx');
print Dumper \@data_array;
```

Output:

```
$VAR1 = [
          23,
```

```
        45,
        67,
        'mnb',
        'vcx'
    ];
```

Chapter 6: Variables

Section 6.1: Scalars

Scalars are Perl's most basic data type. They're marked with the sigil $ and hold a single value of one of three types:

- **a number** (`3`, `42`, `3.141`, etc.)
- **a string** (`'hi'`, `"abc"`, etc.)
- **a reference** to a variable (see other examples).

```
my $integer = 3;                    # number
my $string = "Hello World";         # string
my $reference = \$string;           # reference to $string
```

Perl converts between numbers and strings on the fly, based on what a particular operator expects.

```
my $number = '41';                  # string '41'
my $meaning = $number + 1;          # number  42
my $sadness = '20 apples';          # string '20 apples'
my $danger = $sadness * 2;          # number '40', raises warning
```

When converting a string into a number, Perl takes as many digits from the front of a string as it can – hence why `20 apples` is converted into `20` in the last line.

Based on whether you want to treat the contents of a scalar as a string or a number, you need to use different operators. Do not mix them.

```
# String comparison                 # Number comparison
'Potato' eq 'Potato';               42 == 42;
'Potato' ne 'Pomato';               42 != 24;
'Camel'  lt 'Potato';               41 < 42;
'Zombie' gt 'Potato';               43 > 42;

# String concatenation              # Number summation
'Banana' . 'phone';                 23 + 19;

# String repetition                 # Number multiplication
'nan' x 3;                          6 * 7;
```

Attempting to use string operations on numbers will not raise warnings; attempting to use number operations on non-numeric strings will. Do be aware that some non-digit strings such as `'inf'`, `'nan'`, `'0 but true'` count as numbers.

Section 6.2: Array References

Array References are scalars ($) which refer to Arrays.

```
my @array = ("Hello"); # Creating array, assigning value from a list
my $array_reference = \@array;
```

These can be created more short-hand as follows:

```
my $other_array_reference = ["Hello"];
```

Modifying / Using array references require dereferencing them first.

```
my @contents = @{ $array_reference };              # Prefix notation
my @contents = @$array_reference;                  # Braces can be left out
```

Version ≥ 5.24.0

New postfix dereference syntax, available by default from v5.24

```
use v5.24;
my @contents = $array_reference->@*; # New postfix notation
```

When accessing an arrayref's contents by index you can use the `->` syntactical sugar.

```
my @array = qw(one two three);      my $arrayref = [ qw(one two three) ]
my $one = $array[0];                my $one = $arrayref->[0];
```

Unlike arrays, arrayrefs can be nested:

```
my @array = ( (1, 0), (0, 1) )  # ONE array of FOUR elements: (1, 0, 0, 1)
my @matrix = ( [1, 0], [0, 1] ) # an array of two arrayrefs
my $matrix = [ [0, 1], [1, 0] ] # an arrayref of arrayrefs
# There is no namespace conflict between scalars, arrays and hashes
# so @matrix and $matrix _both_ exist at this point and hold different values.

my @diagonal_1 = ($matrix[0]->[1], $matrix[1]->[0])     # uses @matrix
my @diagonal_2 = ($matrix->[0]->[1], $matrix->[1]->[0]) # uses $matrix
# Since chained []- and {}-access can only happen on references, you can
# omit some of those arrows.
my $corner_1 = $matrix[0][1];   # uses @matrix;
my $corner_2 = $matrix->[0][1]; # uses $matrix;
```

When used as Boolean, references are always true.

Section 6.3: Scalar References

A **reference** is a scalar variable (one prefixed by $) which “refers to” some other data.

```
my $value     = "Hello";
my $reference = \$value;
print $value;     # => Hello
print $reference; # => SCALAR(0x2683310)
```

To get the referred-to data, you **de-reference** it.

```
say ${$reference};                 # Explicit prefix syntax
say $$reference;                   # The braces can be left out (confusing)
```

Version ≥ 5.24.0

New postfix dereference syntax, available by default from v5.24

```
use v5.24;
say $reference->$*; # New postfix notation
```

This "de-referenced value" can then be changed like it was the original variable.

```
${$reference} =~ s/Hello/World/;
print ${$reference};  # => World
print $value;         # => World
```

A reference is always **truthy** – even if the value it refers to is falsy (like `0` or `""`).

You may want a Scalar Reference If:

- You want to pass a string to a function, and have it modify that string for you without it being a return value.
- You wish to explicitly avoid Perl implicitly copying the contents of a large string at some point in your function passing (especially relevant on older Perls without copy-on-write strings)
- You wish to disambiguate string-like values with specific meaning, from strings that convey content, for example:
    - Disambiguate a file name from file content
    - Disambiguate returned content from a returned error string
- You wish to implement a lightweight inside out object model, where objects handed to calling code don't carry user visible metadata:

```
our %objects;
my $next_id = 0;
sub new {
   my $object_id = $next_id++;
   $objects{ $object_id } = { ... }; # Assign data for object
   my $ref = \$object_id;
   return bless( $ref, "MyClass" );
}
```

Section 6.4: Arrays

Arrays store an ordered sequence of values. You can access the contents by index, or iterate over them. The values will stay in the order you filled them in.

```
my @numbers_to_ten = (1,2,3,4,5,6,7,8,9,10); # More conveniently: (1..10)
my @chars_of_hello = ('h','e','l','l','o');
my @word_list = ('Hello','World');

# Note the sigil: access an @array item with $array[index]
my $second_char_of_hello = $chars_of_hello[1]; # 'e'

# Use negative indices to count from the end (with -1 being last)
my $last_char_of_hello = $chars_of_hello[-1];

# Assign an array to a scalar to get the length of the array
my $length_of_array = @chars_of_hello; # 5

# You can use $# to get the last index of an array, and confuse Stack Overflow
my $last_index_of_array = $#chars_of_hello; # 4

# You can also access multiple elements of an array at the same time
# This is called "array slice"
# Since this returns multiple values, the sigil to use here on the RHS is @
my @some_chars_of_hello = @chars_of_hello[1..3]; # ('H', 'e', 'l')
my @out_of_order_chars = @chars_of_hello[1,4,2]; # ('e', 'o', 'l')

# In Python you can say array[1:-1] to get all elements but first and last
# Not so in Perl: (1..-1) is an empty list. Use $# instead
```

```
my @empty_list = @chars_of_hello[1..-1];                          # ()
my @inner_chars_of_hello = @chars_of_hello[1..$#chars_of_hello-1]; # ('e','l','l')

# Access beyond the end of the array yields undef, not an error
my $undef = $chars_of_hello[6]; # undef
```

Arrays are mutable:

```
use utf8; # necessary because this snippet is utf-8
$chars_of_hello[1] = 'u';              #     ('h','u','l','l','o')
push @chars_of_hello, ('!', '!');      #     ('h','u','l','l','o','!','!')
pop @chars_of_hello;                   #     ('h','u','l','l','o','!')
shift @chars_of_hello;                 #         ('u','l','l','o','!')
unshift @chars_of_hello, ('¡', 'H');   # ('¡','H','u','l','l','o','!')
@chars_of_hello[2..5] = ('O','L','A'); # ('¡','H','O','L','A',undef,'!') whoops!
delete $chars_of_hello[-2];            # ('¡','H','O','L','A',      '!')

# Setting elements beyond the end of an array does not result in an error
# The array is extended with undef's as necessary. This is "autovivification."
my @array;           # ()
my @array[3] = 'x';  # (undef, undef, undef, 'x')
```

Finally, you can loop over the contents of an array:

```
use v5.10; # necessary for 'say'
for my $number (@numbers_to_ten) {
  say $number ** 2;
}
```

When used as booleans, arrays are true if they are not empty.

Section 6.5: Typeglobs, typeglob refs, filehandles and constants

A typeglob `*foo` holds references to the contents of *global* variables with that name: `$foo`, `@foo`, `$foo`, `&foo`, etc. You can access it like an hash and assign to manipulate the symbol tables directly (evil!).

```
use v5.10; # necessary for say
our $foo = "foo";
our $bar;
say ref *foo{SCALAR};     # SCALAR
say ${ *foo{SCALAR} };    # bar
*bar = *foo;
say $bar;                 # bar
$bar = 'egg';
say $foo;                 # egg
```

Typeglobs are more commonly handled when dealing with files. `open`, for example, produces a reference to a typeglob when asked to create a non-global filehandle:

```
use v5.10; # necessary for say
open(my $log, '> utf-8', '/tmp/log') or die $!; # open for writing with encoding
say $log 'Log opened';

# You can dereference this globref, but it's not very useful.
say ref $log;                   # GLOB
say (*{$log}->{IO} // 'undef'); # undef
```

```
close $log or die $!;
```

Typeglobs can also be used to make global read-only variables, though **use constant** is in broader use.

```
# Global constant creation
*TRUE = \('1');
our $TRUE;
say $TRUE;  # 1
$TRUE = ''; # dies, "Modification of a read-only value attempted"

# use constant instead defines a parameterless function, therefore it's not global,
# can be used without sigils, can be imported, but does not interpolate easily.
use constant (FALSE => 0);
say FALSE;        # 0
say &FALSE;       # 0
say "${\FALSE}";  # 0 (ugh)
say *FALSE{CODE}; # CODE(0xMA1DBABE)

# Of course, neither is truly constant when you can manipulate the symbol table...
*TRUE = \('');
use constant (EVIL => 1);
*FALSE = *EVIL;
```

Section 6.6: Sigils

Perl has a number of sigils:

```
$scalar = 1; # individual value
@array = ( 1, 2, 3, 4, 5 ); # sequence of values
%hash = ('it', 'ciao', 'en', 'hello', 'fr', 'salut'); # unordered key-value pairs
&function('arguments'); # subroutine
*typeglob; # symbol table entry
```

These look like sigils, but aren't:

```
\@array; # \ returns the reference of what's on the right (so, a reference to @array)
$#array; # this is the index of the last element of @array
```

You can use braces after the sigil if you should be so inclined. Occasionally, this improves readability.

```
say ${value} = 5;
```

While you use different sigils to define variables of different types, the same variable can be accessed in different ways based on what sigils you use.

```
%hash;            # we use % because we are looking at an entire hash
$hash{it};        # we want a single value, however, that's singular, so we use $
$array[0];        # likewise for an array. notice the change in brackets.
@array[0,3];      # we want multiple values of an array, so we instead use @
@hash{'it','en'}; # similarly for hashes (this gives the values: 'ciao', 'hello')
%hash{'it','fr'}; # we want an hash with just some of the keys, so we use %
                  # (this gives key-value pairs: 'it', 'ciao', 'fr', 'salut')
```

This is especially true of references. In order to use a referenced value you can combine sigils together.

```
my @array = 1..5;                    # This is an array
my $reference_to_an_array = \@array; # A reference to an array is a singular value
```

```
push @array, 6;                       # push expects an array
push @$reference_to_an_array, 7;      # the @ sigil means what's on the right is an array
                                      # and what's on the right is $reference_to_an_array
                                      # hence: first a @, then a $
```

Here's a perhaps less confusing way to think about it. As we saw earlier, you can use braces to wrap what's on the right of a sigil. So you can think of `@{}` as something that takes an array reference and gives you the referenced array.

```
# pop does not like array references
pop $reference_to_an_array; # ERROR in Perl 5.20+
# but if we use @{}, then...
pop @{ $reference_to_an_array }; # this works!
```

As it turns out, `@{}` actually accepts an expression:

```
my $values = undef;
say pop @{ $values };       # ERROR: can't use undef as an array reference
say pop @{ $values // [5] } # undef // [5] gives [5], so this prints 5
```

...and the same trick works for other sigils, too.

```
# This is not an example of good Perl. It is merely a demonstration of this language feature
my $hashref = undef;
for my $key ( %{ $hashref // {} } ) {
  "This doesn't crash";
}
```

...but if the "argument" to a sigil is simple, you can leave the braces away.

```
say $$scalar_reference;
say pop @$array_reference;
for keys (%$hash_reference) { ... };
```

Things can get excessively extravagant. This works, but please Perl responsibly.

```
my %hash = (it => 'ciao', en => 'hi', fr => 'salut');
my $reference = \%hash;
my $reference_to_a_reference = \$reference;

my $italian = $hash{it};                              # Direct access
my @greets = @$reference{'it', 'en'};                 # Dereference, then access as array
my %subhash = %$$reference_to_a_reference{'en', 'fr'} # Dereference ×2 then access as hash
```

For most normal use, you can just use subroutine names without a sigil. (Variables without a sigil are typically called "barewords".) The `&` sigil is only useful in a limited number of cases.

- Making a reference to a subroutine:

  ```
  sub many_bars { 'bar' x $_[0] }
  my $reference = \&many_bars;
  say $reference->(3); # barbarbar
  ```

- Calling a function ignoring its prototype.

- Combined with goto, as a slightly weird function call that has the current call frame replaced with the caller. Think the linux `exec()` API call, but for functions.

Section 6.7: Hash References

Hash references are scalars which contain a pointer to the memory location containing the data of a hash. Because the scalar points directly to the hash itself, when it is passed to a subroutine, changes made to the hash are not local to the subroutine as with a regular hash, but instead are global.

First, let's examine what happens when you pass a normal hash to a subroutine and modify it within there:

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my %hash = @_;

    $hash{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper(\%hash);

    return;
}

my %example_hash = (
    old_value   => 1,
);

modify(%example_hash);

print Dumper("After exiting the subroutine");
print Dumper(\%example_hash);
```

Which results in:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'old_value' => 1
        };
```

Notice that after we exit the subroutine, the hash remains unaltered; all changes to it were local to the modify subroutine, because we passed a copy of the hash, not the hash itself.

In comparison, when you pass a hashref, you are passing the address to the original hash, so any changes made within the subroutine will be made to the original hash:

```
use strict;
use warnings;
use Data::Dumper;
```

```
sub modify
{
    my $hashref = shift;

    # De-reference the hash to add a new value
    $hashref->{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper($hashref);

    return;
}

# Create a hashref
my $example_ref = {
    old_value   => 1,
};

# Pass a hashref to a subroutine
modify($example_ref);

print Dumper("After exiting the subroutine");
print Dumper($example_ref);
```

This will result in:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
```

Section 6.8: Hashes

Hashes can be understood as lookup-tables. You can access its contents by specifiying a key for each of them. Keys must be strings. If they're not, they will be converted to strings.

