Perl

Perl = **P**ractical **E**xtraction and **R**eport **L**anguage
Developed by Larry Wall (late 80's) as a replacement for `awk`.
Has grown to become a replacement for `awk`, `sed`, `grep`, other filters, shell scripts, C programs, ... (i.e. "kitchen sink").
An extremely useful tool to know because it:

- runs on Unix variants (Linux/Android/OSX//..), Windows variants, IOS, Plan 9, OS2, OS390, VMS..
- very widely used for complex scripting tasks
- has standard libraries for many applications (Web, DB, ...)

Perl has been influential: PHP, Python, Ruby,

Perl

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.

Perl

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 very different language (unlike Python 2 versus Python 3)
- Perl 6 development started 2000, v1.0 released end 2015 (
- little serious adoption of Perl 6
- COMP[29]041 will cover subset of Perl 5

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

Reference Material

- *Wall, Christiansen & Orwant* , Programming Perl (4ed), O'Reilly, 2012. (Original & best Perl reference manual)
- *Schwartz, Phoenix & Foy*, Learning Perl (6ed), O'Reilly, 2011. (gentle & careful introduction to Perl)
- *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)

Running Perl

Perl programs can be invoked in several ways ...

- giving the filename of the Perl program as a command line argument:

```
perl PerlCodeFile.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 PerlCodeFile
./PerlCodeFile
```

Running Perl

Advisable to *always* use `-w` option.
Causes Perl to print warnings about common errors.

```
perl -w PerlCodeFile.pl
perl -w -e 'PerlCode'
```

Can use options with `#!`

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

PerlCode
```

you can also get warnings via a pragma:

```
use warnings;
```

To catch other possible problems. Some programmers always use strict, others find it too annoying.

```
use strict;
```

Syntax Conventions

Perl uses non-alphabetic characters to introduce various kinds of program entities (i.e. set a context in which to interpret identifiers).

Char	Kind	Example	Description
#	Comment	`# comment`	rest of line is a comment
$	Scalar	`$count`	variable containing simple value
@	Array	`@counts`	list of values, indexed by integers
%	Hash	`%marks`	set of values, indexed by *strings*
&	Subroutine	`&doIt`	callable Perl code (& optional)

Syntax Conventions

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.

```
$x = abc;   is the same as   $x = "abc";
```

Variables

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:

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

Use warnings (-w) and easy to spell variable names.

Variables

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."

Arithmetic & Logical Operators

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`

Scalars

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 $y` is invalid (Perl has no empty infix operator) (unlike predecessor languages such as awk, where `$x $y` meant string concatenation)

Perl Truth Values

False: '' and '0'
True: everything else.
Be careful, subtle consequences:
False: 0.0, 0x0
True: '0.0' and "0\n"

Scalars

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
```

Logical Operators

Perl has two sets of logical operators, one like C, the other like "English".
The second set has very low precedence, so can be used between statements.

Operation	**Example**	**Meaning**
And	`x && y`	false if `x` is false, otherwise `y`
Or	`x || y`	true if `x` is true, otherwise `y`
Not	`! x`	true if `x` is not true, false otherwise
And	`x and y`	false if `x` is false, otherwise `y`
Or	`x or y`	true if `x` is true, otherwise `y`
Not	`not x`	true if `x` is not true, false otherwise

Logical Operators

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

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

is often replaced by Perl idiom

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

Note this doesn't work:

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

because its equivalent to:

```
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";
}
```

Opening Files

Handles can be explicitly attached to files via the open command:

```
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:

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

Opening a file may fail - always check:

```
open DATA, '<', 'data' or die "Can't open data: $!";
```

Reading and Writing a File: Example

```
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:

```
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.

Close

Handles can be explitly closed with `close(`*HandleName*`)`

- 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.

<> give Unix Filter behavior

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:`

```
#!/usr/bin/perl
# Copy stdin to stdout

while ($line = <>) {
    print $line;
}
```

`Displays the contents of the files a, b, c on stdout.`

Control Structures

All single Perl statements must be terminated by a semicolon, e.g.

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

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

```
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}
```

Control Structures

Selection is handled by `if ... elsif ... else`

```
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`

```
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;
```

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

```
# 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:

```
if (! -r "myFile") {
    die "Can't read myFile: $!\n";
}
# or
die "Can't read myFile: $!\n" if ! -r "myFile";
# or
-r "myFile" or die "Can't read myFile: $!\n"
```

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

Perl and External Commands

External command examples:

```
$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";
}
```

File Test Operators

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

Cf. the Unix `test` command.
Used in checking I/O operations, e.g.

```
-r "dataFile" && open my $data, '<', "dataFile";
```

Special Variables

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

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

Special Variables

Example (`echo` in Perl):

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

or

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

or

```
print "@ARGV\n";
```