Introduction to Perl for Programmers

Damian Conway

Table of Contents

Course information

- **Summary**:

  In this class you will learn the fundamentals of programming in Perl,
  with emphasis on the linguistic underpinnings of the many features of the language.
  In particular, you will gain deeper insight into the theory and practical use of one of
  Perl's most compelling components: regular expressions. We will also look briefly at
  the new Raku programming language, which evolved from Perl, and discuss
  the language design principles that make it so different from its progenitor.

- **Critical dates:**
  - Lectures and exercises: 31 August to 11 September, 2020
  - Tutorials and Q&A sessions: 1, 3, 7, 9, and 11 September, 2020
  - Entrance exam: 31 August, 2020
  - Final project: 1 September to 25 September, 2020

- **Instructor:**

  Dr Damian Conway
  `damian.conway@fhnw.ch`

  Due to the current pandemic, it is not possible for Dr Conway to leave Australia,
  so the class will be delivered on-line via recorded lectures
  and live video meetings (via Zoom).

- **Teaching language:**

  English
  (Dr Conway does not speak German, French, or Italian)

- **Reading:**
  - ***Required preliminary reading:***

    These class notes

  - ***Recommended textbook:***

    "Modern Perl, 4th edition", chromatic, 2015, Onyx Neon
    Familiarity with chapters 1 to 6 and 8 to 9 is **strongly recommended** for entrance exam.
    Available as a free PDF:
    `https://pragprog.com/titles/swperl/`
    Available as free HTML:
    `http://modernperlbooks.com/books/modern_perl_2016/`

  - ***Other recommended reading and learning resources:***

    See Appendix E

- **Lectures:**
    - Lectures will consist of pre-recorded presentations by Dr Conway.
    - These may be viewed (and re-viewed) on SWITCHtube at any time convenient to the student during the two class weeks: 31 August to 11 September, 2020.
    - Questions and requests for clarification or further information from Dr Conway may be made during tutorial sessions, or via email (with an expected 24-hour turnaround).

- **Exercises:**
    - Exercises will not be assessed.
    - Exercises may be attempted in any order, and whenever the student wishes (though a suggested schedule will be supplied as part of the recorded lectures).
    - Students are encouraged to discuss these problems amongst themselves and to solve them collaboratively, if they wish.
    - Sample solutions will be provided for each exercise.
    - Feedback and assistance from Dr Conway will be available during tutorial sessions, or via email (with an expected 24-hour turnaround).

- **Tutorials:**
    - Tutorials will not be directly assessed and are not compulsory, however attendance and active participation in all five tutorial sessions is expected.
    - Five live-streamed 40-minute sessions with Dr Conway, via fhnw.zoom.us (sign in using SSO with your fhnw.ch email address).
    - Tutorials will consist of general question-and-answer discussion.
    - Dr Conway will also review particular class topics, as time permits.
    - Students will be randomly assigned into one of three tutorial groups.
    - Tutorials will start at 08:00 or 10:00 or 12:00 CEST *(your tutorial group will meet at the same time each tutorial day)*.
    - Tutorial dates: 1, 3, 7, 9, and 11 September, 2020
    - Complete login details for your tutorial group Zoom meetings will be sent prior to the commencement of the class.

- **Assessment:**
  - *Grading*:
    - Final grade in the range 1.0 to 6.0
      (Sum of raw exam and project scores, then rounded to one decimal place)

  - *Entry exam:*
    - 20% of final assessment
    - Open book, but students must not collaborate with any other person
    - 30 multiple-choice questions
    - Exam will be available from 09:00 CEST on 31 August 2020
    - Distributed and returned electronically (via email)
    - Students will complete the exam in their own time on their own machine
    - The completed exam to be emailed to Dr Conway
    - **Exam must be submitted to Dr Conway by email no later than 23:59 CEST on 31 August 2020**

  - *Final project:*
    - 80% of final assessment
    - Duration: 40 hours (design, implementation, testing, and documentation)
    - Students may either complete the project on their own or may work on it in a small project group of two or three people.
    - Students who elect to work in a group will all receive the same group grade for their project (and the submitted software and documentation will be assessed against a higher standard when two or more people have collaborated on it).
    - Project description will be available from 09:00 CEST on 1 September 2020
    - **Final project source code, test suite, and documentation must be submitted by email to Dr Conway no later than 23:59 CEST on 25 September 2020**

0. What Is Perl?

- Perl is a free, high-level, interpreted programming language
- Oh, and it's "Perl", not "PERL"
- Unless we're referring to the implementation of the interpreter, which is `perl`

A Brief History of Perl

- Originally designed as a data-extraction and report-generating replacement for `awk`
- Perl 1: December 1987: Offered variables, file I/O, subroutines, loops, simple pattern matching
- Perl 2: June 1988: Added sophisticated pattern matching, recursion, interpolation, iteration, warnings
- Perl 3: October 1989: Added GPL, pass-by-reference arguments, binary data processing
- Perl 4: May 1991: Added contexts, access to call stack information
- Perl 5: October 1994: Added object orientation, lexical scoping, modules, embedded documentation, closures, signal handlers, operator overloading, threads, exception handling, Unicode support
- Perl 5.8: July 2002: Reworked most of the internals, provided much better Unicode support, usable thread model, significant increase in the number of standard modules
- Perl 5.10: December 2007: Added better forward compatibility of new features, defined-or operator, switch control structure, smartmatching, recursive pattern matching, named submatch captures, named subpatterns, many other pattern matching features and improvements, static variables
- Perl 5.14: May 2011: Added Unicode 6.0 support, fully re-entrant pattern matching, package blocks, safer exceptions, built-in I/O is fully OO
- Perl 5.20: May 2014: Added subroutine parameter lists, KV slicing, postfix dereferencing, significant performance and memory improvements
- Perl 5.24: May 2016: Added Unicode 8.0 support, many performance improvements, many internal security improvements.
- Raku (formerly: Perl 6): December 2016: A completely separate language in the Perl family; not the successor to Perl 5, but a "sister language". Akin to the C vs C++ divergence (except that Raku doesn't suck all hope out of life the way C++ does).
- Perl 5.32: June 2020: Added Unicode 13.0 support, many more performance and internal security improvements. Aliasing via references, indented heredocs, state container initialization, lexical subroutines, `isa` operator, chained comparisons. The latest stable release.

Why learn Perl?

- Makes easy things trivial
- Makes (many) hard things easy
- Makes (some) impossible things merely hard
- Many programmers find it "fits their brain" very well
- Optimized for getting stuff done
- Highly portable to 100's of systems: A/UX, Acorn RISCOS, AIX, AmigaOS, AOS, BeOS, BSD/OS, ConvexOS, CX/UX, Cygwin, DC/OSx, DDE SMES, DG/UX, DomainOS, DOS DJGPP, DOS EMX, Dynix, DYNIX/ptx, EP/IX, EPOC R5, ESIX, FPS, FreeBSD, GENIX, Greenhills, HP-UX, Hurd, IRIX, ISC, Linux, LynxOS, LynxOS, Mac OS Classic, Mac OS X, MachTen, MachTen 68k, MiNT, MPC, MPE/iX, NetBSD, NetWare, NEWS-OS, NextSTEP, NonStop-UX, OpenBSD, OpenSTEP, OpenVMS, Opus, OS/2, OS/400, Plan 9, POSIX-BC, PowerMAX, PowerUX, QNX, ReliantUNIX, RISC/os, SCO ODT/OSR, SCO SV, Solaris, Stellar, SunOS 4, SUPER-UX, SVR2, SVR4, Tandem Guardian, TI1500, TitanOS, Tru64 UNIX, Ultrix, UNICOS, Unisys Dynix, Unixware, UTS, VM/ESA, VOS, WinCE, Window XP, Windows 2K, Windows 3.1, Windows 95, Windows 98, Windows ME, z/OS, 3b1
- A huge public archive of free software, known as CPAN (see Appendix A)

Perl Slogans

- "There Is More Than One Way To Do It (TIMTOWTDI)
- "Easy things should be easy; Hard things should be possible
- "The three virtues of the programmer: Laziness, Impatience, Hubris"
- "Perl: The Swiss Army Chainsaw"

Getting Perl

- You almost certainly already have it
- Binary distributions available for many platforms
- Start at: http://www.cpan.org/ports/
- Otherwise download source code from a local CPAN mirror
- Start at: http://www.cpan.org/src/README.html
- Better still, use `perlbrew` or `berrybrew` (on Windows) to install and switch between multiple versions of Perl (without messing up your system's own Perl installation)
- Start at: https://perlbrew.pl/
  or: https://github.com/dnmfarrell/berrybrew

Setting up your environment

- On many systems these are unnecessary (or already preset)
- The `PATH` environment variable: where to look for `perl` when it's run
- The `PERL5LIB` or `PERLLIB` environment variables: where to look for additional Perl modules that the program may use
- The `PERL5OPT` environment variable: command-line options to use every time `perl` is run
- The `PERL5SHELL` environment variable (Windows ports only): specifies which command interpreter is used for embedded system commands

Running Perl

- Depending on environment, Perl may be run from command-line (all command-line OSes) or as a separate application (Windows, Macintosh)
- Command-line form:

```
> perl [perl options] program.pl [program args]
```

- The `.pl` suffix is commonly used to denote a Perl program
- Because Perl is JIT-compiled, on systems that support shebang lines, you can also turn programs into scripts:

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

# your program here
```

- Then just:

```
> program.pl [program args]
```

Understanding Perl

- The Perl distribution includes a large amount of documentation: over 160 separate files comprising over 3000 pages of reference and tutorial information
- The documentation is accessible from the command-line via the `perldoc` program (which comes with Perl)
- The documentation is accessable on-line at: http://perldoc.perl.org/
- The best place to start is:

```
> perldoc perl
```

"Hello world!"

- Enough background!
- Let's write some Perl:

```
use v5.32;

say "Hello, world!";
```

"Hello Worlds"

- Boring! Everyone starts with that. How about:

```
use v5.32;

my %worlds = (
        Mercury  => "Fleet in the heat",
        Venus    => "Beauty veiled in mystery",
        Earth    => "Cradle of life",
        Mars     => "Bringer of war",
        Jupiter  => "Lord of the system",
        Saturn   => "Ringéd wonder",
        Neptune  => "The not-particularly interesting",
        Uranus   => "Bringer of rude puns",
        Pluto    => "...is no longer a planet!",
);

for my $planet (sort keys %worlds) {
    say "Hello, $planet: $worlds{$planet}";
}
```

- Perl has a large number of legacy "short-cuts" from its origins as a quick-and-dirty scripting language; features that can seem incredibly convenient in small programs…
- ...but which can lead to nasty, hard-to-find bugs in larger code:

```
# Evil old-style unsafe Perl code...

%worlds = (
        Mercury  => hot,
        Venus    => cloudy,
        Earth    => comfy,
        Mars     => red,
        Jupiter  => big,
        Saturn   => pretty,
        Neptune  => cold,
        Uranus   => rude,
        Pluto    => banished,
);

for $planet (sort keys %worlds) {
    say "$plant: $worlds{$planet}";
}
```

Strictures

- The `use strict` specifier turns many of these dubious historical conveniences off, so it's a very good idea to turn it on in every program you write:

```
use strict;

# Evil old-style unsafe Perl code that now won't even compile!

%worlds = (                             # error: undeclared variable
        Mercury  => hot,                # error: bareword
        Venus    => cloudy,             # error: bareword
        Earth    => comfy,              # error: bareword
        Mars     => red,                # error: bareword
        Jupiter  => big,                # error: bareword
        Saturn   => pretty,             # error: bareword
        Neptune  => cold,               # error: bareword
        Uranus   => rude,               # error: bareword
        Pluto    => banished,           # error: bareword
);

for $planet (sort keys %worlds) {
    say "$plant: $worlds{$planet}";     # error: undeclared variable
}
```

- We'll examine the specifics of how `use strict` helps as we encounter the particular (mis)features it defuses.

Explicit versioning

- Note that, if you explicitly declare which version of Perl your code was developed under:

```
use v5.32;
```

- …then `use strict` is turned on automatically
- …provided that version is v5.12 or later
- Explicitly declaring your version also automatically turns on all the forward-compatibility features available in that version, so it's an excellent habit to get into

Warnings

- Perl is a very accommodating language in which to program
- But there are plenty of traps that even very experienced Perl programmers can easily fall into
- (Some of which spring specifically from that over-helpfulness)
- So it's a good idea to run with Perl's built-in warning system enabled
- Perl can detect and report nearly 900 types of potential mistakes
- You can read about them all in the `perldiag` documentation
- The easiest way to tell Perl to watch for problems is to run `perl` with warnings enabled
- Add a `use warnings` statement to turn them on
- Or run Perl with the `-w` flag in the command-line or in the shebang: `perl -w`
- You can also specify a `no warnings` statement to turn them off again
- You can even enable/disable specific types of warnings:

```
use warnings 'io';   # Report I/O related warnings

no warnings 'once';  # Don't report "var used only once" warnings
```

- Note that the effect of any of these specifications is lexical: from its point to declaration to end of the surrounding block or file
- So they're very handy for honing in on specific problems, or for tuning out things that Perl thinks might be a problem but which you're confident are okay

Diagnostics

- The only problem with warnings is that they can occasionally be rather terse or opaque
- If we add a `use diagnostics` statement to our program, any error is augmented with a useful description
- The description is taken from the `perldiag` documentation

The standard safety prologue

- Give the above recommendations, every Perl program you write should almost certainly start with:

```
use v5.32;      # Or whatever version you're developing under
use strict;
use warnings;
```

- And, for the first few months at least, maybe add this as well:

```
use v5.32;      # Or whatever version you're developing under
use strict;
use warnings;
use diagnostics;
```

- Note that the `use strict` isn't actually necessary there (as long as your version number is v5.12 or later), but being explicit about it anyway is always a good habit

1. Basic components of Perl programs

Comments

- Comments are line-based
- They start with an octothorpe: #
- They end with a newline
- …just like shell or Python comments

```
say "Hello, world!";    # Initiate geek greeting ritual
```

- Perl does not provide either delimited or multi-line comments (such as `/*…*/`)
- However, you *can* comment out a chunk of code using Perl's built-in documentation syntax (see Appendix B)

Scalar variables

- A scalar variable is able to store a single value…
- ...of any standard Perl type
- A number
- A character string
- A reference (*i.e.* a pointer)

- The value of a variable is accessed or assigned by giving its name…
- ...preceded by a dollar sign:

```
$unit             = "km";
$earth_diameter   = 12734.89;
$height           = 18;
$horizon_distance = sqrt($height * $earth_diameter);
```

- The dollar sign is known as a *"sigil"*
- There are several others (as we'll see later)
- The name can be *any* length (provided: 0 < *any* < 252)
- But restricted alphabetical characters, digits, and underscores
- And can't start with a digit
- Case matters
- Scalar variables have no fixed (or *"static"*) type
- Dynamically take on the type of the value they are currently storing
- It's not strictly necessary to predeclare a scalar variable before using it
- But we can if we want to (and we always should!):

```
my    $unit;               # lexical variable
our   $earth_diameter;     # global variable
state $karma;              # static variable
```

- Variables can also be initialized:

```
my    $unit           = "km";
our   $earth_diameter = 12734.89;
state $karma          = -42;
```

Lexicals vs global vs static variables

- The three keywords – `my` and `our` and `state` – declare three distinct kinds of scalar variable
- They all have the same storage behaviours;
  they all have different scope, initialization, and duration
- `my` means: *"lexically scoped, visible to the end of surrounding block, initialized every time execution reaches the declaration, and deallocated when execution reaches the end of it";*
- `our` means *"global, visible throughout the current namespace, initialized every time execution reaches the declaration, and deallocated at the end of the program"*
- `state` means: *"lexically scoped, visible to the end of surrounding block, initialized only the first time execution reaches the declaration, and deallocated at the end of the program";*
- For now, we'll just use `my`

Declaring multiple variables

- We can create two or more variables at once
- Just list them – in parentheses – after the `my`:

```
my ($name, $age, $shoesize);
```

- Can also initialize them all at once:

```
my ($name, $age, $shoesize) = ("Damian", 38, 9.5);
```

- Whitespace almost never matters in Perl, so sometimes we might prefer to write that:

```
my ($name,     $age,  $shoesize)
 = ("Damian",  38,    9.5      );
```

- ***Note: The parens are essential around both the variable list and the initializer list***

Requiring strictly declared variables

- By default, some kinds of Perl variables (*e.g.* `our` variables) are created "on demand"
- That is, if you use a variable without declaring it, Perl just autogenerates it as a global variable
- But that means it's frighteningly easy to write something like this:

```
my $nextline = readline();

if (substr($next_line,0,1) eq '#') {
    say "Skipping comment";
}
elsif ($next_line eq 'END' ) {
    exit(1);
}
```

- The `if` statements will be testing the variable `$next_line`,
  which has nothing to do with the variable `$nextline`
- Even if `$next_line` doesn't exist at the point where the `if` first tests it, there won't be a warning or error; Perl will simply create a global variable of that name on-the-fly
- But we could add a `use strict 'vars'` to the program:

```
use strict 'vars';

my $nextline = readline();

if (substr($next_line,0,1) eq '#') {   # Error: undeclared variable
    say "Skipping comment";
}
elsif ($next_line eq 'END' ) {         # Error: undeclared variable
    exit(1);
}
```

- That causes the compiler to fail if any variable is accessed but wasn't explicitly declared
- Note that specifying `use strict` automatically implies `use strict 'vars'` (as well as other useful safeguards) so it's always best to always just use the general form

Special variables

- A.K.A. "Punctuation variables"
- Instead of a name after the dollar sign, they have a punctuation character
- (Yeah, we know, it's a really stupid idea…but just wait: it gets worse)
- `$_ $+ $/ $] $# $$` *etc. etc.*
- Perl has a *lot* of these
- Each one globally controls some kind of special behaviour
- For example, normally the Perl `readline` function reads up to a newline character:

```
$next_line = readline();
```

- But we can change the character that it considers an end-of-line marker, by changing the value of the global `$/` variable:

```
$/ = " ";
$next_word = readline();    # Read to first space character

$/ = ",";
$next_CSV = readline();     # Read to first comma
```

- Read all about the special variables in the `perlvar` documentation that comes with Perl
- ***Note: in general, avoid using special variables, and especially avoid changing their values, which injects global changes in core behaviours via an obscure, hard-to-understand syntax***

Strings

- Perl provides a built-in variable-length character string type
- It is **not** null-delimited (so we can have nulls in our strings, anywhere we need them)
- Two syntaxes:

```
$str = "This is a string";

$str = 'This is a string too';
```

- The difference is that `"..."` strings interpolate variables, while `'...'` strings don't

String interpolation

- Interpolation allows variables to be placed inside strings, whereupon the contents of those variables are expanded into the string in place of the variable name
- For example, that means that the string:

