Perl

- Perl = Practical Extraction and Reporting Language
- Developed by Larry Wall in late 80’s as a replacement for awk.
- Grew to become a general purpose language focused on text processing
- Filled an key role with the rise of internet in 1990s
- Importance declining
- But still a useful tool to know because it is:
  - widely available on Unix-like and other operating systems
  - Perl scripts occur many places in existing systems
  - libraries available for many purposes
  - influenced languages including: PHP, Python, Ruby
  - successfuly(??) challenges popular notions of language design
Some of the language design principles for Perl:

- make it easy/concise to express common idioms
- provide many different ways to express the same thing
- use defaults where every possible
- exploit powerful concise syntax & accept ambiguity/obscurity in some cases
- create a large language that users will learn subsets of

Many of these conflict with design principles of languages for teaching.
So what is the end product like?

- a language which makes it easy to build useful systems
- readability can sometimes be a problem (language is too rich?)
- interpreted slow/high power consumption (although still remarkably efficient)
- modest footprint - can be used embedded - but not ideal

Summary: it’s easy to write concise, powerful, obscure programs in Perl
Which Perl

- Perl 5 - first stable widely used version of Perl
- huge number of software libraries available
- CPAN (https://www.cpan.org) has 60,000+
- Perl 6 a very different language (unlike Python 2 versus Python 3)
  - Perl 6 so different its name has been changed to **Raku** not Perl
- Perl 6 development started 2000, v1.0 released end 2015 (little serious adoption of Perl 6)
- COMP(2041|9044) will cover subset of Perl 5

Summary: it’s easy to write concise, powerful, obscure programs in Perl
Books


- Christiansen & Torkington, Perl Cookbook (2ed), O’Reilly, 2003. (Lots and lots of interesting Perl examples)

- Schwartz & Phoenix, Learning Perl Objects, References, and Modules (2ed), O’Reilly, 2003. (gentle & careful introduction to parts of Perl mostly not covered in this course)

- Schwartz, Phoenix & Foy, Intermediate Perl (2ed), O’Reilly, 2012. (good book to read after 2041 - starts where this course finishes)

- Orwant, Hietaniemi, MacDonald, Mastering Algorithms with Perl, O’Reilly, 1999. (Algorithms and data structures via Perl)
Perl programs can be invoked in several ways ...

- giving the filename of the Perl program as a command line argument:
  
  ```
  $ perl code.pl
  ```

- giving the Perl program itself as a command line argument:
  
  ```
  $ perl -e 'print "Hello, world\n";' 
  ```

- using the `#!` notation and making the program file executable:
  
  ```
  $ chmod 755 code.pl
  $ ./code.pl
  ```
Running Perl

Always use `-w` option when running Perl.

It causes Perl to print warnings about common errors.

```
$ perl -w code.pl
```

Can use options with `#!`

```
#!/usr/bin/perl -w
...
```

You can also get warnings via a pragma:

```
use warnings;
```

You can catch more possible problems with the `strict` pragma:

```
use strict;
```

Some coders find the `strict` pragma too annoying, some projects insist on it. The `strict` pragma is not required in COMP(2041|9044)
Perl uses non-alphabetic characters to introduce various kinds of program entities (i.e. set a context in which to interpret identifiers).

<table>
<thead>
<tr>
<th>Char</th>
<th>Kind</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Comment</td>
<td># comment</td>
<td>rest of line is a comment</td>
</tr>
<tr>
<td>$</td>
<td>Scalar</td>
<td>$count</td>
<td>variable containing simple value</td>
</tr>
<tr>
<td>@</td>
<td>Array</td>
<td>@counts</td>
<td>list of values, indexed by integers</td>
</tr>
<tr>
<td>%</td>
<td>Hash</td>
<td>%marks</td>
<td>set of values, indexed by strings</td>
</tr>
<tr>
<td>&amp;</td>
<td>Subroutine</td>
<td>&amp;doIt</td>
<td>callable Perl code (&amp; optional)</td>
</tr>
</tbody>
</table>
Any unadorned identifiers are either

- names of built in (or other) functions ~ (e.g. chomp, split)
- control-structures ~ (e.g. if, for, foreach)
- literal strings (like the shell!)

The latter can be confusing to C/Java/PHP programmers e.g.

```plaintext
$x = abc;  is the same as  $x = "abc";
```
Perl provides these basic kinds of variable:

- **scalars** ... a single atomic value (number or string)
- **arrays** ... a list of values, indexed by number
- **hashes** ... a group of values, indexed by string

Variables do not need to be declared or initialised. If not initialised, a scalar is the empty string (0 in a numeric context).