If you give the hash simply a known key, it will serve you its value.

```
# Elements are in (key, value, key, value) sequence
my %inhabitants_of = ("London", 8674000, "Paris", 2244000);

# You can save some typing and gain in clarity by using the "fat comma"
# syntactical sugar. It behaves like a comma and quotes what's on the left.
my %translations_of_hello = (spanish => 'Hola', german => 'Hallo', swedish => 'Hej');
```

In the following example, note the brackets and sigil: you access an element of `%hash` using `$hash{key}` because the *value* you want is a scalar. Some consider it good practice to quote the key while others find this style visually noisy. Quoting is only required for keys that could be mistaken for expressions like `$hash{'some-key'}`

```
my $greeting = $translations_of_hello{'spanish'};
```

While Perl by default will try to use barewords as strings, `+` modifier can also be used to indicate to Perl that key

should not be interpolated but executed with result of execution being used as a key:

```
my %employee = ( name => 'John Doe', shift => 'night' );
# this example will print 'night'
print $employee{shift};

# but this one will execute [shift][1], extracting first element from @_,
# and use result as a key
print $employee{+shift};
```

Like with arrays, you can access multiple hash elements at the same time. This is called a *hash slice*. The resulting value is a list, so use the `@` sigil:

```
my @words = @translations_of_hello{'spanish', 'german'}; # ('Hola', 'Hallo')
```

Iterate over the keys of an hash with `keys` `keys` will return items in a random order. Combine with `sort` if you wish.

```
for my $lang (sort keys %translations_of_hello) {
  say $translations_of_hello{$lang};
}
```

If you do not actually need the keys like in the previous example, `values` returns the hash's values directly:

```
for my $translation (values %translations_of_hello) {
  say $translation;
}
```

You can also use a while loop with `each` to iterate over the hash. This way, you will get both the key and the value at the same time, without a separate value lookup. Its use is however discouraged, as `each` can break in mistifying ways.

```
# DISCOURAGED
while (my ($lang, $translation) = each %translations_of_hello) {
  say $translation;
}
```

Access to unset elements returns undef, not an error:

```
my $italian = $translations_of_hello{'italian'}; # undef
```

`map` and list flattening can be used to create hashes out of arrays. This is a popular way to create a 'set' of values, e.g. to quickly check whether a value is in `@elems`. This operation usually takes O(n) time (i.e. proportional to the number of elements) but can be done in constant time (O(1)) by turning the list into a hash:

```
@elems = qw(x y x z t);
my %set = map { $_ => 1 } @elems;   # (x, 1, y, 1, t, 1)
my $y_membership = $set{'y'};       # 1
my $w_membership = $set{'w'};       # undef
```

This requires some explanation. The contents of `@elems` get read into a list, which is processed by `map`. `map` accepts a code block that gets called for each value of its input list; the value of the element is available for use in `$_`. Our code block returns *two* list elements for each input element: `$_`, the input element, and `1`, just some value. Once you account for list flattening, the outcome is that `map { $_ => 1 } @elems` turns `qw(x y x z t)` into `(x => 1, y => 1, x => 1, z => 1, t => 1)`.

As those elements get assigned into the hash, odd elements become hash keys and even elements become hash values. When a key is specified multiple times in a list to be assigned to a hash, the *last* value wins. This effectively discards duplicates.

A faster way to turn a list into a hash uses assignment to a hash slice. It uses the `x` operator to multiply the single-element list `(1)` by the size of `@elems`, so there is a `1` value for each of the keys in the slice on the left hand side:

```
@elems = qw(x y x z t);
my %set;
@set{@elems} = (1) x @elems;
```

The following application of hashes also exploits the fact that hashes and lists can often be used interchangeably to implement named function args:

```
sub hash_args {
  my %args = @_;
  my %defaults = (foo => 1, bar => 0);
  my %overrides = (__unsafe => 0);
  my %settings = (%defaults, %args, %overrides);
}

# This function can then be called like this:
hash_args(foo => 5, bar => 3); # (foo => 5, bar => 3, __unsafe ==> 0)
hash_args();                   # (foo => 1, bar => 0, __unsafe ==> 0)
hash_args(__unsafe => 1)       # (foo => 1, bar => 0, __unsafe ==> 0)
```

When used as booleans, hashes are true if they are not empty.

Chapter 7: File I/O (reading and writing files)

Mode	Explaination
`>`	**Write (trunc)**. Will overwrite existing files. Creates a new file if no file was found
`>>`	**Write (append)**. Will not overwrite files but append new content at the end of it. Will also create a file if used for opening a non existing file.
`<`	**Read**. Opens the file in read only mode.
`+<`	**Read / Write**. Will not create or truncate the file.
`+>`	**Read / Write (trunc)**. Will create and truncate the file.
`+>>`	**Read / Write (append)**. Will create but not truncate the file.

Section 7.1: Opening A FileHandle for Reading

Opening Generic ASCII Text Files

Version ≥ 5.6.0

```
open my $filehandle, '<', $name_of_file or die "Can't open $name_of_file, $!";
```

This is the basic idiom for "default" File IO and makes `$filehandle` a readable input stream of `bytes`, filtered by a default system-specific decoder, which can be locally set with the `open` pragma

Perl itself does not handle errors in file opening, so you have to handle those yourself by checking the exit condition of `open`. `$!` is populated with the error message that caused open to fail.

On Windows, the default decoder is a "CRLF" filter, which maps any "\r\n" sequences in the input to "\n"

Opening Binary Files

Version ≥ 5.8.0

```
open my $filehandle, '<:raw', 'path/to/file' or die "Can't open $name_of_file, $!";
```

This specifies that Perl should *not* perform a `CRLF` translation on Windows.

Opening UTF8 Text Files

Version ≥ 5.8.0

```
open my $filehandle, '<:raw:encoding(utf-8)', 'path/to/file'
   or die "Can't open $name_of_file, $!";
```

This specifies that Perl should both avoid `CRLF` translation, and then decode the resulting bytes into strings of *characters* (internally implemented as arrays of integers which can exceed 255), instead of strings of *bytes*

Section 7.2: Reading from a file

```
my $filename = '/path/to/file';

open my $fh, '<', $filename or die "Failed to open file: $filename";

# You can then either read the file one line at a time...
while(chomp(my $line = <$fh>)) {
    print $line . "\n";
}

# ...or read whole file into an array in one go
chomp(my @fileArray = <$fh>);
```

If you know that your input file is UTF-8, you can specify the encoding:

```
open my $fh, '<:encoding(utf8)', $filename or die "Failed to open file: $filename";
```

After finished reading from the file, the filehandle should be closed:

```
close $fh or warn "close failed: $!";
```

See also: Reading a file into a variable

Another and **faster** way to read a file is to use File::Slurper Module. This is useful if you work with many files.

```
use File::Slurper;
my $file = read_text("path/to/file"); # utf8 without CRLF transforms by default
print $file; #Contains the file body
```

See also: [Reading a file with slurp]

Section 7.3: Write to a file

This code opens a file for writing. Returns an error if the file couldn't be opened. Also closes the file at the end.

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

# Open "output.txt" for writing (">") and from now on, refer to it as the variable $fh.
open(my $fh, ">", "output.txt")
# In case the action failed, print error message and quit.
or die "Can't open > output.txt: $!";
```

Now we have an open file ready for writing which we access through `$fh` (this variable is called a *filehandle*). Next we can direct output to that file using the `print` operator:

```
# Print "Hello" to $fh ("output.txt").
print $fh "Hello";
# Don't forget to close the file once we're done!
close $fh or warn "Close failed: $!";
```

The `open` operator has a scalar variable (`$fh` in this case) as its first parameter. Since it is defined in the `open` operator it is treated as a *filehandle*. Second parameter `">"` (greater than) defines that the file is opened for writing. The last parameter is the path of the file to write the data to.

To write the data into the file, the `print` operator is used along with the *filehandle*. Notice that in the `print` operator there is no comma between the *filehandle* and the statement itself, just whitespace.

Section 7.4: "use autodie" and you won't need to check file open/close failures

`autodie` allows you to work with files without having to explicitly check for open/close failures.

Since Perl 5.10.1, the `autodie` pragma has been available in core Perl. When used, Perl will automatically check for errors when opening and closing files.

Here is an example in which all of the lines of one file are read and then written to the end of a log file.

```
use 5.010;    # 5.010 and later enable "say", which prints arguments, then a newline
use strict;   # require declaring variables (avoid silent errors due to typos)
use warnings; # enable helpful syntax-related warnings
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding
use autodie;  # Automatically handle errors in opening and closing files

open(my $fh_in, '<', "input.txt"); # check for failure is automatic

# open a file for appending (i.e. using ">>")
open( my $fh_log, '>>', "output.log"); # check for failure is automatic

while (my $line = readline $fh_in) # also works: while (my $line = <$fh_in>)
{
     # remove newline
     chomp $line;

     # write to log file
     say $fh_log $line or die "failed to print '$line'"; # autodie doesn't check print
}

# Close the file handles (check for failure is automatic)
close $fh_in;
close $fh_log;
```

By the way, you should technically always check `print` statements. Many people don't, but `perl` (the Perl interpreter) doesn't do this automatically and [neither does `autodie`](#).

Section 7.5: Rewind a filehandle

Sometimes it is needful to backtrack after reading.

```
# identify current position in file, in case the first line isn't a comment
my $current_pos = tell;

while (my $line = readline $fh)
{
    if ($line =~ /$START_OF_COMMENT_LINE/)
    {
        push @names, get_name_from_comment($line);
    }
    else {
        last; # break out of the while loop
    }
    $current_pos = tell; # keep track of current position, in case we need to rewind the next line
read
}

# Step back a line so that it can be processed later as the first data line
seek $fh, $current_pos, 0;
```

Section 7.6: Reading and Writing gzip compressed files

Writing a gzipped file

To write a gzipped file, **`use`** the module `IO::Compress::Gzip` and create a filehandle by creating a new instance of `IO::Compress::Gzip` for the desired output file:

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Compress::Gzip;

my $fh_out = IO::Compress::Gzip->new("hello.txt.gz");

print $fh_out "Hello World!\n";

close $fh_out;

use IO::Compress::Gzip;
```

Reading from a gzipped file

To read from a gzipped file, **use** the module `IO::Uncompress::Gunzip` and then create a filehandle by creating a new instance of `IO::Uncompress::Gunzip` for the input file:

```
#!/bin/env perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Uncompress::Gunzip;

my $fh_in = IO::Uncompress::Gunzip->new("hello.txt.gz");

my $line = readline $fh_in;

print $line;
```

Section 7.7: Setting the default Encoding for IO

```
# encode/decode UTF-8 for files and standard input/output
use open qw( :encoding(UTF-8) :std );
```

This `pragma` changes the default mode of reading and writing text (files, standard input, standard output, and standard error) to UTF-8, which is typically what you want when writing new applications.

ASCII is a subset of UTF-8, so this is not expected to cause any problems with legacy ASCII files and will help protect you the accidental file corruption that can happen when treating UTF-8 files as ASCII.

However, it is important that you know what the encoding of your files is that you are dealing with and handle them accordingly. (Reasons that we should not ignore Unicode.) For more in depth treatment of Unicode, please see the Perl Unicode topic.

Chapter 8: Reading a file's content into a variable

Section 8.1: Path::Tiny

Using the idiom from The Manual Way several times in a script soon gets tedious so you might want to try a module.

```
use Path::Tiny;
my $contents = path($filename)->slurp;
```

You can pass a `binmode` option if you need control over file encodings, line endings etc. - see `man perlio`:

```
my $contents = path($filename)->slurp( {binmode => ":encoding(UTF-8)"} );
```

`Path::Tiny` also has a lot of other functions for dealing with files so it may be a good choice.

Section 8.2: The manual way

```
open my $fh, '<', $filename
    or die "Could not open $filename for reading: $!";
my $contents = do { local $/; <$fh> };
```

After opening the file (read `man perlio` if you want to read specific file encodings instead of raw bytes), the trick is in the `do` block: `<$fh>`, the file handle in a diamond operator, returns a single record from the file. The "input record separator" variable `$/` specifies what a "record" is—by default it is set to a newline character so "a record" means "a single line". As `$/` is a global variable, `local` does two things: it creates a temporary local copy of `$/` that will vanish at the end of the block, and gives it the (non-)value `undef` (the "value" which Perl gives to uninitialized variables). When the input record separator has that (non-)value, the diamond operator will return the entire file. (It considers the entire file to be a single line.)

Using `do`, you can even get around manually opening a file. For repeated reading of files,

```
sub readfile { do { local(@ARGV,$/) = $_[0]; <> } }
my $content = readfile($filename);
```

can be used. Here, another global variable(`@ARGV`) is localized to simulate the same process used when starting a perl script with parameters. `$/` is still `undef`, since the array in front of it "eats" all incoming arguments. Next, the diamond operator `<>` again delivers one record defined by `$/` (the whole file) and returns from the `do` block, which in turn return from the sub.

The sub has no explicit error handling, which is bad practice! If an error occurs while reading the file, you will receive `undef` as return value, as opposed to an empty string from an empty file.

Another disadvantage of the last code is the fact that you cannot use PerlIO for different file encodings—you always get raw bytes.

Section 8.3: File::Slurp

Don't use it. Although it has been around for a long time and is still the module most programmers will suggest, it is broken and not likely to be fixed.

Section 8.4: File::Slurper

This is a minimalist module that only slurps files into variables, nothing else.

```
use File::Slurper 'read_text';
my $contents = read_text($filename);
```

`read_text()` takes two optional parameters to specify the file encoding and whether line endings should be translated between the unixish LF or DOSish CRLF standards:

```
my $contents = read_text($filename, 'UTF-8', 1);
```

Section 8.5: Slurping a file into an array variable

```
open(my $fh, '<', "/some/path") or die $!;
my @ary = <$fh>;
```

When evaluated in list context, the diamond operator returns a list consisting of all the lines in the file (in this case, assigning the result to an array supplies list context). The line terminator is retained, and can be removed by chomping:

```
chomp(@ary); #removes line terminators from all the array elements.
```

Section 8.6: Slurp file in one-liner

Input record separator can be specified with `-0` switch (*zero*, not *capital O*). It takes an octal or hexadecimal number as value. Any value `0400` or above will cause Perl to slurp files, but by convention, the value used for this purpose is `0777`.

```
perl -0777 -e 'my $file = <>; print length($file)' input.txt
```

Going further with minimalism, specifying `-n` switch causes Perl to automatically read each line (in our case — the whole file) into variable `$_`.

```
perl -0777 -ne 'print length($_)' input.txt
```

Chapter 9: Exception handling

Section 9.1: eval and die

This is the built-in way to deal with "exceptions" without relying on third party libraries like Try::Tiny.

```
my $ret;

eval {
  $ret = some_function_that_might_die();
  1;
} or do {
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};

# use $ret
```

We "abuse" the fact that `die` has a false return value, and the return value of the overall code block is the value of the last expression in the code block:

- if `$ret` is assigned to successfully, then the `1;` expression is the last thing that happens in the `eval` code block. The `eval` code block thus has a true value, so the `or do` block does not run.
- if `some_function_that_might_die()` does `die`, then the last thing that happens in the `eval` code block is the `die`. The `eval` code block thus has a false value and the `or do` block does run.
- The first thing you *must* do in the `or do` block is read `$@`. This global variable will hold whatever argument was passed to `die`. The `|| "Zombie Error"` guard is popular, but unnecessary in the general case.

This is important to understand because some not all code does fail by calling die, but the same structure can be used regardless. Consider a database function that returns:

- the number of rows affected on success
- `'0 but true'` if the query is successful but no rows were affected
- `0` if the query was not successful.

In that case you can still use the same idiom, but you *have* to skip the final `1;`, and this function *has* to be the last thing in the eval. Something like this:

```
eval {
  my $value = My::Database::retrieve($my_thing); # dies on fail
  $value->set_status("Completed");
  $value->set_completed_timestamp(time());
  $value->update(); # returns false value on fail
} or do { # handles both the die and the 0 return value
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};
```

Chapter 10: Pack and unpack

Section 10.1: Manually Converting C Structs to Pack Syntax

If you're ever dealing with C Binary API's from Perl Code, via the `syscall`, `ioctl`, or `fcntl` functions, you need to know how to construct memory in a C Compatible way.