```
"the value $val is next"
```

- ...consists of the characters `'t','h','e',' ','v','a','l','u','e',' '`...
- ...followed by the contents of `$val` converted to a string…
- ...followed by the characters `' ','i','s',' ','n','e','x','t'`
- This variable expansion doesn't happen in single-quoted strings
- So the string:

```
'the value $val is next'
```

- ...just consists of the characters `'t','h','e',' ','v','a','l','u','e',' ','$','v','a','l',' ','i','s',' ','n','e','x','t'`

String escapes

- Apart from variables, there are other special non-literal sequences that are recognized within interpolating strings
- Most characters in a string represent themselves
- But Perl also offers C-like support for specifying certain special characters in strings in interpolated (`"..."`) strings
- The special characters include:

```
"\t is a tab"
"\n is a newline"
"\r is a carriage return"
"\f is a form feed"
"\[c is a control-C"
"\[z is a control-Z"
"\e is an escape"
"\033 is also an escape (specified by its octal ASCII code)"
"\x1b is also an escape (specified by its hexadecimal ASCII code)"
"\x{2625} is Unicode ankh character (specified by its code point)"
"\N{ANKH} is also a Unicode ankh character (specified by name)"
```

String concatenation

- Perl uses the "dot" operator (`.`) to concatenate strings:

```
$full_name = $first_name . " " . $initial . " " . $family_name;
```

- Though it's often easier and cleaner to use interpolation instead:

```
$full_name = "$first_name $initial $family_name";
```

- There's no difference in performance (interpolation actually uses "dot" internally)
- The "dot" operator is needed when concatenating strings generated by subroutine calls:

```
$full_name = get_first_name() . " "
           . get_initial()    . " "
           . get_family_name();
```

- There's also an assignment version of the "dot" operator, which appends to a string already in a variable:

```
$full_name  = $family_name;
if ($first_name ne "") {
    $full_name .= ", $first_name";
    if ($initial ne "") {
        $full_name .= " $initial";
    }
}
```

Variant string syntaxes

- It's often annoying to write a string with quote characters in it
- You have to use \ to force Perl to interpret the quote as a quote character, rather than as a string delimiter:

```
$str = "$name said \"Did you ask 'What's \"interpolation\"'\"";

$str = 'Brian said "Did you ask \'What\'s "interpolation"\'"';
```

- So there are also two "bracketed" versions of the quote markers:

```
$str = qq{$name said "Did you ask 'What's "interpolation"'"};

$str =  q{Brian said "Did you ask 'What's "interpolation"'"};
```

- You can use any brackets you like, so long as they match:

```
$str = qq($name said "Did you ask 'What's "interpolation"'");

$str = qq[$name said "Did you ask 'What's "interpolation"'"];

$str = qq<$name said "Did you ask 'What's "interpolation"'">;
```

- You can also use a non-bracketing character, so long as it's used at both ends:

```
$str = qq/$name said "Did you ask 'What's "interpolation"'"/;

$str = qq!$name said "Did you ask 'What's "interpolation"'"!;

$str = qq#$name said "Did you ask 'What's "interpolation"'"#;
```

- But it's probably best to stick with a pair of brackets

Substrings

- Character strings *aren't* arrays (they way they are in C or C++)
- So we can't access individual characters by subscripting
- Instead we use the `substr` function:

```
$first_char     = substr($str,  0, 1);

$second_char    = substr($str,  1, 1);

$first_2_chars  = substr($str,  0, 2);

$chars_4_to_7   = substr($str,  3, 4);

$last_char      = substr($str, -1, 1);

$last_char      = substr($str, -1);

$last_six_chars = substr($str, -6);
```

- A negative index means "...counting back from the end"
  (this is a common idiom in many access functions…as we'll see later)
- `substr` is unusual in that it returns an *alias* for the actual substring (rather than a copy)
- That is, we can assign a new substring over an old one:

```
if (substr($URL,0,9) eq "link:www.") {
      substr($URL,0,5) = "http://";
}
```

- As the above example implies, the string will automatically grow or shrink as necessary to accommodate the changed substring

String length

- The `length` function returns the number of characters in a string
- That's not necessarily the number of bytes (if string contains Unicode data)

String formatting

- Perl provides a `sprintf` function that is (largely) the same as C's `sprintf`
- It takes a format string
- …and zero or more data strings
- …and then interpolates the data into the format:

```
$message = sprintf("We made %6.2f profit on %08d transactions",
                           $profit,        $trans_count      );

say $message;
```

- Also `printf`:

```
printf("We made %6.2f profit on %08d transactions",
               $profit,        $trans_count      );
```

- Perl has no `sscanf` or `scanf` (more on that shortly)

Case changes

- The `lc` function converts all characters in its string argument to lower case
- The `lcfirst` function converts only the first character of its argument to lower case
- The `uc` function converts all characters to upper case
- The `ucfirst` function: converts only the first character to upper case
- The `fc` function converts all characters to Unicode case-folded lower case
- A common usage is:

```
say ucfirst lc $sentence;
```

Searching a string

- Perl provides a way to find the index (from zero) of the first occurrence of a particular character in a particular string
- The `index` function:

```
$index1 = index($str, "c");    # Returns index of first "c" in $str
```

- If the specified character doesn't appear in the string at all, `index` returns `-1`
- So a typical usage is:

```
if (index($str, $target_char) >= 0) {
    say qq("$str" contains '$target_char');
}
```

- And a typical mistake is:

```
if (index($str, $target_char)) {    # Oops!
    say qq("$str" contains '$target_char');
}
```

- This second version fails to detect the target character if it appears at the very beginning of the string (*i.e.* at index zero)
- The `index` function's target (*i.e.* its second argument) doesn't have to be a single character
- It can be a string of any length
- (Recall that a single character in Perl is really just a one-character string)
- For example:

```
if (index($message, "Nigeria") >= 0) {
    warn "Probable spam:\n$message";
}
else {
    say $message;
}
```

- There is also an `rindex` function ("reverse index") that returns the index of the final occurrence of a specified substring

Numbers

- Integers: `12345`, `+12345`, `-12345`
- Floating point: `1.2345`, `-12.345`, `.2345`, `1`
- Exponent notation: `1e2`, `1.e2`, `.1e2`, `1.1e2`, `1.1e+2`, `1.0e-2`
- The `e` is shorthand for "...by ten to the power of..."
- Can also put underscores in numbers (they're ignored):

```
$operating_capital = 120_000_000_000;

$nano_pi = 314_159_265_358_979_323_846e-1_000_000;
```

Integers in other bases

- Binary: `0b101011`
- Octal: `07707`
- Hexadecimal: `0xFFA2`

Converting floating point to integer

- The `int` function
- Returns the integer component of a number
- Hence, it always rounds towards zero:

```
say int(3.1415);      # outputs: 3

say int(-3.1415);     # outputs: -3
```

- For rounding use the `ceil` and `floor` functions from the standard POSIX module

Mathematical operations

- All of the common ones are built into the language
  (no need to load a library to use them)
- The `abs`, `sqrt`, `log`, `exp`, `sin`, *etc*. functions
- Magnitude, square root, logarithm (base 10), e to the power of..., trigonometric sine, *etc*.

Truths and falsehoods

- No specific boolean values in Perl
- Like C/C++, Perl uses zero for *false*, and `1` for *true*
- Also uses the string `"0"` for *false*
- Also uses the empty string for *false*
- Also uses an empty list for *false*
- Also uses every other value for *true*
- Works a lot better than you might expect
- Typically Perl itself uses `1` for *true* and the empty string for *false*

Negation

- Uses `!` to represent the unary prefix NOT operator
- Always returns `1` if its argument false
- Always returns `""` if its argument true
- ***Note: ...but you probably shouldn't rely on those particular return values***

Comparisons

- Perl has 12 comparison operators that yield boolean values

```
$x <  $y        $a lt $b
$x >  $y        $a gt $b
$x <= $y        $a le $b
$x >= $y        $a ge $b
$x == $y        $a eq $b
$x != $y        $a ne $b
```

- The operators built out of symbols always perform numeric comparison;
  the operatots built out of alphabetics always perform string comparison
- There are also two *"3-way comparison"* operators that yield integers

```
$x <=> $y       $a cmp $b
```

- These last two return `-1` (if the first operand is less than the second operand), `+1` (if the first operand is greater than the second), or zero (if the operands are equal)
- We'll see later how that's sometimes useful
- ***Note: Don't use == (numeric equality) when you mean `eq` (string equality)***
- ***Note: Don't use = (assignment) when you mean == or `eq` (equality)***

Definedness of variables and values

- Variables that are not explicitly initialized are implicitly initialized
  to a special "undefined" value
- You can get your own copy of that value by calling the `undef` function:

```
if (bad($value)) {
    $value = undef;
}
```

- If the `undef` function is passed one or more variable arguments, it sets their values to the undefined value:

```
if (bad($value)) {
    undef $value;       # Same effect as previous example
}
```

- We can test whether a variable contains a defined value with the built-in `defined` function:

```
if (defined($value)) {
    say "The value was $value";
}
else {
    say "Bad value! No biscuit!";
}
```

- The `undef` value is often used as a function return value indicating failure of the function
- That way, the function can successfully return an empty string or the value zero as well
- For example:

```
$count = get_count();

if (!defined $count) {
     say "Couldn't get count";
     exit(1);
}
```

String-to-number conversions

- Whenever we use a string value somewhere that a number is expected, the string is automatically converted to a number
- Whenever we use a numeric value somewhere that a string is expected, the number is automatically converted to a string
- May sound like a recipe for disaster, but it rarely is
- Most of the time it just *DWYM*s ("Does What You Mean"):

```
my $input = readline();

say sqrt($input);
```

2. Advanced components of Perl

Statements

- Very like statements in other procedural languages
- As in C/C++, statements are really just expressions whose side effects and result we are conveniently ignoring
- All statements in Perl are *separated* by a semi-colon, or *terminated* by the end of a block (or both)
- ***Note: get into the habit of terminating every Perl statement with a semi-colon, even if it's not necessary; that way you will prevent subtle bugs from being injected later when that code is maintained***

Issuing warnings

- In addition to the warnings generated by the `use warnings` pragma, we can raise our own warnings too, using the `warn` function
- It takes one or more arguments and prints them (just like `say`)
- But also adds a line and file report and a newline to the end of the message:

```
if ($count!=0) {
    $average = $sum / $count;
}
else {
    $average = 0;
    warn "No values to average";
}
```

- If `$count` was zero, we would get:

```
No values to average at prog.pl line 6.
```

- ***Note: An explicit `warn` is not affected by they `no warnings` statement***

Warning with extreme prejudice

- Sometimes a warning isn't enough
- Sometimes we simply can't get over a failure; we just can't go on; it's time to end it all
- So Perl provides a warning function that is immediately fatal: the `die` function:

```
if ($count < 0) {
     die "Can't use negative count: $count";
}
```

- Like `warn`, the `die` function takes one or more optional arguments, typically used to specify an error message (as if it were a `say`)
- The message (if any) is placed in the special variable `$@`
- Then it throws an exception
- If the exception is not caught (more on that later), it propagates out through any surrounding scopes
- When it reaches the outermost program scope, it prints out the contents of `$@` and then immediately terminates the program
- If the `$@` message ends in a newline, it is printed "as is"
- Otherwise, the message has a location string appended of the form: `"...at filename, line N"`

Array variables

- In addition to variables that store a single value, we can also create variables that store an ordered collection of values
- Such variables are known as *arrays*
- An array variable is able to store one or more values…
- ...with each value uniquely identified by an integer (its *index*)
- Array indexes in Perl start at zero (not at `1`)
- Arrays use a different sigil than scalar variables: `@`, instead of `$`
- They are typically initialized with a list of values:

```
my @dwarfs
   = ("Happy", "Sleepy", "Grumpy", "Dopey", "Sneezy", "Bashful", "Doc");

my @deadly_sins
   = ("Gluttony", "Sloth", "Anger", "Envy", "Lust", "Greed", "Pride");

say "@dwarfs never commit @deadly_sins";
```

Accessing array elements

- To access an element, we change the leading `@` to a `$`, and append the desired element's index number in square brackets:

```
say  $dwarfs[0], " was accused of ", $deadly_sins[6];
say  $dwarfs[1], " was accused of ", $deadly_sins[5];
say  $dwarfs[2], " was accused of ", $deadly_sins[4];
say  $dwarfs[3], " was accused of ", $deadly_sins[3];
say  $dwarfs[4], " was accused of ", $deadly_sins[2];
say  $dwarfs[5], " was accused of ", $deadly_sins[1];
say  $dwarfs[6], " was accused of ", $deadly_sins[0];
```

- ***Note: The `@` changes to `$` because an array look-up returns a single value, not a list of values***
- Elements can also be individually assigned to...
- ...and will automatically be created if they do not already exist:

```
$dwarfs[7] = "Funky";

$deadly_sins[11] = "Incompetence";
```

- Array elements are always consecutive (never sparse)
- So assigning a value to `$deadly_sins[11]` causes the `@deadly_sins` array to grow to a length of twelve elements
- The extra three elements (`$deadly_sins[8]`, `$deadly_sins[9]`, and `$deadly_sins[10]`) are initialized to `undef`

- Because scalars and arrays use different sigils, in a single scope we can use a scalar variable and an array with the same name:

```
my @name = ("Damian", "Conway", "Hey You!");

my $name = $name[0];
say "Call me $name...";

$name = $name[1];
say "or $name...";

$name = $name[2];
say "or $name...\n";
```

- That's almost always a really bad idea

Array interpolation

- Like scalar variables, array elements will also interpolate directly into strings:

```
say "Call me $name[0]...or $name[1]...or $name[2]...";
# Call me: Damian...or Conway...or Hey You...
```

- And we can interpolate entire arrays:

```
say "Call me: @name...";
# Call me: Damian Conway Hey You...
```

- When an entire array is interpolated at once, its elements are concatenated together with a single space character between each

Array size

- There are two ways of obtaining information about the length of an array
- The scalar variable `$#array_name` always stores the last index of the array `@array_name`:

```
# Compare last and third-last deadly sins...
say $deadly_sins[$#deadly_sins],
    " is worse than ",
    $deadly_sins[$#deadly_sins-2];
```

- We can get the actual length of the array (usually `$#array_name + 1`) by using the array itself as a scalar value

```
$sin_count = @deadly_sins;

while ($n < @dwarfs) {
     say $dwarfs[$n++];
}
```

Inverted access

- We can also access the end of an array by using a negative index
- Negative indices count backwards from the end of the array:

```
# Compare last and third-last deadly sins...
say $deadly_sins[-1],
    " is worse than ",
    $deadly_sins[-3];
```

- This is the recommended way of accessing from the end: it's not only cleaner and clearer, it's less error prone too

Lists

- Arrays are closely related to (but not the same as!) lists
- A Perl list is an ordered collection of comma separated values
- Usually in a set of parentheses:

```
my @dwarfs
  = ("Doc","Happy","Sleepy","Grumpy","Sneezy","Bashful","Dopey");

my @digits = (0,1,2,3,4,5,6,7,8,9);
```

- We can also build a list of consecutive integers or strings in a specified range, by giving just the end-points:

```
my @digits  = (0..9);

my @letters = ("A".."Z", "a".."z");

my @digrams = ("aa".."zz");
```

- You can use the range notation anywhere a list is needed:

```
for (1..1000000) {
    say "Are we there yet?";
    sleep 60;
}
```

Lists vs arrays

- A Perl list is a sequence of immutable values
- In contrast, a Perl array is a **container** for a sequence of mutable values
- Lists are commonly used to initialize arrays
- Assigning a list to an array places each item in the list into a successive element of the array

Assigning to lists

- Lists may also be used to split arrays apart
- We can use a list of scalar variables as the left operand of an assignment:

```
($dw1, $dw2, $dw3, $dw4, $dw5, $dw6, $leader) = @dwarfs;
```

- This assigns `$dwarfs[0]` to `$dw1`, `$dwarfs[1]` to `$dw2`, `$dwarf[2]` to `$dw3`, *etc*.
- There may be more variables in the list than elements in the array
- In that case the extra variables are assigned `undef`:

```
($dw1, $dw2, $dw3, $dw4, $dw5, $dw6, $leader, $dw8) = @dwarfs;
# $dw8 now undef
```

- If there are fewer variables than array elements on the right, the extra RHS elements are ignored:

```
($mad, $bad, $dangerous_to_know) = @deadly_sins;
```

- Can also assign lists of literal values to lists of variables:

```
($sanguine, $saline, $doleful) = ("blood", "sweat", "tears");
```

- That's particularly handy for swapping the values in two variables without the need for a temporary:

```
if ($bigger < $smaller) {
    ($bigger, $smaller) = ($smaller, $bigger);
}
```

String lists

- Lists of literal character strings like `("blood", "sweat", "tears")` can be annoying to type in... and difficult to read
- So Perl provides another way to create such lists:
- The `qw` operator:

```
($sanguine, $saline, $doleful) = qw(blood sweat tears);

my @dwarfs = qw(Doc Happy Sleepy Grumpy Sneezy Bashful Dopey);
```

- After a `qw`, the contents of the parens are treated as a list of whitespace-separated words
- As with `q(...)` and `qq(...)`, the delimiters can be anything we like
- ***Note: Don't try to use commas in a `qw`.***
  They're treated as words too (though you do get a warning about that)

List flattening

- Suppose a list contains another nested list:

```
@virtues = ( "faith", "hope", ("love", "charity") );
```

- Many people expect this to produce a hierarchical list of three elements (where the third element is itself a two-element list), but it doesn't
- That's because each element of a list or array must be a scalar, not a list or array
- So instead, the entire nested list is "flattened"
- That is, all internal bracketing is simply ignored
- So the above example is actually equivalent to:

```
@virtues = ( "faith", "hope", "love", "charity" );
```

- Any arrays that appear in a list are also expanded in this way:

```
@virtues = ( "faith", "hope", @deadly_sins );  # nine elements
```

- It *is* possible to build hierarchical lists in Perl, using references (more on that later)

Arrays as stacks and queues

- Perl provides support for using its built-in arrays to implement stacks and queues
- Via the built-in functions `push`, `pop`, `shift`, and `unshift`
- The `push` function takes an array, and a list of elements to append to it
- It appends them and then returns the new length of the array
- The `pop` function removes the last element of an array and returns it
- If the array is empty, it returns `undef`
- For example:

```
$stack_height = push @stack, $item;
$stack_height = push @stack, ($item1, $item2, $item3);

while (my $next = pop @stack) {
    say $next;
}
```

- The `unshift` and `shift` functions work just like `push` and `pop` (respectively)...
- ...except that they add elements to the start of an array instead of to the end
- They're also slightly slower (especially `unshift`)

- The combination of pushing elements onto the end of an array and shifting them off the start of it, produces a queue:

