Perl Programming

Andrew Arensburger

http://www.cfar.umd.edu/~arensb/text/talks/perl/

Preramble

```
perl5 -e 'print "Hello, world!\n"'

#!/usr/imports/bin/perl5
print "Hello, world!\n";
```

- Perl is semi-compiled.
- Perl is case-sensitive.
- Perl is the Swiss Army Chainsaw.

- Variable types include:
  - Scalars
  - Arrays
  - Hashes
  - References

  ... and a few others.
- Variables need not be declared in advance.
- The type of a variable is determined from context.

Scalars

- A scalar is any single value: a number or a string.
- Numbers are internally represented as strings.

```
$var = "Just another Perl hacker.";
print $var;

($a, $b, $c) = (1, 2, 3);
($a, $b, $c) = ($c, $b, $a);
```

Arrays

- An array is an ordered list of scalars.
- It's an array, it's a linked list, it's a deque.
- No need to declare an array's size because:
- Arrays grow and shrink dynamically.
- Indices begin at 0.

```
@array = @otherarray;
@mixed = ( "arensb", 2072, 10 );
@empty = ();

$foo              = $array[9];      # Resize array
$array[3]         = $bar;           # Assign to element
@other            = @array[2,0,1];  # Get some elements
@baz              = @array[4..8];   # Get a range
($a, $b, @rest) = @array;           # Array assignment
$foo              = ("a", "b", "c")[1];

$#foo             = 35;             # Resize array
```

- The deque functions:

`push @array, $value`		array	$\Leftarrow$	value
`$value = pop @array`		array	$\Rightarrow$	value
`$value = shift @array`	value $\Leftarrow$	array		
`unshift @array, $value`	value $\Rightarrow$	array		

- Getting the size of an array:

```
$size = $#array + 1;
$size = @array;
$size = scalar(@array);
```

Hashes

- A hash table is an associative array.
- It's a look-up table.
- It's an unordered set.

```
%user2group = (
        arensb  => 10,
        root    => 0,
        bin     => 3,
);
$user2group{"arnie"} = 199;
%empty = ()

$group  = $user2group{"arensb"};
@groups = @user2group{"root", "bin"};
```

- `keys %hash` returns a list of the keys.
- `values %hash` returns a list of the values.
- `each %hash` returns each entry in turn.
- `defined` and `each` tell you if a hash entry exists.

```
defined($user2group{"arnie"})
exists($user2group{"arnie"})

delete($user2group{"arnie"});
```

Barewords

- Avoid them.

Numbers

```
  123
 0456           # Octal
0x8a0           # Hex
0.998
 .998
9.98e-1
1_234_567_890
```

Contexts

- Scalar (expects a scalar value).
- List (expects an array value).

... and a few others.

```
@array = ("a", "b", "c");
 $foo  = @array;    # Set $foo to size of @array
($bar) = @array;    # Set $bar to first element
                    # of @array

$time = localtime;  # Returns "Sat Jun 13 14:19:35 1998"
@time = localtime;  # Returns (35 19 14 13 5 98 6 163 1)
```

- Use `scalar` to force scalar context.
- `wantarray` will tell you if you're in a list context.

Special Variables, Part 1

`$ARG`	`$_`	The Universal Default.
	`$?`	Child error status.
`$ERRNO`	`$!`	Error returned by last system call.
`$PID`	`$$`	Process ID.
`$UID`	`$<`	Real UID.
`$EUID`	`$>`	Effective UID.
`$GID`	`$(`	Real GID.
`$EGID`	`$)`	Effective GID.
	`@ARGV`	Command-line arguments.
	`$0`	Name of the script.
	`%ENV`	Environment.
	`%SIG`	Signal handlers.
	`@INC`	Include directories.
	`%INC`	Included files.
	`$]`	Perl version.
	`$^O`	OS name.
	`$^X`	Perl executable.
	`$^T`	Start time.
	`$^W`	Is `-w` on?

