Advanced Perl

What's New in Perl v5.10?

Tom Christiansen

tchrist@perl.com
http://training.perl.com/

Course Outline

- These notes hold about *2-3x more* than we'll cover.
- We'll skip orientating review material from the larger chapters which isn't new to v5.10.

Introduction

- Perl version 5.10.0 was released 2007-Dec-18, exactly 20 years to the day after the very first version hit *mod.sources* on 1987-Dec-18.
- The last version of Perl fully documented by Perl's reference book, the Camel, meaning *Programming Perl* (3rd edition; July 2000; O'Reilly), covered Perl version 5.6, itself released on 2000-Mar-22.
- Its companion volume of examples, *Perl Cookbook* (2nd edition; August 2003; O'Reilly) covered Perl version 5.8.1, itself released on 2003-Sep-25.
- No matter which of those dates you start from, a truly staggering amount of fine and hard work by many, many people has gone into making Perl better.
- Not content with merely crunching bugs and adding tests and docs, entire sub-systems have been rewritten and several new ones added, new built-in functions, operators, and syntax, extensions to the object interface made, and many new modules added from CPAN and elsewhere.

But What about Perl6?

- Yes, there is indeed another team working on producing something called "Perl6".
  - This is *not* a Perl6 talk; it is a Perl5 talk, and mostly a Perl v5.10 talk.
  - Perl6 should be consider a work-in-progress, and has been for many years.
- In July 2000, Larry Wall announced to the world that there'd be Perl6, and began accepting RPCs for it. He received 361 of them. They've been working on it ever since.
- Although Perl 5.10 has borrowed a few elements from Perl6, it is entirely its own creature, and shall remain so.
  - Perl5 is not going away; Perl6 is not here yet. When Perl6 gets here, Perl5 is *still* not going away.
  - Perl6 will have a Perl 5 compatibility mode to handle any Perl5 it recognizes, mostly modules.
- For more information on the Perl6 effort, start at *http://dev.perl.org/perl6/faq.htm* and then work your way around the site.

Early History

- **Here's a brief synopsis of Perl's early release history.**

1.000	1987-Dec-18
2.000	1988-Jun-05
3.000	1989-Oct-18
4.000	1991-Mar-21
4.020	1992-Jun-08
4.036	1993-Feb-05
5.000	1994-Oct-17
5.001	1995-Mar-13
5.002	1996-Feb-29
5.003	1996-Jun-25
5.004	1997-May-15

Twice the Fun

- **For a while, we had concurrently-released odd-numbered development releases, even-numbered maintenance ones.**

```
5.005    1998-Jul-22      5.6.0  2000-Mar-22
5.005_03 1999-Mar-28      5.6.1  2001-Apr-08
5.005_04 2004-Feb-23      5.6.2  2003-Nov-15
5.005_50 1998-Jul-26
5.5.670  2000-Feb-29      5.8.0  2002-Jul-18
                          5.8.1  2003-Sep-25
5.7.0    2000-Sep-02      5.8.2  2003-Nov-05
5.7.1    2001-Apr-09      5.8.3  2004-Jan-14
5.7.2    2001-Jul-13      5.8.4  2004-Apr-21
5.7.3    2002-Mar-05      5.8.5  2004-Jul-19
                          5.8.6  2004-Nov-27
5.9.0    2003-Oct-27      5.8.7  2005-May-30
5.9.1    2004-Mar-16      5.8.8  2006-Jan-31
5.9.2    2005-Apr-01
5.9.3    2006-Jan-28
5.9.4    2006-Aug-15
5.9.5    2007-Jul-07
```

- **At last, the grand unification finally arrived on 2007-Dec-18 with the release of v5.10.0!**

Building and Installing Perl v5.10

- The build process is the same as it ever was:
- First you run *Configure*, which asks a million questions unless you pass it *-des*, or else you run *configure.gnu*, a wrapper that does it for you:

```
$ ./Configure -Dusethreads -Duse64bitint -des
```

```
$ ./configure.gnu
```

- Now you have your makefiles, so just type make!

```
$ make test && sudo make install
```

- Oh, and *before* you do any of that, first carefully read the entire *INSTALL* file for important details, :)

How Big Is it?

- The tarfile, which is on your CD, isn't too big, only about 15M. However, it does contain 4,680 files!
- Once built, the build directory takes up about 6,000 files and 108M. That's not the installation size, though.
  - ∗ Here are some sample file-sizes of ealier Perl releases:

```
% cd /usr/local/bin
% ls -1sh perl1 perl4.036 perl5.00502 perl5.8.7 perl5.10.0
 160 perl1
 320 perl4.036
 544 perl5.00502
 936 perl5.8.7
1184 perl5.10.0
```

  - ∗ More specifically, here are exact sizes for different platforms and configurations:

```
1,033,526   OpenBSD.i386-openbsd
1,198,294   OpenBSD.i386-openbsd-thread-multi-64int
1,209,952   i686-linux-thread-multi-64int
1,264,091   OpenBSD.sparc-openbsd
```

How Big Does it Run?

Here are sample in-core footprints:

```
   PID  VSZ   RSS TSIZ COMMAND
% perl1 -e 'system "ps wwv$$";'
 26051  224   484   92 perl1 -e /tmp/perl-ei26051
% perl4.036 -e 'system "ps wwv$$";'
  6152  348   592  244 perl4.036 -e system "ps wwv$$"
% perl5.00502 -e 'system "ps wwv$$";'
  4791  600   772  452 perl5.00502 -e system "ps wwv$$"
% perl5.8.7 -e 'system "ps wwv$$";'
  7264  920  1120  744 perl5.8.7 -e system "ps wwv$$"
% perl5.10.0 -e 'system "ps wwv$$";'
  1759 1128  1288  972 perl5.10.0 -e system "ps wwv$$"
```

How Much Gets Installed

- This varies considerably from one system to the next.
- On one of my OpenBSD files, where I installed everything except HTML versions of the manpages, it installed 2,339 files comprising 48,770,987 bytes. Of these:
  - 39 were commands placed in */usr/local/bin*,
  - 682 were manpages placed in man1 and man3,
  - 524 were podpages placed in */usr/local/lib/perl5/5.10.0/*
  - 184 were `.pm` Perl modules placed somewhere under */usr/local/lib/perl5/5.10.0/*, of which 73 were pragmata.
- The *unicore* directory holds 510 files, of which 427 are `.pl` files. These are just datafiles, really, for the the Unicode Character Database, upgraded to version 5.0 for this release.

Compatibility Concerns

- **Just as Perl v5.8.0 was not binary-compatible with earlier releases before it, so too is Perl v5.10.0 binary-incompatible with all previous releases.**
- **You must therefore recompile and reinstall all XS modules you may have added. Non-XS modules will be picked up from earlier *site_lib* installations, but compiled ones won't.**
- **Alternatives include:**
  - **Don't link */usr/local/bin/perl* to */usr/local/bin/perl5.10.0* and specify an exact version number on the `#!` line.**
    ```
    #!/usr/local/bin/perl5.10.0
    ```
  - **Give it an entirely different name at install time:**
    ```
    $ sudo make install PERLNAME=myperl
    ```
  - **Give it an entirely different directory struct at build time:**
    ```
    $ sh Configure -Dprefix=/opt/perl5.10.0
    $ sh Configure -Dprefix=/opt/newperl -des
    ```
- **But you'll still confuse your users.**

Painless Autobundling

- **If this sounds like a pain, you're right; it is — if you try to do it piecewise and by hand.**
- **The *CPAN* module's autobundling feature does much toward making it a "painless upgrade".**
- **Choose the latest version of Perl you last installed, and do this:**

```
$ sudo perl5.8.8 -MCPAN -e autobundle
   CPAN: Storable loaded ok
   Going to read /home/tchrist/.cpan/Metadata
   Database generated on Mon, 19 May 2008 13:31:58 GMT

Package namespace    installed  latest in CPAN file

Class::Multimethods    1.70      1.70
       D/DC/DCONWAY/Class-Multimethods-1.70.tar.gz
Coy                    undef    undef
       D/DC/DCONWAY/Coy-0.05.tar.gz
Getopt::Declare        1.09      1.11
        D/DC/DCONWAY/Getopt-Declare-1.11.tar.gz
Inline::Files          0.60      0.62
        D/DC/DCONWAY/Inline-Files-0.62.tar.gz
Lingua::EN::Inflect    1.88      1.89
        D/DC/DCONWAY/Lingua-EN-Inflect-1.89.tar.gz
```

Now what?

- **In my case, after 568 lines of this, it said:**

```
Wrote bundle file
/home/tchrist/.cpan/Bundle/Snapshot_2008_05_20_00.pm
```

- **I was "lucky" because those 568 different package namespaces were spread across only 182 modules, some already in the core.**
- **Now just type**

```
$ sudo perl5.10.0 -MCPAN \
      -e 'install Bundle::Snapshot_2008_05_20_00'
```

- **Assuming you've got your CPAN module configged to automagically follow dependencies, it *will* Do The Right Thing — eventually.**
- **But best take a coffee break. Tomorrow morning. In a city on another continent. And don't come back for a while. Maybe a *long* while.**
- **I think mine may *still* be running, especially on my truly slow and ancient machines.**

So What Else Did You Break?

- Hey! That’s not a very constructive attitude, now is it? :-)
- The answer depends on where you’re coming from and what code you have.
- Perl v5.8 changed/broke a few things, but nobody much seemed to notice these.
- That because v5.8 mostly just added *lots* of threatening deprecations about what it was *intending* to break someday.
- The bad news is that today is that day, because you’ve just installed v5.10.
- Good for you!

Major Incompatible Changes in v5.10

- **Semantics of `pack` and `unpack` with respect to Unicode data changed, so programs may behave silently differently.**
- **Two ancient, misunderstood, seldom-used, and long-ago deprecated built-in variables were removed entirely.**
  - `$#`**, the default output format for numbers interpolated into strings**
  - `$*`**, which has always been emulatable on a per-pattern basis using the** `/s` **and** `/m` **modifiers.**
- **Attempts to use either of these will generate a "severe" warning. That's one that's on even if you don't ask for it.**

```
% perl -E '$* = 1; say "that is all"'
$* is no longer supported at -e line 1.
that is all
```

- **Because there was a time when you *had* to use these, several of my most ancient of all scripts needed tweaking, both to shut the severe warnings off and to make the scripts work correctly.**

Other Incompatible Changes in v5.10

- The old, unreliable `use Thread` module from v5.5 — the pthreads model — has been completely replaced by the ithreads model available with the `use threads` pragma.
- Some compatibility remains: the Thread module is now a wrapper for the pragma. But the two models were fundamentally different, and old code that used the previous incarnation of the Thread module are not guaranteed to work under the new one.
- Lexical pragmas now propagate into `eval("STRING")` constructs. This includes any overloadedness of constants.
- Some things are now rightfully tainted which should have been.
- Signal-handling has again changed. See that unit.
- Pseudo-hashes have been removed.

Some Prototypes Have Changed

- **Starting with v5.10, some core functions return different prototypes than they used to.**

```
% perl5.6.1 -le 'print prototype("CORE::int")'
;$

% perl5.10.0 -E 'say prototype("CORE::int")'
_
```

- **As you might guess, it's for allowing your own functions to default to `$_`.**
- **As the last character of a prototype, or just before a semicolon, a `_` means if that argument is left off, you'll get `$_` instead.**
- **Several more functions now default to `$_` if left without an argument, like `unpack` and `mkdir`.**

Simultaneous Prototypes

- **You may also backslash several prototypes at once by backslashing a bracketed set of any of the valid prototype indicators, as in**

  ```
  sub hash_or_array(\[%@]) { ... }
  sub anyref(\[$@%&*]) { ... }
  ```

- **This lets those functions accept any of those types, all of which will be passed by reference.**
- **This way you can emulate functions like `tie` that take anything and have it passed to them by reference.**

No More perlcc & Co

- `perlcc`, the byteloader, and supporting modules are no longer distributed with the Perl core distribution sources. Here's why:
  - They never worked reliably.
  - No one stepped up to the job of keeping them in line with interpreter developments.
  - They were therefore removed rather than shipping broken versions.
  - The last version of those modules can be found with perl v5.9.4.
- However, the *framework* that made the B compiler possible remains supported in the perl core.
- That way the many useful modules that grew out of that effort are still available, such as `B::Deparse`, `B::Lint`, `B::Xref`, etc.

Remaining B Modules: `B::Deparse`

- **`B::Deparse` is like the old *cparen* program, a code formatter, and disassembler, too.**
- **It's used by marshalling modules to serialize CODE refs in user data structures.**
- **To demo, make an alias to call the module correctly:**

```
% alias unperl "perl -MO=Deparse,-q,-p,-x=9"

% unperl -e '$a = $b + $c * $d ** $e'
($a = ($b + ($c * ($d ** $e))));

% unperl -e 'print -$a ** $b ** $c'
print((-($a ** ($b ** $c))));

% unperl -lane 'print $F[0] + $F[-1]'
BEGIN { $/ = "\n"; $\ = "\n"; }
LINE: while (defined(($_ = <ARGV>))) {
    chomp($_);
    (our(@F) = split(' ', $_, 0));
    print(($F[0] + $F[-1]));
}
```

More `B::Deparse` Examples

- **It’s not just interesting with command-line arguments:**

```
% unperl -le 'print "This $pkg and those @things are ready,"'
BEGIN { $/ = "\n"; $\ = "\n"; }
print(((('This ' . $pkg) . ' and those ') . join($", @things)) . ' are ready,'));
```

```
% unperl -E 'say "This $pkg and those @things are ready,"'
BEGIN {
    $^H{'feature_say'} = q(1);
    $^H{'feature_state'} = q(1);
    $^H{'feature_switch'} = q(1);
}
say(((('This ' . $pkg) . ' and those ') . join($", @things)) . ' are ready,'));
```

- **You may also run it on entire files:**

```
% unperl EG/tchrist-scripts/fstops
foreach $delta ((1, 2, 3, 6)) {
    print((('1/' . $delta) . ":\t"));
    for (($stop = (-1)); ($stop <= 12); ($stop += (1 / $delta))) {
        ((not (sprintf('%6.4g', (1.4142135623730951455 ** $stop)) =~ /\./))
            and print("\n"));
        printf('%6.4f ', (1.4142135623730951455 ** $stop));
    }
    print("\n\n");
}
```

Possible Upgrade Glitches

- **The good news is that I had no problems successfully configuring, building, and installing perl v10.0.**
- **The bad news is that trouble showed up when I tried to update my sitelib modules from CPAN.**
  - ∗ **My process limits were set too low.**
  - ∗ **My old list of CPAN mirror sites was out of date.**
  - ∗ **If the default fetcher is *lynx*, it does the wrong thing with temp file names.**
- **All these required manual intervention to fix.**

New Core Functionality

- Due to the very strong urge to preserve backwards compatibility with nearly all existing code, it's always been next-to-impossible to add new features to core perl.
- To do so and be guaranteed it was safe, you had to pick something that was previously syntactically illegal.
- This occurred with variables names and with patterns, but was especially problematic with keywords.
- This precept led to an ever more odd-looking and hard-to-remember things to boggle new Perl programmers, and sometimes old ones, too.
- To address this, the `use feature` pragma has been added.

The use feature Pragma

- It’s a lexical pragma, just like `use strict` and `use warnings`, and like them, is also rescindable via `no feature`. The following features are currently available:
- `use feature "switch"` adds a Perl6-like switch construct to Perl via `when(){...}` and `given`.
- `use feature "say"` adds a new built-in function, `say`.
- `use feature "static"` adds a new `state` keyword.