```
push @menu, qw(appetizer soup entree main dessert cheese coffee);

while (my $next = shift @menu) {
    say "Now serving: $next";
}
```

Splicing an array

- There is also a general array access function: `splice`
- It takes four arguments: an array, an index, a length, and a replacement list
- It removes *length* elements from *array* starting at position *index*
- It then replaces the removed elements with the elements of *replacement list*
- In the process it will grow or shrink the array as necessary
- Finally, it returns the removed elements
- For example:

```
@words = ("I", "do", "not", "like", "green", "eggs", "and", "ham");
say "Started with:\n\t @words\n";

@removed = splice @words, -1, 1, "spam";
say "Replaced '@removed' to produce:\n\t @words\n";

@removed = splice @words, 4, 3, ("all", "kinds", "of");
say "Replaced '@removed' to produce:\n\t @words\n";

@removed = splice @words, 1, 3, "hate";
say "Replaced '@removed' to produce:\n\t @words\n";
```

- Or, to replace Deadly Sins #3 and #4 with the four Virtues, we would write:

```
@former_sins = splice @deadly_sins, 4, 2, @virtues;
```

- In other words: take the `@deadly_sins` array, go to index 4, remove 2 elements (*i.e.* elements 4 and 5, containing Sins #3 and #4), and replace them with the elements in `@virtues`
- The two elements that were removed are returned by `splice` and saved in the `@former_sins` array
- All four of `push` and `pop`, `shift` and `unshift` are just convenient wrappers around `splice`

```
splice @array, 0, 0, $newval;      # same as: unshift @array, $newval
splice @array, 0, 1;               # same as: shift @array
splice @array, -1, 1;              # same as: pop @array
splice @array, @array, 0, $newval; # same as: push @array, $newval
```

Slicing an array

- ***Note: "Slicing" an array is not the same as "splicing" an array***
- Sometimes it's useful to be able to extract or assign to only part of an array
- For example, we might want to print out elements 3 through 5 of a particular array:

```
say $tragedy[3], $tragedy[4], $tragedy[5];
```

- or perhaps reassign them:

```
($tragedy[3], $tragedy[4], $tragedy[5])
     = ("Macburger", "King Leer", "Omelet, Prince of Breakfasts");
```

- This happens often enough that Perl provides a special syntax to simplify it
- Instead of a single index, we can put a list of integers in the indexing square brackets
- Then Perl produces a subset of the original array–known as an *array slice*–instead of a single array element:

```
say @tragedy[3,4,5];  # same as previously, but less cluttered
```

- A "slice" is like a temporary (sub)array; hence it still has the `@` prefix
- Elements of that temporary array are not copies, but the same scalar containers that were in the original array
- So assignments to a slice work too:

```
@tragedy[3,4,5]
    = ("Macburger", "King Leer", "Omelet, Prince of Breakfasts ");
```

- The list of indices in the slice need not be comma-separated, explicit, or even sequential
- For example:

```
@squares[1..4]     = (1, 4, 9, 16);

@sqrt[1,49,9,16,4] = (1, 7, 3, 4, 2);

@inverse[@squares] = (1, 0.25, 0.1111, 0.0625);
```

- We can also slice a list, so long as it's in parentheses:

```
@notes = ("do", "ray", "mi", "far", "so", "la")[3,4,5,2,0,3];
```

List processing

- Perl provides a set of handy vector processing operations
- It can sort elements
- Or join a list into a single string with some specified separator
- Or perform an operation on each element of a list, producing a new list
- Or select only certain elements from a list

Sorting lists

- The `sort` function sorts the elements of a list:

```
@ordered = sort @disordered;
```

- By default, it sorts lexicographically (alphabetically)

```
@name = sort @name;   # @name now: ("Conway", "Damian", "Hey You")
```

- Unfortunately, it does that even for numbers
- That is, it treats each number as a string:

```
@nums = sort 1..11;              # (1, 10, 11, 2, 3, 4, 5, 6, 7, 8, 9)
```

- We can control how `sort` sorts by specifying how it compares each pair of elements:

```
@nums = sort {$a <=> $b} 1..11;  # (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

- Immediately after the `sort` keyword, we specify a brace-delimited block
- ***Note: There's no comma after the block's closing brace***
- In the block, we write an expression that compares two special variables: `$a` and `$b`
- Then, each time the `sort` needs to compare two elements, it assigns them to `$a` and `$b` and evaluates the expression
- The expression is supposed to return -1 if `$a` should come before `$b`, +1 if `$a` should come after `$b`, and zero if they have no preferred order
- That's where the `<=>` and `cmp` operators are useful
- For example, if we wanted to sort lexicographically, but case-insensitively:

```
@sorted = sort { lc($a) cmp lc($b) } @words;
```

- Or numerically by decreasing length of words:

```
@sorted = sort { length($b) <=> length($a) } @words;
```

- Recent versions of Perl use a stable mergesort algorithm that is *O(N log N)* in the worst case

Concatenating lists of strings

- The `join` function joins a list into a single string, interspersing a specified separator between the elements:

```
$str = join "sep", @list;
```

- For example:

```
say "Call me: ", join(", ", @name);
# Call me: Damian, Matthew, Conway

say join " is less than ", 1..100;
# 1 is less than 2 is less than 3 is less than 4 is less than…
```

- Or, the slightly more sophisticated:

```
say join( " is less than ",
            1, join(", which is less than ", 2..100)
        );
# 1 is less than 2, which is less than 3, which is less than 4…
```

Selecting elements from a list

- The `grep` function is analogous to the Unix `grep` utility
- It selects elements from a list
- Specifically, those elements for which a particular block returns a true value
- For example, all elements greater that zero:

```
@positives = grep { $_ > 0 } @numbers;
```

- ***Note: As with `sort`, there's no comma after the block's closing brace***
- Each element of the list is sequentially aliased to the special `$_` variable
- Then the code in the block is evaluated
- If the code evaluates to a true value, the corresponding element is included in the list that `grep` returns
- Another example: select only those words that start with "Q":

```
@q_words = grep { substr($_,0,1) eq "Q" } @words;
```

Processing the elements of a list

- The `map` function performs a specified operation on each element of a list, producing a new list
- This function is often called "apply" in other list processing languages
- For example, to create a list `(2,4,6,8,10,...100)`:

```
@twos = map {$_ * 2}  1..50;
```

- As with `grep`, each element of the list is sequentially aliased to `$_`
- As with `grep`, the block is evaluated once per element
- Unlike `grep`, the results of those evaluations are returned (*`grep`* returns the elements for which the evaluation was true)

- Another example: to generate a list of greetings:

```
@greetings = map { "Dear ". ucfirst(lc($_))}  @names_list;
```

- Or to create the lookup tables we saw before:

```
@numbers = 1..4;

@squares[@numbers] = map { $_**2 } @numbers;

@inverse[@squares] = map { 1/$_ }  @squares;
```

3. Input and output

- Perl offers the full range of Unix I/O options (on all systems)
- Buffered or unbuffered
- Byte-based or character-based
- Single character or line-by-line
- All Perl I/O is stream-based
- The stream abstraction is known as a "filehandle"
- We'll just cover the basic forms of I/O here

Character-based input

- Perl supplies the standard Unix `getc` function
- It returns a single character (or a *code-point* for Unicode input streams), or `undef` on failure
- It has same buffering as the stream it's reading (*i.e*. it's buffered by default)
- We can specify the filehandle explicitly
- Otherwise, `getc` uses `STDIN`
- For example:

```
while (my $char = getc()) {
    if (rand > 0.5) {
        print uc $char;
    }
    else {
        print lc $char;
    }
}
```

- The `print` function used here is identical to the `say` function, except it doesn't add a newline to the end of whatever it prints out, which is what we want when printing one character at a time
- ***Note: `getc` is expensive. Use line-based input where possible***
- ***Note: If you want to read a single unbuffered character immediately, you almost certainly want to use the `Term::ReadKey` module from CPAN instead***

Line-based input

- To read in a line of text, we use the `readline` function
- The `readline` function takes a single optional argument: a filehandle
- It reads characters from the corresponding stream until it encounters a newline
- It then returns all the characters it has read
- *...including* the newline
- For example:

```
while (my $line = readline()) {
    if (index(lc $line, $search_target) >= 0) {
        print $line;
    }
}
```

- Again, we use `print` here instead of `say`, because the line we read in via `readline` already has a newline at the end

Tidying up an input line

- The `chomp` function bites off the tail of its argument(s)
- Specifically, it removes any trailing end-of-line marker
- It doesn't return the truncated string; instead it modifies its argument in place

```
$nextline = readline();
chomp($nextline);
say qq(Read in: "$nextline");
```

- If `chomp` is called without any arguments it chomps the special variable `$_`
(we'll see why that's useful a little later)
- ***Note: `chomp` modifies its own argument; it doesn't return a modified copy of that argument. So this is a common mistake:***

```
$nextline = chomp($nextline);  # $nextline = 1
```

Line reads in list contexts

- Sometimes a call to `readline` will used at a point in our code where a list is expected
- Such points in the code are called *"list contexts"*
- Likewise there are *"scalar contexts"*: points in the code where a scalar value is expected

- A typical list context is the argument list of a `say` or `print`:

```
print readline();
```

- Another is the right hand side of an assignment to an array or list:

```
@input = readline();
```

- In such a context, `readline` doesn't read a single line
- Instead, it reads as many lines as it can...
- ...and then returns a list of those lines, one line per element:

```
print readline();           # reads and then prints ALL lines
```

- Likewise:

```
@input = readline();        # read ALL lines, store them in @input
```

- That's often **not** the desired behaviour
- So there's a way to change the context in which Perl thinks a particular function is being called
- Specifically, to tell it to evaluate something in a scalar context
- The `scalar` function:

```
print scalar(readline());   # reads one line and prints it
```

- `scalar` expects a single scalar argument and returns its value unaltered in any way

The input record separator

- The special `$/` variable stores the input record separator
- This allows a `readline` to read lines that are terminated by something other than a newline
- In fact, `readline` doesn't actually read up to a newline at all
- It reads up to whatever sequence of characters is currently in the special variable `$/`
- Which just happens to default to `"\n"`
- But we can change that very easily:

```
while (1) {

    $/ = ": ";    # Read up to first colon
    my $header_name = readline();
    chomp($header_name);

    $/ = "\n";    # Then eead up to end-of-line
    my $header_data = readline();
    chomp($header_data);

    say "Header was: $header_name";
    say "  Data was: $header_data";
}
```

- As a special feature, setting `$/` to an empty string means: *read the next paragraph*
- That is, until two or more consecutive newlines
- As an even more special feature, setting `$/` to `undef` means: *read until end-of-file*
- So instead of:

```
my $text;
while (!eof($filehandle)) {
    $text .= readline($filehandle);
}
```

- ...we can write:

```
my $oldslash = $/;
$/ = undef;
my $text = readline($filehandle);
$/ = $oldslash;
```

- ...which is **much** quicker
- …but messy
- So Perl allows us to temporarily replace a global variable, within the locality of a single block:

```
my $text;
{
    local $/ = undef;
    $text = readline($filehandle);
}
```

- At the end of the block in which the variable was `local`-ized, that variable is automatically reverted to it's pre-localized value
- Often you will see that code reduced to its most compact form:

```
my $text = do { local $/; readline($filehandle); };
```

- In a subsequent section we'll examine the various behaviours of the `do`-block that allow this idiom to work

Input line numbers

- The special `$.` variable tracks the current line number in the file we're currently reading
- It's especially handy for error messages:

```
while (my $nextline = readline()) {
   chomp $nextline;
   if ( something_is_wrong_with($nextline) ) {
       say "Error in input at line $.";
   }
}
```

The diamond operator

- Although most Perl programmers use `readline` most of the time...
- ...many Perl programmers never actually use `readline` explicitly
- That's because Perl provides an operator with the same behaviour, but much less syntax
- The circumfix diamond operator: `<>`
- By itself, it's usually equivalent to a `readline(STDIN)`
- (Actually it's not entirely equivalent, but that's a close enough approximation for the moment; we'll explore the differences later)
- So we could rewrite our earlier example:

```
while (my $nextline = <>) {
   chomp $nextline;
    if ( something_is_wrong_with($nextline) ) {
        say "Error in input at line $.";
    }
}
```

- Or, if we wanted to read from a different filehandle, we'd just put that filehandle between the angles:

```
while (my $nextline = <$filehandle>) {
   chomp $nextline;
    if ( something_is_wrong_with($nextline) ) {
        say "Error in input at line $.";
    }
}
```

- That's admittedly not much shorter (and certainly less readable) than an explicit `readline`, but Perl has another trick up its sleeve
- If we put a diamond operator by itself in a `while` condition, Perl arranges for the result of the `readline` to be stored in the special `$_` variable automatically
- So we could reduce the read-from-STDIN version to:

```
while (<>) {
    chomp;
    if ( something_is_wrong_with($_) ) {
        say "Error in input at line $.";
    }
}
```

- And that's how many Perl input loops are written
- *(Note the argumentless `chomp`, which therefore chomps `$_`)*
- However, though this code is very idiomatically Perlish, the implicit behaviours of the `<>` and the `chomp`, and the `$_` variable's complete lack of a meaningful name, will also make this code much harder to comprehend in six months' time

- You are strongly encouraged to use explicitly declared, well-named variables, and explicitly named functions instead:

```
while (my $nextline = readline()) {
   chomp $nextline;
    if ( something_is_wrong_with($nextline) ) {
        say "Error in input at line $.";
    }
}
```

Files

- Perl provides a wide range of built-in functions and operators to simplify dealing with files
- There are also numerous standard modules and others on the CPAN available to help
- Those modules are not covered here (look for them in the IO:: namespace)

Globbing filenames

- The `glob` function
- It is given a string representing a shell-ish file pattern
- It returns a list of all the files that matched the file pattern
- For example:

```
@log_files = glob("/cache/date_*/*[1-4].log");
```

Renaming files

- The `rename` function:

```
rename($oldname, $newname);
```

- Names can full pathnames, specified as strings
- Or simple names (which are taken to be the names of files in the current directory)
- Limitations vary according to the operating system you're using

Removing files

- The `unlink` function:

```
unlink(@filenames);
```

- The `rmdir` function:

```
rmdir(@dirnames);
```

File test operators

- File tests are built-in, but have an unusual syntax: *-X*
- They may be given a file name or a filehandle
- These operators determine if something is true about the file:

```
if (-e $file) { say "File exists";                  }
if (-r $file) { say "File is readable";             }
if (-w $file) { say "File is writable";             }
if (-f $file) { say "File is a plain file";         }
if (-d $file) { say "File is a directory";          }
if (-T $file) { say "File is an ASCII text file";   }
if (-B $file) { say "File is a 'binary' file";      }
if (-t $fh)   { say "Filehandle is opened to a tty"; }
```

- There are numerous others as well (read about them all via: `perldoc -X`)
- A few of these operators return non-boolean information about the file:

```
$file_size = -s $file;

$how_many_days_ago_file_was_modified = -M $file;
$how_many_days_ago_file_was_accessed = -A $file;
```

File contents

- To get at the contents of a file, we first need to open it
- The `open` function takes a variable that will eventually hold the filehandle...
- ...and the name of the file to be opened
- ...and (optionally) an extra intermediate argument, specifying the mode
- It returns a boolean value indicating success or failure
- To open for input:

```
$opened = open(my $filehandle, $filename);
```

- Or:

```
$opened = open(my $filehandle, "<", $filename);
```

- To open for output:

```
$opened = open(my $filehandle, ">", $filename);
```

- To open for appending:

```
$opened = open(my $filehandle, ">>", $filename);
```

- On failure the special `$!` variable is set with an error message:

```
$filename = "no_such_file";
$opened = open(my $infile, "<", $filename);

if (! $opened) {
    warn "\nUnable to open file '$filename':\n\t$!\nFailed";
}

$opened = open(my $outfile, ">", "read_only_file");

if (! $opened) {
    warn "\nUnable to open file 'read_only_file':\n\t$!\nFailed";
}
```

Output to a file

- The `say` and `print` builtins will also work with files
- Each of them optionally takes a filehandle before the first item to be printed
- The `say` or `print` knows it's a filehandle (and not something to be printed) because we don't put a comma after it:

```
say $filehandle "This is printed ", "to", " the filehandle";
```

- To be extra safe, and to make things clearer, we can wrap the filehandle in curly braces:

```
say {$filehandle} "This is printed ", "to", " the filehandle";
```

- ***Note: The filehandle has to be already open for writing***

File pointer positioning

- If necessary, we can easily determine the current read or write position
- And also change it
- The `seek` and `tell` functions do that:

```
$prev_pos = tell $filehandle;

# and later...

seek $filehandle, $prev_pos, 0;
```

- The `tell` function returns the current position of the specified filehandle
- That position is the number of bytes that the "next available character" is from the start of the file
- The `seek` function moves the "next available character" pointer to the specified position
- The second argument of `seek` is the number of bytes to move the pointer

- The third argument to `seek` specifies where the move is relative to:

```
seek $filehandle, +10, 0;  # seek forward 10 bytes from current pos
seek $filehandle, -10, 0;  # seek back 10 bytes from current pos
seek $filehandle, +10, 1;  # seek forward 10 bytes from start of file
seek $filehandle, -10, 2;  # seek back 10 bytes from end of file
```

Closing files

- The `close` function takes the name of the filehandle to be closed:

```
$closed = close $filehandle;
```

- If the close fails, `close` returns false and sets an error message in `$!`
- Filehandles are also automatically closed if the lexical variable containing them goes out of scope

In-source data files

- Source code is (almost always) in a file
- And Perl allows us to access that file directly
- At the end of the code we can add a line consisting of just:

```
__DATA__
```

- When the compiler finds this marker in a Perl program, it opens a special filehandle named `DATA`
- The filehandle is opened (read-only) to the file containing the source code itself...
- ...at the first line after the `__DATA__` marker
- Thereafter we can read from it, seek it, *etc., etc*.

```
my %default;

while (my $next_config = readline(DATA)) {    # or just: … = <DATA>
    chomp $next_config;

    my $field = substr($next_config,0,6);
    my $value = substr($next_config,8);

    $default{$field} = $value;
}

say "Using $default{'FORMAT'} playback for $default{'LENGTH'}";

__DATA__
FORMAT: NTSC
LENGTH: 60
AUTOFX: 1
AUTORW: 0
```

Executing shell commands

- The `system` function takes a string...
- ...exactly as we might type it on the shell command-line...
- ...and executes it:

```
system("lint *.c");
```

- As in Unix, this system call returns zero on success, or a non-zero error code on failure
- It also puts that non-zero return value in the special `$?` variable
- If `system` can't start the command at all, it sets the special `$!` instead
- For example:

```
system($some_cmd);
if ($!) { warn "Couldn't start up $some_cmd" }
if ($?) { warn "Couldn't complete $some_cmd" }
```

- We can also pass two or more arguments to `system`
- In that case, it executes the command specified by the first argument, passing it the arguments specified by the remaining arguments:
- ...and executes it:

```
system("lint", @files_to_be_linted);
```

- Unlike the single-argument version, the multi-argument version does no shell-globbing on any of its arguments, which makes it safer, but less convenient

Retrieving data from shell commands

- The `system` command is fine if we just want to tell the shell to do something for us
- But not so useful if we want to ask the shell to tell us something
- For example:

```
system("ls");
```

- This causes the shell to run a "list directory" command, printing the directory contents to the terminal

- But if we wanted to capture the list of files listed by a `system` command, we would have to use a temporary file:

```
my $tmpfile = "/tmp/temp" . $$;

if (system("ls > $tmpfile") == 0) {
     open(my $fh, $tmpfile);
     @files = readline($fh);
     close($fh);
     system("rm -rf $tmpfile");
}
```

- (The special *$$* variable always stores the process ID for the program itself)
- Because that's so tedious, there's a shortcut: the "backticks" operator
- If we specify the required shell command in backticks (`` `...` ``) then it is run as a `system` command
- But the output from the command is returned as the (list) value of the backtick operation:

```
@files = `ls`;
```

- There's also a "long" form of the backticks:

```
@files = qx{ ls };
```

- Don't put the backticks inside a `system` command – **`system(`ls`)`** – since that means: "run the `ls` command, get back the list of files, then use that list as the arguments to `system`"

4. Operators

- Perl provides most of the standard operators found in other languages
- And it uses mostly the same symbols for them
- But adds a few extras of its own
- Mostly out of Laziness

Numeric operators

- All the usual numeric operators are available
- Each one imposes a "numeric context" on their (scalar) operands
- Exponential: *`$x ** $n`*
- Unary: *`+$n, -$n`*
- Multiplicative: *`$x*$y`*, *`$x/$y`*, *`$x%$y`*
- Additive: *`$x+$y`*, *`$x-$y`*
- Functions: `sin`, `cos`, `atan2`, `abs`, `log`, *etc*.