For instance, if you were ever dealing with some function that expected a `timespec`, you'd look into `/usr/include/time.h` and find:

```
struct timespec
{
    __time_t tv_sec;            /* Seconds.  */
    __syscall_slong_t tv_nsec;  /* Nanoseconds.  */
};
```

You do a dance with `cpp` to find what that really means:

```
cpp -E /usr/include/time.h -o /dev/stdout | grep __time_t
# typedef long int __time_t;
cpp -E /usr/include/time.h -o /dev/stdout | grep __syscall_slong_t
# typedef long int __syscall_slong_t
```

So its a (signed) int

```
echo 'void main(){ printf("%#lx\n", sizeof(__syscall_slong_t)); }' |
  gcc -x c -include stdio.h -include time.h - -o /tmp/a.out && /tmp/a.out
# 0x8
```

And it takes 8 bytes. So 64bit signed int. And I'm on a 64Bit Processor. =)

Perldoc `pack` says

> ```
>             q  A signed quad (64-bit) value.
> ```

So to pack a timespec:

```
sub packtime {
    my ( $config ) = @_;
    return pack 'qq', @{$config}{qw( tv_sec tv_nsec )};
}
```

And to unpack a timespec:

```
sub unpacktime {
   my ( $buf ) = @_;
   my $out = {};
   @{$out}{qw( tv_sec tv_nsec )} = unpack 'qq', $buf;
   return $out;
}
```

Now you can just use those functions instead.

```
my $timespec = packtime({ tv_sec => 0, tv_nsec => 0 });
syscall(  ..., $timespec ); # some syscall that reads timespec
```

```
later ...
syscall( ..., $timespec ); # some syscall that writes timespec
print Dumper( unpacktime( $timespec ));
```

Section 10.2: Constructing an IPv4 header

Sometimes you have to deal with structures defined in terms of C data types from Perl. One such application is the creation of raw network packets, in case you want to do something fancier than what the regular socket API has to offer. This is just what `pack()` (and `unpack()` of course) is there for.

The obligatory part of an IP header is 20 octets (AKA "bytes") long. As you can see behind this link, source and destination IP address make up the last two 32-bit values in the header. Among the other fields are some with 16 bits, some with 8 bits, and a few smaller chunks between 2 and 13 bits.

Assuming we have the following variables to stuff into our header:

```
my ($dscp, $ecn, $length,
    $id, $flags, $frag_off,
    $ttl, $proto,
    $src_ip,
    $dst_ip);
```

Note that three fields from the header are missing:

- The version is always 4 (it's IPv4 after all)
- IHL is 5 in our example as we don't have an *options* field; length is specified in units of 4 octets so 20 octets gives a length of 5.
- The checksum can be left at 0. Actually we'd have to calculate it but the code to do this doesn't concern us here.

We could try and use bit operations to construct e.g. the first 32 bits:

```
my $hdr = 4 << 28 | 5 << 24 | $dscp << 18 | $ecn << 16 | $length;
```

This approach only works up to the size of an integer though, which is usually 64 bits but can be as low as 32. Worse, it depends on the CPU's endianness so it will work on some CPUs and fail on others. Let's try `pack()`:

```
my $hdr = pack('H2B8n', '45', sprintf("%06b%02b", $dscp, $ecn), $length);
```

The template first specifies `H2`, a *2-character hex string, high nybble first*. The corresponding argument to pack is "45"—version 4, length 5. The next template is `B8`, an *8-bit bit string, descending bit order inside each byte*. We need to use bit strings to control layout down to chunks smaller than a nybble (4 bits), so the `sprintf()` is used to construct such a bit string from 6 bits from `$dscp` and 2 from `$ecn`. The last one is `n`, an *unsigned 16-bit value in Network Byte Order*, i.e. always big-endian no matter what your CPU's native integer format is, and it is filled from `$length`.

That's the first 32 bits of the header. The rest can be built similarly:

Template	Argument	Remarks
`n`	`$id`	
`B16`	`sprintf("%03b%013b", $flags, $frag_off)`	Same as DSCP/ECN
`C2`	`$ttl, $proto`	Two consecutive unsigned octets
`n`	`0` / `$checksum`	`x` could be used to insert a null byte but `n` lets us specify an argument should we choose to calculate a checksum

`N2`	`$src_ip, $dst_ip`	use `a4a4` to pack the result of two `gethostbyname()` calls as it is in Network Byte Order already!

So the complete call to pack an IPv4 header would be:

```
my $hdr = pack('H2B8n2B16C2nN2',
    '45', sprintf("%06b%02b", $dscp, $ecn), $length,
    $id, sprintf("%03b%013b", $flags, $frag_off),
    $ttl, $proto, 0,
    $src_ip, $dst_ip
);
```

Chapter 11: Strings and quoting methods

Section 11.1: String Literal Quoting

String literals imply no escaping or interpolation (with the exception of quoting string terminators)

```
print 'This is a string literal\n'; # emits a literal \ and n to terminal

print 'This literal contains a \'postraphe '; # emits the ' but not its preceding \
```

You can use alternative quoting mechanisms to avoid clashes:

```
print q/This is is a literal \' <-- 2 characters /;  # prints both \ and '
print q^This is is a literal \' <-- 2 characters ^;  # also
```

Certain chosen quote characters are "balanced"

```
print q{ This is a literal and I contain { parens! } }; # prints inner { }
```

Section 11.2: Double-quoting

Double-quoted strings use **interpolation** and **escaping** – unlike single-quoted strings. To double-quote a string, use either double quotes `"` or the `qq` operator.

```
my $greeting = "Hello!\n";
print $greeting;
# => Hello! (followed by a linefeed)

my $bush = "They misunderestimated me."
print qq/As Bush once said: "$bush"\n/;
# => As Bush once said: "They misunderestimated me." (with linefeed)
```

The `qq` is useful here, to avoid having to escape the quotation marks. Without it, we would have to write...

```
print "As Bush once said: \"$bush\"\n";
```

... which just isn't as nice.

Perl doesn't limit you to using a slash `/` with `qq`; you can use any (visible) character.

```
use feature 'say';

say qq/You can use slashes.../;
say qq{...or braces...};
say qq^...or hats...^;
say qq|...or pipes...|;
# say qq ...but not whitespace. ;
```

You can also interpolate arrays into strings.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
say "I like these letters: @letters.";
```

```
# => I like these letters: a b c.
```

By default the values are space-separated – because the special variable `$"` defaults to a single space. This can, of course, be changed.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
{local $" = ", "; say "@letters"; }    # a, b, c
```

If you prefer, you have the option to **use** English and change `$LIST_SEPARATOR` instead:

```
use v5.18; # English should be avoided on older Perls
use English;

my @letters = ('a', 'b', 'c');
{ local $LIST_SEPARATOR = "\n"; say "My favourite letters:\n\n@letters" }
```

For anything more complex than this, you should use a loop instead.

```
say "My favourite letters:";
say;
for my $letter (@letters) {
  say " - $letter";
}
```

Interpolation does *not* work with hashes.

```
use feature 'say';

my %hash = ('a', 'b', 'c', 'd');
say "This doesn't work: %hash"          # This doesn't work: %hash
```

Some code abuses interpolation of references – **avoid it**.

```
use feature 'say';

say "2 + 2 == @{[ 2 + 2 ]}";            # 2 + 2 = 4 (avoid this)
say "2 + 2 == ${\( 2 + 2 )}";           # 2 + 2 = 4 (avoid this)
```

The so-called "cart operator" causes perl to dereference `@{ ... }` the array reference `[ ... ]` that contains the expression that you want to interpolate, `2 + 2`. When you use this trick, Perl builds an anonymous array, then dereferences it and discards it.

The `${\( ... )}` version is somewhat less wasteful, but it still requires allocating memory and it is even harder to read.

Instead, consider writing:

- `say "2 + 2 == " . 2 + 2;`
- `my $result = 2 + 2; say "2 + 2 == $result"`

Section 11.3: Heredocs

Large Multi-Line strings are burdensome to write.

```
my $variable = <<'EOF';
this block of text is interpreted literally,
no \'quotes matter, they're just text
only the trailing left-aligned EOF matters.
EOF
```

NB: Make sure you ignore stack-overflows syntax highlighter: It is very wrong.

And Interpolated Heredocs work the same way.

```
my $variable = <<"I Want it to End";
this block of text is interpreted.
quotes\nare interpreted, and $interpolations
get interpolated...
but still, left-aligned "I Want it to End" matters.
I Want it to End
```

Pending in 5.26.0* is an "Indented Heredoc" Syntax which trims left-padding off for you

Version ≥ 5.26.0

```
my $variable = <<~"MuchNicer";
    this block of text is interpreted.
    quotes\nare interpreted, and $interpolations
    get interpolated...
    but still, left-aligned "I Want it to End" matters.
MuchNicer
```

Section 11.4: Removing trailing newlines

The function `chomp` will remove *one* newline character, if present, from each scalar passed to it. `chomp` will mutate the original string and will return the number of characters removed

```
my $str = "Hello World\n\n";
my $removed = chomp($str);
print $str;     # "Hello World\n"
print $removed; # 1

# chomp again, removing another newline
$removed = chomp $str;
print $str;     # "Hello World"
print $removed; # 1

# chomp again, but no newline to remove
$removed = chomp $str;
print $str;     # "Hello World"
print $removed; # 0
```

You can also `chomp` more than one string at once:

```
my @strs = ("Hello\n", "World!\n\n"); # one newline in first string, two in second

my $removed = chomp(@strs); # @strs is now  ("Hello", "World!\n")
print $removed;             # 2

$removed = chomp(@strs); # @strs is now ("Hello", "World!")
print $removed;          # 1

$removed = chomp(@strs); # @strs is still ("Hello", "World!")
```

```
print $removed;          # 0
```

But usually, no one worries about how many newlines were removed, so `chomp` is usually seen in void context, and usually due to having read lines from a file:

```
while (my $line = readline $fh)
{
    chomp $line;

    # now do something with $line
}

my @lines = readline $fh2;

chomp (@lines); # remove newline from end of each line
```

Chapter 12: Split a string on unquoted separators

Section 12.1: parse_line()

Using `parse_line()` of Text::ParseWords:

```
use 5.010;
use Text::ParseWords;

my $line = q{"a quoted, comma", word1, word2};
my @parsed = parse_line(',', 1, $line);
say for @parsed;
```

Output:

```
"a quoted, comma"
 word1
 word2
```

Section 12.2: Text::CSV or Text::CSV_XS

```
use Text::CSV; # Can use Text::CSV which will switch to _XS if installed
$sep_char = ",";
my $csv = Text::CSV->new({sep_char => $sep_char});
my $line = q{"a quoted, comma", word1, word2};
$csv->parse($line);
my @fields = $csv->fields();
print join("\n", @fields)."\n";
```

Output:

```
a quoted, comma
 word1
 word2
```

NOTES

- By default, Text::CSV does not strip whitespace around separator character, the way `Text::ParseWords` does. However, adding `allow_whitespace=>1` to constructor attributes achieves that effect.

  ```
  my $csv = Text::CSV_XS->new({sep_char => $sep_char, allow_whitespace=>1});
  ```

  Output:

  ```
  a quoted, comma
  word1
  word2
  ```

- The library supports escaping special characters (quotes, separators)

- The library supports configurable separator character, quote character, and escape character

Documentatoin: http://search.cpan.org/perldoc/Text::CSV

Chapter 13: Object-oriented Perl

Section 13.1: Defining classes in modern Perl

Although available, defining a class from scratch is not recommended in modern Perl. Use one of helper OO systems which provide more features and convenience. Among these systems are:

- `Moose` - inspired by Perl 6 OO design
- `Class::Accessor` - a lightweight alternative to Moose
- `Class::Tiny` - truly minimal class builder

Moose

```
package Foo;
use Moose;

has bar => (is => 'ro');                # a read-only property
has baz => (is => 'rw', isa => 'Bool');  # a read-write boolean property

sub qux {
    my $self = shift;
    my $barIsBaz = $self->bar eq 'baz';  # property getter
    $self->baz($barIsBaz);               # property setter
}
```

Class::Accessor (Moose syntax)

```
package Foo;
use Class::Accessor 'antlers';

has bar => (is => 'ro');                # a read-only property
has baz => (is => 'rw', isa => 'Bool');  # a read-write property (only 'is' supported, the type is
ignored)
```

Class::Accessor (native syntax)

```
package Foo;
use base qw(Class::Accessor);

Foo->mk_accessors(qw(bar baz));  # some read-write properties
Foo->mk_accessors(qw(qux));      # a read-only property
```

Class::Tiny

```
package Foo;
use Class::Tiny qw(bar baz);  # just props
```

Section 13.2: Creating Objects

Unlike many other languages, Perl does not have constructors that allocate memory for your objects. Instead, one

should write a class method that both create a data structure and populate it with data (you may know it as the Factory Method design pattern).

```
package Point;
use strict;

sub new {
    my ($class, $x, $y) = @_;
    my $self = { x => $x, y => $y }; # store object data in a hash
    bless $self, $class;             # bind the hash to the class
    return $self;
}
```

This method can be used as follows:

```
my $point = Point->new(1, 2.5);
```

Whenever the arrow operator `->` is used with methods, its left operand is prepended to the given argument list. So, `@_` in **`new`** will contain values `('Point', 1, 2.5)`.

There is nothing special in the name **`new`**. You can call the factory methods as you prefer.

There is nothing special in hashes. You could do the same in the following way:

```
package Point;
use strict;

sub new {
    my ($class, @coord) = @_;
    my $self = \@coord;
    bless $self, $class;
    return $self;
}
```

In general, any reference may be an object, even a scalar reference. But most often, hashes are the most convenient way to represent object data.

Section 13.3: Defining Classes

In general, classes in Perl are just packages. They can contain data and methods, as usual packages.

```
package Point;
use strict;

my $CANVAS_SIZE = [1000, 1000];

sub new {
    ...
}

sub polar_coordinates {
    ...
}

1;
```

It is important to note that the variables declared in a package are class variables, not object (instance) variables.

Changing of a package-level variable affects all objects of the class. How to store object-specific data, see in "Creating Objects".

What makes class packages specific, is the arrow operator `->`. It may be used after a bare word:

```
Point->new(...);
```

or after a scalar variable (usually holding a reference):

```
my @polar = $point->polar_coordinates;
```

What is to the left of the arrow is prepended to the given argument list of the method. For example, after call

```
Point->new(1, 2);
```

array `@_` in `new` will contain three arguments: `('Point', 1, 2)`.

Packages representing classes should take this convention into account and expect that all their methods will have one extra argument.