Loading Up with Features, Explicitly

- **You may ask for one or more named features:**

```
use feature "switch";
use feature qw(switch say state);
```

- **Or you can enable all features from a particular release using a colon group:**

```
use feature ":5.10.0";

use feature ":5.10";
```

- **Those two are currently equivalent, but as the subversion goes higher, like 5.10.3, the second is meant to load all features up to and including the 5.10.*X*, not just 5.10.0 as it does now.**

Loading Up with Features, Implicitly

- You'll implicitly enable all features from a given release (and before), in several ways.
- An explicit `use VERSION` loads all features up to that release:

```
use v5.10;
use 5.10.0;
use 5.010;
use 5.01;
```

- Or from the command line, as though it were a module:

```
$ perl -M5.10.0  ....
```

- Or using the new `-E` command-line switch, which is exactly like the old `-e` switch apart from implicitly enabling all optional features.

Caution with Version Numbers

- **Be careful with how loosely you sling around version numbers.**
- **This blows up, because it thinks 5.10 means five dot one-hundred.**

```
% perl5.10.0  -e 'use 5.10'

Perl v5.100.0 required (did you mean v5.10.0?)--this is
only v5.10.0, stopped at -e line 1.
BEGIN failed--compilation aborted at -e line 1.
```

- **Make sure to use a leading `v`, a 3-part dotted version of major version, minor version, and subversion numbers, or else a correct floating representation:**

```
% perl5.10.0 -e 'use v5.10'
% perl5.10.0 -e 'use 5.10.0'
% perl5.10.0 -e 'use 5.010'
% perl5.10.0 -e 'use 5.010_000'

% perl5.10.0 -e 'use v5.8'
% perl5.10.0 -e 'use v5.9'
% perl5.10.0 -e 'use 5.8.8'
% perl5.10.0 -e 'use 5.009_005'
% perl5.10.0 -e 'use 5.9.5'
```

use feature

- **The `say` built-in function works just like the `print` function, but always implicitly appends a newline.**
- **That is, `say LIST` is in all regards nothing just syntactic sugar for**

```
{
    local $\ = "\n";
    print LIST;
}
```

- **Its formal prototype can be described as:**

```
say FILEHANDLE LIST
say LIST
say
```

- **Remember that being a "feature", it’s not available until and unless enabled, either explicitly or implicitly.**
- **This protects old programs with functions of their own named `say`.**

Using Closures for Persistent, Private Variables

- **Before the `state` feature, lexically private variables that retained their values — which a C or C++ programmer would always use the `static` storage class for — could only be implemented using closures, a somewhat esoteric approach to a beginner.**

```
{
    my $n = 42;
    sub next_number { return $n++ }
}
```

- **Ordering of code became important, as setting `$n` had to happen before the function could be usefully called.**
- **This requirement chafes, so you should label that bare-block as an INIT or BEGIN clause to guarantee it will run before your main program:**

```
INIT {
    my $n = 42;
    sub next_number { return $n++ }
}
```

How NOT to Do it

- Those employing the following undocumented, unsupported, and rather unintentional trick may be sadly disappointed someday when we break it: :-)

```
sub next_number {
    my $n if 0;
    $n = 42 unless defined $n;
    return $n++;
}
```

use feature

- A `state` variable is nothing more than a `my` variable that sticks around. That is, it's like a C or C++ variable of class `static` instead of one of class `auto`.
- Use `state` just as you'd use `my`, but remember it's `"sticky"`, so it doesn't get cleared when the block is done (er, rather, when refcount goes to 0).
- Initialization of simple scalars is skipped the next time through, unlike what would occur with a `my` variable.

```
sub next_number {
    state $n = 42;
    return ++x;
}
```

- However, that sort of assignment is undefined when applied to non-scalar variables. This is not a problem using the closure approach.

If Wishes Were Fishes

- **When it came out twenty-odd years ago, Perl had this `Wishlist`:**

```
% pwd
/usr/src/local/perls/perl1.0

% ls -l Wishlist
-rw-r--r--  1 tchrist  wsrc  84 Jan 30  1988 Wishlist

% cat -n Wishlist
     1  date support
     2  case statement
     3  ioctl() support
     4  random numbers
     5  directory reading via <>
```

- **Wishes #1, #3, and #4 have long been fullfilled. Wish #5 may remain unimplemented, but it's easy to override `readline` or overload the angle operator itself, `< >` .**
- **Yet for 20 years, Wish #2 languished. It'd've been no trouble making a C `switch/case` statement or an `sh case/esac`. But Larry wanted something much better for Perl, and now thanks to Damian Conway, we have one.**

Historic Work-Arounds for the Missing Switch

- **Though Perl never had a built-in switch statement, the more creative saw this not so much as hardship as opportunity.**
- **It's really easy to build a simple one — arguably perhaps too simple.**
- **There were, as in all things Perl, *many* ways of writing switch statements, and people used all of them.**
- **While you could always use a big chain of `elsifs`, this was unconsidered a bit uncreative.**

Emulating switch Using for

- **You might even say that the word `for` (or `foreach`) can sometimes be pronounced `switch`:**

```
SWITCH: for ($where) {
    /Kitchen/     && do { push @flags, ’-e’; last; };
    /Lounge/      && do { push @flags, ’-h’; last; };
    /Porch/       && do {                    last; };
    die qq#unknown value for variable \$where: "$where"#;
}
```

- **Like a series of `elsifs`, a switch-like construct should *always* have a default case, even if the default case "can’t happen". For those I often include a**

```
else { die "XXX: not reached!" }
```

Really Fancy Emulation of switch Using for

- **The *rename* script included on your CD has a particularly ornate form of this style:**

```
ARG: while (@ARGV && $ARGV[0] =~ s/^-(?=.)//) {
OPT:    for (shift @ARGV) {
            m/^$/       && do {                                 next ARG; };
            m/^-$/      && do {                                 last ARG; };

            s/^0//      && do { $nullpaths++;                   redo OPT; };
            s/^f//      && do { $force++;                       redo OPT; };
            s/^l//      && do { $reslinking++;                  redo OPT; };
            s/^I//      && do { $inspect++;                     redo OPT; };
            s/^i//      && do { $careful++;                     redo OPT; };
            s/^v//      && do { $verbose++;                     redo OPT; };
            s/^V//      && do { $verbose += 2;                  redo OPT; };  # like two -v's
            s/^m//      && do { $renaming++;                    redo OPT; };
            s/^n//      && do { $nonono++;                      redo OPT; };
            s/^N//      && do { $nonono += 2;                   redo OPT; };  # like two -n's
            s/^q//      && do { $quiet++;                       redo OPT; };

            s/^F(.*)//s && do { push @flist, $1 || shift @ARGV; redo OPT; };

            usage("Unknown option: $_");
        }
    }
```

- **See how it looks a bit a "real" switch with hanging case labels?:-)**
- **Hm... not buying it, eh? :-(**

Emulating Switch by Using do{} Creatively

- **Another interesting approach to rolling your own switch is to use the return value from a `do BLOCK` construct.**

```
$amode = do {
    if     ($f & O_RDONLY)   { "r" }       # XXX: isn't this 0?
    elsif  ($f & O_WRONLY)   { ($f & O_APPEND) ? "a" : "w"   }
    elsif  ($f & O_RDWR)     {
        if ($f & O_CREAT)    { "w+" }
        else                 { ($f & O_APPEND) ? "a+" : "r+" }
    }
};
```

Switch with for via && and ||

- Another way would be with a series of carefully connected logicals.
- Be careful that the RHS of `&&` is always true!

```
$dir = "http://www.wins.uva.nl/~mes/jargon";

for ($ENV{HTTP_USER_AGENT}) {
    $page  = /Mac/            && "m/Macintrash.html"
          || /Win(dows )?NT/  && "e/evilandrude.html"
          || /Win|MSIE|WebTV/ && "m/MicroslothWindows.html"
          || /Linux/          && "l/Linux.html"
          || /HP-UX/          && "h/HP-SUX.html"
          || /SunOS/          && "s/ScumOS.html"
          ||                     "a/AppendixB.html";
}
```

Emulating Switch Using Perl's Ternary Operator

- You *can* use the `?:` operator's right-associativity — if you're careful enough. A famous example of nigh-unto-impenetrable code:

```
$leapyear =
    $year % 4
        ? 0
        : $year % 100
          ? 1
          : $year % 400
            ? 0
            : 1;
```

- It's arguably more legible to align all the conditional THEN and ELSE parts vertically:

```
$leapyear =
    $year %   4 ? 0 :
    $year % 100 ? 1 :
    $year % 400 ? 0 : 1;
```

Emulating Switch with a Hash or Array of CODE Refs

- **You could always set up a hash of references to subroutines:**

```
%commands = (
    "happy" => \&joy,
    "sad"   => \&sullen,
    "done"  => sub { die "See ya!" },
    "mad"   => \&angry,
);
```

- **Then call the appropriate function:**

```
$commands{$string}->($arg);
```

- **You could do the same sort of thing using an array of CODE refs instead of a hash of them.**

Emulating Switch Using for and do{} Even More Creatively

- **Hey, sometimes ASCII-art aesthetics counts. :-)**

```
for ($^O) {
    *struct_flock =          do                         {
                  /bsd/  &&  \&bsd_flock
                         ||
              /linux/    &&    \&linux_flock
                         ||
            /sunos/      &&      \&sunos_flock
                         ||
         die "unknown operating system $^O, bailing out";
    };
}
```

use Switch through a Filter Module

- **That's *way* creative enough already, so by now you must all be thoroughly convinced that Perl needs its very own, built-in switchy construct.**
- **Beginning in the v5.8 release of Perl, an optional and super-versatile `Switch.pm` module was introduced. Don't worry, it gets *even better* for v5.10.**
- **This module was implemented by Damian Conway, and is one of the things Perl5 has borrowed from the Perl6 spec.**
- **Use it in these ways, plus far more:**

```
use Switch;
switch ($value) {
    case 17         { print "number 17"       }
    case "snipe"    { print "a snipe"         }
    case /[a-f]+/i  { print "pattern matched" }
    case [1..10,42] { print "in the list"     }
    case (@array)   { print "in the array"    }
    case (%hash)    { print "in the hash"     }
    else            { print "no case applies" }
}
```

The Switch Module, continued

- The various `cases` with the `switch` compare themselves, in order, with whatever you've supplied to the `switch`.
- Which flavor of comparison gets run can be a bit complicated to explain precisely, so for now, think of it as DWIM "smart-match".
- We'll examine this more closely a few slides from now when discussing v5.10's new built-in `switch feature`.
- Notice how on the previous page, you didn't have to use something like C's `break` to avoid fall-through behavior?
- That's because for safety's sake, the `switch` provided by the Switch module — by default — executes only the first applicable `case` it encounters, as though they were a bunch `elsifs`.

Falling Through to the Next case Element

- But what if you *wanted* C’s full-through semantics?
- One way is to `next` out of the `switch`:

```
use Switch;
%traits = (pride => 2, sloth => 3, hope => 14);
switch (%traits) {
    case "impatience"          { print "Hurry up!\\n";         next }
    case ["laziness","sloth"] { print "Maybe tomorrow!\\n";   next }
    case ["hubris","pride"]   { print "Mine′s best!\\n";      next }
    case ["greed","cupidity","avarice"] { print "More!\\n";   next }
}
```

- Another is tell the module at import time you’d always like fall-through behavior:

```
use Switch "fallthrough";
%traits = (pride => 2, sloth => 3, hope => 14);
switch (%traits) {
    case "impatience"          { print "Hurry up!\\n";       }
    case ["laziness","sloth"] { print "Maybe tomorrow!\\n"; }
    case ["hubris","pride"]   { print "Mine′s best!\\n";    }
    case ["greed","cupidity","avarice"] { print "More\!";   }
}
```

The Switch Module's Double-Underscore

- Consider a sequence of `ifs` like this:

```
if ($n % 2 == 0) { print "two "   }
if ($n % 3 == 0) { print "three " }
```

- One *could* use closures for the `cases` after `switching` on `$n`:

```
case sub{$_[0] % 2 == 0} { print "two "   ; next   }
case sub{$_[0] % 3 == 0} { print "three " ; next   }
```

- But fortunately, you never have be so ugly: instead, import `__` (that's two adjacent underscores):

```
use Switch "__";
```

- Once you've done that, the next two lines are equivalent:

```
             __ < 2
    sub { $_[0] < 2 }
```

But It Gets Even Better

- **What’s the best way to arrange for a bunch of tests against the same value? One way’d be:**

```
sub beverage {
    switch (shift) {
        case { $_[0] < 10 } { return "milk" }
        case { $_[0] < 20 } { return "coke" }
        case { $_[0] < 30 } { return "beer" }
        case { $_[0] < 40 } { return "wine" }
        case { $_[0] < 50 } { return "malt" }
        case { $_[0] < 60 } { return "Moet" }
        else                { return "milk" }
    }
}
```

- **Simpler to import `__` instead, with optional `fallthrough`:**

```
use Switch qw( __  fallthrough );
$n = 30;
print "Factors of $n include: ";
switch ($n) {
    case __ % 2 == 0 { print "two "   }
    case __ % 3 == 0 { print "three " }
    case __ % 5 == 0 { print "five "  }
    case __ % 7 == 0 { print "seven " }
}
```

The v5.10 switch feature

- **The `Switch.pm` module just described was first available in v5.8.**
- **However, it can suffer esoteric problems because it's implemented via source filters.**
- **In v5.10, we've a better alternative: a feature built right into the language proper — if you ask for it.**
- **This time instead of using `switch` and `case` the way the `Switch` module does, the keywords are instead `given` and `when`.**
- **Here's a simple example from the *perlsyn* manpage:**

```
use feature "switch";

given($_) {
    when (/^abc/) { $abc     = 1; }
    when (/^def/) { $def     = 1; }
    when (/^xyz/) { $xyz     = 1; }
    default       { $nothing = 1; }
}
```

C’s break and continue Finally Enter Perl

- Unlike `Switch.pm`’s control-flow strategy, with the v5.10 `given` feature, `break` exits the current `given` entirely, while `continue` falls through to the next `when` clause.
- These work if *and only if* you are within a `given` block enabled by the `switch` feature.
- *All* `given`s are of the break-out variety. For safety’s sake, there’s no way to make them all become of the full-through type, as you can with the module.
- If you ever *want* fall through to the next test, use `continue`. Again taken from the manpage:

```
given ($foo) {
    when (/x/) { say '$foo contains an x';       continue  }
    when (/y/) { say '$foo contains a  y';                 }
    default    { say '$foo contains neither an x nor a y'; }
}
```

Smart Matching

Consider how smart the `given` construct really has to be for the following code to work:

```
use feature qw(switch say);

given ($val) {

    when (1)          { say "number 1"                  }
    when ("a")        { say "string a"                  }
    when ([1..10,42]) { say "number in list"            }
    when (@array)     { say "number in list"            }
    when (/\w+/)      { say "pattern"                   }
    when (qr/\w+/)    { say "pattern"                   }
    when (%hash)      { say "entry in hash"             }
    when (\%hash)     { say "entry in hash"             }
    when (\&sub)      { say "arg to subroutine"         }

    default           { say "previous when not true"    }

}
```

Semantics of given and when

- `given(EXPR)` assigns EXPR to a lexical `$_` scoped to the block governed by that `given`.
- This makes it act somewhat like

```
 do { my $_ = EXPR; ... }
```

- Except that unlike the `do{}` , `given`'s block exits whenever any of its `when`'s smart-matches succeeds, or immediately by explicit `break`.
- It's because a `do{}` block isn't subject to customary loop-control constructs `next`, `last`, and `redo`, why you see people write strange things like:

```
do {{
    next if $x > 8;
    ...
}} while $y < 20;
```

given More Than Scalars

- **The impressive part of `given` and `when` is that they aren't restricted to scalars.**
- **Instead of blindly imposing scalar context on literal array and hash variables supplied arguments to `given`, a *reference* to the aggregate is placed in the lexical `$_`.**
- **You could just as easily give it an array:**

```
given (@array) {
    # ... insert when clauses
}
```

- **Likewise with a hash:**

```
given (%table) {
    # ... insert when clauses
}
```

But What Does It All Mean??