String operators

- All impose "string context" on their (scalar) operands
- Concatenation: `$str1 . $str2`
- Repetition: `$str x $count` (means: `$str . $str . $str …`)
- By the way, the repetition operator can also be applied to a list:

```
@rose = ("is", "a", "rose") x 3;

say "A rose @rose";   # A rose is a rose is a rose is a rose
```

- In that case it creates a new list that repeats the sequence the specified number of times

Bitwise operators

- These can operate on either strings or integers
- Complement: `~$val`
- AND: `$val1 & $val2`
- Inclusive OR: `$val1 | $val2`
- Exclusive OR: `$val1 ^ $val2`
- ***Note: Bitwise operators will treat string operands as numbers if those operands have previously been treated as numbers anywhere in the program***

```
say 42  |  13;        # 101010    (42)
                      # 001101    (13)
                      # ======
                      # 101111    (47)

say "42" | "13";      # 00110100 00110010    ("42")
                      # 00110001 00110011    ("13")
                      # ======== ========
                      # 00110101 00110011    ("53")
```

Boolean operators

- Comparators: `==`, `!=`, `<`, `>`, `eq`, `ne`, `lt`, `gt`, *etc*.
- Negation: `!$bool`
- Connectives: `$bool1 && $bool2`, `$bool1 || $bool2`,
- ***Note: There's no ^^ (logical XOR) operator.***
  Most people use: `!$bool1 != !$bool2`
- ***Note: But don't use: `$bool1 == $bool2`***
- The binary logical connectives "short-circuit"
- That is, they don't evaluate their right operand unless they have to:

```
$attractive  =  $Australian || $tall && handsome() && $dark;
```

Low precedence boolean operators

- The boolean connectives have higher precedence than list operations
- So this doesn't quite work as we might hope:

```
say $name, $rank, $serial_number  ||  die "Unable to report!";
```

- We'd like that to mean:

```
(say $name, $rank, $serial_number) ||  die "Unable to report!";
```

- That is, try to execute the `say` and if it fails (*i.e.* returns false) then throw an exception
- But the precedence of `||` is higher than that of the listifying comma
- So the unparenthesized version actually means:

```
say $name, $rank, ($serial_number ||  warn "Unable to report!");
```

- Because Perl programmers like to avoid parens, Perl provides another OR operator that has the correct (*i.e.* very low) precedence:

```
say $name, $rank, $serial_number  or  warn "Unable to report!";
```

- Which means the precedence is the same as:

```
(say $name, $rank, $serial_number) or  warn "Unable to report!";
```

- By far the commonest use of this operator is in opening files:

```
open(my $fh, $filename)
   or die "Couldn't open $filename: $!";
```

- There are also low precedence versions of `&&` and `!`
- No prizes for guessing their names: `and`, `not`
- Curiously, there's also a low precedence `xor`,
  even though there's no high precedence equivalent

Defined-or operator

- In addition to the regular *OR* operator (`||`), Perl has an operator that tests for definedness, rather than truth: the `//` operator
- This operator is especially useful because Perl often uses undefined values to indicate failure
- For example:

```
my $next_line = readline() // "<end of input>";

my $bin_path  = $PATH // "$HOME/bin:/usr/local/bin";

my $top_score = $student_score[$top_index] // 0;
```

Mutators

- Mutators are those operators that change one of their operands
- That operand must, of course, be a variable
- Most of them were stolen from C
- Increment: `$n++`, `++$n`
- Decrement: `$n--`, `--$n`
- Assignment: `$var = $value`
- Op-and-assign: `+=`, `-=`, `*=`, `**=`, `||=`, `.=`, *etc.*
- Op-and-assign performs the operation between its two arguments, then assigns the result back to the left operand:

```
$count  += 7;              # Means: $count = $count + 7

$lines  .= $next_line;     # Means: $lines = $lines . $next_line

$large ||= $size > 1e6;    # Means: $large = $large || size > 1e6
```

The selection (ternary) operator

- Also assimilated from C
- Then augmented
- Like an in-line `if` statement
- Takes three operands: *condition* **?** *expr* **:** *expr*
- If its left operand is true, the ternary operator evaluates to its middle operand
- Otherwise evaluates to right operand:

```
$message = $is_verbose ? "I seem to have a problem, Dave" : "Oops!";
```

- The operator is particularly handy for making decisions in the middle of a larger expression:

```
warn "There ", ($count==1 ? "was" : "were"),
     " $count error", ($count==1 ? "" : "s");
```

- We can "chain together" ternaries to select in longer sequences
- For example, rather than:

```
   if ($is_verbose) { $message = "I seem to have a problem, Dave" }
elsif ($is_curt)    { $message = "Something's wrong, dummy!" }
elsif ($is_scared)  { $message = "Arrrrrgggggggghhhhh!" }
else                { $message = "Oops!" }
```

- We can write:

```
$message = $is_verbose ? "I seem to have a problem, Dave"
         : $is_curt    ? "Something's wrong, dummy!"
         : $is_scared  ? "Arrrrrgggggggghhhhh!"
         :               "Oops!"
         ;
```

- The ternary operation can also be used as the **left** operand (*i.e.* the lvalue) of an assignment
- For example, rather than:

```
if ($dealer_busts) { $player += $pot }
else               { $house  += $pot }
```

- We could write:

```
($dealer_busts ? $player : $house) += $pot;
```

- ***Note: though that doesn't mean we <u>should</u> write that. The cleverness is probably not worth the loss of maintainability.***

5. Hashes

- A hash is variable that stores an unordered collection of mutable values, each of which is indexed by a string
- In other languages, hashes are sometimes called *"associative arrays"* or *"dictionaries"*
- A hash is perhaps best thought of as a two-column look-up table or database
- The left column stores unique string identifiers called *keys*
- The right column stores an associated scalar *value*
- (It's called a "hash" because a hashing algorithm is used to map each key string to some internal index into the table)
- Like scalars and arrays, hash variables have their own prefixing sigil: `%`
- They're declared like any other Perl variable:

```
my %sound;

my %fury;
```

- We can initialize them by assigning a list:

```
my %sound = ("cat", "meow", "dog", "woof", "goldfish", undef);
```

- The list must be an alternating sequence of keys and their associated values
- So, for example, in `%sound`, the key `"cat"` is associated with the value `"meow"`, the key `"dog"` with the value `"woof"`, and the key `"goldfish"` with the value `undef`

- Typically that's written more like a two-column table:

```
my %sound = (
   "cat",       "meow",
   "dog",       "woof",
   "goldfish",   undef,
);
```

- ***Note: The keys have to be strings, but the values can be any type of scalar value (and they don't all have to be of the same type either)***

Fat commas

- It can be easy to lose track of which elements of that initializer list are keys and which are values
- So Perl has a second form of comma: `=>`
- It's used between key and value, like this:

```
my %sound = ("cat"=>"meow", "dog"=>"woof", "goldfish"=>undef);
```

- It's exactly the same as a regular comma
- With one useful extra property…
- It treats any identifier on its immediate left as if it were a quoted character string
- So we can just write:

```
my %sound = (cat=>"meow", dog=>"woof", goldfish=>undef);
```

- Or, even more readably, we can lay the data out in two columns:

```
my %sound = (
   cat       => "meow",
   dog       => "woof",
   goldfish  =>  undef,
);
```

- ***Note: The value after the => still requires quotation marks if it's a string***

Hash entries

- The individual values stored in a hash are called *entries*
- They're accessed in much the same way as the individual elements of an array
- That is, we change the original `%` sigil to a `$` sigil...and then specify which entry to look up
- But instead of specifying a numerical index in square brackets (as for an array)…
- ...we specify a string key in curly braces

- For example:

```
say "The cat replied: ", $sound{'cat'};

$sound{"cat"} = "purr";
say "The cat replied: ", $sound{'cat'};

$animal = 'cat';
say "The $animal replied: ", $sound{$animal};
```

- ***Note: A hash entry such as `$sound{'cat'}` has nothing whatsoever to do with the scalar variable `$sound`.***
- Hash look-ups can be interpolated inside double-quoted strings, so we could have just written:

```
say "The cat replied: $sound{'cat'}";

say "The cat replied: $sound{$animal}";
```

- The insides of the braces will also interpret a single enclosed identifier as a quoted string
- (Just as the left operand of a `=>` is treated as a string if it's a bare identifier)
- So we could simply write:

```
say "The cat replied: ", $sound{cat};
$sound{cat} = "purr";
```

- But we can't just put anything in the braces and expect it to be stringified:

```
$sound{homo sapiens vendax} = "have a nice day";        # Wrong!
```

- As we'll see later, that's actually a series of nested subroutine calls; the equivalent of :

```
$sound{homo(sapiens(vendax()))} = "have a nice day";
```

- So you need to say:

```
$sound{'homo sapiens vendax'} = "have a nice day";
```

- Generally speaking, it's safer to get into the habit of always explicitly quoting your hash keys (even though most Perl programmers don't seem to do so)

Existence and extinction

- We can check whether a hash has an entry for a particular key
- To do so, we use the `exists` function:

```
if (exists $sound{'aardvark'}) {
     say "An aardvark says: $sound{'aardvark'}";
}
else {
     say "Say what?";
}
```

- We can remove entries too, using the `delete` function:

```
my $former_sound = delete $sound{"aardvark"};

say qq(The aardvark said "$former_sound" on its way out);
```

- Observe that `delete` returns the value of the entry it just deleted

Processing all hash elements

- In arrays, the elements are ordered by their sequential indices
- In hashes, the entries are not stored in any obvious or useful order
- In particular, they're not stored alphabetically, nor chronologically
- So it's trickier to step through the contents of a hash
- Perl provides several built-in functions that can assist
- The most widely used are `keys` and `values`
- The `keys` function takes a hash as its argument and returns a list of its keys...
- ...in some (apparently random) order
- The `values` function takes a hash and returns a list of its values...
- ...in the same (apparently random) order
- So we could print out all the key/value pairs in a hash:

```
for my $key (keys %sound) {
    say "The key $key has the value $sound{$key}";
}
```

- Or we could just print the values without the keys:

```
for my $val (values %sound) {
    say $val;
}
```

Iterating the keys of a hash

- The third hash-access function is `each`
- It takes a hash as its argument and returns one distinct key from that hash every time it's called
- The next call to `each` then returns the next key
- Subsequent calls eventually return every possible key…
- ...once again in that mysterious (apparently random) sequence
- After all the hash's keys have been returned, the next call to `each` returns `undef` to indicate that there are no more keys
- Calling `each` again after that, resets the iterator and returns the first key again

- For example:

```
while (my $nextkey = each %sound) {
    say "The key $nextkey has the value $sound{$nextkey}";
}
```

- ***Note: `each` stores its iterator inside the hash itself, so you can't safely nest two loops if those loops both use `each` on the same hash, nor can you safely terminate the loop early (as that will leave the iterator half way through the hash)***

Iterating the keys and values of a hash

- We can also call `each` in a list context
- That is, we can call it and assign its return value to a list or array
- In which case, `each` returns a two-element list containing the next key and its associated value
- For example:

```
while ( my ($nextkey, $nextval) = each %sound ) {
    say "The key $nextkey has the value $nextval";
}
```

- At the end of the set of key/value pairs, in a list context call to `each` returns an empty list
- …which is also false in Perl, so the `while` loop terminates

Another use for `keys`

- From Perl v5.14 onwards, we can also use `keys` on arrays
- When passed an array as its argument, `keys` returns a list of the arrays indices
- So instead of:

```
for my $n (0..$#wordlist) {
    say "$n: $wordlist[$n]";
}
```

- …we could write:

```
for my $n (keys @wordlist) {
    say "$n: $wordlist[$n]";
}
```

- Apart from the improved readability, using `keys` in this way also greatly decreases the chance of committing an "out-by-one" error:

```
for my $n (1..@wordlist) {
    say "$n: $wordlist[$n]";
}
```

Slicing a hash

- Recall that the built-in `values` function returns hash values in an apparently random order
- So to print out those values in a useful order (for example: sorted by key), we can't just write:

```
say values %sound;
```

- We need to be specific:

```
say $sound{"cat"}, $sound{"dog"}, $sound{"dolphin"}, $sound{"fish"};
```

- …or more generally:

```
say map { $sound{$_} } sort @animal_names;
```

- But that's either tediously explicit or moderately obscure, so there's a short-hand:

```
say @sound{"cat", "dog", "dolphin", "fish"};
say @sound{ sort @animal_names };
```

- This is known as a *hash slice*
- And it's closely analogous to an array slice
- Instead of a single key in the curly braces, a list of keys is specified
- ***Note: Hash slices use the sigil `@` (not `$` or `%`) because the resulting slice is an array-like sequence of aliases to values, not a single scalar, or a hash***
- Like an array slice, a hash slice may be the target of a list assignment:

```
@sound{"mouse", "bird"} = ("ariba!", "itortitawapuddytat");
```

Sigils demystified

- By now, variable sigils are either obvious to you…or a total mystery
- Scalars are okay: you always just use `$`
- But for arrays, you use `@`, unless you're looking up an element, in which case it's `$`, unless your looking up multiple elements in a slice, in which case it's `@` again
- And hashes use all three sigils: `%` for the hash itself, `$` for looking up a key, and `@` for looking up multiple keys in a slice
- So most beginners struggle mightily with sigils (and sometimes just try them at random until the compile stops complaining)
- But the rule really is extremely simple…once you know the secret
- The secret is: sigils don't tell you what a variable *is*; sigils tell you what an operation on a variable will *return*
- `$` means: *"the following returns a single value"*
- `@` means: *"the following returns a list of values"*
- `%` means: *"the following returns a list of alternating keys and values"*

Key/value slices

- The sigil rules described in the preceding section imply that there must be a third type of look-up as well…one that returns an alternating key/value list
- And, from Perl v5.20 onwards, there is
- It's called a *"key/value slice"*
- If you take an ordinary array slice or hash slice and change the leading `@` to a `%`, then the look-up returns all the requested values, but precedes each by the corresponding key
- So for example, we can slice a hash to get either just values, or else full key/values:

```
@sound{'cat', 'dog'}  # (       $sound{'cat'},        $sound{'dog'})

%sound{'cat', 'dog'}  # ('cat', $sound{'cat'}, 'dog', $sound{'dog'})
```

- And likewise we can slice an array in two ways:

```
@wordlist[1..2]       # (    $wordlist[1],    $wordlist[2] )

%wordlist[1..2]       # ( 1, $wordlist[1], 2, $wordlist[2] )
```

- Key/value slices are most often used to construct a new hash from some subset of an existing hash:

```
my %selected_data = %full_data{@selected_keys};
```

6. Control structures

- Perl has most of the usual suspects: `if`, `while`, and `for`
- Actually, it has *two* kinds of `for`
- The C-like *"init-test-step"* kind...and Algolish *"eat my list"* kind
- Unusually, Perl also has "negative" control structures: `unless` (an anti-`if`) and `until` (an anti-`while`)

The `if` statement

- Selection statements take the form:

```
if ( condition ) {
    # conditionally executed statements here
}
```

- An `if` can be followed by a single `else` statement, whose block executes if the preceding `if` block **doesn't:**

```
if ( condition ) {
    # conditionally executed statements here
}
else {
    # alternative statements here
}
```

- ***Note: There's no "statement" form of the `if`, as there is in C. A block is always required even if it contains only one statement.***
- We can also place one or more `elsif` statements after the `if` block (and before any `else`):

```
   if ( condition1 ) {...}
elsif ( condition2 ) {...}
elsif ( condition3 ) {...}
elsif ( etcetera   ) {...}
else                 {...}
```

- ***Note: That's `elsif` not `elseif` or `else if`***

The `unless` statement

- Occasionally it's easier to test a negative condition:

```
if (! ($pot > 1e6 || $hand < 16) )  { stand(); }
```

- …but that's more brackets than we'd prefer
- We *could* use the low precedence `not`:

```
if (not $pot > 1e6 || $hand < 16 )  { stand(); }
```

- …but that's probably just confusing
- So Perl provides an "anti-`if`":

```
unless ($pot > 1e6 || $hand < 16 )  { stand(); }
```

- Any statement of the form:

```
unless ( condition ) {...}
```

- Is identical to:

```
if (! (condition) ) {...}
```

- We can also put `elsif` and `else` after an `unless`
- ***Note: `unless` can easily get confusing. Usually it's best to avoid it.***

The `while` and `until` loops

- Perl has a typical `while` statement that keeps looping as long as the associated expression is true at the start of each iteration:

```
$T_minus = 10;

while (!$holding && $T_minus > 0) {
    say "T-$T_minus, and counting";
    $holding = check_for_hold();
}
```

- But Perl also has an "anti-`while`": `until`
- It keeps looping **until** the associated expression is true at the start of an iteration:

```
$T_minus = 10;

until ($holding || $T_minus == 0) {
    say "T-$T_minus, and counting";
    $holding = check_for_hold();
}
```

- Sometimes using the negative form makes the termination condition clearer
- ***Note: ...and sometimes it doesn't. Usually it's best to avoid it.***