*Beware*: spelling mistakes in variable names, e.g:

```perl
print "abc=$acb\n";  # rather than  print "abc=$abc\n";
```

Use warnings (-w) and easy to spell variable names.
Many scalar operations have a "default source/target". If you don’t specify an argument, variable $_ is assumed. This makes it

- often very convenient to write brief programs (minimal syntax)
- sometimes confusing to new users ("Where’s the argument??")

$_ performs a similar role to "it" in English text.

E.g. “The dog ran away. It ate a bone. It had lots of fun.”
Perl arithmetic and logical operators are similar to C.

Numeric:  ==  !=  <  <=  >  >=  <=>

String:  eq  ne  lt  le  gt  ge  cmp

Most C operators are present and have similar meanings, e.g:
+  -  *  /  %  ++  – +=

Perl string concatenation operator:  .
equivalent to using C’s malloc + strcat

C strcmp equivalent to Perl cmp
Examples:

$x = '123'; # $x assigned string "123"
$y = "123 "; # $y assigned string "123 ">
$z = 123; # $z assigned integer 123
$i = $x + 1; # $x value converted to integer
$j = $y + $z; # $y value converted to integer
$a = $x == $y; # numeric compare $x,$y (true)
$b = $x eq $y; # string compare $x,$y (false)
$c = $x.$y; # concat $x,$y (explicit)
$c = "$x$y"; # concat $x,$y (interpolation)

- Note: $c = $x $y is invalid (Perl has no empty infix operator)
  - unlike awk or shell
Perl Truth Values

False: " and ‘0’

True: everything else.

Be careful, subtle consequences:

False: 0.0, 0x0

True: ‘0.0’ and "0\n"
A very common pattern for modifying scalars is:

```
$var = $var op expression
```

**Compound assignments** for the most common operators allow you to write

```
$var op= expression
```

Examples:

```
$x += 1;    # increment the value of $x
$y *= 2;    # double the value of $y
$a .= "abc" # append "abc" to $a
```
Perl has two sets of logical operators, one like C, the other like

The second set has very low precedence, so can be used between statements.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>x &amp;&amp; y</td>
<td>false if x is false, otherwise y</td>
</tr>
<tr>
<td>Or</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Not</td>
<td>! x</td>
<td>true if x is not true, false otherwise</td>
</tr>
<tr>
<td>And</td>
<td>x and y</td>
<td>false if x is false, otherwise y</td>
</tr>
<tr>
<td>Or</td>
<td>x or y</td>
<td>true if x is true, otherwise y</td>
</tr>
<tr>
<td>Not</td>
<td>not x</td>
<td>true if x is not true, false otherwise</td>
</tr>
</tbody>
</table>
Logical Operators

The lower precedence of or/and enables common Perl idioms.

```perl
if (!open FILE, '<', "a.txt") {
  die "Can't open a.txt: $!";
}
```

is often replaced by Perl idiom

```perl
open FILE, '<', "a" or die "Can't open a: $!";
```

Note this doesn’t work:

```perl
open FILE, '<', "a" || die "Can't open a: $!";
```

because its equivalent to:

```perl
open FILE, '<', ("a" || die "Can't open a: $!");
```
Stream Handles

Input & output are accessed via **handles** - similar to FILE * in C.

```
$line = <IN>; # read next line from stream IN
```

Output file handles can be used as the first argument to **print**:

```
print OUT "Andrew\n"; # write line to stream OUT
```

Note: no comma after the handle

Predefined handles for stdin, stdout, stderr

```
# STDOUT is default for print so can be omitted
print STDOUT "Enter your a number: ";
$number = <STDIN>;
if (number < 0) {
    print STDERR "bad number\n";
}
```
Handles can be explicitly attached to files via the `open` command:

```ruby
open DATA, '<', 'data';  # read from file data
open RES, '>', 'results'; # write to file results
open XTRA, '>>', 'stuff'; # append to file stuff
```

Handles can even be attached to pipelines to read/write to Unix commands:

```ruby
open DATE, "date\"";  # read output of date command
open FEED, "|more";  # send output through "more"
```