- **Perl v5.10 has a new operator, called the smart-match operator and spelt with a double-tilde.**
- **It's this operator, which you haven't seen yet, that's at work behind the scenes to provide the smartness.**
- **The older `Switch` module was an earlier attempt to fully implement the Perl6 construct, but was of only limited success for several reasons, including being a source filter.**
- **This `~~` operator, along with the `switch feature`, were both borrowed from the Perl6 spec, and are now part of Perl5 (providing you enable that feature).**
- **Like Perl's notorious `..` and `...` flip-flop operators, the new `~~` smart-match operator is much simpler to use than it is to provide precise rules like those only a language-lawyer would love.**
- **But here they are anyway.**

Rules for the `~~` Smart-Match Operator

- Like addition or multiplication (but not their inverses), smart-matching is a *commutative* operation. That guarantees the equivalence of `$X ~~ $Y` and `$Y ~~ $X` for all `$X` and `$Y`.
- The table on the next page and its annotations are borrowed from the *perlsyn* manpage.
  - ∗ Entries marked with `[+]` must be a code reference whose prototype, if present, must *not* be `""`. Subs that *do* have a `""` prototype are handled by the `Code()` entry lower down the chart.
  - ∗ In the entry marked `[*]`, each element of one array is compared with the corresponding element in the other array.
  - ∗ The entry marked `nummish[!]` must be a real number or a string that "looks" like a real number; ie, one that can be safely "nummified" (as in, `0+$string`) without provoking warning or error. These include strings like `"4."`, `"6.02e23"`, or `"-4.3"`, but not `"3 blind mice"`, `"0xFF"`, `"deadbeef"`, or `"01777"` — presuming you intend the last to be octal.

The Smart-Match Table

```
$a      $b       Type of Match Implied    Matching Code
======  =====    =====================    =============
(overloading trumps everything)

Code[+] Code[+]  referential equality     $a == $b
Any     Code[+]  scalar sub truth         $b->($a)

Hash    Hash     hash keys identical      [sort keys %$a]~~[sort keys %$b]
Hash    Array    hash slice existence     grep {exists $a->{$_}} @$b
Hash    Regex    hash key grep            grep /$b/, keys %$a
Hash    Any      hash entry existence     exists $a->{$b}

Array   Array    arrays are identical[*]
Array   Regex    array grep               grep /$b/, @$a
Array   Num      array contains number    grep $_ == $b, @$a
Array   Any      array contains string    grep $_ eq $b, @$a

Any     undef    undefined                !defined $a
Any     Regex    pattern match            $a =~ /$b/
Code()  Code()   results are equal        $a->() eq $b->()
Any     Code()   simple closure truth     $b->() # ignoring $a
Num    nummish[!] numeric equality        $a == $b
Any     Str      string equality          $a eq $b
Any     Num      numeric equality         $a == $b

Any     Any      string equality          $a eq $b
```

Built-in Variables That Changed in v5.10

- We've already mentioned that `$*` and `$#` are completely gone, and so provoke a severe warning if mentioned.
- The `$^V` variable (aka `$PERL_VERSION`) is now an object of class `version`, not a vstring as it'd been before.

```
% perl -E 'say $]'
5.010000

% perl -E 'say $^V'
v5.10.0

% perl -E 'say ref $^V'
version
```

Lexical $_

- The special variable `$_` is at last `my()`able, just like any normal user variable.
- You used to have to use `local $_` to give a temporary run-time value to what was still a global variable — and in all packages, too.
- The other punctuational variables are still not lexicalizable.
- As with other lexical variables, if `$_` winds up as the loop variable in a a `foreach()` loop, but there's already a lexical version of that variable, it's a new lexical private to that loop.
- Otherwise, it's merely a `local`ized global.
- This same property also extends to `$_` when used in `map` and `grep`.

use English fixed

- **One used to incur a severe penalty for loading the `English` module: by forcing the compiler to notice the $‘, $&, and $’ variables, (respectively aka `$PREMATCH`, `$MATCH`, and `$POSTMATCH`), *all* matches in the entire program had to load them up.**
- **This dissuaded most people from ever loading that module.**
- **A special directive to the module now loads all aliases *except* for those dangerous ones:**
  ```
  use English "-no_match_vars";
  ```
- **To compensate, a new regex modifier, `/p`, has been added. Patterns compiled or executed with `//p` load three new special variables: `${^PREMATCH}`, `${^MATCH}`, and `${^POSTMATCH}`.**
- **These otherwise act in all ways like the more costly versions.**
- **You don’t need to `use English` to access variables of the form `${^XXX}`.**

New Built-in Variables: %!

- The `%!` variable, originally from the `Errno.pm` module, is now built-in. Its English aliases are `%OS_ERROR` and `%ERRNO`.
- Find out what signals your system knows about:

```
$ perl -E 'say join ", " => sort keys %!'
E2BIG, EACCES, EADDRINUSE, EADDRNOTAVAIL, EAFNOSUPPORT, EAGAIN,
EALREADY, EAUTH, EBADF, EBADRPC, EBUSY, ECHILD, ECONNABORTED,
ECONNREFUSED, ECONNRESET, EDEADLK, EDESTADDRREQ, EDOM, EDQUOT,
EEXIST, EFAULT, EFBIG, EFTYPE, EHOSTDOWN, EHOSTUNREACH, EINPROGRESS,
EINTR, EINVAL, EIO, EIPSEC, EISCONN, EISDIR, ELAST, ELOOP, EMFILE,
EMLINK, EMSGSIZE, ENAMETOOLONG, ENEEDAUTH, ENETDOWN, ENETRESET,
ENETUNREACH, ENFILE, ENOATTR, ENOBUFS, ENODEV, ENOENT, ENOEXEC,
ENOLCK, ENOMEM, ENOPROTOOPT, ENOSPC, ENOSYS, ENOTBLK, ENOTCONN,
ENOTDIR, ENOTEMPTY, ENOTSOCK, ENOTTY, ENXIO, EOPNOTSUPP, EPERM,
EPFNOSUPPORT, EPIPE, EPROCLIM, EPROCUNAVAIL, EPROGMISMATCH,
EPROGUNAVAIL, EPROTONOSUPPORT, EPROTOTYPE, ERANGE, EREMOTE, EROFS,
ERPCMISMATCH, ESHUTDOWN, ESOCKTNOSUPPORT, ESPIPE, ESRCH, ESTALE,
ETIMEDOUT, ETOOMANYREFS, ETXTBSY, EUSERS, EWOULDBLOCK, EXDEV
```

- Or to inspect `$!` symbolically to avoid locale issues.

```
unless (open(FH, "/fangorn/spouse")) {
    if ($!{ENOENT}) {
        warn "Get a wife!\n";
    } else {
        warn "This path is barred: $!";
    }
}
```

New Internal Variables

- `${^RE_DEBUG_FLAGS}` is related to the `use re "debug"` pragma.
- `${^CHILD_ERROR_NATIVE}` Works just like `$?` (`$CHILD_ERROR`) except it always returns the current system native status, not a canonicalized value.
- `${^RE_TRIE_MAXBUF}` is related to a clever, new optimization in regex searches involving alternations of particular strings, as in `/cat|dog|cow/`
- `${^WIN32_SLOPPY_STAT}` governs how `stat()` does its job under Win32 systems.

New Pattern-related Variables

- Besides the numbered variables and the unmentionables, Perl already knew about
  - `@+        @LAST_MATCH_END`
  - `@-        @LAST_MATCH_END`
  - `$+        $LAST_PAREN_MATCH`
  - `$^R       $LAST_REGEXP_CODE_RESULT`
- Now it also knows about
  - `$^N       LAST_SUBMATCH_RESULT`
  - `%+` (a hash of strings for named capture buffers)
  - `%-` (a hash of arrays of strings for for named capture buffers)
- The last two, named capture buffers, are covered in the section on pattern matching.

New Operators in v5.10

- The fancy ~~ smart-match operator was already described under the new switch feature using `given` and `when`.
- The new `//` operator is for people who keep using `$a || C<$b>` and want it to check whether `$a` is defined rather than whether it's true. Thus, it's syntactic sugar for:

```
$x = defined($a) ? $a : $b
$x = $a // $b       # same thing
```

- Similarly, these now do the same thing:

```
$c   = $d unless defined($c);
$c //= $d;
```

- In case you're wondering, the unary backlash operator for taking references, `\`, can stack.

```
$a  = "stuff";
$r  =  \$a;
$rr = \\$a;
```

You Can’t Win For Trying

- **Like the original `||` and `&&`, `//` is of no use for arrays or hashes.**
- **That’s because their first operand is in boolean (scalar) context.**
  - ∗ **You can use them for scalars only:**

```
$a = $b || $c;   # this is ok
$a = $b // $c;   # as is this
```

  - ∗ **But not on aggregates — arrays or hashes. Also, it doesn’t make sense to ask whether an aggregate is defined.**

```
@a = @b || @c;            # but this is NOT!
@a = scalar(@b) || @c;  # for it really means this
```

- **So you’re back to the standard idiom again anyway:**

```
@a = @b ? @b : @c;
```

Operator Precedence

- **This is the operator precedence and associativity table as of 5.10:**

1	left	*terms and list operators (leftward)*
2	left	`->`
3	nonassoc	`++ --`
4	right	`**`
5	right	`! ~ \` *unary* `+` *and unary* `-`
6	left	`=~ !~`
7	left	`* / % x`
8	left	`+ - .`
9	left	`<< >>`
10	nonassoc	*named unary operators*
11	nonassoc	`< > <= >= lt gt le ge`
12	nonassoc	`== != <=> eq ne cmp ~~`
13	left	`&`
14	left	`| ^`
15	left	`&&`
16	left	`|| //`
17	nonassoc	`..  ...`
18	right	`?:`
19	right	`= += .= x= %= &= |= ^= -= *= **= ||= //=` *etc*
20	left	`, =>`
21	nonassoc	*list operators (rightward)*
22	right	`not`
23	left	`and`
24	left	`or xor`

Who Needs Precedence, Anyway?

- **Anybody ever seen code like this and not questioned it?**

```
    if ($x & $y == 0) { ... }
```

- **Well, you should have, and it now elicits an optional warning:**

```
    Possible precedence problem on bitwise & operator
```

- **The complete diagnostic is:**

```
(W precedence) Your program uses a bitwise logical operator in
conjunction with a numeric comparison operator, like this :

    if ($x & $y == 0) { ... }

This expression is actually equivalent to $x & ($y == 0), due
to the higher precedence of ==. This is probably not what you want.
(If you really meant to write this, disable the warning, or, better,
put the parentheses explicitly and write $x & ($y == 0)).
```

See Rule 2

- Got that? We have 24 precedence levels. Isn't that just swell?
- I can tell you how to easily keep them all straight, for only two rules matter:
  - Rule 1: Multiplicative operators bind more tightly than additive operators.
  - Rule 2: Otherwise use parentheses, don't be a danged fool! :-)
- There, no more precedence troubles. Try to think about the people who'll come after you — before they do so.

Stackable Filetest Operators

- The so-called "filetest operators" borrowed from the shell, along with `stat` and `lstat`, have always cached their stat-buffer in a filehandle called `_`.
- That way to determine whether a file is plain, writable, and executable, instead of saying

```
if ( -f $file && -w $file && -x $file) { ... }
```

- You could save two stat calls this way:

```
if ( -f $file && -w _ && -x _ ) { ... }
```

- Now there's a bit of syntactic sugar so you can write that as:

```
if ( -f -w -x $file ) {  ... }
```

New and Improved Pragmas

- **A pragma is a module whose name is in all lower case. These are typically lexically scoped, and are intended to tell the compiler to do something differently than it's doing.**
- **After you install Perl 5.10, the following pragmatic modules will have been installed.**

```
        KEY: * new since 5.6; + changed; ! is cool,important
  attributes  +constant        +lib        +subs
 *attrs       +diagnostics!    +locale     *threads!!
 *autouse     +encoding!       *mro!       *...::shared!!
 +base        *...::warnings    open!      +utf8
 *bigint!     *feature!        *ops         utils::pl2pm
 *bignum!     -fields           overload!  +vars
 *bigrat!      filetest        +re!        *version
  blib        *if               sigtrap!    vmsish
 +bytes        integer         *sort!      +warnings!
 +charname!   +less            +strict!     ...::register
```

- **Some are new. Some are indispensable. Some are small. And some open up huge, huge areas of further study.**

What Do Pragmata Do?