C-like `for`

- Perl provides a version of the C three-part `for` statement:

```
say "We're in a hurry...";

for (my $i=100; $i>0; $i-=3) {
    say "Lift-off in $i seconds";
}
```

- As in C, that's really just a short-hand for:

```
my $i=100; while ($i>0) {
    say "Lift-off in $i seconds";
    $i-=3;
}
```

- Except that any lexical variable (like `$i` above) that is declared as part of the `for` initializer is scoped to the `for` block, **not** to the block surrounding the `for`

Algolish `for`

- Perl has another form of `for` statement
- This second form takes a list of values and steps through it
- One value per iteration
- On each iteration, the special variable `$_` is aliased to (*i.e.* …it becomes another name for…) each of the values in turn:

```
for (1,3,6,7,9) {
    if ($_ % 2) { say "$_ is odd" }
    else        { say "$_ is even" }
}
```

- The list to be iterated can come from a range operator…

```
@nums = qw(zero one two three four five six george eight nine ten);

for (0..$#nums) {
    say $nums[$_];
}
```

- Or from a flattened array...

```
@nums = qw(zero one two three four five six george eight nine ten);

for (@nums) {
    say $_;
}
```

- Or from the return value of a function or subroutine...

```
@nums = qw(zero one two three four five six george eight nine ten);

for (reverse(@nums)) {
    say $_;
}
```

The `foreach` *loop*

- Perl has a synonym for the `for` keyword: `foreach`
- It can be used for either kind of `for` statement (C-like or Algolish)
- Typically its used to make the code read more naturally
- That is, you can use `for` when the list reads like a plural:

```
for (@names) {
    say "Have you met $_?";
}
```

- And use `foreach` when the list reads like a singular:

```
foreach (@person) {
    say "Have you met $_?";
}
```

- However, it's probably a better idea to stick to one or the other throughout your code: most people seem to just use `for`, though some prefer to use `for` for the C-like construct and `foreach` for the Algol-ish version

Naming the iterator variable

- Aliasing `$_` on each iteration is certainly Lazy
- But it's not especially maintainable
- Because `$_` is a *terrible* name for a variable; it tells us nothing at all about the purpose, contents, or use of the container
- So Perl also provides a way of aliasing the iterated value to a more meaningful name
- We can place a lexical variable between the `for` keyword and the list:

```
for my $person (@names) {
     say "Have you met $person?";
}
```

- Naming iterator variables is an excellent practice, and highly recommended
- Using the default `$_` isn't any faster, and it's almost always much less clear
- ***Note: You can also omit the `my`, in which case Perl will (sometimes, but not always!) autogenerate a global variable to be used as the iterator variable. That's a really bad idea.***

Blocks as expressions

- The `do` block is a way of grouping a series of statements together as if they were a single statement
- Or, from another point of view, it's a way of treating a block of code as if it were an inlined function call (*i.e.* as if it returned a value)
- A `do` block executes its statements and then returns the value of the final expression it evaluates within the block:

```
my $contents = do {
    open(my $filehandle, $filename) or die $!;
    my $text = join "", readline($filehandle);
    close $filehandle or die $!;
    $text;
};
```

- This explains how the "file slurping" idiom shown earlier works:

```
my $text = do {
    local $/;
    readline($filehandle);
};
```

- Localizing `$/` temporarily resets its value (to `undef`) for the remainder of the block
- Which causes the `readline` to read until end-of-file
- And because the `readline` is the final expression evaluated in the block, it's return value *(i.e.* the contents of the file) becomes the value of the entire `do` block
- ***Note: Unlike all other Perl blocks, a `do` block is an expression. It's almost always part of a larger statement, and usually requires a semicolon after it.***

Statement qualifiers

- Perl allows any statement to be *qualified*
- A qualified statement is a regular Perl statement…
- ...with an `if`, `unless`, `while`, `until`, or (Algolish) `for` keyword tacked on to the end of it
- For example:

```
say "Moo!" if ($animal eq "cow");

die "Unknown format" unless (recognizable($input));

$bar_chart .= "*" for (1..$count);

sleep(1) while (no_data_yet());

sleep(1) until (data_available());
```

- The statement is only executed if/while/unless/until the condition is true
- Or, in the case of a `for` qualifier, the statement is executed once for each value in the list
- And the `for` statement still aliases each element of its list to `$_` in turn
- In other words, qualifiers provide another way of writing what would otherwise be single statement control blocks:

```
if ($animal eq "cow") { say "Moo!" }

unless (recognizable($input)) { die "Unknown format" }

for (1..$count) { $bar_chart .= "*" }

while (no_data_yet()) { sleep(1) }

until (data_available()) { sleep(1) }
```

- The (otherwise mandatory) parentheses around the conditional or list are optional in a qualifier, allowing us to further reduce "line-noise":

```
say "Moo!" if $animal eq "cow";

die "Unknown format" unless recognizable($input);

$bar_chart .= "*" for 1..$count;

sleep(1) while no_data_yet();

sleep(1) until data_available();
```

- Qualifiers also have the advantage of making the action prominent, rather than the condition, which can sometimes make the code easier to maintain
- The only restriction on qualifying statements is that we can't re-qualify an already-qualified statement:

```
say $_ if length($_) > 0 for @words;      # Compile-time error!
```

- To get the same effect, use a `grep` instead:

```
say $_ for grep {length($_) > 0} @words;
```

- Whether to use a single statement control block, or a qualified statement, is largely a matter of taste (or policy)
- Sometimes it depends on whether you need to emphasize the condition or the response
- However, qualifiers can only be applied to single statements, so if you later discover you need to do two or more things in response to a successful qualifier, you'll then have to rewrite the code as a block-form control statement anyway

Loop control

- Perl provides three ways to alter the flow of control within a loop
- The `next` statement short-circuits the rest of the current iteration and immediately starts the next one
- That is: it jumps out of the current iteration and immediately executes any increment, test, or motion through a `for` list
- A common usage is:

```
for my $filename (@filenames) {
    open(my $fh, '<', $filename)
      or next;

    # process file here
}
```

- The `last` statement immediately terminates the loop:

```
while (my $next_value = readline()) {
    chomp $next_value;

    last if $next_value eq "";

    $sum += $next_value;
}
```

- The `redo` statement repeats the *current* loop iteration immediately
- That is, if the `redo` is in a `while`, `unless`, or C-like `for`, the next iteration of the loop starts at once, but without retesting its condition or incrementing its iterated value
- And if the `redo` is in an Algolish `for`, the list is not iterated; the loop block starts again with the same element of the list is aliased to `$_` a second time
- A `redo` is useful for handling "incomplete" iterations
(such as the discovery of a line-continuation marker on an input loop)
- For example:

```
while (my $next_line = readline()) {
    chomp $next_line;

    if (substr($next_line,-1) eq "\\") {
         substr($next_line,-1) = readline();
         redo;
    }

    say "Now processing: $next_line";
}
```

The `goto` command

- Perl provides a `goto` statement
- You can label a statement using: ***`LABEL`*`:`**
- Then you can jump to that statement with: `goto` ***`LABEL;`***
- Just say: *"No!"*

Running code in other execution phases

- Normally, when you execute a Perl program, it is first fully compiled and only then is it run
- However the execution of a Perl program is actually divided into five distinct phases...
- The `BEGIN` phase is the compilation phase, during which the source code is parsed and an executable version of it constructed
- The `CHECK` phase occurs immediately after compilation is complete
- This phase is sometimes used to apply extra tests to determine that the compilation was satisfactory

- The `INIT` phase occurs immediately before execution begins
- This phase is sometimes used to pre-initialize certain variables on the basis of information determined at compile-time
- The execution phase (which has no special name) is the phase in which the code actually runs
- The `END` phase occurs after the execution is complete
- It is often used to clean up external resources (for example, to write an end-marker to a file or device)
- We can specify that certain parts of the code are to be executed "out-of-sequence" during one of these named phases
- That is, we can mark blocks of code that ought to be run in the compilation, checking, initialization, or clean-up phases, rather than in the execution phase
- To specify these special blocks of code, we simply prefix the block with the name of the phase in which we want it to run:

```
BEGIN { say "This will be executed during compilation"; }

CHECK { say "This will be executed at the end of compilation"; }

INIT  { say "This will be executed just before the program runs"; }

END   { say "This will be executed after the program finishes"; }

      { say "This will be executed as the program runs"; }
```

- Note that, if we specify more than one `BEGIN` or `INIT` block, those blocks run (within their respective phases) in the order they are encountered by the compiler
- However, if we specify more than one `CHECK` or `END` block, they run in the ***reverse*** order from that in which the compiler found them
- That is, the last `CHECK` block to be compiled runs before the second last, which run before the third last, *etc*. *etc*. ...until finally the first `CHECK` block runs
- Though it may seen odd, it's actually remarkably handy because it ensures that any clean-up happens in the correct "first-setup-last-torndown" sequence
- A `BEGIN` block is a special case because it is supposed to run in the compilation phase, the body of a `BEGIN` block is executed as soon as possible
- That is, as soon as the closing curly brace on its own block has been successfully recognized (in other words, **before** the rest of the program is even parsed, let alone compiled)
- So be careful of traps like:

```
BEGIN { say foo() }

sub foo { return "foo!" }
```

Terminating a program

- Perl provides three distinct ways to exit a program
- The simplest is to do nothing
- In that case when the last executable statement finishes, so does the program
- To explicitly terminate the program, we use the `exit` function:

```
if (no_more_to_do()) {
     exit();
}
```

- The `exit` function takes an optional integer argument which sets its exit status (on systems that have an exit status):

```
if (checking_complete() && $error_count > 0) {
     exit($error_count);
}
```

- Note however that, before the `exit` terminates the program, it runs any `END` blocks and the destructors of any objects
- The third (and most common) alternative is to call `die`, as we have already seen

Catching exceptions

- Calling `die` is the Perl way of throwing an exception
- And the Perl way of *catching* an exception is with an `eval` block
- If an exception is thrown inside an `eval` block (or inside code called from an `eval` block) then the `eval` block intercepts the exception
- When it catches an exception, the `eval` block itself immediately terminates, returning an `undef` value
- If no exception is thrown, the `eval` block runs to completion and returns the final value its last executed statement evaluated to (*i.e.* like a `do` block does):

```
while (my $next_num = readline()) {
    chomp $next_num;

    my $reciprocal = eval { 1/$next_num } // "not defined";

    say "The reciprocal of $next_num is $reciprocal";
}
```

- As mentioned earlier, the `die` built-in sets the special `$@` variable to the propagating exception

- So you can access and handle any caught exception through that variable:

```
my $okay = eval {
    call_that_might_fail();
    another_chancy_call();
    1; # If we get to here, evaluate to true (to indicate "okay")
};

if (!$okay) {
     say "Error: $@";
     recover_if_possible() or die;
}
```

- Note that calling `die` without an argument rethrows the exception already in `$@`
- This construction is admittedly clumsy (and error-prone)…mainly because Perl has no built-in `try` or `catch` or `finally` blocks
- However there are CPAN modules that extend the language to provide these features
- The two most popular are: Try::Tiny and TryCatch
- For example:

```
use TryCatch;

try {
    call_that_might_fail();
    another_chancy_call();
}
catch ($error) {
    say "Error: $error";
    recover_if_possible() or die $error;
}
```

- Using such modules is recommended because Perl's built-in `eval` has several limitations and a couple of nasty edge-cases that make it easy to introduce bugs when handling exceptions

Another use for `eval`

- Alternatively, the `eval` function can be given a string argument, instead of a block
- In that case it treats the string as if it were more Perl source code
- It compiles that code, and then executes it
- If the compilation fails, or the executed code throws an exception, the `eval` returns `undef` and sets the special `$@` variable with the appropriate error message
- Otherwise `eval` returns the value of the last statement it executed (just as the block form does)

- For example:

```
$/ = undef;  # Hereafter, readline will read entire input

for my $file (@ARGV) {
     open(my $fh, $file) or die $!;
     my $file_contents = readline($fh);    # i.e. read entire file
     close $fh or die $!;

     my $result = eval($file_contents);

     if (defined $result) { say "$file: $result" }
     else                 { die "Bad code: $@\nDetected" }
}
```

- Hence, both forms of `eval` have the same fundamental behaviour (*i.e.* that of a `try` block in other languages)
- The only difference is that the code governed by the block form is pre-compiled at compile-time
- Whereas the code governed by the string form is compiled only at run-time
- ***Note: String `eval` is slow. Try to avoid it, especially in loops***
- ***Note: Passing `eval` a string that originated outside the program is extremely insecure. Avoid it.***

7. References and nested data structures

- Sometimes it's important to be able to access a variable indirectly
- We want to be able to refer to a variable generically ("that thing there"), rather than directly using its name
- To support that, Perl provides a special scalar datatype called a *reference*
- It's a pointer (more or less)
- That is, it's like the traditional Zen notion of the "finger pointing at the moon"
- The finger (reference) isn't the moon (the variable), merely a means of locating it

Creating references

- We create a reference using the unary backslash (\) operator
- (That is, outside a string it's an operator, not an escape)
- The operator takes a variable (or value) as its single argument and returns a reference to it
- The original variable or value is then known as the *referent* to which that reference refers
- For example:

```
my $ref = \$var;
```

- Now `$ref` stores a reference...
- ...whose referent is `$var`

- We can also take references to arrays and hashes
- Any of these kinds of references fits into a single scalar variable:

```
my $slr_ref = \$s;

my $arr_ref = \@a;

my $hsh_ref = \%h;
```

- Given a reference, we can access its referent by simply prefixing the reference (usually in curly braces) with the appropriate sigil:

```
say ${$slr_ref};      # same as: say $s

say @{$arr_ref};      # same as: say @a

say %{$hsh_ref};      # same as: say %h
```

- References are strictly typed (scalar, array, or hash),
  so using the wrong sigil throws an exception

The "arrow" operator

- Suppose we want to access an element of (for example) a hash, via a reference to that hash
- We first construct an access to the hash itself:

```
%{$hsh_ref}
```

- Then apply the usual rule: change the `%` to a `$`, then append the key in braces:

```
${$hsh_ref}{"key"}
```

- Likewise for array references:

```
${$arr_ref}[$index]
```

- Perl has a second – and cleaner – syntax
- We can use the "arrow operator" (`->`):

```
$hsh_ref->{"key"}

$arr_ref->[$index]
```

- In recent releases of Perl, you can also dereference the entire container via an arrow:

```
use v5.32;

say $slr_ref->$*;      # same as: say $s
say $arr_ref->@*;      # same as: say @a
say $hsh_ref->%*;      # same as: say %h
```

- Suppose we now want to access a "slice" of two or more elements from a hash or array, via a reference
- Once again we first construct an access to the hash or array itself:

```
%{$hsh_ref}
@{$arr_ref}
```

- Then we apply the usual rule for slices: change the sigil to a `@`, then append the list of keys/indexes in braces/brackets:

```
@{$hsh_ref}{"key1", "key2", "etc"}
@{$arr_ref}[$index1, $index2, $etc]
```

- Once again, in recent releases of Perl, there's a cleaner, arrow-oriented syntax:

```
use v5.32;

$hsh_ref->@{"key1", "key2", "etc"}
$arr_ref->@[$index1, $index2, $etc]
```

Requiring non-symbolic references

- There is a historical problem with references
- If we attempt to dereference something that isn't a reference (*e.g.* a number or a string), Perl converts that non-reference to a string, then attempts to look up the string as the name of a variable in its global symbol table
- This is known as a "symbolic reference"
- For example, consider this code:

```
my $hash_ref = get_hash();


if ($hash_ref->{'unreported'}) {
        say "$hash_ref->{'ID'} was seen";
}
```

- But suppose we didn't realize that the call to `get_hash` sometimes returns the string `"null"`
- Would Perl subsequently complain that we gave it a string where it expected a hash reference?
- By default: no, it would not!
- By default, the above code would happily look up the global hash whose name is `%null`
- …and if it didn't exist, Perl would happily autogenerate such a global hash for us
- (Good luck tracking that bug down!)
- To prevent those kinds of mishaps, we can specify: `use strict 'refs';`

- That pragma generates a run-time error whenever we attempt use a symbolic reference:

```
use strict 'refs';

my $hash_ref = get_hash();

if ($hash_ref->{'unreported'}) {      # error: symbolic reference
    say "$hash_ref->{'ID'} was seen";
}
```

- Note that specifying `use strict` automatically implies `use strict 'refs'` (as well as other useful safeguards) so it's best to always just use the general form

Identifying a referent

- Scalar variables can store references to any kind of data (including things we haven't encountered yet, like subroutines and typeglobs and objects)
- But using the wrong sigil when dereferencing is instantly fatal
- So it's useful to be able to ask what kind of referent a reference refers to
- We do that with the `ref` function:

```
my $type_name = ref($reference);
```

- `ref` returns a string: `"SCALAR"`, `"ARRAY"`, or `"HASH"`
- That's particularly handy for improving the error messages of deadly mistakes:

```
REF:
for my $scalar_ref (@list_of_refs) {

    next REF if ref($scalar_ref) ne 'SCALAR';

    ${$scalar_ref}++;
}
```

Printing a reference

- ...is usually a mistake
- Except when debugging
- If a reference is used where a string is expected, it evaluates to a string consisting of reference type followed by hexadecimal internal address of referent:

```
say $hsh_ref;   # outputs: HASH(0x10027588)
```

- Likewise, if a reference is used where a number is expected, the reference evaluates to an integer corresponding to the referent's address in memory

- This is useful for comparing references:

```
if ($next_ref != $prev_ref) {
    push @references, $next_ref;
}
```

- But in most other cases it doesn't produce the desired effect:

```
say $array[$ref];  # outputs contents of $array[140333946068520]
```

- If you need to print out a reference (or an array or hash directly), you probably need one of the "data dumping" CPAN modules: `Data::Dumper`, `Data::Dump`, `Data::Show`, *etc*.