- `use English;` if you don't like line noise.
- Those are carets, not control characters (phew!)
- See mnemonics in perlvar(1).

Quoting

’abc’ q{abc}	Literal quotes. ’It\’s \\quot"ing’ ⇒ It’s \quot"ing
"ab$c" qq{ab$c}	Interpolates variables and backticks. "Say $greeting" ⇒ Say hello
‘abc‘ qx{abc}	Runs a command, returns what it printed. \`date\` ⇒ Tue Jun 16 18:37:11 EDT 1998
qw{a b c}	Literal array. qw/foo bar baz/ = (’foo’, ’bar’, ’baz’)

- Interpolated arrays use `$"` (`$LIST_SEPARATOR`).
- “Here” documents:

```
# Wrong
print <<end
Just another Perl hacker.
end;
```

```
# Right
print <<EOT;
Just another Perl hacker.
EOT
```

- Quoting the end-of-text word gives you those kinds of quotes.
- You can have multiple “here” documents on a line.

Math Operators

- `+, -, *, /` — The usual arithmetic operators.
- `%` — Modulus.
- `**` — Exponentiation.
- `&, |, ^, ~` — Bit-wise and, or and exclusive-or, and not.
- `<<, >>` — Left and right bit-shift.
- `+=, -=`, etc. — *operator*-and-assign. Works just as in C, with all of the operators above.
- `++, --` — Pre- and post-increment and -decrement. `++` is magical.
- `<, <=, ==, >=, >, !=` — Numeric comparison. Non-numeric strings are equal to 0.
- `<=>` — Generalized numeric comparison operator. `$a <=> $b` returns -1, 0 or 1, depending on whether `$a` is less than, equal to, or greater than `$b`.

String Operators

. (dot)	Concatenate two strings.
`.=`	Concatenate and assign.
`x`	String multiplier: `"ab" x 3` ⇒ `"ababab"`
`x=`	String multiply and assign.
`lt`, `le`, `eq`, `ge`, `gt`, `ne`	String comparison operators.
`cmp`	Generalized string comparison operator. `$a cmp $b` returns -1, 0 or 1, depending on whether `$a` is less than, equal to, or greater than `$b`.

File Operators

- *-X FILEHANDLE* or *-X filename*.
- Use _ to reuse `stat()` results.

`-r, -w, -x, -o`	File is readable, writable, executable, owned by euid.
`-R, -W, -X, -O`	File is readable, writable, executable, owned by euid.
`-e`	File exists.
`-z`	File has zero size.
`-s`	Size of file (hence functions as a "file has non-zero size" test).
`-M, -A, -C`	File's mtime, atime, ctime.
`-f`	File is a plain file.
`-d`	File is a directory.
`-l`	File is a symbolic link.
`-p`	File is a named pipe.
`-S`	File is a socket.
`-b`	File is a block special file.
`-c`	File is a character special file.
`-t`	File is a filehandle opened to a terminal (`isatty()`).
`-u`	File has the setuid bit set.
`-g`	File has the setgid bit set.
`-k`	File has the sticky bit set.

Other Operators

`=~`, `!~`	Match on something other than `$_`. `!~` negates the result.
`&&`, `||`	Logical and, or.
`!`	Not.
`&`, `|`, `^`, `~`	Bitwise and, or, xor; one's complement.
`?:`	Inline if-then-else.
`..` (dot dot)	Range operator, or flip-flop.
`and`, `or`, `xor`, `not`	Low-precedence `&&`, `||`, `^`, `!`

- Don't memorize the precedence hierarchy. Just use parens.

`if` and `unless`

`if` (*condition*) `{`
  commands
`}`
[`elsif` (*condition*) `{`
  commands
`}`...]
[`else {`
  commands
`}`]

- Parentheses are mandatory (except in postfix version).

  command `if` *condition*;
  command `unless` *condition*;

```
if ($foo eq "abc") {
        print "Foo is abc\n";
}

print "foo is abc\n" if $foo eq "abc";
```

What is Truth?