- **When you load up a pragma, it can potentially change how the compiler handles the compilation.**
- **It's as though they propagate their effects into their caller's lexical scope — and they can pretty much do anything they'd like.**
- **It is now possible to write one's own pragmata. The implementation of the special internal hints variables, `$^H` and `%^H` has been made available to the user so you can write your own pragmata. See the *perlpragma* manpage.**
- **As of `v5.10.0`, these propagate into `eval STRING` operations so things like overloaded constants become visible where they previously were not.**

Demo of Change of Propagation

- **First try spilling our ints into floats:**

```
% perl5.8.8 -le 'print 2**100'
1.26765060022823e+30
```

- **Now make that not happen anymore using the `bignum` pragma:**

```
% perl5.8.8 -Mbignum -le 'print 2**100'
1267650600228229401496703205376
```

- **Check for whether the `eval`ed strings keep their bignums:**

```
 % perl5.8.8 -Mbignum -le 'eval( q{print 2**100} ) || die'
Constant(undef): $^H{integer} is not defined at (eval 6) line 1, at end of line
Constant(undef): $^H{integer} is not defined at (eval 6) line 1, at end of line
  ...propagated at -e line 1.
 Exit 255.
```

- **Ah but now, they do.**

```
% perl5.10.0 -Mbignum -E 'eval( q{say 2**100} ) || die'
1267650600228229401496703205376
```

More Big Math Fun

- **Compare default precisions — and notice the rounding.**

```
% perl          -E 'say sqrt(33)'
 5.74456264653803
% perl -Mbignum -E 'say sqrt(33)'
 5.744562646538028659850611468218929318220
```

- **Even simple stuff may need complex help**

```
% perl          -E 'say sqrt(-33)'
 Can't take sqrt of -33 at -e line 1.
% perl -Mbignum -E 'say sqrt(-33)'
 NaN

% perl -MMath::Complex -E 'say sqrt(-33)'
 5.74456264653803i
% perl -MMath::Complex -E 'say sqrt(-33)*i'
 -5.74456264653803
```

Numbers as Objects

- **Here, this should impress the python people.**

```
% perl -Mbignum -E 'say 123->is_odd()'
 1
% perl -Mbignum -E 'say "Is 1 less than a gross odd: ",
   (12**2-1)->is_odd()'
 Is 1 less than a gross odd: 1
```

- **Rational numbers can work automatically:**

```
% perl -Mbigrat -E '$r = 3/7 + 5/8 + 8/3;
 say "inverse of $r is ", 1/$r;'
   inverse of 625/168 is 168/625
```

Still More Big Fun

- The 8th Mersenne Prime (prime numbers of the form 2 ** p-1) was found by Euler. You probably think it’s a pretty regular number, but it isn’t. Plus once you have a Mersenne Prime, you also have a new, and probably very large, perfect number, too.

```
perl -Mbignum -E '$p = 31; $M = (2**$p)-1; $P = $M * (2**($p-1));
    say "Euler found M8 to be $M, so we know that $P is perfect" '
Euler found M8 to be 2147483647, so we know that 2305843008139952128 is perfect
```

- Powers found the 11th Mersenne Prime. It’s way bigger than Euler’s.

```
perl -Mbignum -E '$p =107; $M = (2**$p)-1; $P = $M * (2**($p-1));
    say "Powers found M11 to be $M,so we know that $P is perfect" '
Powers found M11 to be 162259276829213363391578010288127,
  so we know that 13164036458569648337239753460458722910223472318386943117783728128 is perfect
```

Upgrading Standard and Dual-Life Modules

- **Modules in the standard distribution can also be located on CPAN.**
- **There are many good reasons for this, including the testing, rating, commenting, and searching facilities found there.**
- **Most of all, it allows for the inclusion of CPAN modules in the standard distribution without requiring the distro manager to answer every little module bug.**
- **These modules are called *dual-life* modules, because the original owners still maintain them, not the p5p team.**

Auto-upgrading Modules

- The `cpan` shell has facilities for asking what needs upgrading. Even though I just reinstalled everything, I just ran it and found all these:

```
 % cpan

 CPAN[1]> r
Package namespace         installed    latest  in CPAN file
Archive::Extract               0.24      0.26  KANE/Archive-Extract-0.26.tar.gz
DateTime                     0.4301    0.4302  DROLSKY/DateTime-0.4302.tar.gz
ExtUtils::ParseXS           2.18_02      2.19  KWILLIAMS/ExtUtils-ParseXS-2.19.tar.gz
Inline::Files                  0.60      0.62  DCONWAY/Inline-Files-0.62.tar.gz
Math::BigInt::FastCalc         0.16      0.19  TELS/math/Math-BigInt-FastCalc-0.19.tar.gz
Module::CoreList               2.13      2.15  RGARCIA/Module-CoreList-2.15.tar.gz
Module::Load::Conditional      0.22      0.26  KANE/Module-Load-Conditional-0.26.tar.gz
Module::Pluggable               3.6       3.8  SIMONW/Module-Pluggable-3.8.tar.gz
Object::Accessor               0.32      0.34  KANE/Object-Accessor-0.34.tar.gz
PerlIO::via::Rotate            0.04      0.06  ELIZABETH/PerlIO-via-Rotate-0.06.tar.gz
Time::Piece                    1.12      1.13  MSERGEANT/Time-Piece-1.13.tar.gz
```

- You can then use the `upgrade` command to pull in and install those in need of upgrading.

- Alternately, use the `autobundle` feature previously described.

New Modules

- Quite a few of these migrated into the core from CPAN.
- Between 5.6.0 and 5.8.8, the *lib* directory in Perl's build environment went from a 142 *.pm* modules to 362 of them. That's 220 more modules!
- Between 5.8.8 and 5.10.0, that same number rose from 362 to 552. That's 190 more modules in 5.10.0 since 5.8.8, but 410 more counting since 5.6.0!
- Those numbers are misleading: most are inner parts of upper-level modules, and many deal with the installation infrastructure itself or are code development tools.
- For example:

```
Module::Load::Conditional::t::to_load::MustBe::Loaded
CPANPLUS::Shell::Default::Plugins::CustomSource
Attribute::Handlers::demo::Descriptions
Pod::Simple::t::testlib2::hinkhonk::Vliff
ExtUtils::CBuilder::Platform::Windows
Module::Build::t::bundled::Tie::CPHash
Encode::MIME::Header::ISO_2022_JP
ExtUtils::CBuilder::Platform::VMS
IO::Uncompress::Adapter::Identity
```

Modules New to Perl v10.0.0

- The *perldelta* manpage lists 28 only of these, so you can relax.

```
Archive::Extract                    Math::BigInt::Calc
Archive::Tar                        Math::BigInt::FastCalc
Compress::Zlib                      Module::Build
CPANPLUS                            Module::CoreList
Digest::SHA                         Module::Load
encoding::warnings                  Module::Load::Conditional
ExtUtils::CBuilder                  Module::Loaded
ExtUtils::MakeMaker                 Module::Pluggable
ExtUtils::ParseXS                   Object::Accessor
File::Fetch                         Package::Constants
Hash::Util::FieldHash               Params::Check
IO::Zlib                            Pod::Escapes
IPC::Cmd                            Pod::Simple
Locale::Maketext::Lexicon           Term::UI
Locale::Maketext::Simple            Time::Piece
Log::Message                        Win32API::File
```

Modules New to Perl 5.6.1+ before v5.10

- On the other hand, the list of modules added to the core since Perl v5.6 not mentioned on the previous page, is (*ahem*) rather longer:

```
Attribute::Handlers          I18N::Langinfo
B::Concise                   JPL
Class::ISA                   List::Util
Data::Dumper                 Locale::Constants
Devel::DProf                 Locale::Country
Devel::PPPort                Locale::Currency
Devel::Peek                  Locale::Language
Digest                       Locale::Maketext
Digest::MD5                  MIME::Base64
Encode                       MIME::QuotedPrint
ExtUtils::Constant           Memoize
File::Temp                   NEXT
Filter::Simple               PerlIO::Scalar
Filter::Util::Call           PerlIO::Via
Hash::Util                   PerlIO::via::QuotedPrint
I18N::LangTags               Pod::Checker
```

More Semi-New Modules

- **And that's not all. We also have these:**

```
Pod::Find                    Text::Balanced
Pod::InputObjects            Tie::File
Pod::Man                     Tie::Memoize
Pod::ParseLink               Tie::RefHash::Nestable
Pod::ParseUtils              Time::HiRes
Pod::Parser                  Time::Local
Pod::Select                  Time::Seconds
Pod::Text                    Unicode::Collate
Pod::Text::Overstrike        Unicode::Normalize
Pod::Usage                   Unicode::UCD
Scalar::Util                 UnicodeCD
Storable                     Win32
Switch                       XS::APItest
Term::ANSIColor              XS::Typemap
Test::More                   XSLoader
Test::Simple
```

Semi-New Pragmata

- **In the same category, the following pragmatic modules appeared after v5.6 and before v5.10:**

```
assertions::compat        if
attributes                open
bigint                    sort
bignum                    threads
bigrat                    threads::shared
constant
```

  * ***use attributes*** **to get/set subroutine or variable attributes**
  * ***use constant*** **to declare constants at compile-time**
  * ***use big*`foo`** **to make your numbers `Math::Big`*Foo* objects**
  * ***use if*** **to conditionally `use` a Module**
  * ***use open*** **to set default PerlIO layers**
  * ***use sort*** **to control the built-in `sort`'s function's behavior**
  * ***use threads*** **to be discussed in that unit**

The Utils Modules

- **Take a look at what's in the Util modules for each of the three types provided.**
- **`Scalar::Util` gives access to these functions:**

```
blessed    readonly     weaken         tainted
dualvar    refaddr      isweak         looks_like_number
           reftype      isvstring      set_prototype
```

- **`List::Util` gives access to these functions:**

```
first       max         min         shuffle
reduce      maxstr      minstr      sum
```

- **`Hash::Util` gives access to these functions to support, which are mostly for implementing secure/restricted hashes:**

```
hash_seed      all_keys      lock_keys         unlock_keys
hash_locked    legal_keys    lock_value        unlock_value
               hidden_keys   lock_hash         unlock_hash
                             lock_keys_plus
```

The first Construct

- The `first` and `reduce` functions provided by `List::Util` are particularly cool.
- `first {}` works a lot like `grep {}`. It takes a code block to apply to each element in the list, temporarily setting `$_` to that element.
- But `first` short-circuits! So instead of returning *all* the elements that test true, it returns only the *first* such:

```
use List::Util qw(first);

$bingo = first { $_ > $maxvalue } @list;
```

The reduce Construct

- `reduce {}` is fancier; most `List::Util` functions are defined using it.
- Instead of using `$_` for each call like `grep` and `map`, it uses `$a` and `$b` like `sort`.
- The first call to `reduce`'s code block sets `$a` and `$b` to the first two list values.
- After that, `$a` is the result of the previous call and `$b` is the next list element.

```
use List::Util qw(reduce);

$littlest = reduce { $a < $b ? $a : $b } 1..10;
$total    = reduce { $a + $b } 1 .. 10;
$pasted   = reduce { $a . $b } @strings;
```

Factorial Using reduce

- Rather than writing the factorial function recursively, one could just use `reduce`:

```
use List::Util qw(reduce);
sub factorial($) {
    my $n = shift;
    return 1 if $n == 0 || $n == 1;
    return reduce { $a * $b }  1 .. $n;
}
```

- Where this blows up depends on whether you have 64-bit integers support compiled into your build of Perl.
- Consider the following printf:

```
printf "%d! is d=%d u=%lu g=%g
       f=%.0f s=%s\n", $num, ($f) x 7;
```

Platform Math Issues, 1

- **Under OpenBSD.i386-openbsd w/o large integer support:**

```
12! is d=479001600 u=479001600 g=4.79002e+08
  f=479001600 s=479001600
13! is d=-1 u=4294967295 g=6.22702e+09
  f=6227020800 s=6227020800
14! is d=-1 u=4294967295 g=8.71783e+10
  f=87178291200 s=87178291200
```

- **Under OpenBSD.i386-openbsd-thread-multi-64int**

```
12! is d=479001600 u=479001600 g=4.79002e+08
        f=479001600 s=479001600
13! is d=6227020800 u=1932053504 g=6.22702e+09
        f=6227020800 s=6227020800
14! is d=87178291200 u=1278945280 g=8.71783e+10
        f=87178291200 s=87178291200
15! is d=1307674368000 u=2004310016 g=1.30767e+12
        f=1307674368000 s=1307674368000
```

Platform Math Issues, 2

- **But even this eventually fails:**

```
19! is d=121645100408832000 u=109641728 g=1.21645e+17
        f=121645100408832000 s=121645100408832000
20! is d=2432902008176640000 u=2192834560 g=2.4329e+18
        f=2432902008176640000 s=2432902008176640000
21! is d=-1 u=4294967295 g=5.10909e+19
        f=51090942171709440000 s=5.10909421717094e+19
```

- **Unless you use bigint:**

```
19! is d=121645100408832000 u=109641728 g=1.21645e+17
        f=121645100408832000 s=121645100408832000
20! is d=2432902008176640000 u=2192834560 g=2.4329e+18
        f=2432902008176640000 s=2432902008176640000
21! is d=-1 u=4294967295 g=5.10909e+19
        f=51090942171709440000 s=51090942171709440000
```

- **Notice how the `%s` string rep changed.**

Platform Math Issues, 3

- **Even as inaccurate floats, these eventually run out of bits.**
- **For the same reason, floats offer only limited precision; notice the last digits:**

```
170! is d=-1 u=4294967295 g=7.25742e+306
        f=72574[275 digits]1922516097717752406978O6848
        s=7.257415615308e+306

171! is d=-1 u=4294967295 g=Inf
        f=Inf s=Inf
```

- **Whereas under bigint:**

```
170! is d=-1 u=4294967295 g=7.25742e+306
        f=72574[275 digits]6141721875932183456791920064
        s=72574[275 digits]0000000000000000000000000000

171! is d=-1 u=4294967295 g=Inf
        f=Inf
        s=12410[278 digits]0000000000000000000000000000
```

The Memoize Module

- **Functions returning the same value given the same arguments can sometimes be sped up with Memoize. Recursive Fibonacci is a good demo, as it bifurcates.**

```
# must predeclare any proto on recursive funcs
sub fibo($);
sub fibo($) {
    my $n = shift;
    return $n if $n == 0 || $n == 1;
    return fibo($n - 1) + fibo($n - 2);
}
```

- **Just say `memoize("fibo")` and compare timings:**

```
% time perl fibo 25
fib 25 is d=75025 g=75025 f=75025 s=75025
10.507u 0.023s 0:10.70 98.3%    0+0k 26+6io 0pf+0w

% time perl mfibo 25
fib 25 is d=75025 g=75025 f=75025 s=75025
0.195u 0.039s 0:00.26 84.6%     0+0k 5+7io 0pf+0w
```

Conditional Module Loading

- **In the next slides of demos, instead of always loading bigint:**

```
use bigint;
```

- **I set an envariable to conditionally load my bigints this way:**

```
use if $ENV{tchrist_BIG} => "bigint";
sub am_big() {
    return defined &bigint::in_effect
               &&  &bigint::in_effect;
}
warn "bigint IN EFFECT"      if  am_big;
warn "bigint NOT in effect" if !am_big;
```

Platform Math Issues, 4

- **You can still run-over:**

```
% ( setenv tchrist_BIG 0 ; time perl mfibo 1300)
fib 1300 is d=-1 g=2.15997e+271
  f=215996[250 digits]8175139269509120
  s=2.15996802831617e+271
0.093u 0.000s 0:00.13 69.2%     0+0k 0+7io 0pf+0w

%  ( setenv tchrist_BIG 0 ; time perl mfibo 2000)
fib 2000 is d=-1 g=Inf
  f=Inf s=Inf
 0.117u 0.007s 0:00.15 73.3%     0+0k 0+7io 0pf+0w
```

- **Here with bigints loaded:**

```
% ( setenv tchrist_BIG 1 ; time perl mfibo 1300 )
fib 1300 is d=-1 g=2.15997e+271
  f=21599[250 digits]71794938102546432
  s=21599[250 digits]69574650368333475
0.593u 0.007s 0:00.65 90.7%     0+0k 0+2io 0pf+0w

% ( setenv tchrist_BIG 1 ; time perl mfibo 2000 )
  fib 2000 is d=-1 g=Inf
  f=Inf s=422469[400 digits]082516817125
  0.859u 0.031s 0:00.94 93.6%     0+0k 0+2io 0pf+0w
```

Is It a Number?

- **A perennial FAQ was how to find out if a string seemed nummish; answers usually involved elaborate patterns like:**

```
if (/\D/)            { print "has nondigits\n" }
if (/^\d+$/)         { print "is a whole number\n" }
if (/^-?\d+$/)       { print "is an integer\n" }
if (/^[+-]?\d+$/)    { print "is a +/- integer\n" }
if (/^-?\d+\.?\d*$/) { print "is a real number\n" }
if (/^-?(?:\d+(?:\.\d*)?|\.\d+)$/) { print "is a decimal number\n" }
if (/^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/)
   { print "a C float\n" }
```

- **You may now access the function Perl itself uses internally using `Scalar::Util`'s `looks_like_a_number`.**

- **Be careful with `Scalar::Util`'s `looks_like_a_number` function. It doesn't like bigint objects:**

```
sub check_whole_int($) {
    my $N = shift;
    return if am_big;
    croak "$N doesn't look like a number to me!"
            unless looks_like_number($N);
    croak "$N must be positive"  if $N < 0;
    croak "$N must be a integer" if $N != int($N);
}
```

New for Objects: NEXT

- Perl v5.10 offers a couple of new object facilities, including the `NEXT` pseudo-class.
- Implemented as a module, you must `use NEXT` to get it.
- Works somewhat like `SUPER`, but `SUPER` looks only at the current class's ancestry. `NEXT` restarts the original method search as though it hadn't been caught.
- This allows backtracking down and then up a right-hand parent class that `SUPER` would have missed.

```
$self->meth();                  # Call wherever first meth is found
$self->Where::meth();           # Start looking in package "Where"
$self->SUPER::meth();           # Call overridden version
$self->NEXT::meth();            # Restart dispatch from current point
```

Being Demanding

- If `SUPER` can't find a method, it will raise and exception. This forces people to write strange code like:

  ```
  if (@ISA && $self->SUPER::can("some_meth")) {
      $self->SUPER::some_meth();
  }
  ```

- `NEXT`, in contrast, is normally silent. To change this, simple invoke the redispatch as:

  ```
     $self->NEXT::ACTUAL::method();
  ```

  rather than:

  ```
     $self->NEXT::method();
  ```

- The `ACTUAL` tells `NEXT` that there must actually be a next method to call, or it should throw an exception.