References and anonymous arrays

- References are particularly useful for creating multidimensional data structures
- Recall that nested lists are automatically flattened
- So this doesn't work:

```
my @power_table = (
                    ( 1, 0,  0,  0 ),
                    ( 1, 1,  1,  1 ),
                    ( 1, 2,  4,  8 ),
                    ( 1, 3,  9, 27 ),
                    ( 1, 4, 16, 64 ),
                  );
```

- Because it's flattened to a single list:

```
my @power_table = (1,0,0,0,1,1,1,1,1,2,4,8,1,3,9,27,1,4,16,64);
```

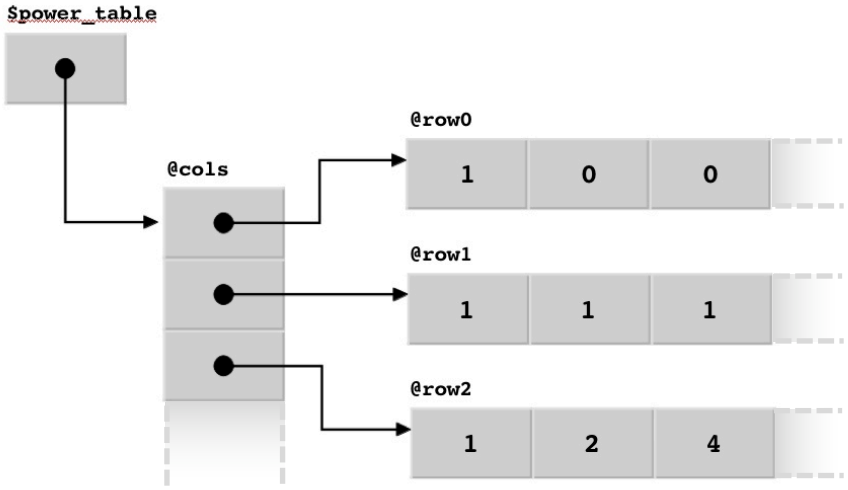
- But each element in a Perl array can store any kind of scalar value
- And a reference is just a special kind of scalar value
- So we can write:

```
my @row0 = ( 1,  0,  0,  0 );
my @row1 = ( 1,  1,  1,  1 );
my @row2 = ( 1,  2,  4,  8 );
my @row3 = ( 1,  3,  9, 27 );
my @row4 = ( 1,  4, 16, 64 );

my @cols = (\@row0, \@row1, \@row2, \@row3, \@row4);

my $power_table = \@cols;
```

- That creates a data structure that looks something like this:



- Note that we used `$power_table`, not `@power_table` as the top-level container
- That was merely to make everything consistently reference-based
- Then we can use the "arrow" notation to access elements of the table:

```
say "3**4 is ", $power_table->[3]->[4];
```

- The `$power_table->[3]` notation means: find the array referred to by the reference in $power_table (i.e. *`@cols`*), then get the element at index *`3`*
- That element stores another reference (a reference to the `@row3`)
- The final `->[4]` means: find the array referred to by *`$power_table->[3]`* (i.e. *`@row3`*)...
- ...then get the element at index *`4`*
- Multidimensional tables are popular, so Perl provides syntactic assistance
- We can specify a list of values in square brackets instead of parens:

```
[1, 3, 9, 27]
```

- The result is not a list, but a reference to a nameless ("anonymous") array
- That array is automatically initialized to the list of values in the square brackets
- So we could rewrite the table:

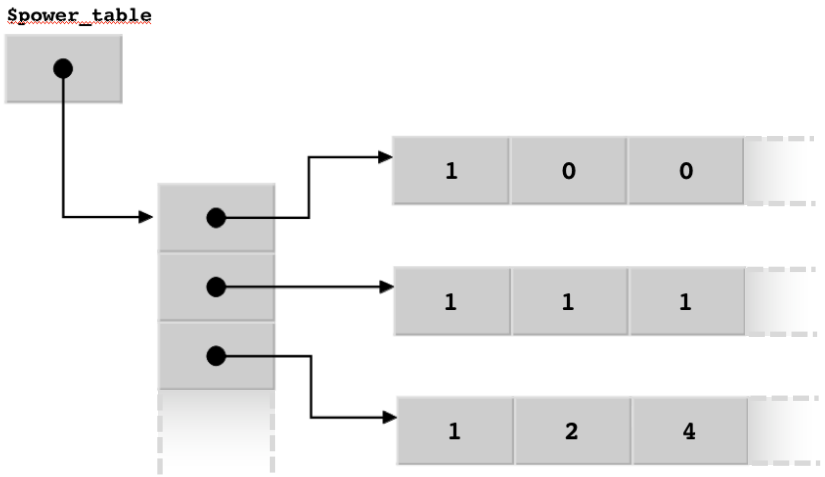
```
my $row0 = [ 1,  0,  0,  0 ];
my $row1 = [ 1,  1,  1,  1 ];
my $row2 = [ 1,  2,  4,  8 ];
my $row3 = [ 1,  3,  9, 27 ];
my $row4 = [ 1,  4, 16, 64 ];

my $power_table = [$row0, $row1, $row2, $row3, $row4];
```

- Or, better still:

```
my $power_table = [
                    [ 1,  0,  0,  0 ],
                    [ 1,  1,  1,  1 ],
                    [ 1,  2,  4,  8 ],
                    [ 1,  3,  9, 27 ],
                    [ 1,  4, 16, 64 ],
                  ];
```

- The resulting data structure now looks like this:



- That is, it's exactly the same as the previous data structure, except that now only the "top" component (`$power_table`) actually has a name
- Perl provides one further piece of syntactic assistance
- In any expression like:

```
say $power_table->[$x]->[$y];
```

- ...the arrow between any closing bracket and following opening bracket is optional:

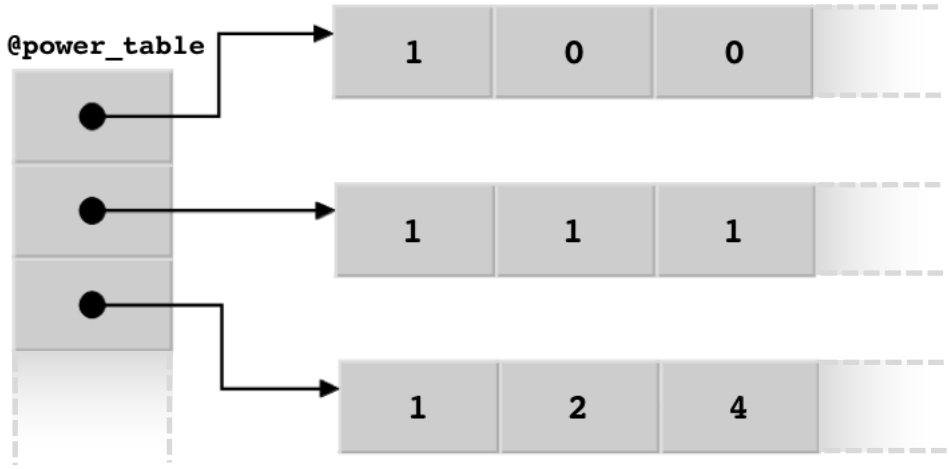
```
say $power_table->[$x][$y];
```

- As a final step, could use an array (instead of a scalar) for the top level:

```
my @power_table = (
                    [ 1,  0,  0,  0 ],
                    [ 1,  1,  1,  1 ],
                    [ 1,  2,  4,  8 ],
                    [ 1,  3,  9, 27 ],
                    [ 1,  4, 16, 64 ],
                  );

# and later...
say $power_table[$x][$y];
```

- The resulting data structure would now look like this:



- Note too that the former square brackets of the top-level of the initialisation have now reverted to parentheses
- (Otherwise, `@power_table` would contain only one element: a reference to the outermost anonymous array)

References and anonymous hashes

- It's also possible to create references to anonymous hashes
- For example:

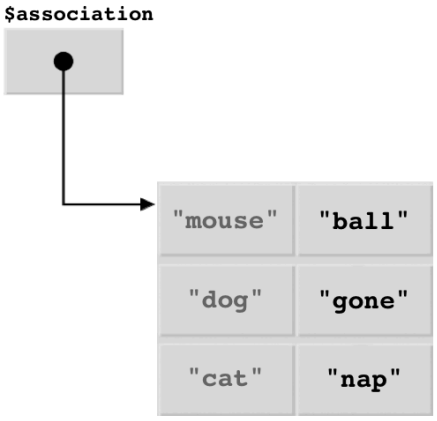
```
my $association = { cat=>"nap", dog=>"gone", mouse=>"ball" };
```

- Which is the same as:

```
my %nameless = ( cat=>"nap", dog=>"gone", mouse=>"ball" );

my $association = \%nameless;
```

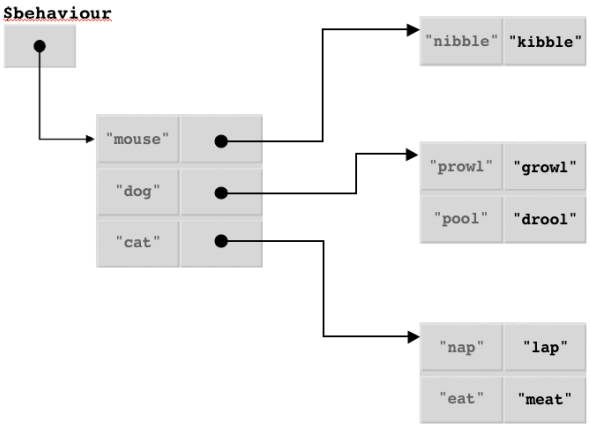
- And creates a data structure like:



- ***Note: Don't confuse blocks (sequence of statements in braces) with anonymous hashes (sequence of key/value pairs in braces)***
- We can even create multilevel hashes:

```
my $behaviour
    = {
        cat   => { nap    => "lap",     eat => "meat"  },
        dog   => { prowl  => "growl",  pool => "drool" },
        mouse => { nibble => "kibble"                  },
      };
```

- The data structure for that would be:



- Accessing that data now requires a chain of arrow operators:

```
say "A cat eats ", $behaviour->{cat}->{eat};
```

- Just as in nested arrays, second and subsequent arrows are optional:

```
say "A mouse nibbles ", $behaviour->{mouse}{nibble};
```

8. Subroutines

- A subroutine is a small, self-contained, user-defined subprogram
- They are often called "functions" or "procedures" in other languages
- The call syntax for subroutines is like that for Perl's in-built functions (`map`, `pop`, `chomp`, *etc*.)
- They are invoked by name
- …and may be passed arguments
- …and may return a scalar or list value

Defining subroutines

- Subroutines defined using the `sub` keyword, followed by subroutine name, followed by the subroutine code in curly braces:

```
sub rev_sort {
    my @ordered = reverse sort @_;
    return @ordered;
}
```

- Arguments to the call are accessed within the subroutine via a special array: `@_`
- The `return` keyword causes execution of the subroutine to finish immediately
- Even if it's not the final statement
- Values specified as arguments to the `return` (if any) are returned as the call's result
- A `return` is always optional
- If no `return` statement is encountered during the subroutine's execution, the subroutine automatically returns the value of the last statement it actually executed
- So we could reduce the previous example to:

```
sub rev_sort { reverse sort @_ }
```

- Subroutines can be defined anywhere in a program, including ***after*** the point they're used

Calling subroutines

- Subroutines are called by specifying their name, followed by a list of arguments:

```
@sorted = rev_sort("eat", "at", "Joes");

@sorted = rev_sort(@unsorted);

@sorted = rev_sort(@sheep, "shepherd", $goatherd);
```

- If the subroutine has already been defined earlier in the code (*i.e.* before it is called), the parens are not required:

```
@sorted = rev_sort "eat", "at", "Joes";

@sorted = rev_sort @unsorted;

@sorted = rev_sort @sheep, "shepherd", $goatherd;
```

- If a subroutine does not require any arguments, it can be called with an empty argument list:

```
sub get_two_lines { return readline() . readline() }

# and later...

$nextlines = get_two_lines();
```

- If it was defined in the code earlier than the call, the argument list can be omitted completely:

```
sub get_two_lines { return readline() . readline() }

prompt();                   # always okay
$next2 = get_two_lines();   # always okay

prompt;                     # error: Perl hasn't seen definition yet
$next2 = get_two_lines;     # okay: sub definition already seen

sub prompt { say "(next 2 lines please)" }
```

- In general, it's best to always supply the parens on any subroutine call, even when they're not required; this makes it much easier to detect that a particular piece of code *is* a subroutine call

Requiring subroutine declarations

- There's another reason to always use parentheses to indicate a subroutine call, even if the call takes no arguments:

```
my $hash_ref = get_hash;
```

- That's because, historically, Perl has always treated any identifier for which it cannot not find a corresponding subroutine as if it were a character string:

```
my @words = ( Look, Ma, No, Quotes );     # Ugh! (but valid Perl)
```

- *(i.e. Perl treats unknown identifiers just like your shell's command line does: as strings)*
- So, if that `get_hash` subroutine hadn't actually been defined anywhere in the program…
- ...or was only defined later, after the actual call…
- ...then the Perl would treat what looks like call to `get_hash` if it were actually the string `"get_hash"`
- *(Yes, really…even though that's crazy)*

- That "helpfulness" is almost always a very bad idea, so there's a `use strict 'subs'` pragma that causes unknown *barewords* like this to trigger compiler errors:

```
use strict 'subs';

my $hash_ref = get_hash;      # Error: undeclared subroutine
```

- Note that specifying `use strict` automatically implies `use strict 'subs'` (as well other useful safeguards) so it's always best to always just to use the general form

Passing arrays (or hashes) as arguments

- An argument list is just like any other list
- If the argument list has nested lists or arrays or hashes, they will be "flattened"
- As mentioned earlier, arrays and lists flatten down to their elements
- Hashes flatten to a list of alternating key/value pairs:

```
say %hash;     # outputs: key1 val1 key2 val2 key3 val3 etc.
```

- To pass arrays or hashes to a subroutine but keep them "unflattened", we need to use references:

```
if ( arrays_equal(\@wanted, \@caught) ) {
    say "We done rounded up the whole dang gang, sheriff!";
}
```

- The implementation of the subroutine would then look like:

```
sub arrays_equal {
   my ($arr1_ref, $arr2_ref) = @_;

   return 0 if @{$arr1_ref} != @{$arr2_ref};

   for my $index (0..$#{$arr1_ref}) {
       return 0 if $arr1_ref->[$index] ne $arr2_ref->[$index];
   }

   return 1;
}
```

Explicit parameter lists

- For most of Perl's history, unpacking the elements of `@_` into a series of my variables was the only way of defining parameter variables
- However, modern versions of Perl (from v5.20 onwards) also allow parameters to be explicitly declared

- This is still an experimental feature, so you must request it explicitly:

```
use experimental 'signatures';

sub arrays_equal ($arr1_ref, $arr2_ref) {

    return 0 if @{$arr1_ref} != @{$arr2_ref};

    for my $index (0..$#{$arr1_ref}) {
         return 0 if $arr1_ref->[$index] ne $arr2_ref->[$index];
    }

    return 1;
}
```

- The specified parameters are treated as lexical variables within the block of the subroutine
- Each parameter variable must be specified as a scalar, except the final one, which may also be specified as an array or a hash
- A final array parameter will be assigned all the remaining arguments that were passed to the subroutine:

```
use experimental 'signatures';

sub sort_as ($direction, $type, @data) {
    @data = ($type eq 'numbers')
               ? sort { $a <=> $b } @data
               : sort @data;

    return reverse @data if $direction eq 'descending';

    return @data;
}

@sorted = sort_as('descending', 'numbers', @unsorted);

@sorted = sort_as('ascending',  'numbers', @unsorted);

@sorted = sort_as('descending', 'strings', @unsorted);
```

- A final hash parameter will be assigned all the remaining named arguments (*i.e.* `name => value, name => value`) that were passed to the subroutine call:

```
use experimental 'signatures';

sub hash_string ($str, %options) {

    $str = fc($str) if $options{'nocase'};

    my $hash_val = 0;
    if ($options{'fast'}) {
        for my $nextchar (split //, $str) {
           $hash_val = $hash_val*33 + ord($nextchar);
        }
    }
    else {
       $hash_val = 2166136261;
       for my $nextchar (split //, $str) {
           $hash_val = ($hash_val ^ ord($nextchar)) * 16777619 % 2**53;
       }
    }

    return $hash_val;
}


$hash_index = hash_string('Perl rocks');

$hash_index = hash_string('Perl rocks', fast=>1);

$hash_index = hash_string('Perl rocks', nocase=>1);

$hash_index = hash_string('Perl rocks', nocase=>1, fast=>1);
```

- Parameters may be marked as optional, by appending an = and a default value:

```
use experimental 'signatures';

sub fill_str($str, $fill_width=length($str)+2, $fill_char=' ') {

    my $str_width   = length($str);
    my $left_width  = int(($fill_width - $str_width) / 2);
    my $right_width = $fill_width - $str_width - $left_width;


    return $fill_char x $left_width
         . $str
         . $fill_char x $right_width;
}

say fill_str('This page intentionally left blank');
say fill_str('This page intentionally left blank', 80);
say fill_str('This page intentionally left blank', 80, '~');
```

- Note that Perl doesn't support parameter types, or other more sophisticated styles of parameter specification or argument passing
- However, there are several modules on CPAN that add these features to the language, for example: Method::Signatures, Kavorka, and Dios

Named arguments

- Sometimes a subroutine will have a much larger number of optional arguments than `fill_str` does
- Indeed, over time `fill_str` itself might accumulate a much larger number of optional arguments (string justification, left vs right fill character, inter-word filling, *etc*. *etc*.)
- For example, consider a `list_dir` subroutine that provides the functionality of a typical directory listing command
- It might take arguments specifying a file name or pattern to list, what type of files to list, whether to list "hidden" files, whether to list details of each file or just its name, whether to separate listing of files and directories, whether to recursively list the contents of directories, how to sort the output, how many columns to format the listing into, and whether to page the output a screen at a time
- We certainly don't want to have to specify every one of those nine parameters every time we call `listdir`:

```
list_dir("./*", "any", 1, 1, 0, 0, "alpha", 4, 1);
```

- Of course, we might arrange that specifying `undef` for a particular argument would select an appropriate default behaviour for that argument
- But such calls would still be ugly and error-prone:

```
list_dir(undef, undef, 1, 1, undef, undef, undef, 4, 1);
```

- Named arguments are the answer:

```
list_dir(cols=>4, page=>1, hidden=>1, sep_dirs=>1);
```

- In this approach, we only specify those arguments whose values differ from the default values
- Extracting the right arguments for each option is easy

- Inside the `list_dir` subroutine, we simply capture the named arguments from `@_` array into a hash:

```
sub list_dir {
    # Unpack named arguments
    my %arg = @_;

    # Use defaults for missing arguments...
    $arg{match} //= "*";
    $arg{cols}  //= 1;
    # etc.

    # Use arguments to control behavior...
    @files = files_matching( $arg{match} );

    push @files, hidden_files_matching( $arg{match} )
            if $arg{hidden};
    # etc.
}
```

- Or, in modern Perl:

```
use experimental 'signatures';

sub list_dir (%arg) {
     # rest of code as previously
}
```

- Named arguments are highly recommended
- Calls become self-documenting
- And we no longer need to remember the order of the nine arguments; just their names
- …which is much easier for (most) humans
- Provided, of course, the names are chosen sensibly
- Better still, if we have several calls that require the same subset of arguments. we can store that subset in a separate hash and reuse it:

```
my %std_listing = (cols=>2, page=>1, sort_by=>"date");

list_dir( file=>"*.txt", %std_listing );

list_dir( file=>"*.log", %std_listing );

list_dir( file=>"*.dat", %std_listing );
```

- We can even choose to "override" specific elements of standard set
- By placing the overriding values **after** the standard set:

```
list_dir( file=>"*.exe", %std_listing, sort_by=>"size" );
```