- Empty arrays and hashes are false.
- Undefined variables are false.
- 0 is false.
- "" (empty string) and `undef` are false.
- Anything that isn't false is true.

- To undefine a variable:

```
undef $var;
$var = undef;
```

`while` and `until`

```
while (condition) {
        commands
}
[continue {
        commands
} ]
```

command `while` *condition*;
command `until` *condition*;

- Note: `do` *BLOCK* `until` *condition* will run at least once.

`for` and the other `for`

`for` (*init*; *condition*; *continue*) `{`
    commands
`}`

`for` [*var*] (*list*) `{`
    commands
`}`

- `foreach` is a synonym for `for`.

`do` BLOCK

- executes BLOCK and returns the value of the last statement

```
open INFILE, "/my/file" or do {
        print STDERR "I don't know what to do!\n";
        exit 1;
    }
```

- A bare block is a loop that runs once.

Loop Control

- `last`   Break out of a loop.
- `next`   Go on to the next iteration.
- `redo`   Go back to the top.

- Use block labels to break out of nested loops:

```
OUTER: for ($i = 0; $i < 100; $i++)
{
    INNER: for ($j = 0; $j < 10; $j++)
    {
        if (&an_error_has_occurred)
        {
            last OUTER;
        }
    }
}
```

File I/O

The canonical file read loop:

```
open INFILE, "/my/file";
while ($line = <INFILE>)
{
        # Do something
}
close INFILE;
```

- Open for reading: `open IN, "</my/file";`
- Open for writing: `open IN, ">/my/file";`
- Open for appending: `open IN, ">>/my/file";`
- Open pipe for reading: `open IN, "command|";`
- Open pipe for writing: `open IN, "|command";`

- `STDIN`, `STDOUT` and `STDERR` come pre-opened.
- `print` *[FILEHANDLE] [expr]*`;`
- Note: no comma after *FILEHANDLE*.
- Remove trailing `\n` with `chop` or `chomp`.

Brackety Magic

- <*FILEHANDLE*> reads the next line (in a scalar context).
- <*FILEHANDLE*> reads whole file (in a list context).
- `while (<FILEHANDLE>)` puts the line in `$_`.
- `<>` reads all of the files in `@ARGV`, or `-` (dash) if there are none.
- `-` is magic: it's either `STDIN` or `STDOUT`.
- Yes, you can set `@ARGV` to mess with `<>`'s mind.
- <*glob*> expands filenames.

```
while (</tmp/*>) {
        print "Just another temp file: $_\n";
}
```

- `eof(`*FILEHANDLE*`)`, `eof` and `eof()`.
- `perl5 -ne` *script* yields

```
while (<>) {
        script
}
```

- `perl5 -pe` *script* yields

```
while (<>) {
        script
} continue { print }
```

- `perl -i`*extension* edits in-place.

Formats

```
format NEWHOST =
Name:   @>>>>>>>>>  IP: @<<<<<<<<<<<<<<< Ether: @||||||||||||||||
        $hostname,      $ip,             $ether
Domain:    @>>>>>>          CPU Tag: @<<<<<<<<<<<<<<<<<<<<<
        $domain,        $cpu_tag,
                        Monitor Tag: @<<<<<<<<<<<<<<<<<<<<<
                        $mon_tag
Problems with the installation:
    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $problems
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $problems
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        $problems
~   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<...
        $problems
# Should there be more than four lines for problems?

Comments:
@*
$comments
.

$~ = NEWHOST;                   # Set the format

$hostname = "glitnir";          # Initialize the format
$domain   = "CfAR";
$ip       = "128.8.132.40";
...
write;                          # Write a record
```

- Use `@#####` or `@####.##` for numbers.
- Each filehandle has a default format, with the same name as the filehandle.
- Each format has an associated top-of-page format, *FORMAT*`_TOP` by default.