Solving the Diamond Inheritance Problem

- **In a situation where a class has more than one ancestor that themselves share a common ancestor, `NEXT` would call that same method more than once.**
- **When this is undesirable, just use:**

```
$self->NEXT::DISTINCT::method();
```

- **For unique-invocation *and* exception-on-failure semantics, use one of**

```
$self->NEXT::ACTUAL::DISTINCT::method();
$self->NEXT::DISTINCT::ACTUAL::method();
```

EVERY

- Another pseudo-class provided by `NEXT` is `EVERY.`
- Use it this way

```
 $obj->EVERY::some_method();
```

- Using EVERY is better than rolling your own method dispatch all the time.

```
sub some_method {
    my $self = shift;
    my %seen;
    print "some_method($self): checking all ancestors\\n";
    for my $parent (our @ISA) {
        if (my $code = $parent->can("some_method")) {
            $self->$code(@_) unless $seen{$code}++;
        }
    }
```

- Normally EVERY is left-to-right, *breadth*-first, and so executes those methods closer before those further away in the inheritance chain.
- To get the opposite behavior, use

```
$obj->EVERY::LAST::some_method();
```

Changing Method Resolution Order

- **New to Perl v5.10, the `use mro` pragma lets you change Perl's default order of resolving methods for the current class:**

  ```
  use mro 'DFS';
  ```

  ```
  use mro 'C3';
  ```

  - **DFS is Perl's traditional "depth-first-search".**
  - **C3 is different in complex inheritance situations.**

- **Perl6's default MRO will apparently be C3, not DFS.**

- **The original C3 paper was given OOPSLA '95 and can be found at *http://192.220.96.201/dylan/linearization-oopsla96.html*.**

Unicode

- Original design goals for adding Unicode support to Perl were:
  - Goal #1: Old byte-oriented programs should not spontaneously break on the old byte-oriented data they used to work on.
  - Goal #2: Old byte-oriented programs should magically start working on the new character-oriented data when appropriate.
  - Goal #3: Programs should run just as fast in the new character-oriented mode as in the old byte-oriented mode.
  - Goal #4: Perl should remain one language, rather than forking into a byte-oriented Perl and a character-oriented Perl.
- While these goals have been mostly met, there remains the occasional snag here and there, and it's been an incremental process.

Perl Unicode Support

- Remembering the Perl maxim that *easy things should be easy, and hard things possible*, here's the history of Perl's Unicode support:
  - ∗ Perl 5.6.1 - limited support for Unicode
  - ∗ Perl 5.8.1 - much more comprehensive support
  - ∗ Perl 5.10.0 - more refinement, UCD 5.0; some changes
  - ∗ Perl 5.10.1 - upgrade to UCD 5.1; a few changes
- String and regex functions/operators now operate on characters instead of octets (8-bit bytes).
- Transparent conversions between external encoding formats through per-filehandle I/O layers.
- Programmers who interchange bytes and characters are guilty of the same class of sin as C programmers who blithely interchange integers and pointers.
- You get in the most trouble when you've raw, binary data that you don't want treated as character data, but are mixing it with the same.

What Unicode Is

- **Computers don't store characters. They store bits, which we interpret as numbers. Sometimes we further interpret these numbers to mean a character.**
- **Many different character sets exist, all sharing the same "namespace", if you would: all represented by integer 0 through either 127 (ASCII) or 255 (ISO-8859-*?*), and sometimes even 65535 for CJK work.**
- **Consider what character is represented by 196 decimal (0xC4)**
  - **In ISO 8859-1 it's Ä, a Latin capital A with diaeresis**
  - **In ISO 8859-7 it's Δ, a Greek capital Delta**
- **Makes it hard to mix different character repertoires within the same document.**
- **Unicode attempts to unify all character sets in the world, including many sets of symbols and even fictional character sets.**
- **Under Unicode, each character is assigned a *distinct* integer, called its *code point*.**

Specifying Unicode Characters

- **If your text editor allows direct entry of Unicode, tell Perl via the pragma:**

```
use utf8;
$name = "Σωκρατης";
```

- **`\xNN` string escape notation has been extended to use braces: "`Delta is \x{394}`" for "Delta is Δ".**

- **The v-string notation can be used for multi-character strings**

```
  $word = "fac\x{327}ade";              # "façade"
  $word = v102.97.99.807.97.100.101;  # exactly equivalent
```

- **Use printf's v modifier to invert that:**

```
% perl -e 'printf "%vd\n", "fac\x{327}ade"'
102.97.99.807.97.100.101
% perl -e 'printf "%vx\n", "fac\x{327}ade"'
66.61.63.327.61.64.65
% perl -e 'printf "%#vx\n", "fac\x{327}ade"'
0x66.0x61.0x63.0x327.0x61.0x64.0x65
```

Using chr on Negative Values

- **Perl has not been very consistent on what to do about negative character values:**

```
% perl5.6.1 -e 'printf "ord of chr -20 is U+%04X\n", ord chr(-20)'
 Malformed UTF-8 character (byte 0xfe) in ord at -e line 1.
 ord of chr -20 is U+0000
```

```
% perl5.8.1 -e 'printf "ord of chr -20 is U+%04X\n", ord chr(-20)'
 ord of chr -20 is U+FFFFFFEC
```

```
% perl5.10.0 -e 'printf "ord of chr -20 is U+%04X\n", ord chr(-20)'
 ord of chr -20 is U+FFFD
```

- **Now any negative argument to `chr` produces `"\x{FFFD}"` the Unicode Replacement Character, usually used to represent an error in text conversions.**

- **If `use "bytes"` should be in effect, the low-order eight bits alone are used and result is unsigned:**

```
% perl5.10.0 -Mbytes -e'printf "ord of chr -20 is %d\n",ord chr(-20)'
 ord of chr -20 is 236
```

tr/// Changes

- If you used to use `tr///U` and `tr///C` to force Unicode or byte semantics, you'll have to stop that, as they're gone now.
- Instead, normally just using `pack` or `unpack` with U0 or C0 suffices.
- In rarer cases, you may need to resort to the Encode module's functions, some of which are described later.

Pack and Unpack for Unicode

- **The U format character handles Unicode data. It’s also a way to force the internal UTF-8 bit on.**

```
say pack("C*", 102,97,0xE7,97,100,101);
façade
say uc pack("C*", 102,97,0xE7,97,100,101);
FAçADE
```

```
say pack("U*", 102,97,0xE7,97,100,101);
façade
say uc pack("U*", 102,97,0xE7,97,100,101);
FAÇADE
```

- **Although there’s a much better way, but you could force it this way:**

```
say uc "fa\x{E7}ade;
FAçADE
```

```
say uc pack("U*", unpack("C*", "fa\x{E7}ade"));
FAÇADE
```

Changes in Pack and Unpack for Unicode

- **As of Perl v5.10, the pack and unpack functions have changed a bit in how they handle Unicode.**
- **In v5.8.X, "C0" and "U0" behaved differently in pack than they did unpack. This has been fixed so they work the same way for both functions.**
- **The new format character, "W", is intended to replace the "c" or "C" for Unicode code points.**

```
 say unpack("c", pack("c", 500))′;
-12
 say unpack("C", pack("C", 500))′
244
 say unpack("W", pack("W", 500))′
500
```

- **Many, *many* new features come with pack and unpack now, including directives for template grouping, endianness, alignment, and much more. See the pack entry in the perlfunc manpage, or the perlpacktut manpage.**

Encodings

- **Many different ways of encoding Unicode characters some variable-width, others fixed-width encodings**

```
UTF-8       UTF-7
UTF-16      UTF-16BE        UTF-16LE
UTF-32      UTF-32LE        UTF-32BE
```

- **Perl uses UTF-8 internally (or close to it), but can easily handle Unicode files with alternate encodings**
- **Specify the encoding either with binmode or open**

```
binmode(FH, ":encoding(UTF-16BE)")
  || die "can't binmode to utf-16be: $!";
open(FH, "< :encoding(UTF-32)", $pathname)
  || die "can't open $pathname: $!";
```

- **That uses the encoding module implicitly; for simple utf-8, specify `":utf8"` instead of an encoding**

Upgrading Legacy Files

- Suppose you have files in legacy encodings such as:
  - ∗ Shift JIS (aka MS Kanji)
  - ∗ cp1252 (aka MS-ASCII or MS-Latin1).
- You probably can’t convert these into Latin1 because they have characters not in that character set
- It’s easy to convert them into Unicode

```
binmode(STDIN, ":encoding(cp1252)")
  || die "can't binmode to cp1252 encoding: $!";
binmode(STDOUT, ":utf8")
  || die "can't binmode to utf8: $!";

print while <STDIN>;
```

The -C Command-Line Switch

- **Emitting a character whose code point exceeds 255 to a file not marked with an encoding triggers a warning**

```
% perl -le 'print "nai\x{308}ve fac\x{327}ade"'
  Wide character in print at -e line 1.
  naïve façade
```

- **Use -C command-line switch to binmode a program's stdin (-CI), stdout (-CO), or stderr (-CE) to ":utf8" those streams:**

```
% perl -COE -E 'print "nai\x{308}ve fac\x{327}ade"'
  naïve façade
```

- **Alternately, set the PERL_UNICODE envariable; see the perlrun manpage for both this and the `-C` switch.**

```
setenv PERL_UNICODE OE
```

- **Still need to arrange to load Unicode fonts (ISO 10646)**

```
% xterm -n unicode +lc -u8 -fn \
      -misc-fixed-medium-r-normal--20-200-75-75-c-100-iso10646-1
```

Using Character Names

- **Allows symbolic names for Unicode characters using `\N{CHARSPEC}` within double-quoted strings**
  - ∗ **Full character names**

```
use charnames ':full';
print "\N{GREEK CAPITAL LETTER DELTA} is delta.\n";
Δ is delta.
```

  - ∗ **Short character names**

```
use charnames ':short';
print "\N{greek:Delta} is an upper-case delta.\n";
Δ is an upper-case delta.
```

- **Any import without a colon tag is taken to be a script (writing language) name, giving case-sensitive shortcuts for those scripts.**

```
use charnames qw(cyrillic greek);
print "\N{Sigma} and \N{sigma} are Greek sigmas.\n";
print "\N{Be} and \N{be} are Cyrillic B's.\n";
```

Translating Long Names and Code Points

- **Translate a Unicode code point into its long name**

```
use charnames qw(:full);
for $code (0xC4, 0x394) {
printf "U+%04X (%s) is %s\\n",
    $code, chr($code), charnames::viacode($code);
}
```

```
U+00C4 (Ä) is LATIN CAPITAL LETTER A WITH DIAERESIS
U+0394 (Δ) is GREEK CAPITAL LETTER DELTA
```

- **Translate a Unicode long name into its code point**

```
use charnames qw(:full);
$name = "MUSIC SHARP SIGN";
$code = charnames::vianame($name);

printf "%s is character U+%04X (%s)\\n",
    $name, $code, chr($code);
```

```
MUSIC SHARP SIGN is character U+266F (♯)
```

- **To find where Perl gets its copy of the Unicode database**

```
% perl -MConfig -E 'say "$Config{privlib}/unicore/NamesList.txt"'
  /usr/local/lib/perl5/5.10.0/unicore/NamesList.txt
```

Case in Unicode

- **Unicode understands not just upper and lower case, but also title case.**
- **In Perl, use ucfirst (or the `\u` escape char)for this:**

```
use charnames ":full";
$sharp_s = "\N{LATIN SMALL LETTER SHARP S}"

printf "lc=%s uc=%s tc=%s\n",
lc($sharp_s), uc($sharp_s), ucfirst($sharp_s);
lc=ß uc=SS tc=Ss
```

- **There's no guarantee that these are round-trippable:**

```
lc(uc("ß"))   # produces ss, not ß!
```

Unicode Character Properties

- May match using character properties in a regex
  - \pX means one character with property X
  - \PX means one character not having property X
- Surround longer property names with braces

```
\p{Alphabetic}      OR      \p{ Upper-case Letter }
```

- Unlike POSIX properties, Unicode properties can occur outside character-class brackets.

```
$is_alpha = $var =~ /^\p{Alphabetic}+$/;
@capwords = ($phrase =~ /\b\p{ Upper-case Letter }+\b/g);
@lowords  = ($phrase =~ /\b\p{ Lower-case Letter }+\b/g);
```

- Unicode properties nicer to read than POSIX

```
$phone =~ /\b[[:digit:]]{3}[[:space:][:punct:]]?[[:digit:]]{4}\b/;
$phone =~ /\b\p{Number}{3}[\p{Space}\p{Punctuation}]?\p{Number}{4}\b/;
```

Combined Characters

- **In Unicode, a base character combines with one or more following non-spacing characters, usually diacritics, such as accent marks, cedillas, tildas, etc.**
- **Because pre-combined characters also exist, there can be two or more ways of writing the same thing**
- **For example, "façade" can be written with just one character between the two a’s, a "ç" (U+00E7) just as in ISO 8859-1**

```
$w1 = "fa\xE7ade";             # "façade"
```

- **But it can also be written as a normal "c" followed by the non-spacing character U+0327**

```
$w2 = "fac\x{327}ade";         # also "façade"
```

Problems with Combined Characters

- Combined characters makes tricky even otherwise simple operations. They look the same when printed, but are treated as unequal strings of different lengths.

```
for $w ($w1, $w2) {
    printf "%s has length %d: [%vx]\n",
        $w, length($w), $w;
}
façade has length 6: [66.61.e7.61.64.65]
façade has length 7: [66.61.63.327.61.64.65]
```

- For example, a naïve reverse would apply combining characters to the wrong character!

```
for $word ("anne\x{301}e", "nin\x{303}o") {
printf "%s simple reversed to %s\n", $word,
    scalar reverse $word;
}
année simple reversed to éenna
niño simple reversed to õnin
```

Problems with Combined Characters, cont.