Section 13.4: Inheritance and methods resolution

To make a class a subclass of another class, use `parent` pragma:

```
package Point;
use strict;
...
1;

package Point2D;
use strict;
use parent qw(Point);
...
1;

package Point3D;
use strict;
use parent qw(Point);
...
1;
```

Perl allows for multiple inheritance:

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);
...
1;
```

Inheritance is all about resolution which method is to be called in a particular situation. Since pure Perl does not prescribe any rules about the data structure used to store object data, inheritance has nothing to do with that.

Consider the following class hierarchy:

```
package GeometryObject;
use strict;
```

```
sub transpose { ...}

1;

package Point;
use strict;
use parent qw(GeometryObject);

sub new { ... };

1;

package PlanarObject;
use strict;
use parent qw(GeometryObject);

sub transpose { ... }

1;

package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new { ... }

sub polar_coordinates { ... }

1;
```

The method resolution works as follows:

1. The starting point is defined by the left operand of the arrow operator.
    - If it is a bare word:

      ```
      Point2D->new(...);
      ```

      ...or a scalar variable holding a string:

      ```
      my $class = 'Point2D';
      $class->new(...);
      ```

      ...then the starting point is the package with the corresponding name (`Point2D` in both examples).

    - If the left operand is a scalar variable holding a *blessed* reference:

      ```
      my $point = {...};
      bless $point, 'Point2D'; # typically, it is encapsulated into class methods
      my @coord = $point->polar_coordinates;
      ```

      then the starting point is the class of the reference (again, `Point2D`). The arrow operator cannot be used to call methods for *unblessed* references.

2. If the starting point contains the required method, it is simply called.

Thus, since `Point2D::new` exists,

```
Point2D->new(...);
```

will simply call it.

3. If the starting point does not contain the required method, depth-first search in the `parent` classes is performed. In the example above, the search order will be as follows:

   - `Point2D`
   - `Point` (first parent of `Point2D`)
   - `GeometryObject` (parent of `Point`)
   - `PlanarObject` (second parent of `Point2D`)

   For example, in the following code:

   ```
   my $point = Point2D->new(...);
   $point->transpose(...);
   ```

   the method that will be called is `GeometryObject::transpose`, even though it would be overridden in `PlanarObject::transpose`.

4. You can set the starting point explicitly.

   In the previous example, you can explicitly call `PlanarObject::transpose` like so:

   ```
   my $point = Point2D->new(...);
   $point->PlanarObject::transpose(...);
   ```

5. In a similar manner, `SUPER::` performs method search in parent classes of the current class.

   For example,

   ```
   package Point2D;
   use strict;
   use parent qw(Point PlanarObject);

   sub new {
       (my $class, $x, $y) = @_;
       my $self = $class->SUPER::new;
       ...
   }

   1;
   ```

   will call `Point::new` in the course of the `Point2D::new` execution.

Section 13.5: Class and Object Methods

In Perl, the difference between class (static) and object (instance) methods is not so strong as in some other languages, but it still exists.

The left operand of the arrow operator `->` becomes the first argument of the method to be called. It may be either a

string:

```
# the first argument of new is string 'Point' in both cases
Point->new(...);

my $class = 'Point';
$class->new(...);
```

or an object reference:

```
# reference contained in $point is the first argument of polar_coordinates
my $point = Point->new(...);
my @coord = $point->polar_coordinates;
```

Class methods are just the ones that expect their first argument to be a string, and object methods are the ones that expect their first argument to be an object reference.

Class methods typically do not do anything with their first argument, which is just a name of the class. Generally, it is only used by Perl itself for method resolution. Therefore, a typical class method can be called for an object as well:

```
my $width = Point->canvas_width;

my $point = Point->new(...);
my $width = $point->canvas_width;
```

Although this syntax is allowed, it is often misleading, so it is better to avoid it.

Object methods receive an object reference as the first argument, so they can address the object data (unlike class methods):

```
package Point;
use strict;

sub polar_coordinates {
    my ($point) = @_;
    my $x = $point->{x};
    my $y = $point->{y};
    return (sqrt($x * $x + $y * $y), atan2($y, $x));
}

1;
```

The same method can track both cases: when it is called as a class or an object method:

```
sub universal_method {
    my $self = shift;
    if (ref $self) {
        # object logic
        ...
    }
    else {
        # class logic
        ...
    }
}
```

Section 13.6: Roles

A role in Perl is essentially

- a set of methods and attributes which
- injected into a class directly.

A role provides a piece of functionality which can be *composed* into (or *applied* to) any class (which is said to *consume* the role). A role cannot be inherited but may be consumed by another role.

A role may also *require* consuming classes to implement some methods instead of implementing the methods itself (just like interfaces in Java or C#).

Perl does not have built-in support for roles but there are CPAN classes which provide such support.

Moose::Role

```
package Chatty;
use Moose::Role;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}


package Parrot;
use Moose;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

Role::Tiny

Use if your OO system does not provide support for roles (e.g. `Class::Accessor` or `Class::Tiny`). Does not support attributes.

```
package Chatty;
use Role::Tiny;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}

package Parrot;
use Class::Tiny;
use Role::Tiny::With;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
```

```
}
```

Chapter 14: Regular Expressions

Section 14.1: Replace a string using regular expressions

```
s/foo/bar/;          # replace "foo" with "bar" in $_
my $foo = "foo";
$foo =~ s/foo/bar/; # do the above on a different variable using the binding operator =~
s~ foo ~ bar ~;      # using ~ as a delimiter
$foo = s/foo/bar/r; # non-destructive r flag: returns the replacement string without modifying the
variable it's bound to
s/foo/bar/g;         # replace all instances
```

Section 14.2: Matching strings

The `=~` operator attempts to match a regular expression (set apart by `/`) to a string:

```
my $str = "hello world";
print "Hi, yourself!\n" if $str =~ /^hello/;
```

`/^hello/` is the actual regular expression. The `^` is a special character that tells the regular expression to start with the beginning of the string and not match in the middle somewhere. Then the regex tries to find the following letters in order `h`, `e`, `l`, `l`, and `o`.

Regular expressions attempt to match the default variable (`$_`) if bare:

```
$_ = "hello world";

print "Ahoy!\n" if /^hello/;
```

You can also use different delimiters is you precede the regular expression with the `m` operator:

```
m~^hello~;
m{^hello};
m|^hello|;
```

This is useful when matching strings that include the `/` character:

```
print "user directory" if m|^/usr|;
```

Section 14.3: Parsing a string with a regex

Generally, it's not a good idea to use a regular expression to parse a complex structure. But it can be done. For instance, you might want to load data into hive table and fields are separated by comma but complex types like array are separated by a "|". Files contain records with all fields separated by comma and complex type are inside square bracket. In that case, this bit of disposable Perl might be sufficient:

```
echo "1,2,[3,4,5],5,6,[7,8],[1,2,34],5" | \
    perl -ne \
        'while( /\[[^,\]]+\,.*\]/ ){
            if( /\[([^\]\|]+)\]/){
                $text = $1;
                $text_to_replace = $text;
                $text =~ s/\,/\|/g;
                s/$text_to_replace/$text/;
```

```
            }
        } print'
```

You'll want to spot check the output:

> 1,2,[3|4|5],5,6,[7|8],[1|2|34],5

Section 14.4: Usage of \Q and \E in pattern matching

What's between \Q and \E is treated as normal characters

```
#!/usr/bin/perl

my $str = "hello.it's.me";

my @test = (
  "hello.it's.me",
    "hello/it's!me",
    );

sub ismatched($) { $_[0] ? "MATCHED!" : "DID NOT MATCH!" }

my @match = (
      [ general_match=> sub { ismatched /$str/ } ],
      [ qe_match    => sub { ismatched /\Q$str\E/ } ],
      );

for (@test) {
    print "\String = '$_':\n";

foreach my $method (@match) {
    my($name,$match) = @$method;
    print "  - $name: ", $match->(), "\n";
}
```

}

Output

```
String = 'hello.it's.me':
  - general_match: MATCHED!
  - qe_match: MATCHED!
String = 'hello/it's!me':
  - general_match: MATCHED!
  - qe_match: DID NOT MATCH!
```

Chapter 15: Perl commands for Windows Excel with Win32::OLE module

Parameters	Details
Cell1 (required)	The name of the range. This must be an A1-style reference in the language of the macro. It can include the range operator (a colon), the intersection operator (a space), or the union operator (a comma).
Cell2 (optional)	If specified, *Cell1* corresponds to the upper-left corner of the range and *Cell2* corresponds to the lower-right corner of the range

These examples introduce the most used commands of Perl to manipulate Excel via Win32::OLE module.

Section 15.1: Opening and Saving Excel/Workbooks

```
#Modules to use
use Cwd 'abs_path';
use Win32::OLE;
use Win32::OLE qw(in with);
use Win32::OLE::Const "Microsoft Excel";
$Win32::OLE::Warn = 3;

#Need to use absolute path for Excel files
my $excel_file = abs_path("$Excel_path") or die "Error: the file $Excel_path has not been found\n";

# Open Excel application
my $Excel = Win32::OLE->GetActiveObject('Excel.Application')
    || Win32::OLE->new('Excel.Application', 'Quit');

# Open Excel file
my $Book = $Excel->Workbooks->Open($excel_file);

#Make Excel visible
$Excel->{Visible} = 1;

#___ ADD NEW WORKBOOK
my $Book = $Excel->Workbooks->Add;
my $Sheet = $Book->Worksheets("Sheet1");
$Sheet->Activate;

#Save Excel file
$Excel->{DisplayAlerts}=0; # This turns off the "This file already exists" message.
$Book->Save; #Or $Book->SaveAs("C:\\file_name.xls");
$Book->Close; #or $Excel->Quit;
```

Section 15.2: Manipulation of Worksheets

```
#Get the active Worksheet
my $Book = $Excel->Activewindow;
my $Sheet = $Book->Activesheet;

#List of Worksheet names
my @list_Sheet = map { $_->{'Name'} } (in $Book->{Worksheets});

#Access a given Worksheet
my $Sheet = $Book->Worksheets($list_Sheet[0]);

#Add new Worksheet
$Book->Worksheets->Add({After => $workbook->Worksheets($workbook->Worksheets->{Count})});
```

```
#Change Worksheet Name
$Sheet->{Name} = "Name of Worksheet";

#Freeze Pane
$Excel -> ActiveWindow -> {FreezePanes} = "True";

#Delete Sheet
$Sheet -> Delete;
```

Section 15.3: Manipulation of cells

```
#Edit the value of a cell (2 methods)
$Sheet->Range("A1")->{Value} = 1234;
$Sheet->Cells(1,1)->{Value} = 1234;

#Edit the values in a range of cells
$Sheet->Range("A8:C9")->{Value} = [[ undef, 'Xyzzy', 'Plugh' ],
                                  [ 42,    'Perl',  3.1415  ]];

#Edit the formula in a cell (2 types)
$Sheet->Range("A1")->{Formula} = "=A1*9.81";
$Sheet->Range("A3")->{FormulaR1C1} = "=SUM(R[-2]C:R[-1]C)";      # Sum of rows
$Sheet->Range("C1")->{FormulaR1C1} = "=SUM(RC[-2]:RC[-1])";      # Sum of columns

#Edit the format of the text (font)
$Sheet->Range("G7:H7")->Font->{Bold}       = "True";
$Sheet->Range("G7:H7")->Font->{Italic}     = "True";
$Sheet->Range("G7:H7")->Font->{Underline}  = xlUnderlineStyleSingle;
$Sheet->Range("G7:H7")->Font->{Size}       = 8;
$Sheet->Range("G7:H7")->Font->{Name}       = "Arial";
$Sheet->Range("G7:H7")->Font->{ColorIndex} = 4;

#Edit the number format
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\@";                                  # Text
$Sheet -> Range("A1:H7") -> {NumberFormat} = "\$#,##0.00";                         # Currency
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\$#,##0.00_);[Red](\$#,##0.00)";      # Currency - red
negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00_);[Red](0.00)";                  # Numbers with
decimals
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0";                           #     Numbers with
commas
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0_);[Red](#,##0)";                # Numbers with
commas - red negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00%";                               # Percents
$Sheet -> Range("G7:H7") -> {NumberFormat} = "m/d/yyyy"                             # Dates

#Align text
$Sheet -> Range("G7:H7") -> {HorizontalAlignment} = xlHAlignCenter;                 # Center text;
$Sheet -> Range("A1:A2") -> {Orientation} = 90;                                     # Rotate text

#Activate Cell
$Sheet -> Range("A2") -> Activate;

$Sheet->Hyperlinks->Add({
   Anchor          =>  $range, #Range of cells with the hyperlink; e.g. $Sheet->Range("A1")
   Address         =>  $adr, #File path, http address, etc.
   TextToDisplay   =>  $txt, #Text in the cell
   ScreenTip       =>  $tip, #Tip while hovering the mouse over the hyperlink
});
```

N.B: to retrieve the list of hyperlinks, have a look at the following post Getting list of hyperlinks from an Excel

Section 15.4: Manipulation of Rows / Columns

```
#Insert a row before/after line 22
$Sheet->Rows("22:22")->Insert(xlUp, xlFormatFromRightOrBelow);
$Sheet->Rows("23:23")->Insert(-4121,0);     #xlDown is -4121 and that xlFormatFromLeftOrAbove is 0

#Delete a row
$Sheet->Rows("22:22")->Delete();

#Set column width and row height
$Sheet -> Range('A:A') -> {ColumnWidth} = 9.14;
$Sheet -> Range("8:8") -> {RowHeight}   = 30;
$Sheet -> Range("G:H") -> {Columns} -> Autofit;

# Get the last row/column
my $last_row = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious, SearchOrder
=> xlByRows})    -> {Row};
my $last_col = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious, SearchOrder
=> xlByColumns}) -> {Column};


#Add borders (method 1)
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlDouble;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThick;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {ColorIndex} = 1;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {LineStyle}  = xlDashDot
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {Weight}     = xlMedium;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {Weight}     = xlThin;

#Add borders (method 2)
my @edges = qw (xlInsideHorizontal xlInsideVertical xlEdgeBottom xlEdgeTop xlEdgeRight);
foreach my $edge (@edges)
```

Chapter 16: Install Perl modules via CPAN

Section 16.1: cpanminus, the lightweight configuration-free replacement for cpan

Usage

To install a module (assuming `cpanm` is already installed):

```
cpanm Data::Section
```

`cpanm` ("cpanminus") strives to be less verbose than `cpan` but still captures all of the installation information in a log file in case it is needed. It also handles many "interactive questions" for you, whereas `cpan` doesn't.

`cpanm` is also popular for installing dependencies of a project from, e.g., GitHub. Typical use is to first `cd` into the project's root, then run

```
cpanm --installdeps .
```

With `--installdeps` it will:

1. Scan and install *configure_requires* dependencies from either
    - META.json
    - META.yml (if META.json is missing)
2. Build the project (equivalent to `perl Build.PL`), generating MYMETA files
3. Scan and install *requires* dependencies from either
    - MYMETA.json
    - MYMETA.yml (if MYMETA.json is missing)

To specify the file 'some.cpanfile', containing the dependencies, run:

```
cpanm --installdeps --cpanfile some.cpanfile .
```

`cpanm` Installation

There are several ways to install it. Here's installation via `cpan`:

```
cpan App::cpanminus
```

`cpanm` Configuration

There is **no** config file for `cpanm`. Rather, it relies on the following environment variables for its configuration:

- `PERL_CPANM_OPT` (General cpanm command line options)
    - `export PERL_CPANM_OPT="--prompt"` # in .bashrc, to enable prompting, e.g.
    - `setenv PERL_CPANM_OPT "--prompt"` # in .tcshrc
- `PERL_MM_OPT` (ExtUtils::MakeMaker command line options, affects module install target)
- `PERL_MB_OPT` (Module::Build command line options, affects module install target)

Section 16.2: Installing modules manually

If you don't have permissions to install perl modules, you may still install them manually, indicating a custom path where you've got writing permissions.