Subroutine sigils

- Like variables, subroutines have a leading symbol that indicates the kind of thing they are: `&`
- It can be used when calling them:

```
@sorted = &rev_sort("eat", "at", "Joes");
```

- (Though this style of calling subroutines has nasty edge-cases and is frowned upon nowadays)
- We also need to use the sigil when taking a reference to a subroutine:

```
my $sub_ref = \&rev_sort;
```

- Because if we omitted the sigil and just wrote:

```
my $sub_ref = \rev_sort;
```

- ...Perl would immediately call `rev_sort`, then take a reference to the *return value* of a call
- By the way, once we have our subroutine reference, we can call the subroutine using the arrow notation:

```
@sorted = $sub_ref->(@unsorted);
```

- The one place that the `&` sigil **cannot** be used when we're defining a subroutine:

```
sub &rev_sort {   # Fatal compile-time error!
     return reverse sort @_;
}
```

- ***Note: Be careful not to accidentally call a subroutine with its sigil and no argument list. That has a special meaning: pass the current subroutine's `@_` to the subroutine we're calling. A better idea is never to use the `&` sigil, except when taking a reference.***

Variadic subroutines

- A subroutine's arguments are passed in the special array `@_`
- But all Perl arrays are inherently dynamically sized
- That implies that any subroutine may be passed any number of arguments
- For example, that's why `rev_sort` could be given different numbers of arguments
- Likewise, we could write a generic `max` subroutine:

```
sub max {
     my $max = shift @_;

     for my $next_candidate ( @_ ) {
         $max = $next_candidate if $max < $next_candidate;
     }

     return $max;
}
```

- Or, in modern Perl:

```
use experimental 'signatures';

sub max ($max, @candidates) {

     for my $next_candidate ( @candidates ) {
         $max = $next_candidate if $max < $next_candidate;
     }

     return $max;
}
```

- Or even in a pure functional programming style:

```
use experimental 'signatures';

sub max ($head, @tail) {
        @tail == 0       ? $head
      : @tail >  1       ? max($head, max(@tail))
      : $head > $tail[0] ? $head
      :                    $tail[0]
}
```

Aliasing of parameters

- The elements of the `@_` array are special
- They are **not** copies of the actual arguments of the function call
- They are *aliases* (*i.e.* just other names) for those arguments
- That means that, if `$_[0]`, `$_[1]`, `$_[2]`, *etc.* are assigned new values inside a subroutine call...that will change the original arguments that were passed to that subroutine call!
- Or we'll get a fatal exception because the original argument was a literal, not a variable
- Either way, that's probably a bad outcome

- Changing variable arguments can be tempting:

```
sub cyclic_incr {
    $_[0] = ($_[0]+1) % 10;
}

# and later..

$next_digit = 8;
say $next_digit;            # outputs: 8

cyclic_incr($next_digit);
say $next_digit;            # outputs: 9

cyclic_incr($next_digit);
say $next_digit;            # outputs: 0

cyclic_incr($next_digit);
say $next_digit;            # outputs: 1

while (readline()) {
    cyclic_incr($next_digit);
    say $next_digit;
}
```

- But don't try:

```
cyclic_incr(7);             # KA-BOOM!
```

- Even when they work correctly, these kinds of subroutines introduce non-obvious state changes that can be very hard to debug
- It's now generally considered bad practice to create subroutines that modify their arguments; pure functional approaches are considered clearer and safer:

```
use experimental 'signatures';

sub next_cyclic_after ($number) {
    return ($number+1) % 10;
}
```

- Mostly because it makes variable modifications in the client code much more explicit:

```
$next_digit = 9;
say $next_digit;                                # outputs: 9

$next_digit = next_cyclic_after($next_digit);
say $next_digit;                                # outputs: 0

$next_digit = next_cyclic_after($next_digit);
say $next_digit;                                # outputs: 1

while (readline()) {
    $next_digit = next_cyclic_after($next_digit);
    say $next_digit;
}
```

Caller information

- Unlike most languages, Perl makes it easy to determine where a subroutine was called from
- The built-in `caller` function returns a list of values
- The first value is the name of the namespace from which the current subroutine was called
- Then comes the name of the file containing the code that called the current subroutine
- And finally the line in that file at which the current subroutine was called
- We could use that information to create audit trails through our code:

```
use experimental 'signatures';

sub initial_sample ($reaction_rate, $temperature, $pressure) {

     my ($package, $file, $line) = caller();

     return {
         rate    => $reaction_rate,
         temp    => $temperature,
         pres    => $pressure,
         _trail  => "\tinitiated ($file:$line)\n",
     };
}

sub catalyze ($state, $catalyst, $rate_multiplier) {

     my ($package, $file, $line) = caller();

     return {
         %{$state},
         rate     => $state->{rate} * $rate_multiplier,
         catalyst => $catalyst,
         _trail   => $state->{_trail}
                   . "\tcatalysed to $state->{rate} "
                   . "with $catalyst ($file:$line)\n",
     };
}

my $reaction = initial_sample(0.02, 283, 1.014);

$reaction = catalyze($reaction, "AlO2", 1.7);

while ($reaction->{rate} < 5) {
    $reaction = catalyze($reaction, "CO2", 3.3);
}

$reaction = catalyze($reaction, "N2", 0);

say $reaction->{_trail};
```

- When called with an integer argument, `caller` returns even more information
- Most usefully, it also returns the name by which the subroutine was called

- And the integer argument tells `caller` how many extra stack frames to look back for that information
- See the `perlfunc` documentation for more details

Stateful subroutines

- We can also create subroutines that retain information between calls
- Normally, any `my` variable within a subroutine is cleaned up at the end of that subroutine, and is recreated and reinitialized the next time the subroutine is called
- So a subroutine like this will return the value `1` every time:

```
sub generate_unique_ID () {
    my $next_ID = 1;
    return $next_ID++;
}

say generate_unique_ID();    # 1
say generate_unique_ID();    # 1
say generate_unique_ID();    # 1
```

- However, all Perl subroutines are automatically also *"closures"*
- That is: they are able to access lexically scoped variables from the surrounding blocks (or file) in which they are declared
- The compiler then arranges for those variables to be preserved between calls
- So we could make our ID generator subroutine work correctly by hoisting the `$next_ID` variable into an outer scope:

```
my $next_ID = 1;

sub generate_unique_ID () {
    return $next_ID++;
}

say generate_unique_ID();    # 1
say generate_unique_ID();    # 2
say generate_unique_ID();    # 3
```

- But that also means that other code in the surrounding scope could see (and mess with) that variable, which might introduce bugs
- So Perl also provides a kind of lexical variable that is scoped to the surrounding block (like a local variable), but which also retains its values between calls (like a global variable)
- In other languages, this is known as a *"static"* variable; in Perl, as a `state` variable
- State variables are initialized the first time their declaration is reached during execution, and then preserve their value (avoiding reinitialization) thereafter
- State variables can be scalars, arrays, or hashes…but until Perl 5.28 only scalar state variables could be initialized before use. From 5.28 onwards, all kinds of state variables can be initialized.

- Using a `state` variable, we could also fix the `generate_unique_ID` subroutine, and preserve the secure locality of the `$next_ID` variable, like so:

```
sub generate_unique_ID () {
    state $next_ID = 1;
    return $next_ID++;
}

say generate_unique_ID();    # 1
say generate_unique_ID();    # 2
say generate_unique_ID();    # 3
```

9. Regular Expressions

- Perl provides full built-it support for regular expressions
- In fact, supporting regular expressions was one of Perl's original *raisons d'etre*
- If you're accustomed to using Unix style regular expressions, Perl's will seem familiar (though considerably extended)

Writing regexes

- Regular expressions can be written in angle brackets:

```
/hello+ worlds?/
```

- Or with a prefixing letter that tells Perl explicitly what to do with them:

```
$rx_ref = qr/hello+ worlds?/;           # Create ref to a regex

$string =~ m/hello+ worlds?/;           # Match against a regex

$string =~ s/hello+ worlds?/hi there!/; # Match and replace
```

- Without the prefixing letter, a regex automatically matches against `$_`:

```
for (@animals) {
    if (/cat/) {
        say "meow!";
    }
}
```

- With the prefixing letter, you can use any delimiters:

```
$rx_ref = qr{hello+ worlds?};           # Create ref to regex

$string =~ m[hello+ worlds?];           # Match against regex

$string =~ s%hello+ worlds?%hi there!%; # Match and replace
```

Components of regexes

- The escape character within regexes is backslash (\)
- Most unescaped characters match themselves (including all alphanumerics and whitespace)
- The most frequently used special characters are:

Special character	Means...	Example	Would match...
`.`	Any character except newline	`/<.>/`	`<a>` *or* `<B>` *or* `<!>` *or* `<~>` *or...*
`?`	Zero-or-one of the preceding atom	`/fiancée?s?/`	`fiancées` *or* `fiancée` *or* `fiancés` *or* `fiancé`
`*`	Zero-or-more of the preceding atom	`/Help!*/`	`Help` *or* `Help!` *or* `Help!!` *or* `Help!!!` *or ...*
`+`	One-or-more of the preceding atom	`/Khan!+/`	`Khan!` *or* `Khan!!` *or* `Khan!!!` *or...*
`{…,…}`	Match a specified number of the preceding atom	`/Kha{3,5}n!/`	`Khaaan!` *or* `Khaaaan!` *or* `Khaaaaan!` *or...*
`(…)`	Group the enclosed pattern into a single atom (and capture what it matches)	`/(No! )+/`	`No!` *or* `No! No!` *or* `No! No! No!` *or...*
`(?:…)`	Group a pattern into a single pattern (without capturing)	`/(?:No )+/`	`No` *or* `No No` *or* `No No No` *or...*
`[…]`	Match any one of the enclosed characters	`/l[aeiou]st/`	`last` *or* `lest` *or* `list` *or* `lost` *or* `lust`
`^`	Match start of string	`/^Dear/`	`Dear` (but only as the first four characters of a string)
`$`	Match end of string	`/THE END$/`	`THE END` (but only as the last seven characters of a string)
`|`	Match either the preceding pattern, or the following pattern	`/cat|dog|ca+mel/`	`cat` *or* `dog` *or* `camel` *or* `caamel` *or…*

- Most other special characters are specified with a leading backslash:

Special character	Means...
`\t`	Match a tab
`\n`	Match a newline
`\r`	Match a return (CR)
`\R`	Match any Unicode end-of-line marker (newline, CR, LF, CR-LF, any vertical space characters)
`\f`	Match a form feed
`\e`	Match the escape character (ESC)
`\033`	Match octal character 33
`\x1B`	Match hex character 1B
`\x{263a}`	Match wide hex character 236A
`\N{ANKH}`	Match a named Unicode character
`\Q...\E`	Quote (*i.e.* disable) metacharacters between markers
`\w`	Match a "word" character (alphanumeric plus "_")
`\W`	Match a non-"word" character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a digit character
`\D`	Match a non-digit character
`\b`	Match a "word" characer boundary
`\B`	Match a non-("word" character boundary)

Capturing parentheses

- An important feature of Perl regular expressions is the ability to "capture" parts of a string, as the regex engine matches them
- Suppose we wanted to find a match for a series of digits, and then use that series of digits as an integer in our program
- For example, we might want to allow the user to type "month 12" and have the element at index 12 of an array (say `@months`) returned

- To do that we'd have to match the user's input with a pattern like this:

```
while (my $nextline = readline()) {
    $nextline =~ /^month\s+\d+/
       or next;

    # somehow use the chars that matched \d
    # to index @months
}
```

- But how do we extract just the input characters that matched the \d+ bit of the pattern?
- For that, we need to take advantage of a special feature of parentheses used in a regex
- As well as collecting together parts of a regular expression, each pair of parentheses causes the matching engine to "remember" the substring that was matched by the sub-pattern in the parentheses
- In other words, we could write our "month decoder" as:

```
while (my $nextline = readline()) {
    $nextline =~ /^month\s+(\d+)/
       or next;

    # somehow use the captured substring
    # to index @months
}
```

- The presence of the parentheses causes the regex engine to remember the sequence of digits that the \d+ subpattern matches
- After the match is finished (successfully), that remembered substring is available to the program through the special variable `$1`:

```
while (my $nextline = readline()) {
    $nextline =~ /^month\s+(\d+)/
       or next;

    say $months[$1];
}
```

- The variable is `$1`, because we want the substring matched by the *first* pair of parentheses
- A regular expression with multiple parentheses would assign values to `$1`, `$2`, `$3`, *etc*.

```
while (my $nextline = readline()) {
    $nextline =~ /(missed|found) (\w+) at (\d+)/
        or next;

    say "    Verb was: $1";
    say "  Object was: $2";
    say "Position was: $3";
}
```

- If two or more pairs of parentheses are nested, the order of the *opening parenthesis* determines the order in which `$1`, `$2`, `$3`, *etc*. are assigned:

```
while (my $nextline = readline()) {

                    #12          3    4             5
    $nextline =~ /((put|get) (the (\w+) on the (\w+)))/
        or next;

    say " Command was: $1";
    say "  Phrase was: $3";
    say "    Verb was: $2";
    say "  Object was: $4";
    say "Location was: $5";
}
```

- Occasionally, we may want to group things together *without* capturing them
- To do that we use the special parentheses `(?:...)`:

```
#1             2                     3
/((?:put|get) (the (?:\w+) on the (\w+)))/
```

- A substring that's been captured is *immediately* available to the remainder of the pattern currently being matched
- To access the *i*th captured substring within the same pattern we use the specifier `\i`
- So, for example, to match a sequence containing a repeated letter (*e.g.* `"codebook"`, `"vacuum"`, *etc*.), we could use:

```
for my $nextword (@words) {
  say $nextword if $nextword =~ /([aeiou])\1/;
}
```

- This regex means: match a vowel, capture the matched letter (into `$1`), and then match against that captured letter (specified as `\1`)
- Note that, because a `for` loop's iterator, and the `say` function's argument, and the implicit regex matching behaviour all default to `$_`, we could also write that as just:

```
for (@words) {
    say if /([aeiou])\1/;
}
```

- For a short loop like this, that's perhaps clearer and easier to read
- But if the loop block were subsequently to grow bigger under ongoing maintenance, the lack of explicit variables might well make the extended code much more obscure
- Generally, it's better to avoid Perl's many implicit `$_` behaviours and always use explicit well-named variables from the start

Interpolated patterns

- The specification of a pattern doesn't have to be set in stone at compile-time
- Patterns interpolate nested variables in the same way that double-quoted strings do
- For example, if we wanted to search for input lines containing the current user's username, or their UID, or just the word `"user"`, we could write:

```
my $name = getpwuid($<); # special $< var contains proc's real UID

for my $user_data (get_users_data()) {
   say $user_data if $user_data =~ /$name|$<|user/;
}
```

- The variables `$name` and `$<` are interpolated *before* the match is attempted, producing a pattern equivalent to something like this:

```
for my $user_data (get_users_data()) {
   say $user_data if $user_data =~ /damian|15327|user/;
}
```

- The ability to interpolate variables explains why we can't just use `$1`, `$2`, *etc*. to refer to previously captured submatches, but instead have to use `\1`, `\2`, *etc*.
- Like all variables, `$1`, `$2`, *etc*. are interpolated into the pattern *before* it starts matching
- Whereas `\1`, `\2`, *etc*. are re-interpolated *every time* the regex engine tries to match them
- So the pattern `/([aeiou])$1)/` means: match a vowel, followed by whatever any previous pattern match left in `$1`
- That may be useful in its own right, but it almost certainly *won't* match repeated vowels

Recognizing newlines

- Patterns can take various options, either appended as modifiers after their closing `/`, or embedded in the pattern itself in special `(?...)` parentheses
- Two of the most important are the `/m` and `/s` options, which alter the way regular expressions interact with newlines
- Normally, the dot metacharacter (`.`) matches any character, *except* a newline
- That's a handy default because it stops a `/.*/` pattern match from "running away" to the very end of a multi-line string
- But it also means that we have to get tricky to specify a subpattern that matches *every* character
- For example, to specify 3 octothorpes followed by absolutely any character, including newline:

```
/###[\w\W]/
```

- To avoid this obscure construction, we can use the `/s` modifier to cause the regex engine to run in "single-line" mode (*i.e*. pretend it's all one line and treat embedded newlines as ordinary characters)

- The only effect of this option is to cause the dot metacharacter to match newlines as well
- So we could rewrite the "### then anything" pattern as:

```
/###./s
```

- The ^ and $ assertions are normally also indifferent to newlines, in that they usually match only at the very beginning and very end of the string, respectively
- However, if we specify the /m modifier, the regex engine runs in "multiline" mode
- In that mode, ^ will match at the beginning of any line (*i.e.* immediately after a newline) and $ will match at the end of any line (*i.e.* immediately before a newline)
- For example, here's a pattern that matches genetic spam, no matter which line of a mail message it appears on:

```
/^Subject:.*MAKE MONEY FAST!+$/m
```

- A good mnemonic is that the /s modifer changes the value of a single metacharacter (.), whilst the /m modifier changes the behaviour of multiple metacharacters (^ and $)
- Note that, despite the seemingly related names, the two modifiers are completely independent of each other: you can specify either or both of them, depending on which changes of behaviour you need

Extended formatting

- Regexes can get unmanageably visually complicated, long before they get unmanageably semantically complicated
- For example, here's the regex to efficiently match a decimal number:

```
/[+-]?([0-9]+\.?[0-9]*|\.[0-9]+)([eE][+-]?[0-9]+)?/
```

- Code like that is a maintenance nightmare
- You ***can*** put whitespace and comments in a regex (without changing the meaning of the regex), via the /x modifier:

```
/ [+-]?          # An plus or a minus, which is optional
  (              # Then group the following...
    [0-9]+       #   One-or-more digits
    \.?          #   Then an optional dot
    [0-9]*       #   Then zero-or-more digits
  |              # or...
    \.           #   A dot
    [0-9]+       #   Then one-or-more digits
  )              # End of group
  (              # Then group the following...
    [eE]         #   The letter 'e' (either case)
    [+-]?        #   Then a plus or minus, which is optional
    [0-9]+       #   Then one-or-more digits
  )?             # End of group, all of which is optional
/x               # The entire regex uses extended formatting
```

- If you're using regexes regularly in your code, then the /x modifier is ***strongly*** recommended

- You need to be careful, though: under `/x` whitespace is *ignored*, so you need to be explicit when you really want to match a space:

```
$document =~ / software \   patents /x;
$document =~ / software [ ] patents /x;
$document =~ / software \s  patents /x;
```

Putting it all together

- These sample regexes are adapted from *Perl Cookbook*:

Match a `key=value` pair	`/(\w+) \s* = \s* (.+)/x`
Match single lines at least `$n` characters long	`/.{$n,}/`
Match a C comment	`/ \/ \* (.*?) \* \/ /sx`
Match leading and/or trailing whitespace	`/^ \s* (.*?) \s* $/x`
Match an IP address	`$quad = qr{ [01]?\d?\d | 2[0-4]\d | 25[0-5] }x;` `/^ $quad \. $quad \. $quad \. $quad $/x`

Match a floating point number (with optional exponent)	`my $dig = qr{ \d+ (?:\.\d*)? | \.\d+ }x;`
	`my $sgn = qr{ [+-]? }x;`
	`my $exp = qr{ [Ee] $sgn \d+ }x;`
	`/ ($sgn) ($dig) ($exp)? /x`

How regexes are used

- Regular expressions are generally used either for simple matching (`/.../` and `m{...}`) or substitution (`s/.../.../`) or string dissection (`split /.../, $str`) or list refinement (`grep /.../, @list`)
- Matching is the simplest application
- Given a string `$str`, we can match the contents against a pattern by *binding* the string to the pattern using the =~ operator
- This performs the match, extracts any parenthesized components, and returns a value indicating whether the match succeeded

- The most common usage is:

```
if ($str =~ /pattern/) { do_something() }
```

- The return value of a match depends on the context in which the match is specified
- In a scalar context (*i.e.* when the result is assigned to a scalar, or used in a test), a successful match returns `1`, and a failed match returns `""`
- In a list context, a successful match returns a list of all the substrings captured by parentheses during the match; a failed match returns the empty list
- So, for example:

```
while (my $keyval = readline()) {
    my ($key, $value) = ($keyval =~ m{ (\w+) \s* = \s* (.+) }x)
        or next;

    $config{$key} = $value;
}
```

- Matching against `$_` is the default behaviour when no binding specified, so could just write:

```
while (readline()) {
    my ($key, $value) = m{ (\w+) \s* = \s* (.+) }x
        or next;
    $config{$key} = $value;
}
```

- Or we could make use of the `$1` and `$2` capture variables:

```
while (readline()) {
    if (m{ (\w+) \s* = \s* (.+) }x) {
        $config{$1} = $2;
    }
}
```

- Or we could even squish it all into a gapless one-liner:

```
/(\w+)\s*=\s*(.+)/&&($config{$1}=$2)while<>;
```

- …which is how Perl gets its undeserved reputation for being executable line-noise
- The `m{pattern}x` syntax is useful because it allows us to use *any* matched or balanced delimiters, rather than just slashes (`/pattern/`), as well as spacing out the various components of the pattern
- Consider our "dancing toothpicks" example of a C comment matcher:

```
$code =~ /\/\*(.*?)\*\//s;
```

- It would become (slightly) less confusing as:

```
$code =~ m{ / \* (.*?) \* / }sx;
```

Substitution

- Often the reason we want to match a pattern is to locate a substring that needs to be changed
- The substitution operator, `s///`, is the normal way to accomplish that
- It takes a pattern and an (interpolated) string, like so:

```
s/pattern/replacement/
```

- When bound to a scalar variable (say `$str`), it looks for a match within the string contained in `$str` and replaces the first such matching substring with the interpolated text
- For example, we could remove a C comment like this:

```
$source_code =~ s/\/\*(.*?)\*\///s;
```

- That's almost unreadable, so `s///` allows us to use different delimiters, just like `m//`:

```
$source_code =~ s{/\*(.*?)\*/}{}s;
```

- If we wanted to replace a single-line C++-like comment with a Perlish one, we could write:

```
$source_code =~ s{//(.*)}{# $1};
```

- Here the replacement string is `"# $1"`, so we replace any comment with an octothorpe, followed by the contents of the variable `$1` (*i.e.* the comment text matched by the parenthesized subpattern)
- ***Note: because the substitution happens after the match is complete and the substitution is effectively a double-quoted string, we use `$1` (which now has the successfully captured substring) instead of `\1` (which, in an interpolating string, is octal code for the Unicode START OF HEADING control character)***
- We can also interpolate hash or array look-ups in a substitution
- For example, the following line replaces a substring of the form `"<<NAME>>"` with the contents of the corresponding entry in the hash `%interpolate`:

```
$template =~ s/ << ([A-Z]+) >> /$interpolate{$1}/x;
```

- Normally, the `s///` operator only substitutes the first match it finds, so to replace every instance of `"<<NAME>>"` we would have to write:

```
1 while $template =~ s/ << ([A-Z]+) >> /$interpolate{$1}/x;
```

- That's ugly and comparatively slow, especially for a task as commonly needed as global replacement
- So Perl provides the `/g` modifier to simplify the task:

```
$template =~ s/ << ([A-Z]+) >> /$interpolate{$1}/gx;
```

String dissection

- A very common use for regular expressions is finding places to hack a string apart
- For example, if we have data that consists of a series of numbers separated by either whitespace or commas, we will probably want to extract just the numbers (as a list)
- The built-in `split` function allows us to break up a string into a list of substrings
- It does this by looking for substring separators that match the pattern specified as its first argument
- For example, to extract our list of numbers:

```
@number_list = split /\s+|,/, $comma_separated_str;
```

- Normally, `split` throws the delimiters away, so if `$data` contains the string `"1,2 3"`, then `@numbers` is assigned `('1', '2', '3')`
- But if the separator pattern contains capturing parentheses, then the parts of the separator matched by each pair of parentheses are added into the returned list, *between* the split substrings
- For example, if it was important to know when two numbers were separated by a comma, we could write:

```
@number_list = split /\s+|(,)/, $comma_separated_str;
```

- Now, if `$comma_separated_str` contained `"1,2 3"`, then `@number_list` would be assigned: `('1', ',', '2', undef, '3')`
- The `undef` appears between the `2` and `3` because in that splitting the capturing parens of the separator pattern didn't capture anything
- If we omit the string to be split from `split`'s argument list, the contents of `$_` are split instead, which is handy (but also potentially confusing) in input processing loops
- For example, here's an easy way to count the number of occurrences of words in an input sequence:

```
my %wordcount;

while (readline()) {
    for (split /\s+/) {
        $wordcount{$_}++;
    }
}
```

- Splitting on whitespace is a common activity, so that is `split`'s default behaviour whenever we leave out the pattern as well:

```
my %wordcount;

while (readline()) {
    for (split) {
        $wordcount{$_}++;
    }
}
```

- At which point a one-liner may seem tempting:

```
my %wordcount; while (readline()) { $wordcount{$_}++ for split }
```

- Yet again this illustrates the short-term convenience, and long-term disadvantages, of using `$_`
- A much more maintainable solution would be:

```
my %wordcount;

while (my $nextline = readline()) {
    for my $nextword (split /\s+/, $nextline) {
        $wordcount{$nextword}++;
    }
}
```

- `split` has another special behaviour when the pattern we use matches an empty string
- In such cases, it splits between every individual character in the string
- For example, to count the frequencies of individual characters (rather than word frequencies), we could write:

```
my %charcount;

while (my $nextline = readline()) {
    for my $nextchar (split //, $nextline) {
        $charcount{$nextchar}++;
    }
}
```

- This is handy, but also a source of peril
- For example, trying to `split` on a *completely optional* pattern (which also matches nothing) is a common mistake:

```
@number_list = split /\s*,?\s*/, $comma_separated_str;
```

- The correct solution is to write a more sophisticated regex that always matches something:

```
@number_list = split / \s*,\s* | \s+ /x, $comma_separated_str;
```

List refinement

- Yet another important use for regular expressions is as the "selector" for a `grep` operation
- A `grep` acts analogously to a Unix grep command: it selects certain elements from a list
- One of the ways in which it can do that is to match each element against a regular expression, and return only those elements that do match
- So if we need only the items in a list that refer to certain topics we could write:

```
my @legalese = grep /su(ed?|ing)|law\s*suit|plaintiff/, @lines;
```

- Or we could grab those lines directly from a file:

```
open(my $news_fh, "headlines")
    or die "No data!";

my @legalese
    = grep /su(ed?|ing)|law suit|plaintiff/, readline($news_fh);
```

- The list context of `grep`'s argument list causes the `readline` to return every line it can read, as a list of strings, which we then filter for suitable word matches
- Using exactly the same structure, we could extract telephone numbers:

```
my $one800 = qr{ 1 [- ]? 800 [- ]? \d{3} [- ]? \d{3} }x;

my @freecalls = grep $one800, readline($phonebook_fh);
```

- If it's called in a scalar context, `grep` returns the count of list elements for which the match was successful
- So we could determine the comparative frequencies of particular top-level domains like so:

```
my %freq;

for my $domain (qw(com org net)) {
    $freq{$domain} = grep /[.]$domain\b/, @URLs;
}
```

Appendix A. The CPAN

- This distributed repository of free Open Source code is perhaps Perl's single greatest strength
- A *huge* archive of free code, much of it very high quality
- As well as binary distributions, documentation, and FAQs
- Archived in the Comprehensive Perl Archive Network
- Universally known as CPAN (pronounced "see-pan")

How to access and search the CPAN

- CPAN has a dedicated on-line search engine: https://metacpan.org/
- On this website you can search the entire CPAN archive
- We can use exact and prefix-only pattern-matching on archive file name, module name, script name, package name, module or script description, contents of module documentation, or author's name or ID

How to install modules from the CPAN

- If you're on a Unix, Linux, or MacOS system, by far the easiest way to locate, download, and install CPAN modules is with the `cpanm` utility
- To download and install `cpanm` see: https://cpanmin.us/
- Then you can install modules like so:

```
> cpanm Module::Name
```

- If you're on a Windows platform, you may also have the `ppm` facility installed:

```
> ppm
```

- If you don't have any of these facilities you may have to do it "by hand"
- Start with the documentation:

```
> perldoc perlmodinstall
```

Appendix B. Documentation

- Perl comes with its own documentation markup format, known as "POD"
- It's deliberately very simple
- And it's fully integrated with both the language and the compiler
- We can intersperse documentation and code in one file
- Documentation processors are available to produce plaintext, HTML, DSR, roff, RTF, HLP, XML, DocBook, LaTeX, PalmDoc, *etc*. *etc*.
- For full details: `perldoc perlpod`
- Here's a summary…

Ordinary paragraphs

- Most documentation is formatted as ordinary blocks of text
- In POD such text starts at column 1
- With a blank line before and after each paragraph
- Such paragraphs will be rewrapped when formatted
- For example:

```
The subroutines provided by this module may be used to
extract various types of delimited substring, possibly after
skipping a specified prefix string. By default, that prefix
is optional whitespace, but you can change it to whatever you
wish (see below)
```

Formatting codes

- There are a small number of "escapes" that allow us to specify simple character formatting within ordinary blocks and headings
- **Bold text**: `B<Bold text>`
- *Italic text*: `I<italic text>`
- Code: `C<Code>`
- HTML éñţïţĭēs: `E<eacute>` E<0x00F1>
- Links: `L<Links|"Some heading in this document">`
  `L<the Perl website|http://www.perl.org>`
- For example:

```
The C<extract_delimited> subroutine should B<not> be
used when attempting to extract so-called
I<bracket-balanced blocks>E<trade>. For a way to extract these
see L<"Extracting bracketed substrings">.
```

- The `E<...>` code can be given an HTML entity name, or an ASCII/Latin-1/Unicode number (in decimal, octal, or hexadecimal)

Verbatim Paragraphs

- These are not rewrapped or reformatted in any way
- Formatting codes inside them are ignored
- They are usually used for presenting code or other types of source
- Specify a verbatim paragraph by indenting (with at least one space or tab):

```
The C<extract_delimited> subroutine is used as follows:

    ($extracted, $prefix, $remainder) =
         extract_delimited($str);
```

Command Paragraphs

- Are used to mark sections, headings, or lists
- All begin with an equals sign (=) in column 1 and end at the next empty line
- Headings: `=head1`, `=head2`, `=head3`, `=head4`

```
=head1 The care and feeding of Camels

This document explains how to care for various species of camels.

=head2 The glorious history of camels

Camels are without a doubt nature's crowning glory and
most wondrous creation.
```

- Lists must start with an `=over` (which also indents them when rendered)

- Each item is marked with an `=item`
- The list ends with a `=back` (which outdents the text from that point)
- Lists can be nested:

```
=head2 The various species of camelids

The camel family tree looks like this:

=over

=item True camels

=over

=item The dromedary

=item The bactrian

=back

=item Other relatives

=item The llama

=item The alpaca

=item The guanaco

=item The vicuE<ntilde>a

=back

=back
```

- To indicate the start of an embedded POD section within a source file use: `=pod`
- To indicated the end of an embedded POD section within a source file use: `=cut`

Appendix C. Debugging

- Whenever the `perl` interpreter is invoked with the `-d` option, it runs in its integrated debugging mode
- It's a very powerful debugger, which is described at length in the `perldebug` documentation
- But we can get most of the benefits by knowing just a small subset of its commands...
- The `h` *`topic`* command prints out help on specified topics
- The `x` *`expr`* command prints out the result of the specified expression
- The `n` command steps to the next statement (treating any subroutine call as atomic)
- The `s` command steps to the next statement (stepping **into** any subroutine calls)
- The `b` *`lineno`* command sets a break-point at the specified line

- The `c` command continues execution until the next break-point
- The `r` command continues execution until after the next `return` statement (*i.e.* till the end of the current subroutine)
- The `l` command lists the next executable line
- The `l` *`subname`* command lists the specified subroutine
- The `l` *`lineno`* command lists the specified line number
- The `l` *`lineno-lineno`* c ommand lists the specified range of line numbers
- The `v` command views a window of code around the next executable line

"Manual" debugging

- One particularly useful debugging technique is to manually hard-code a breakpoint in our own source
- When running under `-d`, the interpreter checks the status of the special `$DB::single` package variable at each step
- If the variable is true, the debugger halts before executing the next step
- So we can set an explicit break-point like so:

```
okay_code();

$DB::single = 1;    # e.g. Debugger single-step-mode becomes true

suspect_code();
```

- Another very useful "manual" technique is to explicitly print out the value of suspect variables, usually with the standard Data::Dumper module:

```
use Data::Dumper 'Dumper';

say Dumper \%suspect_variable;
```

- …or with the more-user-friendly CPAN module, Data::Dump:

```
use Data::Dump 'ddx';

ddx \%suspect_variable;
```

Appendix D. Other features of Perl

Object orientation

- Perl has a sophisticated – but unusual – OO system
- Most languages predefine how object storage is implemented (*i.e.* as records)
- Perl allows us to use *anything* as the basis for objects: hashes, arrays, scalars, even subroutines
- The language also supports class and per-object methods, class and per-object attributes, multiple inheritance, and dynamic reclassification of objects
- Start with the `perlootut` introductory manpage and then read the `perlobj` reference manpage
- Perl's OO system is very "bare bones", which means it can easily be extended or re-interfaced (indeed, it **has** been extended and reiniterfaced) to support declarative class syntaxes, method redispatch, multimethods, persistent objects, classless OO schemes, and aspect-oriented techniques, and much more
- Nowadays, very few people write OO Perl code using only the built-in mechanisms
- Mostly they use some kind of helper CPAN modules that provide more and safer features
- The most popular such modules are (in approximate order of popularity): `Moo`, `Moose`, `Class::Accessor`, `Object::InsideOut`, `Class::Tiny`, and `Dios`
- There are so many because they represent different tradeoffs between power, convenience, extensibility, maintainability, compiletime overhead, runtime performance, and memory footprint

Operator overloading

- An adjunct to Perl's OO
- We can overload built-in operators to vary their behaviour when operands are objects of specific classes

Multiprocessing

- Perl supports "fork" operations
- Even on some systems that don't support multiple processes (*i.e.* via emulation)
- Perl also has a usable (if not very satisfactory) single-process threading system

System calls

- Many Unix system calls are built-in Perl keywords
- And can be used directly from Perl
- These include such favorites as: `alarm`, `chdir`, `chmod`, `chown`, `crypt`, `exec`, `fcntl`, `flock`, `fork`, `gethostbyaddr`, `getppid`, `getpwname`, `gmtime`, `kill`, `link`, `localtime`, `mkdir`, `pipe`, `rmdir`, `select`, `sleep`, `stat`, `truncate`, `umask`, `wait`

IPC

- Perl also has a large range of interprocess communications calls (both BSD and SysV style) already built-in
- And many more available through standard libraries
- For example:

```
use Socket;

my ($remote,$port, $iaddr, $paddr, $proto, $line);

$remote  = shift || 'localhost';
$port    = shift || 2345;  # random port

if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }

die "No port" unless $port;

$iaddr   = inet_aton($remote)  or  die "no host: $remote";
$paddr   = sockaddr_in($port, $iaddr);
$proto   = getprotobyname('tcp');

socket(*SOCK, PF_INET, SOCK_STREAM, $proto)  or  die "socket: $!";

connect(*SOCK, $paddr)  or  die "connect: $!";

while (defined($line = readline(*SOCK))) {
    print $line;
}

close (SOCK) or die "close: $!";
```

Unicode

- Since version Perl v5.6, Perl has allowed Unicode characters in character strings
- The language also supports Unicode properties as character classes in regexes
- Recent versions also support full Unicode identifiers in source code via the `use utf8` pragma

Web development

- There are numerous standard modules and CPAN modules that greatly simplify web programming
- From simple webpage generation tools
- Right up to full content management and generation systems
- The three best and most popular modern web frameworks for Perl are: *Catalyst*, *Mojolicious*, and *Dancer*

Databases

- Perl has strong support for access to just about any kind of database
- Including: Adabas, ADO, CSV, DBase, DB2, EmpressNet, Excel, Fulcrum, Google, Illustra, Informix, Ingres, InterBase, JDBC, LDAP, mSQL, Multiplex, mysql, Oracle, Ovrimos, PgSPI, PrimeBase, QBase, Solid, Sprite, Sybase, Teradata, Unify, XBase
- There's also a uniform access interface (just plug in the appropriate back-end)
- The most commonly used APIs are the CPAN modules: DBI and DBIx::Class

Appendix E. Where to find out more

- *Learning Perl (6th Edition)*, Randal Schwartz, brian d foy, & Tom Phoenix, O'Reilly, 2011
- *Modern Perl (4th edition)*, chromatic, Pragmatic Bookshelf, 2015
- *Perl 5 Pocket Reference* (5th Edition), Johan Vromans, O'Reilly, 2011
- *Programming Perl (4th Edition)*, Tom Christiansen, brian d foy, & Larry Wall, O'Reilly, 2012
- *Perl Best Practices*, Damian Conway, O'Reilly, 2005
- https://www.perlmonks.org
- https://learn.perl.org
- https://www.perl.org
- https://www.metacpan.org

- *Think Raku: How to think like a computer scientist (2nd edition),* Laurent Rosenfeld, with Allen B. Downey, Green Tea Press, 2020
  https://raw.githubusercontent.com/LaurentRosenfeld/think_raku/master/PDF/think_raku.pdf
- https://docs.raku.org
- https://raku.guide/