- One way to deal with this is with the `\X` regex escape: it means a normal character followed by any number of non-spacing combining characters following it.
- Technically, that's just a short-cut for `(?:\PM\pM*)`, or in long-hand:

```
(?x:        # begin non-capturing group
    \PM     # a char w/o the M (mark) property, eg, a letter
    \pM     # a char w/ the M property, eg, an accent
       *    # as many marks as you want
)
```

- Now the bouncing-diacriticals problem can be addressed:

```
for $word ("anne\x{301}e", "nin\x{303}o") {
printf "%s better reversed to %s\n", $word,
    join("", reverse $word =~ /\X/g);
}

année better reversed to eénna
niño better reversed to oñin
```

Unicode::Normalize

- The same character sequence can sometimes be specified in multiple ways.
- Suppose you want the letter "c" to carry both a cedilla and a caron!
- Three different ways of encoding that would be:

```
1. LATIN SMALL LETTER C WITH CEDILLA
   plus COMBINING CARON

2. LATIN SMALL LETTER C
   plus COMBINING CARON
   plus COMBINING CEDILLA

3. LATIN SMALL LETTER C
   plus COMBINING CEDILLA
   plus COMBINING CARON
```

- `Unicode::Normalize` can normalize the different encodings...

Unicode::Normalize, continued

- `Unicode::Normalize` provides functions to rearrange such strings into a reliable ordering. Here are some of them:
  - ∗ NFD for canonical decomposition
  - ∗ NFC for canonical decomposition, followed by canonical composition
  - ∗ NFKD is the compatibility decomposition form
  - ∗ NFKC is compatibility decomposition, followed by canonical composition
- Example: Comparing normalized strings

```
use Unicode::Normalize;

$s1 = "fa\x{E7}ade";
$s2 = "fac\x{327}ade";

if (NFD($s1) eq NFD($s2)) { print "Yup!\n" }
```

Troubleshooting Perl Unicode Issues

- **Providing you're not interfacing with XS modules, and you're using a version of Perl that's within 5 years old, you should be ok.**
- **Here are the four things that may trip you up:**
  - ∗ **1. Knowing *what* Perl documentation to read about Unicode, reading it, and then making the least bit of sense out of what you've just read.**
  - ∗ **2. Understanding I/O Layers and encodings; that is, things like encoding vs Encode, ::via::, `binmode`, -C, envariables, triadic open, etc.**
  - ∗ **3. The specific troubles of getting Unicodish action on codepoints in the `U+0080 .. U+OOFF` range.**
  - ∗ **4. Difficulties comparing and matching Unicode data that hasn't been first laundered, er, normalized, into a standard canonical from, generally due to combining characters, ordering, and pre-combined characters.**

Troubleshooting Perl Unicode Issues, continued

- Mixing true binary data with character data is bound to be problematic, and sometimes you aren't sure what sort of string you have.
- The `encoding::warnings` pragma emits warnings whenever byte-string (octets only) data gets implicitly converted into UTF-8.
- The `Devel::Peek` module will report whether the internal UTF8 flag is set.
- The built-in `utf8::is_utf8(STRING)` function tells you whether Perl's internal utf8 flag is set. You don't need to say `use utf8` to get at it.
- Just because `use utf8` lets you use Unicode in your identifiers doesn't mean you should. `$niño` might be ok, but due to how modules work, package names and even subroutines may need to map to the filesystem, and that's trouble.

Dealing with Codepoints 128 .. 255

- **For legacy reasons, 8-bit data in your own program is treated as binary data; meaning, it doesn't have the UTF8 flag on.**

```
perl -E 'say chr(0xdf)'
ß

% perl -E 'say ucfirst chr(0xdf)'
ß
```

- **What's even more bizarre is that this happens even if you say your whole program is in utf-8!**

```
% perl -E 'use utf8; say ucfirst chr(0xdf)'
ß

% perl -E 'use encoding "latin1"; say ucfirst chr(0xdf)'
Ss
```

The utf8 Pragma

- **This pragma tells Perl the code it's *compiling* is in utf-8.**

```
csh% perl -CS -E 'printf qq{say "%c\\\\u%c";\n}, 0xDF, 0xDF'
say "ß\uß";

ksh$ perl -CS -E 'printf qq{say "%c\\u%c";\n}, 0xDF, 0xDF'
say "ß\uß";
```

- **This proves we really are getting output encoded as utf-8:**

```
% perl     -E 'printf qq{say "%c\\\\u%c";\n}, 0xDF, 0xDF' | wc -c
12

% perl -CS -E 'printf qq{say "%c\\\\u%c";\n}, 0xDF, 0xDF' | wc -c
14
```

- **Now run against itself:**

```
% perl -CS -E 'printf qq{say "%c\\\\u%c";\n}, 0xDF, 0xDF' > /tmp/eg

% perl -Mutf8 -M5.10.0 /tmp/eg
ßSs

% perl -M5.10.0 /tmp/eg | wc -c /tmp/eg /dev/stdin
      14 /tmp/eg
      5 /dev/stdin

% perl -M5.10.0 /tmp/eg | od -c
0000000    Ã 237   Ã 237  \n
```

Forcing Matters with pack

- You don’t always know whether something like `chr(0xDF)` or `"\x{DF}"` will produce a honest-to-goodness UTF-8 character or non-UTF8 byte.
- Instead of using a `chr()`, use one of these to be sure:

```
sub uchar(_) {
    my $cp = shift;
    return pack("U*", $cp);
}

sub byte(_) {
    my $cp = shift;
    return pack("C*", $cp);
}
```

Forcing Matters with the Encode Module

- The `encode` and `decode` functions from the Encode module may be handy, tricky, or both.
- The tricky bit is that when you encode something, the UTF8 bit is off, and when you decode it, it’s on.

```
use Encode;

$data = "fa\xE7ade";
say uc $data;

  FAçADE

$data = decode("iso-8859-1", $data);
say uc $data;

 FAÇADE
```

Græcum est: non potest legi!

- **Many languages have strangenesses in their corners, like whether you use a special form of a letter in certain positions:**
  - **German has terminal ß for ss**
  - **Greek has two lower-case sigmas: ς at the end, but σ normally.**
- **Sometimes getting this all working seems like playing whack-a-mole, a Sisyphean task:**

```
use utf8;
use Unicode::Normalize;

$ηρωσ = "Περικλης"; # Pericles -> Selcirep
$χθων = "Σισυφος";  # Sisyphus -> Suphysis
for $word ($ηρωσ, $χθων) {
    $drow = join("" => reverse(NFKC($word) =~ /\X/g));
    $drow = "\u\L$drow";
    $drow =~ s/σ\b/ς/;   # change to stigma
    print "$word [\U$word\E] flips to $drow\n";
}
Περικλης [ΠΕΡΙΚΛΗΣ] flips to Σηλκιρεπ
Σισυφος [ΣΙΣΥΦΟΣ] flips to Σοφυσις
```

What Unicode Isn't: i18n != l10n,g11n

- Before we had Unicode, we'd set locale variables that handled among other things, 8-bit data, such as ISO 8859-1 vs 8859-7.
- Unicode helps internationalization ("i18n") but specifically does not address local issues, including:
  - Numeric, date, and time formats
  - Collation and sorting
- Perl can be asked to honor one's LC_* locale variables as needed,
- Not all systems offer equal locale support.
- It may be tough to find adequate fonts.
- One has to locate and learn who to use Unicode-aware editors and tools — or write them oneself.
- HINT: launder Unicode data via `NFKC` from `Unicode::Normalize` for better support in shells and browsers.

Example of Sorting (Collation) Troubles

- **Between 1803-1997, the Spanish alphabet placed *chocolate* before *color* because their alphabet said so:**
  - **Old: a b c ch d e f g h i j (k) l ll m n ñ o p q r s t u v (w) x y z.**
  - **New: a b c d e f g h i j (k) l m n ñ o p q r s t u v (w) x y z.**
- **So while *chocolate* now precedes *color*, *piñata* still comes after *pinza*: *¡Qué lo paséis bien!* ("Have a nice day!" :-).**
- **Spanish ignores its acute accents and the occasional diaeresis, but *NOT* the tilde on the eñe, so this is the correct sequence :**
  `radio ráfaga rana ranúnculo raña rápido rastrillo`
- **Other challenges include hyphenating words, a *very* hard problem. I suggest the `Tex::Hyphen` module as a starting point for English; it uses Knuth's algorithm by default.**

And That Was the Easy Stuff

- **The Icelandic alphabet, in contrast, is ordered this way: a á b d ð e é f g h i í j k l m n o ó p r s t u ú v x y ýþæö   .**
- **That's right: diacritical markings count, and the letters after ý are surprises. Like Old English and Old Norse, Icelandic treats æ as a single character, whereas in current English, we decompose it into an a + e.**
- **In German, umlauted characters are treated as the original vowel + e for collation — but only for telephone books; otherwise they ignore the umlauts. ß is always ss.**
- **Danish and Norwegian similarly decompose the Å/å (the angstrom A with the circle) into a double-a sequence, and vice-versa.**
- **It gets even more interesting in Nordic countries: things like where W or ø falls, not just on country and language, but on the year.**

Help for non-Unicode l10n Troubles

- **A good start is at *http://en.wikipedia.org/wiki/Collating_sequence***
- **Standards are good:**
  - ∗ **The standard on Unicode Collation Algorithm - UTS #10, is at *http://www.unicode.org/reports/tr10/***
  - ∗ **The Default Unicode Collation Element Table (DUCET) can always be found at *http://www.unicode.org/Public/UCA/latest/allkeys.txt***
  - ∗ **For info on Unicode Normalization Forms - UAX #15, see *http://www.unicode.org/reports/tr15/* and Public Review Issue #29: Normalization Issue at *http://www.unicode.org/review/pr-29.html***
  - ∗ **For canonical equivalence in applications, see UTN #5 at *http://www.unicode.org/notes/tn5/***
- **Perl's `Unicode::Collate` and `Unicode::Normalize` manpages**
  - ∗ **`man 3 C<Unicode::Collate>` and `man 3 C<Unicode::Normalize>` if you're lucky**
  - ∗ **Otherwise *http://kobesearch.cpan.org/htdocs/perl/Collate.html* and *http://kobesearch.cpan.org/htdocs/perl/Normalize.pm.html***

Perl’s Unicode Documentation

- **Standard manpages on *perlrun*, *perllocale*, *perlunicode*, *perlunitut*, *perluniintro*, and *perlunifaq*, all in section 1.**
- **Variables in *perlvar* including `${^UNICODE}`, `${^UTF8CACHE}`, `${^UTF8LOCALE}`, `${^OPEN}`, and the internals-only `${^ENCODING}`.**
- **A bit in the *perlre* manpage, and more in *perlretut*.**
- **Functions in *perlfunc* such as ord & chr, open & binmode, the case-related functions, (s)printf, and pack & unpack.**
- **Unicode-related pragmata with their own manpages in section 3 (probably): *utf8* and *bytes*, *open* and *charnames* ,and *encoding* and *encoding::warnings*. Oh, and the *locale* pragma.**
- **Unicode-related modules that Perl v5.10 ships with tend to fall under the *Encode::* and *Unicode::* hierarchies, but you might want to check *Locale::* modules, too.**

Pattern Matching Novelties

- Perl v5.10 introduced quite a few new pattern-matching features.
- The new but already-discussed `/p` modifier, and the `${^PREMATCH}`, `${^MATCH}`, and `${^POSTMATCH}`) it loads
- Your back-referenced capture buffers can be referred to relatively instead of absolutely.
- You can even name your capture buffers.
- You no longer need to use `(??{ CODE })` for recursive patterns.
- Much finer control over what does or doesn't get back-tracked into is available that the (?>...) construct permits.
- More explicit white-space matching through `\v`, `\h` (and their complements) and `\R`.

Vertical White Space

- **Perl has never recognized "`\v`" to mean a vertical tab, even though the ASCII character 11 is used for that.**
- **In patterns, now, `\v` now matches any sort of vertical white space, while `\V` means *not* vertical white space.**

```
% perl -wE 'say "\v"'
Unrecognized escape \v passed through at -e line 1.
v

% perl -wE 'say chr(11) =~ /\v/'
1
```

- **Form feeds, newlines, and returns also count, plus various Unicode stuff:**

```
% perl -wE 'say "\n" =~ /\v/'
1
% perl -wE 'say "\r" =~ /\v/'
1
% perl -wE 'say chr(0x2029) =~ /\v/'
1
% perl -Mcharnames=:full -wE \
    'say "\N{PARAGRAPH SEPARATOR}" =~ /\v/'
1
```

Horizontal Whitespace

- **Similarly, `\h` in a pattern means horizontal white-space, also known as `\p{HorizSpace}`.**
- **It can be negated, so `\H` (or `\P{HorizSpace}`) matches anything that isn't horizontal white space.**
- **The new pattern escape, `\R`, matches any "return sequence".**
- **It's syntactic sugar for vertical white space or CRLF:** `(?:\x0D\x0A|\v)`
- **Remember that `/$/` is syntactic sugar for /(?=>\n?)\z/, so use `\z` if that's what you really need.**
- **Similarly, `/./` is syntactic sugar for `/[^\n]/` unless you write it as `/./s` instead.**

Capture Buffers

- **When you build up patterns piecewise, a problem arises with numbered capture buffers.**

```
$dupword = qr{ \b ( ( \w+ ) ( \s+ \2 )+ ) \b }xi;
while (<>) {
    printf "%s %d: %s\n" => ( $ARGV => $. => ${^MATCH} )
        while /$dupword/pg;
} continue {
    close ARGV if eof;
}
```

- **Although we could have used non-capturing groups for some of those:**

```
$dupword = qr{ \b (?:
                      ( \w+ )
                      (?:
                          \s+ \1
                      )+
                   ) \b
   }xi;
```

Combining Capture Buffers

- **It still can't be embedded very easily. This, for example, won't work:**

```
$dupword = qr{ \b (?: ( \w+ ) (?: \s+ \1 )+ ) \b }xi;
$quoted  = qr{ ( ["'] ) $dupword  \1 }x;
```

- **Because `$dupword` should be using `\2` there if it's going to wind up embedded in `$quoted`.**
- **But you can't do that because when it's being compiled, there's *NOT YET* a second capture buffer to refer back to! It won't even compile.**
- **The new notation, `\g{N}`, refers back to capture group *N*.**
  - ∗ **When N is positive, it's the same as `\N`.**
  - ∗ **When N is negative, it's the *Nth*-to-the-last buffer; ie, the *Nth* previous one.**

Relative Capture Buffers

- **To fix that code to run correctly, do this:**

```
$dupword = qr/\b(?:(\w+)(?:\s+\g{-1})+)\b/i;
```

- **Well, or spaced out more via `/x`:**

```
$dupword = qr{
    \b (?:
            ( \w+ )
            (?:
                \s+
                \g{-1}
            )+
       ) \b
}xi;
```

Relative Capture Buffer Demo

- **Now the following program runs correctly:**

```
use 5.10.0;

$dupword = qr{ \b (?: ( \w+ ) (?: \s+ \g{-1} )+ ) \b }xi;
$quoted = qr{ ( ["'] ) $dupword  \1 }x;
$/ = q[];  # cross paragraphs

while (<>) {
    printf "%s %d: %s\n" => ( $ARGV => $. => ${^MATCH} )
        while /$quoted/pg;
} continue {
    close ARGV if eof;
}
```

- **But there's still a niggle — or two:**
  - ∗ **If `$quoted` is used in a larger pattern, its use of `\1` will be wrong.**
  - ∗ **Yet it can't know how many to count back because it shouldn't have to know how many `$dupword` used.**

Named Capture Buffers

- **The solution is to name your capture buffers.**
  ```
  (?<NAME>...)
  ```
  is like a regular parenthesized grouping, but its name is *NAME*.
- **Now to refer back to it, use `g{NAME}`.**
- **This fixes the previous issue:**
  ```
  $quoted = qr{ (?<quote> ["'] )
                 $dupword
                \g{quote}
  }x;
  ```

What About Outside the Pattern?