Fist, download and unzip module archive:

```
wget module.tar.gz
tar -xzf module.tar.gz
cd module
```

Then, if the module distribution contains a `Makefile.PL` file, run:

```
perl Makefile.PL INSTALL_BASE=$HOME/perl
make
make test
make install
```

or if you have `Build.PL` file instead of a `Makefile.PL`:

```
perl Build.PL --install_base $HOME/perl
perl Build
perl Build test
perl Build install
```

You also have to include the module path in `PERL5LIB` environment variable in order to use it in your code:

```
export PERL5LIB=$HOME/perl
```

Section 16.3: Run Perl CPAN in your terminal (Mac and Linux) or command prompt (Windows)

Command line

You can use `cpan` to install modules directly from the command line:

```
cpan install DBI
```

This would be followed by possibly many pages of output describing exactly what it is doing to install the module. Depending on the modules being installed, it may pause and ask you questions.

Interactive Shell

You can also enter a "shell" thus:

```
perl -MCPAN -e "shell"
```

It will produce output as below:

```
Terminal does not support AddHistory.

cpan shell -- CPAN exploration and modules installation (v2.00)
Enter 'h' for help.

cpan[1]>
```

Then you can install the modules which you want by the easy command `install <module>`.

Example: `cpan[1]> install DBI`

After installing successfully, type `exit` to quit.

Chapter 17: XML Parsing

Section 17.1: Parsing with XML::Twig

```
#!/usr/bin/env perl

use strict;
use warnings 'all';

use XML::Twig;

my $twig = XML::Twig->parse( \*DATA );

#we can use the 'root' method to find the root of the XML.
my $root = $twig->root;

#first_child finds the first child element matching a value.
my $title = $root->first_child('title');

#text reads the text of the element.
my $title_text = $title->text;

print "Title is: ", $title_text, "\n";

#The above could be combined:
print $twig ->root->first_child_text('title'), "\n";

## You can use the 'children' method to iterate multiple items:
my $list = $twig->root->first_child('list');

#children can optionally take an element 'tag' - otherwise it just returns all of them.
foreach my $element ( $list->children ) {

   #the 'att' method reads an attribute
   print "Element with ID: ", $element->att('id') // 'none here', " is ", $element->text,
     "\n";
}

#And if we need to do something more complicated, we an use 'xpath'.
#get_xpath or findnodes do the same thing:
#return a list of matches, or if you specify a second numeric argument, just that numbered match.

#xpath syntax is fairly extensive, but in this one - we search:
# anywhere in the tree: //
#nodes called 'item'
#with an id attribute [@id]
#and with that id attribute equal to "1000".
#by specifying '0' we say 'return just the first match'.

print "Item 1000 is: ", $twig->get_xpath( '//item[@id="1000"]', 0 )->text, "\n";

#this combines quite well with `map` to e.g. do the same thing on multiple items
print "All IDs:\n", join ( "\n", map { $_ -> att('id') } $twig -> get_xpath('//item'));
#note how this also finds the item under 'summary', because of //

__DATA__
<?xml version="1.0" encoding="utf-8"?>
<root>
  <title>some sample xml</title>
  <first key="value" key2="value2">
```

```
    <second>Some text</second>
  </first>
  <third>
    <fourth key3="value">Text here too</fourth>
  </third>
  <list>
     <item id="1">Item1</item>
     <item id="2">Item2</item>
     <item id="3">Item3</item>
     <item id="66">Item66</item>
     <item id="88">Item88</item>
     <item id="100">Item100</item>
     <item id="1000">Item1000</item>
     <notanitem>Not an item at all really.</notanitem>
  </list>
  <summary>
     <item id="no_id">Test</item>
  </summary>
</root>
```

Section 17.2: Consuming XML with XML::Rabbit

With `XML::Rabbit` it is possible to consume XML files easily. You *define* in a declarative way and with an XPath syntax what you are looking for in the XML and `XML::Rabbit` will return objects according to the given definition.

Definition:

```
package Bookstore;
use XML::Rabbit::Root;
has_xpath_object_list books => './book' => 'Bookstore::Book';
finalize_class();

package Bookstore::Book;
use XML::Rabbit;
has_xpath_value bookid => './@id';
has_xpath_value author => './author';
has_xpath_value title => './title';
has_xpath_value genre => './genre';
has_xpath_value price => './price';
has_xpath_value publish_date => './publish_date';
has_xpath_value description => './description';
has_xpath_object purchase_data => './purchase_data' => 'Bookstore::Purchase';
finalize_class();

package Bookstore::Purchase;
use XML::Rabbit;
has_xpath_value price => './price';
has_xpath_value date => './date';
finalize_class();
```

XML Consumption:

```
use strict;
use warnings;
use utf8;

package Library;
use feature qw(say);
use Carp;
use autodie;
```

```
say "Showing data information";
my $bookstore = Bookstore->new( file => './sample.xml' );

foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author, "\n";
}
```

Notes:

Please be careful with the following:

1. The first class has to be `XML::Rabbit::Root`. It will place you inside the main tag of the XML document. In our case it will place us inside `<catalog>`

2. Nested classes which are optional. Those classes need to be accessed via a try/catch (or `eval / $@` check) block. Optional fields will simply return `null`. For example, for `purchase_data` the loop would be:

```
foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author;
    try {
        say "Purchase price: ". $book->purchase_data->price, "\n";
    } catch {
        say "No purchase price available\n";
    }
}
```

sample.xml

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
```

```
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
      <description>After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.</description>
   </book>
   <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2001-03-10</publish_date>
      <description>In post-apocalypse England, the mysterious
      agent known only as Oberon helps to create a new life
      for the inhabitants of London. Sequel to Maeve
      Ascendant.</description>
      <purchase_data>
        <date>2001-12-21</date>
        <price>20</price>
      </purchase_data>
   </book>
</catalog>
```

Section 17.3: Parsing with XML::LibXML

```
# This uses the 'sample.xml' given in the XML::Twig example.

# Module requirements (1.70 and above for use of load_xml)
use XML::LibXML '1.70';

# let's be a good perl dev
use strict;
use warnings 'all';

# Create the LibXML Document Object
my $xml = XML::LibXML->new();

# Where we are retrieving the XML from
my $file = 'sample.xml';

# Load the XML from the file
my $dom = XML::LibXML->load_xml(
    location => $file
);

# get the docroot
my $root = $dom->getDocumentElement;

# if the document has children
if($root->hasChildNodes) {

    # getElementsByLocalName returns a node list of all elements who's
    # localname matches 'title', and we want the first occurrence
    # (via get_node(1))
    my $title = $root->getElementsByLocalName('title');

    if(defined $title) {
        # Get the first matched node out of the nodeList
        my $node = $title->get_node(1);
```

```
        # Get the text of the target node
        my $title_text = $node->textContent;

        print "The first node with name 'title' contains: $title_text\n";
    }

    # The above calls can be combined, but is possibly prone to errors
    # (if the getElementsByLocalName() failed to match a node).
    #
    # my $title_text = $root->getElementsByLocalName('title')->get_node(1)->textContent;
}

# Using Xpath, get the price of the book with id 'bk104'
#

# Set our xpath
my $xpath = q!/catalog/book[@id='bk104']/price!;

# Does that xpath exist?
if($root->exists($xpath)) {

    # Pull in the twig
    my $match = $root->find($xpath);

    if(defined $match) {
        # Get the first matched node out of the nodeList
        my $node = $match->get_node(1);

        # pull in the text of that node
        my $match_text = $node->textContent;

        print "The price of the book with id bk104 is: $match_text\n";
    }
}
```

Chapter 18: Perl one-liners

Section 18.1: Upload file into mojolicious

```
perl -Mojo -E 'p("http://localhost:3000" => form => {Input_Type => "XML", Input_File => {file =>
"d:/xml/test.xml"}})'
```

File `d:/xml/test.xml` will be uploaded to server which listen connections on `localhost:3000` (Source)

In this example:

`-Mmodule` executes `use module;` before executing your program
`-E commandline` is used to enter one line of program
If you have no `ojo` module you can use `cpanm ojo` command to install it

To read more about how to run perl use `perldoc perlrun` command or read here

Section 18.2: Execute some Perl code from command line

Simple one-liners may be specified as command line arguments to perl using the `-e` switch (think "execute"):

```
perl -e'print "Hello, World!\n"'
```

Due to Windows quoting rules you can't use single-quoted strings but have to use one of these variants:

```
perl -e"print qq(Hello, World!\n)"
perl -e"print \"Hello, World!\n\""
```

Note that to avoid breaking old code, only syntax available up to Perl 5.8.x can be used with `-e`. To use anything newer your perl version may support, use `-E` instead. E.g. to use `say` available from 5.10.0 on plus Unicode 6.0 from >=v5.14.0 (also uses `-C0` to make sure **`STDOUT`** prints UTF-8):

```
Version ≥ 5.14.0
perl -C0 -E'say "\N{PILE OF POO}"'
```

Section 18.3: Using double-quoted strings in Windows one-liners

Windows uses only double quotes to wrap command line parameters. In order to use double quotes in perl one-liner (i.e. to print a string with an interpolated variable), you have to escape them with backslashes:

```
perl -e "my $greeting = 'Hello'; print \"$greeting, world!\n\""
```

To improve readability, you may use a `qq()` operator:

```
perl -e "my $greeting = 'Hello'; print qq($greeting, world!\n)"
```

Section 18.4: Print lines matching a pattern (PCRE grep)

```
perl -ne'print if /foo/' file.txt
```

Case-insensitive:

```
perl -ne'print if /foo/i' file.txt
```

Section 18.5: Replace a substring with another (PCRE sed)

```
perl -pe"s/foo/bar/g" file.txt
```

Or in-place:

```
perl -i -pe's/foo/bar/g' file.txt
```

On Windows:

```
perl -i.bak -pe"s/foo/bar/g" file.txt
```

Section 18.6: Print only certain fields

```
perl -lane'print "$F[0] $F[-1]"' data.txt
# prints the first and the last fields of a space delimited record
```

CSV example:

```
perl -F, -lane'print "$F[0] $F[-1]"' data.csv
```

Section 18.7: Print lines 5 to 10

```
perl -ne'print if 5..10' file.txt
```

Section 18.8: Edit file in-place

Without a backup copy (not supported on Windows)

```
perl -i -pe's/foo/bar/g' file.txt
```

With a backup copy `file.txt.bak`

```
perl -i.bak -pe's/foo/bar/g' file.txt
```

With a backup copy `old_file.txt.orig` in the `backup` subdirectory (provided the latter exists):

```
perl -i'backup/old_*.orig' -pe's/foo/bar/g' file.txt
```

Section 18.9: Reading the whole file as a string

```
perl -0777 -ne'print "The whole file as a string: --->$_<---\n"'
```

Note: The `-0777` is just a convention. Any `-0400` and above would de the same.

Chapter 19: Unicode

Section 19.1: The utf8 pragma: using Unicode in your sources

The `utf8` pragma indicates that the source code will be interpreted as UTF-8. Of course, this will only work if your text editor is also saving the source as UTF-8 encoded.

Now, string literals can contain arbitrary Unicode characters; identifiers can also contain Unicode but only word-like characters (see perldata and perlrecharclass for more information):

```
use utf8;
my $var1 = '§?§©????';      # works fine
my $? = 4;                 # works since ? is a word (matches \w) character
my $p§2 = 3;               # does not work since § is not a word character.
say "ya" if $var1 =~ /?§/; # works fine (prints "ya")
```

Note: When printing text to the terminal, make sure it supports UTF-8.*

There may be complex and counter-intuitive relationships between output and source encoding. Running on a UTF-8 terminal, you may find that adding the `utf8` pragma seems to break things:

```
$ perl -e 'print "Møøse\n"'
Møøse
$ perl -Mutf8 -e 'print "Møøse\n"'
M??se
$ perl -Mutf8 -CO -e 'print "Møøse\n"'
Møøse
```

In the first case, Perl treats the string as raw bytes and prints them like that. As these bytes happen to be valid UTF-8, they look correct even though Perl doesn't really know what characters they are (e.g. `length("Møøse")` will return 7, not 5). Once you add `-Mutf8`, Perl correctly decodes the UTF-8 source to characters, but output is in Latin-1 mode by default and printing Latin-1 to a UTF-8 terminal doesn't work. Only when you switch **`STDOUT`** to UTF-8 using `-CO` will the output be correct.

`use` `utf8` doesn't affect standard I/O encoding nor file handles!

Section 19.2: Handling invalid UTF-8

Reading invalid UTF-8

When reading UTF-8 encoded data, it is important to be aware of the fact the UTF-8 encoded data can be invalid or malformed. Such data should usually not be accepted by your program (unless you know what you are doing). When unexpectedly encountering malformed data, different actions can be considered:

- Print stacktrace or error message, and abort program gracefully, or
- Insert a substitution character at the place where the malformed byte sequence appeared, print a warning message to STDERR and continue reading as nothing happened.

By default, Perl will `warn` you about encoding glitches, but it will not abort your program. You can make your program abort by making UTF-8 warnings fatal, but be aware of the caveats in Fatal Warnings.

The following example writes 3 bytes in encoding ISO 8859-1 to disk. It then tries to read the bytes back again as UTF-8 encoded data. One of the bytes, `0xE5`, is an invalid UTF-8 one byte sequence:

```
use strict;
use warnings;
use warnings FATAL => 'utf8';

binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
my $bytes = "\x{61}\x{E5}\x{61}";  # 3 bytes in iso 8859-1: aåa
my $fn = 'test.txt';
open ( my $fh, '>:raw', $fn ) or die "Could not open file '$fn': $!";
print $fh $bytes;
close $fh;
open ( $fh, "<:encoding(utf-8)", $fn ) or die "Could not open file '$fn': $!";
my $str = do { local $/; <$fh> };
close $fh;
print "Read string: '$str'\n";
```

The program will abort with a fatal warning:

```
utf8 "\xE5" does not map to Unicode at ./test.pl line 10.
```

Line 10 is here the second last line, and the error occurs in the part of the line with `<$fh>` when trying to read a line from the file.

If you don't make warnings fatal in the above program, Perl will still print the warning. However, in this case it will try to recover from the malformed byte `0xE5` by inserting the four characters `\xE5` into the stream, and then continue with the next byte. As a result, the program will print:

```
Read string: 'a\xE5a'
```

Section 19.3: Command line switches for one-liners

Enable utf8 pragma

In order to enable `utf8` pragma in one-liner, perl interpreter should be called with `-Mutf8` option:

```
perl -Mutf8 -E 'my $? = "human"; say $?'
```

Unicode handling with -C switch

The `-C` command line flag lets you control Unicode features. It can be followed by a list of option letters.

Standard I/O

- `I` - **`STDIN`** will be in *UTF-8*
- `O` - **`STDOUT`** will be in *UTF-8*
- `E` - **`STDERR`** will be in *UTF-8*
- `S` - shorthand for `IOE`, standard I/O streams will be in *UTF-8*

```
echo "????????? ???????????" | perl -CS -Mutf8 -nE 'say "ok" if /?????????/'
```

Script's arguments

- `A` - treats **`@ARGV`** as an array of *UTF-8* encoded strings

```
perl -CA -Mutf8 -E 'my $arg = shift; say "anteater" if $arg eq "????????"' ????????
```

Default PerlIO layer

- `i` - *UTF-8* is the default PerlIO layer for input streams

- `o` - *UTF-8* is the default PerlIO layer for output streams
- `D` - shorthand for `io`

```
perl -CD -Mutf8 -e 'open my $fh, ">", "utf8.txt" or die $!; print $fh "?? ???"'
```

`-M` and `-C` switches may be combined:

```
perl -CASD -Mutf8 -E 'say "????????? ???????????\n"';
```

Section 19.4: Standard I/O

The encoding to be used for the standard I/O filehandles (**`STDIN`**, **`STDOUT`**, and **`STDERR`**), can be set separately for each handle using `binmode`:

```
binmode STDIN, ':encoding(utf-8)';
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
```

Note: when reading one would in general prefer `:encoding(utf-8)` over `:utf8`, see Remarks for more information.

Alternatively, you can use the `open` pragma.

