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/IOS/..), Windows/DOS variants, Plan 9, OS2, OS390, VMS..
- very widely used for complex scripting tasks
- has standard libraries for many applications (Web/CGI, DB, ...)

Perl has been influential: PHP, Python, Ruby, ... even Java (interpreted)

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 accest 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 efficient? (although still remarkably efficient)

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

Reference Material

- *Wall, Christiansen & Orwant* , Programming Perl (3ed), O'Reilly, 2000. (Original & best Perl reference manual)
- *Schwartz, Phoenix & Foy*, Learning Perl (5ed), O'Reilly, 2008. (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, 2008. (good book to read after 2041 - starts where this course finishes)
- *Sebesta*, A Little Book on Perl, Prentice Hall, 1999. (Modern, concise introduction to Perl)
- *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

Example of using statement-level logical operations:

```
if (!open(FILE,"myFile")) {
    die "Can't open myFile";
}
# or

if (!open(FILE,"myFile"))
    { die "Can't open myFile"; }


# can be replaced by Perl idiom

open(FILE,"myFile")  or  die "Can't open myFile";
```

Input/Output

Files are accessed via *handles* - similar to `FILE *` in the C stdio library.

`<HANDLE>`; for an input file means "read the next line from that file".

E.g. `$line = <HANDLE>`

... stores the next line from standard input in the variable `$line`.

Output file handles are used as the first argument to the `print` command.

E.g. `print REPORT "Report for $today\n";`

... writes a line to the file attached to the `REPORT` handle.

Note: no comma after the handle ID

Input/Output

Example (a simple cat):

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

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

However, this can be simplified to:

```
while (<STDIN>) { print; }

# or even

print <STDIN>;
```

Defaults:

- the default destination variable for input is $_
- the default argument for `print` is also $_

Input/Output

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 attched to pipelines to read/write to Unix commands:

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

Opening a handle may fail:

```
open(DATA, "<< data")  or  die "Can't open data file";
```

Handles are closed using `close(`*HandleName*`)` (or automatically closed on exit).

Input/Output

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>`

Example:

```
perl -e 'print <>;' a b c
```

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 $\{\ldots\}$. 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 ... foreach $_
$pow = 1;
foreach (1..$n) { $pow *= $k; }
# Method 5 ... builtin operator
$pow = $k ** $n;
```

Control Structures

Example of fine-grained loop control:

```
OUTER:
while (i < 1000)
{
    INNER:
    for ($i = 0; $i < 99; $i++)
    {
        last OUTER if $i > 90;    # terminates both loops
        $i += 3;
        next if $i > 80;          # next iteration of INNER
        if ($i > 70) { next; }    # next iteration of INNER
        $i += 4;
        redo if $i < 70;          # next iteration of INNER
        next OUTER if $i == 42;   # next iteration of OUTER
     }
}
```

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`;                 # string output from command

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

open(FILES, "ls $d |");           # display dir contents
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 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";
```