Opening a file may fail - always check:

```ruby
open DATA, '<', 'data' or die "Can't open data: $!";
```
open OUT, '>', 'a.txt' or die "Can't open a.txt: $!";
print OUT "42\n";
close OUT;
open IN, '<', 'a.txt' or die "Can't open a.txt: $!";
$answer = <IN>;
close IN;
print "$answer\n"; # prints 42
Anonymous File Handles

If you supply a uninitialized variable Perl will store an anonymous file handle in it:

```perl
open my $output, '>', 'answer' or die "Can't open ...";
print $output "42\n";
close $output;
open my $input, '<', 'answer' or die "Can't open ...";
$answer = <$input>;
close $input;
print "$answer\n";  # prints 42
```

Use this approach for larger programs to avoid collision between file handle names.
Handles can be explicitly closed with `closeHandleName`

- All handles closed on exit.
- Handle also closed if open done with same name
good for lazy coders.
- Data on output streams may be not written (buffered)until close - hence close ASAP.

Calling `<>` without a file handle gets unix-filter behaviour.

- treats all command-line arguments as file names
- opens and reads from each of them in turn
- no command line arguments, then `~ <> == <STDIN>`
- so this is `cat` in Perl:

```perl
#!/usr/bin/perl
while ($line = <>) {
    print $line;
}
```
Displays the contents of the files a, b, c on stdout.
All single Perl statements must be terminated by a semicolon, e.g.

```perl
$x = 1;
print "Hello";
```

All statements with control structures must be enclosed in braces, e.g.

```perl
if ($x > 9999) {
    print "x is big\n";
}
```

You don’t need a semicolon after a statement group in `{...}`.

Statement blocks can also be used like anonymous functions.
Function Calls

All Perl function calls . . .

- are call by value ~ like C (except scalars aliased to @
- are expressions ~ (although often ignore return value)

Notation(s) for Perl function calls:

```
&func(arg{1}, arg{2}, ... arg{n})
func(arg{1}, arg{2}, ... arg{n})
func arg{1}, arg{2}, ... arg{n}
```
Selection is handled by ~ if ... elsif ... else

```plaintext
if ( boolExpr{1} ) { statements{1} }
elsif ( boolExpr{2} ) { statements{2} }
...
else { statements{n} }

statement if ( expression );
```
Control Structures

Iteration is handled by ~ while, until, for, foreach

```java
while ( boolExpr ) {
    statements
}

until ( boolExpr ) {
    statements
}

for ( init ; boolExpr ; step ) {
    statements
}

foreach var (list) {
    statements
}
```
Control Structures

Example (compute $pow = k^n$):

```
# Method 1 ... while
$pow = $i = 1;
while ($i <= $n) {
    $pow = $pow * $k;
    $i++;
}
```

```
# Method 2 ... for
$pow = 1;
for ($i = 1; $i <= $n; $i++) {
    $pow *= $k;
}
```

```
# Method 3 ... foreach
$pow = 1;
foreach $i (1..$n) {
    $pow *= $k;
}
```

```
# Method 4 ... builtin operator
$pow = $k * $n;
```
Example (compute $pow = k^n$):

```bash
# Method 1 ... while
$pow = $i = 1;
while ($i <= $n) {
    $pow = $pow * $k;
    $i++;
}
```

```bash
# Method 2 ... for
$pow = 1;
for ($i = 1; $i <= $n; $i++) {
    $pow *= $k;
}
```

```bash
# Method 3 ... foreach
$pow = 1;
foreach $i (1..$n) { $pow *= $k; }
```

```bash
# Method 4 ... foreach $_
$pow = 1;
foreach (1..$n) { $pow *= $k; }
```

```bash
# Method 5 ... builtin operator
$pow = $k * $n;
```
Terminating

Normal termination, call: `exit 0`

The `die` function is used for abnormal termination:

- accepts a list of arguments
- concatenates them all into a single string
- appends file name and line number
- prints this string
- and then terminates the Perl interpreter

Example:

```perl
if (! -r "myFile") {
    die "Can't read myFile: $!

} # or
# or
die "Can't read myFile: $!
if ! -r "myFile";
# or
-r "myFile" or die "Can't read myFile: $!
```
Perl and External Commands

Perl is shell-like in the ease of invoking other commands/programs.