```
# Setup such that all subsequently opened input streams will use ':encoding(utf-8)'
# and all subsequently opened output streams will use ':utf8'
# by default
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Make the (already opened) standard file handles inherit the setting
# given by the IO settings for the open pragma
use open ( :std );
# Now, STDIN has been converted to ':encoding(utf-8)', and
# STDOUT and STDERR have ':utf8'
```

Alternatively, to set all filehandles (both those yet to be opened and also the standard ones) to use `:encoding(utf-8)`:

```
use open qw( :encoding(utf-8) :std );
```

Section 19.5: File handles

Setting encoding with open()

When opening a text file, you may specify it's encoding explicitly with a three-argument `open()`. This en-/decoder attached to a file handle is called an "I/O layer":

```
my $filename = '/path/to/file';
open my $fh, '<:encoding(utf-8)', $filename or die "Failed to open $filename: $!";
```

See Remarks for a discussion of the differences between `:utf8` and `:encoding(utf-8)`.

Setting encoding with binmode()

Alternatively, you may use binmode() to set the encoding for individual file handle:

```
my $filename = '/path/to/file';
open my $fh, '<', $filename or die "Failed to open $filename: $!";
```

```
binmode $fh, ':encoding(utf-8)';
```

open pragma

To avoid setting encoding for each file handle separately, you may use the `open` pragma to set a default I/O layer used by all subsequent calls to the `open()` function and similar operators within the lexical scope of this pragma:

```
# Set input streams to ':encoding(utf-8)' and output streams to ':utf8'
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Or to set all input and output streams to ':encoding(utf-8)'
use open ':encoding(utf-8)';
```

Setting encoding with command line -C flag

Finally, it is also possible to run the perl interpreter with a `-CD` flag that applies UTF-8 as the default I/O layer. However, this option should be avoided since it relies on specific user behaviour which cannot be predicted nor controlled.

Section 19.6: Create filenames

The following examples use the UTF-8 encoding to represent filenames (and directory names) on disk. If you want to use another encoding, you should use `Encode::encode(...)`.

```
use v5.14;
# Make Perl recognize UTF-8 encoded characters in literal strings.
# For this to work: Make sure your text-editor is using UTF-8, so
# that bytes on disk are really UTF-8 encoded.
use utf8;

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that your terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

my $filename = 'æ€'; # $filename is now an internally UTF-8 encoded string.

# Note: in the following it is assumed that $filename has the internal UTF-8
#  flag set, if $filename is pure ASCII, it will also work since its encoding
#  overlaps with UTF-8. However, if it has another encoding like extended ASCII,
#  $filename will be written with that encoding and not UTF-8.
# Note: it is not necessary to encode $filename as UTF-8 here
#  since Perl is using UTF-8 as its internal encoding of $filename already

# Example1  -- using open()
open ( my $fh, '>', $filename ) or die "Could not open '$filename': $!";
close $fh;

# Example2 -- using qx() and touch
qx{touch $filename};

# Example3 -- using system() and touch
system 'touch', $filename;

# Example4 -- using File::Touch
use File::Touch;
eval { touch( $filename ) }; die "Could not create file '$filename': $!" if $@;
```

Section 19.7: Read filenames

Perl does not attempt to decode filenames returned by builtin functions or modules. Such strings representing

filenames should always be decoded explicitly, in order for Perl to recognize them as Unicode.

```
use v5.14;
use Encode qw(decode_utf8);

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that you terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

# Example1  -- using readdir()
my $dir = '.';
opendir(my $dh, $dir) or die "Could not open directory '$dir': $!";
while (my $filename = decode_utf8(readdir $dh)) {
    # Do something with $filename
}
close $dh;

# Example2  -- using getcwd()
use Cwd qw(getcwd);
my $dir = decode_utf8( getcwd() );

# Example3  -- using abs2rel()
use File::Spec;
use utf8;
my $base = 'ø';
my $path = "$base/b/æ";
my $relpath = decode_utf8( File::Spec->abs2rel( $path, $base ) );
# Note: If you omit $base, you need to encode $path first:
use Encode qw(encode_utf8);
my $relpath = decode_utf8( File::Spec->abs2rel( encode_utf8( $path ) ) );

# Example4  -- using File::Find::Rule (part1 matching a filename)
use File::Find::Rule;
use utf8;
use Encode qw(encode_utf8);
my $filename = 'æ';
# File::Find::Rule needs $filename to be encoded
my @files = File::Find::Rule->new->name( encode_utf8($filename) )->in('.');
$_ = decode_utf8( $_ ) for @files;

# Example5  -- using File::Find::Rule (part2 matching a regular expression)
use File::Find::Rule;
use utf8;
my $pat = '[æ].$'; # Unicode pattern
# Note: In this case:  File::Find::Rule->new->name( qr/$pat/ )->in('.')
#  will not work since $pat is Unicode and filenames are bytes
#  Also encoding $pat first will not work correctly
my @files;
File::Find::Rule->new->exec( sub { wanted( $pat, \@files ) } )->in('.');
$_ = decode_utf8( $_ ) for @files;
sub wanted {
    my ( $pat, $files ) = @_;
    my $name = decode_utf8( $_ );
    my $full_name = decode_utf8( $File::Find::name );
    push @$files, $full_name if $name =~ /$pat/;
}
```

Note: if you are concerned about invalid UTF-8 in the filenames, the use of `decode_utf8( ... )` in the above examples should probably be replaced by `decode( 'utf-8', ... )`. This is because `decode_utf8( ... )` is a synonym for `decode( 'utf8', ... )` and there is a difference between the encodings `utf-8` and `utf8` (see

Remarks below for more information) where `utf-8` is more strict on what is acceptable than `utf8`.

Chapter 20: Lists

Section 20.1: Array as list

The array is one of Perl's basic variable types. It contains a list, which is an ordered sequence of zero or more scalars. The array is the variable holding (and providing access to) the list data, as is documented in perldata.

You can assign a list to an array:

```
my @foo = ( 4, 5, 6 );
```

You can use an array wherever a list is expected:

```
join '-', ( 4, 5, 6 );
join '-', @foo;
```

Some operators only work with arrays since they mutate the list an array contains:

```
shift @array;
unshift @array, ( 1, 2, 3 );
pop @array;
push @array, ( 7, 8, 9 );
```

Section 20.2: Assigning a list to a hash

Lists can also be assigned to hash variables. When creating a list that will be assigned to a hash variable, it is recommended to use the ***fat comma*** `=>` between keys and values to show their relationship:

```
my %hash = ( foo => 42, bar => 43, baz => 44 );
```

The `=>` is really only a special comma that automatically quotes the operand to its left. So, you *could* use normal commas, but the relationship is not as clear:

```
my %hash = ( 'foo', 42, 'bar', 43, 'baz', 44 );
```

You can also use quoted strings for the left hand operand of the fat comma `=>`, which is especially useful for keys containing spaces.

```
my %hash = ( 'foo bar' => 42, 'baz qux' => 43 );
```

For details see Comma operator at `perldoc perlop`.

Section 20.3: Lists can be passed into subroutines

As to pass list into a subroutine, you specify the subroutine's name and then supply the list to it:

```
test_subroutine( 'item1', 'item2' );
test_subroutine  'item1', 'item2';     # same
```

Internally Perl makes *aliases* to those arguments and put them into the array `@_` which is available within the subroutine:

```
@_ =  ( 'item1', 'item2' ); # Done internally by perl
```

You access subroutine arguments like this:

```
sub test_subroutine {
    print $_[0]; # item1
    print $_[1]; # item2
}
```

Aliasing gives you the ability to change the original value of argument passed to subroutine:

```
sub test_subroutine {
    $_[0] +=  2;
}

my $x =  7;
test_subroutine( $x );
print $x; # 9
```

To prevent inadvertent changes of original values passed into your subroutine, you should copy them:

```
sub test_subroutine {
    my( $copy_arg1, $copy_arg2 ) =  @_;
    $copy_arg1 += 2;
}

my $x =  7;
test_subroutine $x; # in this case $copy_arg2 will have `undef` value
print $x; # 7
```

To test how many arguments were passed into the subroutine, check the size of `@_`

```
sub test_subroutine {
    print scalar @_, ' argument(s) passed into subroutine';
}
```

If you pass array arguments into a subroutine they all will be *flattened*:

```
my @x =  ( 1, 2, 3 );
my @y =  qw/ a b c /; # ( 'a', 'b', 'c' )
test_some_subroutine @x, 'hi', @y; # 7 argument(s) passed into subroutine
# @_ =  ( 1, 2, 3, 'hi', 'a', 'b', 'c' ) # Done internally for this call
```

If your `test_some_subroutine` contains the statement `$_[4] = 'd'`, for the above call it will cause `$y[0]` to have value `d` afterwards:

```
print "@y"; # d b c
```

Section 20.4: Return list from subroutine

You can, of course, return lists from subs:

```
sub foo {
    my @list1 =  ( 1, 2, 3 );
    my @list2 =  ( 4, 5 );

    return    ( @list1, @list2 );
}
```

```
my @list =  foo();
print @list;          # 12345
```

But it is not the recommended way to do that unless you know what you are doing.

While this is OK when the result is in **LIST** context, in **SCALAR** context things are unclear. Let's take a look at the next line:

```
print scalar foo();  # 2
```

Why `2`? What is going on?

1. Because `foo()` evaluated in *SCALAR* context, this list `( @list1, @list2 )` also evaluated in *SCALAR* context
2. In *SCALAR* context, LIST returns its last element. Here it is `@list2`
3. Again in *SCALAR* context, **array** `@list2` returns the number of its elements. Here it is `2`.

In most cases the **right strategy will return references to data structures**.
So in our case we should do the following instead:

```
return    ( \@list1, \@list2 );
```

Then the caller does something like this to receive the two returned *arrayrefs*:

```
my ($list1, $list2) = foo(...);
```

Section 20.5: Hash as list

In list context hash is flattened.

```
my @bar =  ( %hash, %hash );
```

The *array* `@bar` is initialized by list of two `%hash` hashes

- both `%hash` are flattened
- new list is created from flattened items
- `@bar` array is initialized by that list

It is guaranteed that key-value pairs goes together. Keys are always even indexed, values - odd. It is not guaranteed that key-value pairs are always flattened in same order:

```
my %hash =  ( a => 1, b => 2 );
print %hash; # Maybe 'a1b2' or 'b2a1'
```

Section 20.6: Using arrayref to pass array to sub

The arrayref for `@foo` is `\@foo`. This is handy if you need to pass an array and other things to a subroutine. Passing `@foo` is like passing multiple scalars. But passing `\@foo` is a single scalar. Inside the subroutine:

```
xyz(\@foo, 123);
...
sub xyz {
    my ($arr, $etc) = @_;
    print $arr->[0]; # using the first item in $arr. It is like $foo[0]
```

Chapter 21: Control Statements

Section 21.1: Conditionals

Perl supports many kinds of conditional statements (statements that are based on boolean results). The most common conditional statements are if-else, unless, and ternary statements. `given` statements are introduced as a switch-like construct from C-derived languages and are available in versions Perl 5.10 and above.

If-Else Statements

The basic structure of an if-statement is like this:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

For simple if-statements, the if can precede or succeed the code to be executed.

```
$number = 7;
if ($number > 4) { print "$number is greater than four!"; }

# Can also be written this way
print "$number is greater than four!" if $number > 4;
```

Section 21.2: Loops

Perl supports many kinds of loop constructs: for/foreach, while/do-while, and until.

```
@numbers = 1..42;
for (my $i=0; $i <= $#numbers; $i++) {
    print "$numbers[$i]\n";
}

#Can also be written as
foreach my $num (@numbers) {
    print "$num\n";
}
```

The while loop evaluates the conditional *before* executing the associated block. So, sometimes the block is never executed. For example, the following code would never be executed if the filehandle `$fh` was the filehandle for an empty file, or if was already exhausted before the conditional.

```
while (my $line = readline $fh) {
    say $line;
}
```

The `do`/`while` and `do`/`until` loops, on the other hand, evaluate the conditional *after* each time the block is executed. So, a `do`/`while` or a `do`/`until` loop is always executed at least once.

```
my $greeting_count = 0;
do {
    say "Hello";
    $greeting_count++;
} until ( $greeting_count > 1)
```

```
# Hello
# Hello
```

Chapter 22: Interpolation in Perl

Section 22.1: What is interpolated

Perl interpolates variable names:

```
my $name = 'Paul';
print "Hello, $name!\n"; # Hello, Paul!

my @char = ('a', 'b', 'c');
print "$char[1]\n"; # b

my %map = (a => 125, b => 1080, c => 11);
print "$map{a}\n"; # 125
```

Arrays may be interpolated as a whole, their elements are separated by spaces:

```
my @char = ('a', 'b', 'c');
print "My chars are @char\n"; # My chars are a b c
```

Perl does *not* interpolate hashes as a whole:

```
my %map = (a => 125, b => 1080, c => 11);
print "My map is %map\n"; # My map is %map
```

and function calls (including constants):

```
use constant {
    PI => '3.1415926'
};
print "I like PI\n";         # I like PI
print "I like " . PI . "\n"; # I like 3.1415926
```

Perl interpolates *escape sequences* starting with `\`:

```
\t                  horizontal tab
\n                  newline
\r                  return
\f                  form feed
\b                  backspace
\a                  alarm (bell)
\e                  escape
```

Interpolation of `\n` depends on the system where program is working: it will produce a newline character(s) according to the current system conventions.

Perl does *not* interpolate `\v`, which means vertical tab in C and other languages.

Character may be addressed using their codes:

```
\x{1d11e}     ???? by hexadecimal code
\o{350436}    ???? by octal code
\N{U+1d11e}   ???? by Unicode code point
```

or Unicode names:

```
\N{MUSICAL SYMBOL G CLEF}
```

Character with codes from `0x00` to `0xFF` in the *native* encoding may be addressed in a shorter form:

```
\x0a     hexadecimal
\012     octal
```

Control character may be addressed using special escape sequences:

```
\c@      chr(0)
\ca      chr(1)
\cb      chr(2)
...
\cz      chr(26)
\c[      chr(27)
\c\      chr(28) # Cannot be used at the end of a string
                 # since backslash will interpolate the terminating quote
\c]      chr(29)
\c^      chr(30)
\c_      chr(31)
\c?      chr(127)
```

Uppercase letters have the same meaning: `"\cA" == "\ca"`.

Interpretation of all escape sequences except for `\N{...}` may depend on the platform since they use platform- and encoding-dependent codes.

Section 22.2: Basic interpolation

Interpolation means that Perl interpreter will substitute the values of variables for their name and some symbols (which are impossible or difficult to type in directly) for special sequences of characters (it is also known as escaping). The most important distinction is between single and double quotes: double quotes interpolate the enclosed string, but single quotes do not.