- That's fine for within the pattern, but just as buffer `\1` outside the pattern is `$1`, you want to get at the named buffers, too.
- The new, special variable `%+` serves that purpose, so in the previous code, use `$+{quote}` to learn which quote it found.
- Here's another example

```
use 5.010;
$word = "bookkeeper";
$word =~ s{ (?<letter> \p{Alphabetic} )
            \g{letter}
}{$+{letter}}gix;
```

- That will reduce "bookkeeper" to "bokeper", but "bOokKEeper" to "bOkEper".
- To fix, make the replacement portion `{\u$+{letter}}`.
- If you've several capture buffers by the same name, `%-` is a hash that holds array refs to all the values each grabbed.

Recursive Patterns

- **Previous to v5.10, one could write recursive patterns, but only in a rather roundabout fashion:**

```
use re 'eval';
$normal = qr/[^()]*/;
$parened = qr{
    \(                    # opening paren
      ( $normal           # either text
          |               # or
       (??{ $parened })   # nested parens
      ) *                 # multiple times
    \)                    # closing paren
}x;
```

- **For example, to find all function calls plus their arguments:**

```
$ident = qr/\b\w+\b/;
$callfunc = qr{
    & ?              # optional ampersand
    $ident \s*      # func + any amount of whitespace
    $parened         # and a balanced paren group
}x;

while ( <> ) {
    printf "%s %d: %s\n", $ARGV, $., ${^MATCH} while /$callfunc/gp;
    close ARGV if eof;
}
```

What about Malformed Data?

- If a function call started on one line but continued, and you were reading the file linewise, you'd have a big problem.
- The way "normal" was defined allows for far too much backtracking; you'll seem to hang forever.
- One way to fix ths is to use a no-backtracking group:

```
$parened = qr{
    \(                    # opening paren
     (?>                  # start no-backtrack group
      ( $normal           # either text
          |               # or
       (??{ $parened })   # nested parens
      ) *                 # multiple times
     )                    # end no backgrack group
    \)                    # closing paren
}x;
```

Possessive Quantifiers

- **New to Perl 5.10.0 are *possessive quantifiers*, which block backtracking.**
- **These are reminiscent of the minimal-matching quantifiers, but use a plus not a question mark:**

```
    Maximal:    *    +    ?    {n,m}
    Minimal:    *?   +?   ??   {n,m}?
 Possessive:    *+   ++   ?+   {n,m}+
```

- **You could solve the bad-data problem on the previous page two ways:**
  - **Change the definition of normal to a possessive match:**

    ```
    $normal = qr/[^()]*+/;
    ```

  - **Or else add the + to the * where it says "`# multiple times`".**

    ```
    (
      $normal            # either text
         |               #   or
     (??{ $parened })    # nested parens
    ) *+                 # multiple times, w/o backracking
    ```

Cheaper Recursive Patterns

- **New to Perl 5.10.0, you can now recurse without evaluating embedded code.**
- **Do this using `(?N)` where *N* is that buffer you want to recurse on. For example:**

```
$parened = qr{
    \(                        # opening paren
      (                      # start first capture buffer
        $normal              # either text
          |                  #    or
        (?1)                 # recurse on the first buffer
      ) *                    # multiple times
    \)                       # closing paren
}x;
```

Infinitely Recursive Patterns

- **This works fine on good data, but on bad data, you'll die with the exception: `Infinite recursion in regex`.**
- **Now having `$normal` as a possessive match isn't enough; you *must* make the interior a possessive match, too!**

```
$parened = qr{
    \(                         # opening paren
      (                       # start first capture buffer
        $normal               # either text
          |                   #    or
        (?1)                  # recurse on the first buggger
      ) *+                    # multiple times, no BT
    \)                        # closing paren
}x;
```

Signals, safe and otherwise

- Originally, Perl would deliver a signal to your signal handler whenever that signal came in.
- Often C libraries are not re-entrant friendly, nor is the Perl interpreter itself.
- This was a recipe for disaster, because the standards say you can barely twitch in a signal handler; you certainly can't allocate memory.
- Sometimes it worked; sometimes it didn't.
- By default, Perl now delivers signals synchronously, meaning only between executions of individual Perl op-codes. This way you avoid the real threat of memory-corruption.

But I Meant to Do That!

- Unfortunately, this isn't always a good solution: what if you're in a stalled read, or doing a long sort?
- If your system support SA_RESTART, Perl used to always automatically restart syscalls that got interrupted, so there was no way out of them.
- This is no longer true due to safe signals, so certain calls may return EINTR that you weren't expecting them to. Those you'll have to restart yourself.
- However the `:perlio` layer *will* still always restart `read`, `write`, and `waitpid`.

Safe Signals

- **That means code like this is safe to run now:**

```
$SIG{INT} = sub { $finished = 1 } ;
$| = 1;
print "Hit ^C to exit: ";
1 until $finished;
print "\n";
```

- **But this will likely be a problem as you hang in sort a long time:**

```
use sort '_quicksort';      # use a quicksort algorithm

our $finished = 0;
$SIG{INT} = sub { $finished = 1 } ;
@x = (1 .. 500_000);
printf "x has %d elts\n", scalar @x;
$| = 1;
print "Hit ^C to exit: ";
do { @x = sort @x; print "." } until $finished;
print "\n";
```

- **The pragma was because Perl’s new adaptive sort is too fast/smart to slow down this way.**

Unsafe Signals

Code like this *appears* to still work, but isn't trustworthy:

```
sub timeout (&$) {
    my ($code, $duration) = @_;
    local $SIG{ALRM} = sub { die "RRRRING\n" };
    eval {
        alarm $duration;
        eval { $code->(); };
        alarm 0;
    };
      alarm 0;
    die if $@ && $@ !~ /^RRRRING/;
}

my $name;
do {
    timeout {
        print "\nName: ";
        chomp($name=<STDIN>);
    } 5;
} until $name;

print "Your name is $name.\n";
```

Unsafe Signals, continued

- **You have three choices to re-instate unsafe signals if you insist:**
- **1. You can set the environment variable PERL_SIGNALS to "unsafe".**

```
csh% setenv PERL_SIGNALS unsafe
sh% export PERL_SIGNALS=unsafe
```

- **2. You can try out the CPAN module `Perl::Unsafe::Signals`.**
- **3. You can try accessing POSIX stuff directly.**
- **An example of this last is found in the *perlipc* manpage:**

```
   use POSIX qw(SIGALRM);
   POSIX::sigaction(SIGALRM,
    POSIX::SigAction->new(sub { die "alarm" }))
 or die "Error setting SIGALRM handler: $!\n";
```

Perl Threads

- You want to do more than one thing at a time
  - ∗ boss/worker
  - ∗ work crew
  - ∗ pipeline
- A thread is a flow of control through a program with a single execution point
- Can have multiple threads per process
- Each gets a separate copy of the Perl interpreter (ithreads)
- All of a process's threads share process metadata: file handles, user id, process id, current directory, etc.
- Program data *not* shared by default

Perl Threads, continued

- Perl must be compiled to support threads
- sh Configure -Dusethreads
- Earlier Perl5.005 threading model now obsolete

```
use Threads;         # Perl 5.005, avoid; just a wrapper
use threads;         # new ithreads model
```

- ithreads introduced with Perl 5.6.0
- Still evolving, but first became reasonably functional Perl 5.8.1

Creating a Thread

- Create a threads object, pass subroutine and parameters

```
use threads;
$thread = threads->create( \&do_work, ... );
sub do_work {
my @params = @_;
# ...
return @results;
}
```

- Pass subroutine name or reference to threads->create
- could be anonymous subroutine

```
$thread = threads->create( sub { ... }, ... );
```

- threads->new is an alias for threads->create

```
$thread = threads->new( \&do_work, ... );
```

- async is an alias for threads->create(sub { })

```
$thread = async { foreach (@files) { ... } };
```

Joining or Detaching

- Use join to wait for a thread to complete

```
$thread  = threads->create(\&do_work, ...);
@results = $thread->join;
```

- retrieves data returned by thread
- cleans up the thread
- Use detach if you don’t care about thread

```
threads->create(\&do_work, ...)->detach;
```

- cannot retrieve data from thread
- thread will go away when done
- might get warnings

```
A thread exited while 3 other threads were still running
```

- Order in which threads execute/complete unpredictable
- Thread may block on I/O, etc.

Example: Retrieving data from a thread

```
#!/usr/bin/perl5.8.1 -w

use threads;

my @data = (1, 4, 8);
my $thread = threads->create(\&do_work, @data);

print "Main program thread still running...\n";
print "Retrieving results from child thread: ";

my $result = $thread->join;

print "$result\n";

sub do_work {
my @params = @_;
my $total = 0;
$total += $_ for @params;
return $total;
}
```

Working with Threads Objects

- Use self to get the current thread object

```
my $thread = threads->self;
```

- Use tid to get the thread ID (1, 2, 3, etc.)

```
my $id = threads->self->tid;
my $id = threads->tid;          # same
```

- Given a thread ID, fetch the object

```
my $thread = threads->object( $id );
```

- Use list to get a list of thread objects

```
foreach $thread (threads->list) {
my $id = $thread->tid;
print "Joining with thread $id\n";
$result = $thread->join;
}

$_->join for threads->list;
```

Example: Creating multiple threads

```
#!/usr/bin/perl5.8.1 -w

use threads;

my @data = ([1,4,8], [4,7,3], [3,9,6]);

for $i (0..2) {
my $thread = threads->create( \&do_work, @{$data[$i]} );
my $thread_id = $thread->tid;
print "Thread $thread_id created\n";
}

print "Retrieving results from child threads:\n";

for $thread (threads->list) {
my $thread_id = $thread->tid;
my $results   = $thread->join;
print "Results from thread $thread_id: $results\n";
}

# continues...
```

Example: Creating multiple threads, cont.

```
# continued...

sub do_work {
my @params = @_;
my $thread_id = threads->self->tid;
print "In thread: $thread_id, parameters: @params\n";
my $total = 0;
$total += $_ for @params;
return $total;
}
```

Example: Detaching threads

```
#!/usr/bin/perl5.8.1 -w
use threads;

my @data = ([1,4,8], [4,7,3], [3,9,6]);
open(OUT, ">results") || die "cannot create results file";

for $i (0..2) {
my $t = threads->create( \&do_work, @{$data[$i]} );
my $thread_id = $t->tid;
print "Thread $thread_id created\n";
}
$_->detach for threads->list;

sub do_work {
my @params = @_;
my $thread_id = threads->self->tid;
my $total = 0;
$total += $_ for @params;
print OUT "Thread $thread_id results: $total\n";
}
```

Example: Create and immediately detach

```
#!/usr/bin/perl5.8.1 -w
use threads;

my @data = ([1,4,8], [4,7,3], [3,9,6]);

open(OUT, ">results") || die "cannot create results file";

for $i (0..2) {
threads->create( \&do_work, @{$data[$i]} )->detach;
}

sub do_work {
my @params = @_;
my $thread_id = threads->self->tid;
my $total = 0;
$total += $_ for @params;
print OUT "Thread $thread_id results: $total\n";
}
```

Sharing Data Among Threads

- **Variables *not* shared by default**
- **Must declare shared variables with :shared attribute... use threads;**

```
use threads::shared;
my $foo : shared;
my @foo : shared;
my %foo : shared;
```

- **Or, use the share function to add sharing at run-time**

```
use threads;
use threads::shared;
my %foo;
# ...
share(%foo);
```

- **Threads should use lock function for advisory locking**

Example: Sharing data

```
#!/usr/bin/perl5.8.1 -w
use threads;
use threads::shared;

my @Data : shared = (1, 4, 8);
my $t;
$t = threads->create( \&do_work, 9, 6, 2 );
$t = threads->create( \&do_work, 5, 7, 3 );
$t = threads->create( \&do_work, 1, 1, 3 );
$_->join for threads->list;

print "Data: @Data\n";

sub do_work {
my @params = @_;
lock(@Data);
for (my $i = 0; $i < @Data; $i++) {
    $Data[$i] += $params[$i] if defined $params[$i];
}
}
```

Sharing a Data Queue

- **Can create a data queue accessible to all threads**

```
use threads;
use Thread::Queue;
my $DataQueue = Thread::Queue->new;
```

- **Add scalars to queue**

```
$DataQueue->enqueue("aaa", "gct", "ttg");
```

- **Extract scalars from the queue (first-in, first-out)**

```
$val = $DataQueue->dequeue; # blocks if queue empty
$val = $DataQueue->dequeue_nb;      # non-blocking, rtns undef when empty
```

- **Can add undef to signify that thread should stop trying**

```
$DataQueue->enqueue("aaa", "gct", "ttg", undef);
while (defined($val = $DataQueue->dequeue)) { ... }
```

- **Can determine number of items pending in queue `$num_items` = `$DataQueue`->pending;**

Example: Sharing a data queue

```
#!/usr/bin/perl5.8.1 -w

use threads;
use Thread::Queue;

my $DataQueue = Thread::Queue->new;

while (<DATA>) {
chomp;
$DataQueue->enqueue($_);
}
$DataQueue->enqueue(undef) for 0..2;

my $t = threads->create( \&do_work ) for 0..2;
$_->join for threads->list;

# continues...
```

Example: Sharing a data queue, continued

# continued

```
sub do_work {
my ($tid, $seq, $revcom);
$tid = threads->tid;    # same as threads->self->id
while (defined($seq = $DataQueue->dequeue)) {
    ($revcom = $seq) =~ tr/acgt/tgca/;
    print "Thread $tid: reverse compl: $revcom\n";
    sleep 1;            # fake more work...
}
}

__END__
gttaatgcca cttgatgata atagcatacc agatataaag
cttccaattt ggattgttta tggaacaatt aacaacagga
gcgtatagtt aaaaaccgat tacctaagtt ctcaaaattc
ttcatttgat tttattggta taaactatta ctcttctagt
acatggcaat gccaaaccca gttactcaac aaatcctatg
...
```

Legacy Fancy Pattern Matching

- **This section describes Perl's fancy pattern features you may not be familiar with.**
- **They came out before v5.10, but haven't all been around forever.**

Regular Expressions are Eager and Greedy

- **Always seek the leftmost starting point (eager)**

```
$var = "Graham:Rowley:24 Laurier St:Boulder:CO";
$var =~ s/:[^:]*/:Farquharson/;
```

- **Longest match from the leftmost start (greedy)**

```
$var = "Graham:Rowley:24 Laurier St:Boulder:CO";
$var =~ s/.*:/Diana:/;
```

- **Greed sometimes gives way to eagerness**

```
$var = "good food";
```

Non-greedy Quantifiers

- **Add ? to quantifiers to look for leftmost-shortest match**

```
$var = q(It’s raining cats and dogs and cats and dogs!);
$var =~ s/cat.*dog/cow/;    # greedy
print $var;
It’s raining cows!
```

```
$var = q(It’s raining cats and dogs and cats and dogs!);
$var =~ s/cat.*?dog/cow/;   # non-greedy
print $var;
It’s raining cows and cats and dogs!
```

- **Helpful for matching enclosures of text**

```
$html = "<B>Finest</B> French-fried <I>freshest</I> fish!";
$html =~ s/<.*?>//g;      # simplistic tag removal
print "$html\n";
Finest French-fried freshest fish!
```

Flags for Match and Substitute Operators

- **Global**

```
$var =~ m/RE/g    $var =~ s/RE/repl/g
```

- **Case Insensitive**

```
$var =~ m/RE/i    $var =~ s/RE/repl/i
```

- **Multi-line mode**

```
$var =~ m/RE/m    $var =~ s/RE/repl/m
```

- **Single-line mode**

```
$var =~ m/RE/s    $var =~ s/RE/repl/s
```

- **Extended regular expressions**

```
$var =~ m/RE/x    $var =~ s/RE/repl/x
```