Several ways of interacting with external commands/programs:

- `'cmd'`; capture entire output of `cmd` as single string
- `system "cmd"` execute `cmd` and capture its exit status only
- `open F,"cmd |"` collect `cmd` output by reading from a stream
External command examples:

```perl
$files = `ls $d`;  # output captured

$exit_status = system "ls $d";  # output to stdout

open my $files, '-|', "ls $d";  # output to stream
while (<$files>) {
    chomp;
    @fields = split;  # split words in $_ to @_
    print "Next file is $fields[$#fields]\n";
}
```
Perl provides an extensive set of operators to query file information:

- `r` - `w` - `x`  file is readable, writeable, executable
- `e` - `z` - `s`  file exists, has zero size, has non-zero size
- `f` - `d` - `l`  file is a plain file, directory, sym link

- inspired by the shell `test` command.

Makes file operations concise, e.g.:

```
-d $directory or system "mkdir $directory";
```
Perl defines numerous special variables to hold information about its execution environment.

These variables typically have names consisting of a single punctuation character e.g. $!, $@, $#, $$, $% ... (English names are also available)

The $_ variable is particularly important:

- acts as the default location to assign result values (e.g. <STDIN>)
- acts as the default argument to many operations (e.g. print)

Careful use of $_ can make programs concise, uncluttered. Careless use of $_ can make programs cryptic.
### Special Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$_</code></td>
<td>default input and pattern match</td>
</tr>
<tr>
<td><code>@ARGV</code></td>
<td>list (array) of command line arguments</td>
</tr>
<tr>
<td><code>$0</code></td>
<td>name of file containing executing Perl script (cf. shell)</td>
</tr>
<tr>
<td><code>$i</code></td>
<td>matching string for $i^{th}$ regexp in pattern</td>
</tr>
<tr>
<td><code>$!</code></td>
<td>last error from system call such as open</td>
</tr>
<tr>
<td><code>$.</code></td>
<td>line number for input file stream</td>
</tr>
<tr>
<td><code>$/</code></td>
<td>line separator, none if undefined</td>
</tr>
<tr>
<td><code>$$</code></td>
<td>process number of executing Perl script (cf. shell)</td>
</tr>
<tr>
<td><code>%ENV</code></td>
<td>lookup table of environment variables</td>
</tr>
</tbody>
</table>
Example (echo in Perl):

```perl
for ($i = 0; $i < @ARGV; $i++) {
    print "ARGV[$i] ";
}
print "\n";
```

or

```perl
foreach $arg (@ARGV) {
    print "$arg ";
}
print "\n";
```

or

```perl
print "ARGV\n";
```
Simple I/O example

# compute Pythagoras' Theorem
print "Enter x: ";
$x = <STDIN>; 
chomp $x;
print "Enter y: ";
$y = <STDIN>; 
chomp $y;
$pythagoras = sqrt $x * $x + $y * $y;
print "Square root of $x squared + $y squared is $pythagoras\n";

source code for pythagoras.pl
# Read numbers until end of input (or a non-number) is reached
# then print the sum of the numbers
$sum = 0;
while ($line = <STDIN>) {
    $line =~ s/^\s*//;  # remove leading white space
    $line =~ s/\s*$//;   # remove leading trailing white space
    # Test if string looks like an integer or real (scientific notation not handled!)
    if ($line !~ /^\d[.\d]*$/) {
        last;
    }
    $sum += $line;
}
print "Sum of the numbers is $sum\n";

source code for sum_stdin.pl
printf "Enter some input: ";
$line = <STDIN>;
if (!defined $line) {
    die "$0: could not read any characters\n";
}
chomp $line;
$n_chars = length $line;
print "That line contained $n_chars characters\n";
if ($n_chars > 0) {
    $first_char = substr($line, 0, 1);
    $last_char = substr($line, $n_chars - 1, 1);
    print "The first character was '$first_char'\n";
    print "The last character was '$last_char'\n";
}

source code for line_chars.pl
# Reads lines of input until end-of-input
# Print snap! if two consecutive lines are identical

print "Enter line: ";

$last_line = <STDIN>;
print "Enter line: ";
while ($line = <STDIN>) {
    if ($line eq $last_line) {
        print "Snap!\n";
    }
    $last_line = $line;
    print "Enter line: ";
}

source code for snap_consecutive.pl
# create a string of size $2^n$ by concatenation

die "Usage: $0 <n>
" if @ARGV != 1;

$n = 0;
$string = '@';

while ($n < $ARGV[0]) {
    $string = "$string$string";
    $n++;
}

,size = length $string;

printf "String of $2^%d = %d characters created\n", $n, $size;

source code for exponential_concatenation.pl