```
my $name = 'Paul';
my $age = 64;
print "My name is $name.\nI am $age.\n"; # My name is Paul.
                                         # I am 64.
```

But:

```
print 'My name is $name.\nI am $age.\n'; # My name is $name.\nI am $age.\n
```

You can use `q{}` (with any delimiter) instead of single quotes and `qq{}` instead of double quotes. For example, `q{I'm 64}` allows to use an apostrophe within a non-interpolated string (otherwise it would terminate the string).

Statements:

```
print qq{$name said: "I'm $age".}; # Paul said: "I'm 64".
print "$name said: \"I'm $age\"."  # Paul said: "I'm 64".
```

do the same thing, but in the first one you do not need to escape double quotes within the string.

If your variable name clashes with surrounding text, you can use the syntax `${var}` to disambiguate:

```
my $decade = 80;
```

```
print "I like ${decade}s music!"  # I like 80s music!
```

Chapter 23: Simple interaction with database via DBI module

Column	Column
`$driver`	Driver for DB, "Pg" for Postgresql and "mysql" for MySQL
`$database`	your database name
`$userid`	your database id
`$password`	your database password
`$query`	put your query here, ex: "select * from $your_table"

Section 23.1: DBI module

You should make sure that module DBI has been installed on your pc, then follow the bellow steps:

1. use DBI module in your perl script

```
use DBI;
```

2. Declare some primary parameters

```
my $driver = "MyDriver";

my $database = "DB_name";

my $dsn = "DBI:$driver:dbname=$database";

my $userid = "your_user_ID";

my $password = "your_password";

my $tablename = "your_table";
```

3. Connect to your database

```
my $dbh = DBI->connect($dsn, $userid, $password);
```

4. Prepare your query

```
my $query = $dbh->prepare("Your DB query");
```

Ex:

```
$my_query = qq/SELECT * FROM table WHERE column1 = 2/;

my $query = $dbh->prepare($my_query);
```

We can also use variable in the query, like below:

```
my $table_name = "table";

my $filter_value = 2;

$my_query = qq/SELECT * FROM $table_name WHERE column1 = $filter_value/;
```

5. Execute your query

```
$query->execute();
```

*Note: To avoid injection attack, you should use placeholders `?` instead of put your variable in the query.

Ex: you want to show the all data from 'table' where column1=$value1 and column2=$value2:

```
my $query = $dbh->prepare("SELECT * FROM table WHERE column1 = ? AND column2 = ?;");

$query->execute($value1, $value2);
```

6. Fletch your data

`my @row = $query->fetchrow_array();` store data as array

or

`my $ref = $sth->fetchrow_hashref();` store data as hash reference

7. Finish and disconnect DB

```
$sth->finish;

$dbh->disconnect();
```

Chapter 24: Perl Testing

Section 24.1: Perl Unit Testing Example

The following is a simple example Perl test script, that gives some structure to allow for testing of other methods in the class/package under test. The script produces standard output with simple "ok" / "not ok" text, which is called TAP (Test Anything Protocol).

Typically the prove command runs the script(s) and summarises the test results.

```
#!/bin/env perl
# CPAN
use Modern::Perl;
use Carp;
use Test::More;
use Test::Exception;
use Const::Fast;

# Custom
BEGIN { use_ok('Local::MyPackage'); }

const my $PACKAGE_UNDER_TEST => 'Local::MyPackage';

# Example test of method 'file_type_build'
sub test_file_type_build {
    my %arg    = @_;
    my $label  = 'file_type_build';
    my $got_file_type;
    my $filename = '/etc/passwd';

    # Check the method call lives
    lives_ok(
        sub {
            $got_file_type = $PACKAGE_UNDER_TEST->file_type_build(
                filename => $filename
            );
        },
        "$label - lives"
    );

    # Check the result of the method call matches our expected result.
    like( $got_file_type, qr{ASCII[ ]text}ix, "$label - result" );
    return;
} ## end sub test_file_type_build

# More tests can be added here for method 'file_type_build', or other methods.


MAIN: {

   subtest 'file_type_build' => sub {
      test_file_type_build();
      # More tests of the method can be added here.
      done_testing();
   };

   # Tests of other methods can be added here, just like above.
```

```
    done_testing();
} ## end MAIN:
```

Best Practice

A test script should only test one package/class, but there many scripts may be used to test a package/class.

Further Reading

- Test::More - The basic test operations.
- Test::Exception - Testing thrown exceptions.
- Test::Differences - Comparing test results that have complex data structures.
- Test::Class - Class based testing rather than script. Similarities to JUnit.
- Perl Testing Tutorials - Further reading.

Chapter 25: Best Practices

Section 25.1: Using Perl::Critic

If you'd like to start implementing best practices, for yourself or your team, then Perl::Critic is the best place to start. The module is based on the *Perl Best Practices* book by Damien Conway and does a fairly good job implementing the suggestions made therein.

> **Note:** *I should mention (and Conway himself says in the book) that these are suggestions. I've found the book provides solid reasoning in most cases, though I certainly don't agree with all of them. The important thing to remember is that, whatever practices you decide to adopt, you remain consistent. The more predictable your code is, the easier it will be to maintain.*

You can also try out Perl::Critic through your browser at perlcritic.com.

Installation

```
cpan Perl::Critic
```

This will install the basic ruleset and a **perlcritic** script that can be called from the command line.

Basic Usage

The CPAN doc for perlcritic contains full documentation, so I will only be going over the most common use cases to get you started. Basic usage is to simply call perlcritic on the file:

```
perlcritic -1 /path/to/script.pl
```

perlcritic works both on scripts and on modules. The **-1** refers to the severity level of the rules you want to run against the script. There are five levels that correspond to how much Perl::Critic will pick apart your code.

-5 is the most gentle and will only warn about potentially dangerous problems that could cause unexpected results. **-1** is the most brutal and will complain about things as small as your code being tidy or not. In my experience, keeping code compliant with level 3 is good enough to keep out of danger without getting too persnickety.

By default, any failures will list the reason and severity the rule triggers on:

```
perlcritic -3 --verbose 8 /path/to/script.pl

Debugging module loaded at line 16, column 1.  You've loaded Data::Dumper, which probably shouln't
be loaded in production.  (Severity: 4)
Private subroutine/method '_sub_name' declared but not used at line 58, column 1.  Eliminate dead
code.  (Severity: 3)
Backtick operator used at line 230, column 37.  Use IPC::Open3 instead.  (Severity: 3)
Backtick operator used at line 327, column 22.  Use IPC::Open3 instead.  (Severity: 3)
```

Viewing Policies

You can quickly see which rules are being triggered and why by utilizing perlcritic's **--verbose** option:

Setting the level to 8 will show you the rule that triggered a warning:

```
perlcritic -3 --verbose 8 /path/to/script.pl
```

```
[Bangs::ProhibitDebuggingModules] Debugging module loaded at line 16, column 1.  (Severity: 4)
[Subroutines::ProhibitUnusedPrivateSubroutines] Private subroutine/method '_sub_name' declared but
not used at line 58, column 1.  (Severity: 3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 230, column 37.  (Severity:
3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 327, column 22.  (Severity:
3)
```

While a level of 11 will show the specific reasons why the rule exists:

```
perlcritic -3 --verbose 11 /path/to/script.pl

Debugging module loaded at line 16, near 'use Data::Dumper;'.
  Bangs::ProhibitDebuggingModules (Severity: 4)
    This policy prohibits loading common debugging modules like the
    Data::Dumper manpage.

    While such modules are incredibly useful during development and
    debugging, they should probably not be loaded in production use. If this
    policy is violated, it probably means you forgot to remove a `use
    Data::Dumper;' line that you had added when you were debugging.
Private subroutine/method '_svn_revisions_differ' declared but not used at line 58, near 'sub
_sub_name {'.
  Subroutines::ProhibitUnusedPrivateSubroutines (Severity: 3)
    By convention Perl authors (like authors in many other languages)
    indicate private methods and variables by inserting a leading underscore
    before the identifier. This policy catches such subroutines which are
    not used in the file which declares them.

    This module defines a 'use' of a subroutine as a subroutine or method
    call to it (other than from inside the subroutine itself), a reference
    to it (i.e. `my $foo = \&_foo'), a `goto' to it outside the subroutine
    itself (i.e. `goto &_foo'), or the use of the subroutine's name as an
    even-numbered argument to `use overload'.
Backtick operator used at line 230, near 'my $filesystem_diff = join q{}, `diff $trunk_checkout
$staging_checkout`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                    #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
Backtick operator used at line 327, near 'my $output = `$cmd`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';
```

```
        @output = `some_command`;                       #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
```

Ignoring Code

There will be times when you can't comply with a Perl::Critic policy. In those cases, you can wrap special comments, "**## use critic()**" and "**## no critic**", around your code to make Perl::Critic ignore them. Simply add the rules you want to ignore in the parentheses (multiples can be separated by a comma).

```
##no critic qw(InputOutput::ProhibitBacktickOperator)
my $filesystem_diff = join q{}, `diff $trunk_checkout $staging_checkout`;
## use critic
```

Make sure to wrap the entire code block or Critic may not recognize the ignore statement.

```
## no critic (Subroutines::ProhibitExcessComplexity)
sub no_time_to_refactor_this {
    ...
}
## use critic
```

Note that there are certain policies that are run on the document level and cannot be exempted this way. However, they can be turned off...

Creating Permanent Exceptions

Using ## no critic() is nice, but as you start to adopt coding standards, you will likely want to make permanent exceptions to certain rules. You can do this by creating a **.perlcriticrc** configuration file.

This file will allow you to customize not only which policies are run, but how they are run. Using it is as simple as placing the file in your home directory (in Linux, unsure if it's the same place on Windows). Or, you can specify the config file when running the command using the **--profile** option:

```
perlcritic -1 --profile=/path/to/.perlcriticrc /path/to/script.pl
```

Again, the perlcritic CPAN page has a full list of these options. I will list some examples from my own config file:

Apply basic settings:

```
#very very harsh
severity = 1
color-severity-medium = bold yellow
color-severity-low = yellow
color-severity-lowest = bold blue
```

Disable a rule (note the dash in front of the policy name):

```
# do not require version control numbers
[-Miscellanea::RequireRcsKeywords]

# pod spelling is too over-zealous, disabling
[-Documentation::PodSpelling]
```

Modifying a rule:

```
# do not require checking for print failure ( false positives for printing to stdout, not filehandle
)
[InputOutput::RequireCheckedSyscalls]
    functions = open close

# Allow specific unused subroutines for moose builders
[Subroutines::ProhibitUnusedPrivateSubroutines]
private_name_regex = _(?!build_)\w+
```

Conclusion

Properly utilized, Perl::Critic can be an invaluable tool to help teams keep their coding consistent and easily maintainable no matter what best practice policies you employ.

Chapter 26: Dates and Time

Section 26.1: Create new DateTime

Install `DateTime` on your PC and then use it in perl script:

```
use DateTime;
```

Create new current datetime

```
$dt = DateTime->now( time_zone => 'Asia/Ho_Chi_Minh');
```

Then you can access elements's values of date and time:

```
$year   = $dt->year;
$month  = $dt->month;
$day    = $dt->day;
$hour   = $dt->hour;
$minute = $dt->minute;
$second = $dt->second;
```

To get only time:

```
my $time = $dt->hms; #return time with format hh:mm:ss
```

To get only date:

```
my $date = $dt->ymd; #return date with format yyyy-mm-dd
```

Section 26.2: Working with elements of datetime

Set single element:

```
$dt->set( year => 2016 );
```

Set many elements:

```
$dt->set( year => 2016, 'month' => 8);
```

Add duration to datetime

```
$dt->add( hour => 1, month => 2)
```

Datetime subtraction:

```
my $dt1 = DateTime->new(
      year       => 2016,
      month      => 8,
      day        => 20,
);

my $dt2 = DateTime->new(
      year       => 2016,
      month      => 8,
      day        => 24,
);

my $duration = $dt2->subtract_datetime($dt1);
```

```
print $duration->days
```

You will get the result is 4 days

Section 26.3: Calculate code execution time

```
use Time::HiRes qw( time );

my $start = time();

#Code for which execution time is calculated
sleep(1.2);

my $end = time();

printf("Execution Time: %0.02f s\n", $end - $start);
```

This will print execution time of Code in seconds

Chapter 27: Dancer

About:

Dancer2 (the successor of Dancer) is a simple but powerful web application framework for Perl.

It is inspired by Sinatra and written by Alexis Sukrieh.

Key features: ••• Dead Simple - Intuitive, minimalist and very expressive syntax. ••• Flexible - PSGI support, plugins and modular design allow for strong scalability. ••• Few dependencies - Dancer depends on as few CPAN modules as possible making it easy to install.

Section 27.1: Easiest example

```
#!/usr/bin/env perl
use Dancer2;

get '/' => sub {
    "Hello World!"
};

dance;
```

Chapter 28: Attributed Text

Section 28.1: Printing colored Text

```
#!/usr/bin/perl

use Term::ANSIColor;

print color("cyan"), "Hello", color("red"), "\tWorld", color("green"), "\tIt's Me!\n",
color("reset");
```

Chapter 29: Dates and Time

Section 29.1: Date formatting

Time::Piece is available in perl 5 after version 10

```
use Time::Piece;

my $date = localtime->strftime('%m/%d/%Y');
print $date;
```

> Output
> 07/26/2016

Chapter 30: GUI Applications in Perl

Section 30.1: GTK Application

```
use strict;
use warnings;

use Gtk2 -init;

my $window = Gtk2::Window->new();
$window->show();

Gtk2->main();

0;
```

Chapter 31: Easy way to check installed modules on Mac and Ubuntu

Section 31.1: Use perldoc to check the Perl package install path

```
$ perldoc -l Time::Local
```

Section 31.2: Check installed perl modules via terminal

Type below command:

`instmodsh`

It'll show you the guild as below:

```
Available commands are:
   l            - List all installed modules
   m <module>   - Select a module
   q            - Quit the program
cmd?
```

Then type `l` to list all the installed modules, you can also use command `m <module>` to select the module and get its information.

After finish, just type `q` to quit.

Section 31.3: How to check Perl corelist modules

```
$ corelist -v v5.23.1
```

Chapter 32: Memory usage optimization

Section 32.1: Reading files: foreach vs. while

When reading a potentially large file, a `while` loop has a significant memory advantage over `foreach`. The following will read the file record by record (by default, "record" means "a line", as specified by `$/`), assigning each one to `$_` as it is read:

```
while(<$fh>) {
    print;
}
```

The *diamond operator* does some magic here to make sure the loop only terminates at end-of-file and not e.g. on lines that contain only a "0" character.

The following loop seems to work just the same, however it evaluates the diamond operator in list context, causing the entire file to be read in one go:

```
foreach(<$fh>) {
    print;
}
```

If you are operating on one record at a time anyway, this can result in a huge waste of memory and should thus be avoided.

Section 32.2: Processing long lists

If you have a list in memory already, the straightforward and usually sufficient way to process it is a simple `foreach` loop:

```
foreach my $item (@items) {
    ...
}
```

This is fine e.g. for the common case of doing some processing on `$item` and then writing it out to a file without keeping the data around. However, if you build up some other data structure from the items, a `while` loop is more memory efficient:

```
my @result;
while(@items) {
    my $item = shift @items;
    push @result, process_item($item);
}
```

Unless a reference to `$item` directly ends up in your result list, items you shifted off the `@items` array can be freed and the memory reused by the interpreter when you enter the next loop iteration.

Chapter 33: Perl script debugging

Section 33.1: Run script in debug mode

To run script in debug mode you should add `-d` option in the command line:

```
$perl -d script.pl
```

If `t` is specified, it indicates to the debugger that threads will be used in the code being debugged:

```
$perl -dt script.pl
```

Additional info at `perldoc`perlrun

Section 33.2: Use a nonstandard debugger

`$perl -d:MOD script.pl` runs the program under the control of a debugging, profiling, or tracing module installed as `Devel::MOD`.

For example, `-d:NYTProf` executes the program using the `Devel::NYTProf` profiler.

See all available Devel modules here

Recommended modules:

- `Devel::NYTProf` -- Powerful fast feature-rich Perl source code profiler
- `Devel::Trepan` -- A modular gdb-like Perl debugger
- `Devel::MAT` -- Perl Memory Analysis Tool
- `Devel::hdb` -- Perl debugger as a web page and REST service
- `Devel::DebugHooks::KillPrint` -- Allows to forget about debugging by `print` statement
- `Devel::REPL` -- A modern perl interactive shell
- `Devel::Cover` -- Code coverage metrics for Perl

Chapter 34: Comments

Section 34.1: Single-line comments

Single-line comments begin with a pound sign `#` and go to the end of the line:

```
# This is a comment

my $foo = "bar"; # This is also a comment
```

Section 34.2: Multi-line comments

Multi-line comments start with

=

and with the `=cut` statement. These are special comments called POD (Plain Old Documentation).

Any text between the markers will be commented out:

```
=begin comment

This is another comment.
And it spans multiple lines!