- **Compile regular expression once only**

```
$var =~ m/RE/o    $var =~ s/RE/repl/o
```

Extended Regular Expressions

- **Can spread out RE's and comment by adding /x flag**

```
  if ($num =~ /
^           # start of string
(
  [-+]?       # optional sign
  \d+         # 1 or more digits
  \.?         # optional dot
  \d*         # 0 or more digits
|           #    OR:
  [-+]?       # optional sign
  \.          # dot
  \d+         # 1 or more digits
)
$           # end of string
  /x) {
print "Thank you for entering a number!\n";
  }
```

Four Ways to Use the Match Operator

- **Scalar context - Returns true or false**

```
if ($var =~ m/^[yY]/) { print "Yes!\n" }
```

- **List context - Returns text captured with ()'s**

```
$date = "Date of birth: 05/30/91";
($mon, $day, $yr) = $date =~ m#(..)/(..)/(..)#;
```

- **Global match, scalar context - Iterates for each match**

```
$html = "<B>Three</B> free <I>fleas</I> flew!";
while ($html =~ m/(<.*?>)/g) {
    print "Possible tag: $1\n";
}
```

- **Global match, list context - Returns all matching text**

```
@tags = $html =~ m/<.*?>/g;
print "@tags\n";
<B> </B> <I> </I>
```

Determining Position in Global Matching

- Use *pos* to get the index of where next search begins
- Report where matches found within a sequence

```
$pattern = "AC[GTA]";
while ($dna_seq =~ /($pattern)/g) {
    $match = $1;
    $position = pos($dna_seq) - length($match);
    print "Match found: $match  Index: $position\n";
}
```

- Assign to `pos` to change where next search begins
- Identify possibly overlapping matches

```
$pattern = "GCA.{0,25}?[TC]A";
while ($dna_seq =~ /($pattern)/g) {
    $match = $1;
    $position = pos($dna_seq) - length($match);
    pos($dna_seq) = $position + 1;
    print "Match found: $match  Index: $position\n";
}
```

Evaluating Substitute Replacement

- **Substitute operator may have a replacement expression**

```
$var =~ s/RE/repl/e;
```

- **Example - Convert text to Title Case**

```
$title = "northward oVer THE greAT ice";
$title =~ s/\w+/ ucfirst(lc($&)) /ge;
print "$title\n";
Northward Over The Great Ice
```

- **Example - Convert temperatures**

```
$temps = "Daytime low was -20F, high was 5F";
$temps =~ s{
    (-? \d+) F \b
}{
    sprintf("%.0fC", ($1 - 32)*5/9)
}gxe;

print "$temps\n";
Daytime low was -29C, high was -15C
```

Multi-Line Pattern Matching

- **Multi-line strings can be read in and matched**

```
$var = ‘cat fileA‘;

undef $/;                          $/ = "";
$var = <FH>;                       $var = <FH>;
```

- **Assume, for example, three lines in a variable**

```
$var = "xyz\nabc\n123\n";
```

- **Some patterns that match**

```
$var =~ /^xyz/;
$var =~ /123$/;
$var =~ /xyz\nabc/;
```

- **Some patterns that don’t match**

```
$var =~ /^abc$/;
$var =~ /^xyz.*123$/;
```

Single-Line Mode

- **The `/s` flag allows dot to match newline**

```
$var = "xyz\nabc\n123\n";
$var =~ /^xyz.*123$/;        # doesn't match
$var =~ /^xyz.*123$/s;       # matches
$var =~ s/xyz.abc/Hello/s;
print $var;
```

```
  Hello
  123
```

- **Example - Remove C comments**

```
$program =~ s{

    /\*
      .*?
    \*/

}{}gsx;
```

Multi-Line Mode

- **The m flag allows `^` and `$` to find multiple matches**

```
$var = "xyz\nabc\n123\n";
$var =~ /^abc$/;    # doesn't match
$var =~ /^abc$/m;   # matches
$var =~ /^xyz$/m;   # matches
```

- **May use `\A` and `\Z` as anchors to start/end of string**

```
$var =~ /\Axyz/m;   # matches
$var =~ /123\Z/m;   # matches
```

- **Example using both `/m` and `/s` flags**

```
open(FH, "Addr_data") or die "cannot open Addr_data";
$/ = "";        # read in paragraph mode
while ($rec = <FH>) {
    if ($rec =~ /^NAME \t Jim .* ^STATE \t (AZ|Arizona)$/msx) {
        print "Found Jim:\n$rec";
    }
}
```

Assertions and Other Notations

- **Non-capturing parentheses (?:regex)**

```
if (/cat(?:fish|dog|aclysm) ... /) { ... }
```

- **Embedded comments (?#comment)**

```
if (/(?# integer )^[-+]?\d+$/) { ... }
```

- **Embedded modifiers (?imsx)**

```
s{ (?sx) /\* .*? \*/ }[]g;
```

- **Localized use of ismx flags**

```
/((?s-i)Foo.*Bar).*baz/i    # turning flags on/off within a group
/(?s-i:Foo.*Bar).*baz/i     # same, using non-capturing parentheses
```

Case Conversions and Quoting

- **Special escapes used with substitutes or matches**

```
\u  \l    make next character uppercase/lowercase
\U  \L    make uppercase/lowercase until next \E
\Q        quote until next \E
\E        end previous \L, \U, or \Q
```

- **Examples**

```
$var = "Subject: a walk in the park";
$var =~ s/^(\w+): (.*)/\U$1\E: \u$2/;
print "$var\n";
SUBJECT: A walk in the park


$title =~ s/\w+/\u\L$&/g;          # Title Case
$formula = "C*9/5 + 32";
if (/Degrees Fahrenheit equals \Q$formula\E/) { ... }
```

Regular Expression Compilation

- **REs must be compiled.**
- **No interpolation in RE means compiled once:**

```
if (/foo/)    # compiled with your program
```

- **They also undergo qq-like processing:**

```
$name = "Nat";
if (/Name: $name/)  {...}
if (/Name: Nat/)  {...}   # equivalent
```

- **Variable interpolation delays compilation until runtime.**
- **/o (once) flag prevents recompilation:**

```
 if (/Name: $name/o) {...}
```

- **Promises `$name` won't change value.**

Multi-grep

- **Patterns given on command line.**
- **Print lines matching any pattern.**

```
LINE: while (<FH>) {
    foreach $pattern (@ARGV) {
        if (/$pattern/) {
            print;
            next LINE;
        }
    }
}
```

- **Can’t use /o as `$pattern` will change value.**
- **Compile num. patterns times num. lines of regexps.**

Quoting Regular Expressions

- **Solution: `qr//`**
- **Returns compiled RE**
- **Compiled RE can be interpolated without recompilation**

```
@pats = map { qr/$_/ } @ARGV;
LINE: while (<FH>) {
    foreach $pattern (@pats) {
        if (/$pattern/) {
            print;
            next LINE;
        }
    }
}
```

More on qr

- **Can print compiled RE:**

```
$re = qr/foo.*bar/;
print "Pattern is $re\n";
Pattern is (?-xism:foo.*bar)
```

- **Saves flags:**

```
$re = qr/foo.*bar/i;
print "Pattern is $re\n";
Pattern is (?i-xsm:foo.*bar)
```

- **Can be interpolated into larger REs:**

```
$integer = qr/\d+/;
$decimal = qr/\.\d+/;
if (/$integer($decimal)?/) {
    # 3 or 3.2 but not 4.
}
```

Interpolation Details

- **First pass is like double-quote processing.**
- **Variables are expanded, `\n` converted, etc.**
- **Not *quite* `qq`, as `$|`, `$)` and terminal `$` don't mean variable interpolation.**
- **Example:**

```
$foo = "bar";
/$foo$/
```

- **qq-like processing yields:**

```
/bar$/
```

- **This is then compiled as an RE.**

Implications

- **Treated as a qq-like string before treated as RE.**
- **`qq` processor finds closing delimiter and doesn't know about char classes or RE comments.**
- **Consequence: Can't hide RE delimiter:**

```
# this is a syntax error for two reasons
m{
    [{}]  # }
};
```

- **`\Q` and ilk are handled by `qq()`, not RE compiler.**

```
$bit = q(\Qfoo|bar\E);  # a non-interpolated string

if (/$bit/) {...}

# like
if (/\\Qfoo|bar\\E/) { ... }
```

Lookahead and Lookbehind Assertions

- **Positive lookahead assertion: "the next bit must be XYZ, but don’t match it"**

```
/(?=XYZ)/
```

- **Negative lookahead assertion: "the next bit must not be XYZ, but don’t match whatever it is"**

```
/(?!XYZ)/
```

- **Positive lookbehind assertion: "the last bit in the string must be XYZ, don’t match it"**

```
/(?<=XYZ)/
```

- **Negative lookbehind assertion: "the last bit in the string must not be XYZ, don’t match whatever it is"**

```
/(?<!XYZ)/
```

Lookaheads

- **May be any regular expression**
- **Positive lookahead: "if this thing is next"**

```
# remove doubled words
$_ = "Paris in the the spring.";
s/ \b(\w+) \s (?= \1\b ) //gxi;
# mistakenly "fixes" "the game I won won’t work"
```

- **Negative lookahead: "unless this thing is next"**

```
s/ \b(\w+) \s (?= \1\b (?! ’\w))//xgi;
```

- **Alternative way to find overlapping matches**

```
$string = "0123456789";
@triples  =  $string =~ /(?=(\d\d\d))/g;
print "triples are @triples\n";
```

```
triples are 012 123 234 345 456 567 678 789
```

Lookbehinds

- **Positive lookbehind: "after this thing"**

```
# delete "th" from "19th"
$_ = "the 19th street theatre";
s/(?<=\d)th//g;

the 19 street theatre
```

- **Negative lookbehind: "except after this thing"**

```
# delete single quotes except in words
$_ = q('I hurt,' said Tim O'Reilly.  'Bad.');
s/(?<!\w)'//g;

I hurt, said Tim O'Reilly.  Bad.
```

- **Lookbehind pattern must be fixed width**
- **Use lookarounds when you have markers for what you want to match, but the markers aren't part of what you want to match.**

POSIX RE Classes

- **POSIX character classes are now supported:**

`alnum`	`any alphanumeric (alpha or digit)`
`alpha`	`letter`
`ascii`	`"\x00" .. "\x7F"`
`cntrl`	`Control-character`
`digit`	`\d`
`graph`	`alphanumeric or punctuation`
`lower`	`lowercase letter`
`print`	`alphanumeric or punctuation or space`
`punct`	`punctuation character`
`space`	`\d`
`upper`	`uppercase letter`
`word`	`identifier character (alnum and underline)`
`xdigit`	`hexadecimal digit`

More on POSIX Classes

- **Only available inside a regular character class:**

```
if (/[.,[:alpha:][:digit:]]/)
```

- **Negatable:**

```
/[[:^digit:]]/
```

Additional Properties

- `\p{PROP}` matches characters with this property
- `\P{PROP}` matches characters not having property
- Perl defines extra properties:

`IsASCII`	`7-bit bytes`
`IsAlnum`	`alphabetic and numbers`
`IsAlpha`	`alphabetic`
`IsCntrl`	`control characters`
`IsDigit`	`digit`
`IsGraph`	`all but Cntrl and space`
`IsLower`	`lowercase letters`
`IsPrint`	`non-Cntrl chars`
`IsSpace`	`separators and \n\f\r\t`
`IsUpper`	`uppercase`
`IsWord`	`_ and Alnum`
`IsXDigit`	`Hexadecimal digits`

```
 /(\p{IsAlpha}+)/
```

Additional Properties, continued

- Many additional Unicode properties

```
Arabic
BraillePatterns
CanadianAboriginal
CurrencySymbols
Cyrillic
MathematicalOperators
MusicalSymbols
WhiteSpace
... and hundreds more
```

- To find where Perl gets its list of Unicode properties

```
perl -MConfig -le 'print "$Config{privlib}/unicore/Properties"'
```

Code in Regular Expressions

- Perl code can appear within a regex
- Evaluated at Match-time
- Embedded code `(?{ CODE })`
- Code executed as the RE engine reaches it
- Code doesn’t affect the matching process
- Useful for debugging
- Return value stored in `$^R`
- Dynamic regex `(??{ CODE })`
- Dynamic pattern
- Code returns a pattern
- Allows RE to be completed as it’s being evaluated

Embedded Code for Debugging

- `(?{CODE})` **executed when RE engine reaches that point.**
- **Useful for debugging:**

```
$re = qr/ (\d+) (?{ print "Found $1\n" })
      [.]
    /x;

"12 3.2" =~ /$re/;
Found 12
Found 1
Found 2
Found 3
```

- **Backtracking: 12 failed `\d+`, so try matching less**
- **Bumpalong: no match possible beginning at 1, but perhaps at 2**

Measuring a Span of Text

- **Count the characters that match .**

```
$var = 'The food is on the bar in the barn';
# $var =~ /foo.*bar/;

$var =~ m{
    foo
    (?{ $cnt = 0 })   # initialize $cnt
    (
      .
      (?{ local $cnt = $cnt + 1 })  # increment local $cnt
    )
    bar
    (?{ $result = $cnt })   # copy result
}x;

print "$result\n";

23
```

Dynamic Patterns

```
  sub is_palindrome($) {
my $word = shift();
return lc($word) =~ m{
                      ^
                      ( \p{Alphabetic} + )
                        \p{Alphabetic} ?
                      (??{ lc reverse $1 })
                      $
}x;
  }

  @test_words = qw(joe bob otto Joe Bob Otto reviver);

  for $word (@test_words) {
printf "%s is %s palindromic\n",
        $word,
           is_palindrome($word)
           ? "indeed" : "not";
  }
```

Dynamic Patterns, continued

- **Detecting RNA stem loops**

```
$seq = 'gccuaucuggguuuuguagacgt';

if ($seq =~ m{
                (...)                 # any three nucleotides
                (.{4,20}?)            # then any 4-20 nucleotides
                (??{ revcom($1) })    # followed by rev complement
             }x)
{
    print "Stem loop:   $&\n";
    print "Starts with: $1\n";
    print "Ends with reverse complement of $1\n";
    print "Contains within: $2\n";
}

sub revcom {
    my $rev;
    ($rev = reverse shift) =~ tr/ACGUacgu/UGCAugca/;
    return $rev;
}
```

Recursive Patterns

- **Now possible to create recursive patterns**
- **Example: Matching nested enclosures of text**
- **Delay interpolation of a RE chunk with `(??{ CODE })`**
- **CODE evaluates to next RE chunk**

```
$normal = qr/[^()]*/;

$parened = qr{
           \(                         # opening paren

                ( $normal             # either text
                  |                   # or
                 (??{ $parened })     # nested parenths
                ) *                   # multiple times

           \)                         # closing paren

         }x;
```

Security Concerns About Code in RE's

- **Possible security problem:**

```
$re = param("SEARCH");  # from untrusted user
while (< >) {
    print if /$re/;
}
```

- **User could give `(?{ system("rm -rf /") })`**
- **By default, variables in RE not allowed to present code**
  - ∗ **Match-time execution `(?{code})`**
  - ∗ **Delayed RE interpolation `(?{{code}})`**
- **To allow variable interpolation to include code, you must say**

```
use re "eval";
```