=end comment

=cut
```

Chapter 35: Randomness

Section 35.1: Accessing an array element at random

```
my @letters = ( 'a' .. 'z' );                # English ascii-bet

print $letters[ rand @letters ] for 1 .. 5;  # prints 5 letters at random
```

How it works

- `rand EXPR` expects a scalar value, so `@letters` is evaluated in scalar context
- An array in scalar context returns the number of elements it contains (26 in this case)
- `rand 26` returns a random fractional number in the interval `0 ? VALUE < 26`. (It can never be `26`)
- Array indices are always integers, so `$letters[rand @letters]` ? `$letters[int rand @letters]`
- Perl arrays are zero-indexed, so `$array[rand @array]` returns `$array[0]`, `$array[$#array]` or an element in between

(The same principle applies to hashes)

```
my %colors = ( red   => 0xFF0000,
               green => 0x00FF00,
               blue  => 0x0000FF,
             );

print ( values %colors )[rand keys %colors];
```

Section 35.2: Generate a random integer between 0 and 9

Cast your random floating-point number as an int.

Input:

```
my $range = 10;

# create random integer as low as 0 and as high as 9
my $random = int(rand($range));   # max value is up to but not equal to $range

print $random . "\n";
```

Output:

A random integer, like...

```
0
```

See also the perldoc for rand.

Chapter 36: Special variables

Section 36.1: Special variables in perl:

1. `$_` : The default input and pattern-searching space.

Example 1:

```
my @array_variable = (1 2 3 4);
foreach (@array_variable){
    print $_."\n";    # $_ will get the value 1,2,3,4 in loop, if no other variable is supplied.
}
```

Example 2:

```
while (<FH>){
    chomp($_);    # $_ refers to the iterating lines in the loop.
}
```

The following functions use `$_` as a default argument:

```
abs, alarm, chomp, chop, chr, chroot, cos, defined, eval,
evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log,
lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink,
readpipe, ref, require, reverse (in scalar context only), rmdir,
say, sin, split (for its second argument), sqrt, stat, study,
uc, ucfirst, unlink, unpack.
```

2. `@_` : This array contains the arguments passed to subroutine.

Example 1:

```
example_sub( $test1, $test2, $test3 );

sub example_sub {
    my ( $test1, $test2, $test3 ) = @_;
}
```

Within a subroutine the array `@_` contains the **arguments** passed to that subroutine. Inside a subroutine, `@_` is the `default` array for the array operators `pop` and `shift`.

Chapter 37: Sorting

For sorting lists of things, Perl has only a single function, unsurprisingly called `sort`. It is flexible enough to sort all kinds of items: numbers, strings in any number of encodings, nested data structures or objects. However, due to its flexibility, there are quite a few tricks and idioms to be learned for its use.

Section 37.1: Basic Lexical Sort

```
@sorted = sort @list;

@sorted = sort { $a cmp $b } @list;

sub compare { $a cmp $b }
@sorted = sort compare @list;
```

The three examples above do exactly the same thing. If you don't supply any comparator function or block, `sort` assumes you want the list on its right sorted lexically. This is usually the form you want if you just need your data in some predictable order and don't care about linguistic correctness.

`sort` passes pairs of items in `@list` to the comparator function, which tells `sort` which item is larger. The `cmp` operator does this for strings while `<=>` does the same thing for numbers. The comparator is called quite often, on average *n*?*?log(*n*) times with *n* being the number of elements to be sorted, so it's important it be fast. This is the reason `sort` uses predefined package global variables (`$a` and `$b`) to pass the elements to be compared to the block or function, instead of proper function parameters.

If you `use locale`, `cmp` takes locale specific collation order into account, e.g. it will sort `Å` like `A` under a Danish locale but after `Z` under an English or German one. However, it doesn't take the more complex Unicode sorting rules into account nor does it offer any control over the order—for example phone books are often sorted differently from dictionaries. For those cases, the `Unicode::Collate` and particularly `Unicode::Collate::Locale` modules are recommended.

Section 37.2: The Schwartzian Transform

This is probably the most famous example of a sort optimization making use of Perl's functional programming facilities, to be used where the sort order of items depend on an expensive function.

```
# What you would usually do
@sorted = sort { slow($a) <=> slow($b) } @list;

# What you do to make it faster
@sorted =
map { $_->[0] }
sort { $a->[1] <=> $b->[1] }
map { [ $_, slow($_) ] }
@list;
```

The trouble with the first example is that the comparator is called very often and keeps recalculating values using a slow function over and over. A typical example would be sorting file names by their file size:

```
use File::stat;
@sorted = sort { stat($a)->size <=> stat($b)->size } glob "*";
```

This works, but at best it incurs the overhead of two system calls per comparison, at worst it has to go to the disk, twice, for every single comparison, and that disk may be in an overloaded file server on the other side of the planet.

Enter Randall Schwartz's trick.

The Schwartzian Transform basically shoves `@list` through three functions, bottom-to-top. The first `map` turns each entry into a two-element list of the original item and the result of the slow function as a sort key, so at the end of this we have called `slow()` exactly once for each element. The following `sort` can then simply access the sort key by looking in the list. As we don't care about the sort keys but only need the original elements in sorted order, the final `map` throws away the two-element lists from the already-sorted list it receives from `@sort` and returns a list of only their first members.

Section 37.3: Case Insensitive Sort

The traditional technique to make `sort` ignore case is to pass strings to `lc` or `uc` for comparison:

```
@sorted = sort { lc($a) cmp lc($b) } @list;
```

This works on all versions of Perl 5 and is completely sufficient for English; it doesn't matter whether you use `uc` or `lc`. However, it presents a problem for languages like Greek or Turkish where there is no 1:1 correspondence between upper- and lowercase letters so you get different results depending on whether you use `uc` or `lc`. Therefore, Perl 5.16 and higher have a *case folding* function called `fc` that avoids this problem, so modern multi-lingual sorting should use this:

```
@sorted = sort { fc($a) cmp fc($b) } @list;
```

Section 37.4: Numeric Sort

```
@sorted = sort { $a <=> $b } @list;
```

Comparing `$a` and `$b` with the `<=>` operator ensures they are compared numerically and not textually as per default.

Section 37.5: Reverse Sort

```
@sorted = sort { $b <=> $a } @list;
@sorted = reverse sort { $a <=> $b } @list;
```

Sorting items in descending order can simply be achieved by swapping `$a` and `$b` in the comparator block. However, some people prefer the clarity of a separate `reverse` even though it is slightly slower.

Chapter 38: Perlbrew

Perlbrew is a tool to manage multiple perl installations in your `$HOME` directory.

Section 38.1: Setup perlbrew for the first time

Create setup script `~/.perlbrew.sh`:

```
# Reset any environment variables that could confuse `perlbrew`:
export PERL_LOCAL_LIB_ROOT=
export PERL_MB_OPT=
export PERL_MM_OPT=

# decide where you want to install perlbrew:
export PERLBREW_ROOT=~/perlbrew
[[ -f "$PERLBREW_ROOT/etc/bashrc" ]] && source "$PERLBREW_ROOT/etc/bashrc"
```

Create installation script `install_perlbrew.sh`:

```
source ~/.perlbrew.sh
curl -L https://install.perlbrew.pl | bash
source "$PERLBREW_ROOT/etc/bashrc"

# Decide which version you would like to install:
version=perl-5.24.1
perlbrew install "$version"
perlbrew install-cpanm
perlbrew switch  "$version"
```

Run installation script:

```
./install_perlbrew.sh
```

Add to the end of your `~/.bashrc`

```
[[ -f ~/.perlbrew.sh ]] && source ~/.perlbrew.sh
```

Source `~/.bashrc`:

```
source ~/.bashrc
```

Chapter 39: Installation of Perl

I'm going to begin this with the process in Ubuntu, then in OS X and finally in Windows. I haven't tested it on all perl versions, but it should be a similar process.

Use Perlbrew if you like to switch easily beween different versions of Perl.

I want to state that this tutorial is about Perl in it's open-source version. There are other versions like `activeperl` which its advantages and disadvantages, that are not part of this tutorial.

Section 39.1: Linux

There is more than one way to do it:

- Using the package manager:

```
sudo apt install perl
```

- Installing from source:

```
wget http://www.cpan.org/src/5.0/perl-version.tar.gz
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

- Installing in your $home directory (not sudo needed) with Perlbrew:

```
wget -O - https://install.perlbrew.pl | bash
```

  See also Perlbrew

Section 39.2: OS X

There are several options:

- Perlbrew:

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

- Perlbrew with thread support:

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

  After the install of perlbrew, if you want to install Perl with thread support, just run:

```
perlbrew install -v perl-5.26.0 -Dusethreads
```

- From source:

```
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

Section 39.3: Windows

- As we said before, we go with the open-source version. For Windows you can choose `strawberry` or `DWIM`. Here we cover the `strawberry` version, since `DWIM` is based on it. The easy way here is installing from the official executable.

See also berrybrew - the perlbrew for Windows Strawberry Perl

Chapter 40: Compile Perl cpan module sapnwrfc from source code

I'd like to describe the prerequisites and the steps how to build the Perl CPAN module sapnwrfc with the Strawberry Perl environment under Windows 7 x64. It should work also for all later Windows versions like 8, 8.1 and 10.

I use Strawberry Perl 5.24.1.1 64 bit but it should also work with older versions.

It took me some hourse to succeed with several tries (32 vs. 64 bit installation of Perl, SAP NW RFC SDK, MinGW vs. Microsoft C compiler). So I hope some will benefit from my findings.

Section 40.1: Simple example to test the RFC connection

Simple example from http://search.cpan.org/dist/sapnwrfc/sapnwrfc-cookbook.pod

```
use strict;
use warnings;
use utf8;
use sapnwrfc;

SAPNW::Rfc->load_config('sap.yml');
my $conn = SAPNW::Rfc->rfc_connect;

my $rd = $conn->function_lookup("RPY_PROGRAM_READ");
my $rc = $rd->create_function_call;
$rc->PROGRAM_NAME("SAPLGRFC");

eval {
$rc->invoke;
};
if ($@) {
    die "RFC Error: $@\n";
}

print "Program name: ".$rc->PROG_INF->{'PROGNAME'}."\n";
my $cnt_lines_with_text = scalar grep(/LGRFCUXX/, map { $_->{LINE} } @{$rc->SOURCE_EXTENDED});
$conn->disconnect;
```

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

AbhiNickz	Chapter 36
Al.G.	Chapter 14
Alien Life Form	Chapter 8
AntonH	Chapter 2
Ataul Haque	Chapter 5
badp	Chapters 3, 4, 6, 9, 11 and 13
Bill the Lizard	Chapter 3
brian d foy	Chapter 20
callyalater	Chapter 21
Chankey Pathak	Chapter 27
Christopher Bottoms	Chapters 1, 2, 3, 4, 5, 7, 11, 16, 20, 21, 34 and 35
datageist	Chapter 1
Denis Ibaev	Chapter 1
digitalis_	Chapters 4, 6 and 11
Dmitry Egorov	Chapters 13 and 18
dmvrtx	Chapter 6
Drav Sloan	Chapter 17
DVK	Chapters 12 and 20
eballes	Chapter 17
eddy85br	Chapter 1
Eugen Konkov	Chapters 1, 6, 18, 20 and 33
fanlim	Chapters 31 and 39
flamey	Chapter 39
flotux	Chapter 40
Håkon Hægland	Chapters 6, 7, 19 and 38
ikegami	Chapter 3
interduo	Chapter 25
Iván Rodríguez Torres	Chapter 39
Jean	Chapter 15
Jeff Y	Chapter 8
John Hart	Chapter 2
Jon Ericson	Chapters 1, 2, 6 and 14
Kemi	Chapters 7, 8, 16, 18 and 19
Kent Fredric	Chapters 2, 3, 6, 7, 10, 11 and 19
Leon Timmermans	Chapter 1
lepe	Chapter 2
luistm	Chapters 16 and 39
matt freake	Chapter 7
mbethke	Chapters 2, 3, 4, 6, 8, 10, 18, 19, 32 and 37
Michael Carman	Chapter 4
Mik	Chapter 6
msh210	Chapters 3 and 4
Muaaz Rafi	Chapter 20
Nagaraju	Chapter 7
nfanta	Chapter 6
Ngoan Tran	Chapters 16, 23, 26 and 31
nslntmnx	Chapter 24
oals	Chapters 6, 21 and 22
oldtechaa	Chapter 30

Ole Tange	Chapter 3
Otterbein	Chapter 6
pwes	Chapters 8 and 20
Randall	Chapter 16
rbennett485	Chapter 7
Rebecca Close	Chapters 34 and 35
reflective_mind	Chapter 20
Rick James	Chapter 20
Ruslan Batdalov	Chapters 13 and 22
SajithP	Chapters 7, 14, 28 and 29
Sarwesh Suman	Chapter 14
Sebi	Chapter 7
serenesat	Chapter 12
simbabque	Chapter 5
Sobrique	Chapter 17
SREagle	Chapters 7 and 8
Stephen Leppik	Chapter 14
svarog	Chapter 11
Tim Hallyburton	Chapter 7
waghso	Chapters 5 and 26
Wolf	Chapters 4 and 20
xfix	Chapters 3 and 4
xtreak	Chapter 4
yonyon100	Chapter 7
Zaid	Chapter 35
zb226	Chapters 6 and 18
□□□	Chapter 31

You may also like