I/O Special Variables

<table>
<tr><td></td><td>$_</td><td>Default reading/writing space.</td></tr>
<tr><td></td><td>$ARGV</td><td>Name of the current file when reading with <>.</td></tr>
<tr><td>$RS</td><td>$/</td><td>Input record separator (magical).</td></tr>
<tr><td></td><td>$|</td><td>If nonzero, flush after every write.</td></tr>
<tr><td>$OFS</td><td>$,</td><td>Output field separator.</td></tr>
<tr><td></td><td>$"</td><td>Separator for arrays interpolated into strings.</td></tr>
<tr><td>$NR</td><td>$.</td><td>Input line number.</td></tr>
<tr><td></td><td>$=</td><td>Number of lines per page.</td></tr>
<tr><td></td><td>$-</td><td>Number of lines left on the page.</td></tr>
<tr><td></td><td>$%</td><td>Output page number.</td></tr>
<tr><td></td><td>$~</td><td>Name of current format.</td></tr>
<tr><td></td><td>$^</td><td>Name of top-of-page format.</td></tr>
<tr><td></td><td>$:</td><td>Line-break characters for formats.</td></tr>
<tr><td></td><td>$^L</td><td>How to output a formfeed.</td></tr>
<tr><td></td><td>$^A</td><td>write accumulator.</td></tr>
<tr><td></td><td>$^I</td><td>In-place editing extension.</td></tr>
</table>

Regular Expressions

- A regular expression is a pattern of characters.
- An ordinary character matches itself.
- Variables are interpolated into patterns.
- `m/`*pattern*`/` returns true if `$_` matches *pattern*.

(*pattern*)	Group subpatterns, and set backreferences.
`^`	Matches the beginning of a line.
`$`	Matches the end of a line.
`.` (dot)	Matches any character except newline.
pat1`|`*pat2*	Matches either *pat1* or *pat2*.
[*range*]	Matches any character in *range*.
`\`	Escapes the next character.
pattern`*`	Matches zero or more occurrences of *pattern*.
pattern+	Matches one or more occurrences of *pattern*.
pattern?	Matches zero or one occurrences of *pattern*.
pattern{*n*}	Matches exactly *n* occurrences of *pattern*.
pattern{*n*,}	Matches at least *n* occurrences of *pattern*.
pattern{*n*,*m*}	Matches at least *n* to *m* occurrences of *pattern*.

Regular Expressions (cont.)

`*?`, `+?`, `??`	Non-greedy quantifiers.
`(?#`*text*`)`	A comment. *text* is ignored.
`(?:`*pattern*`)`	Groups, but doesn't make a backreference.
`(?=`*pattern*`)`	Zero-width, positive lookahead assertion.
`(?!`*pattern*`)`	Zero-width, negative lookahead assertion.
`\w`, `\W`	Matches a word/non-word character.
`\s`, `\S`	Matches a whitespace/non-whitespace character.
`\d`, `\D`	Matches a digit/non-digit.
`\b`, `\B`	Matches a word break/non-word break.
`\t`, `\n`, `\r`	Tab, newline, carriage return.
`\f`, `\a`, `\e`	Form feed, bell, escape.
`\`*n*	Matches the *n*th parenthesized expression.
`\0`*nnn*	Matches the octal character *nnn*.
`\x`*nn*	Matches the hex character *nn*.
`\u`, `\l`	Convert the next character to upper/lower case (`ucfirst`, `lcfirst`).
`\U`, `\L`	Convert to upper/lower case until `\E` (`uc`, `lc`).
`\Q`	Escape (quote) metacharacters until `\E` (`quotemeta`).

Extended Regexps

- The `/x` option helps make regexps more readable.
- Non-escaped whitespace is ignored.
- `#` introduces a comment.

```
m/^                                    # Anchor beginning
 # Start with the day
 (mon|tue|wed|thu|fri|sat|sun)         # Day of the week
 \.?                                   # Optional dot
 ,\s+                                  # Comma, whitespace
 # Now try to match the date
 ((jan|mar|may|jul|aug|oct|dec)        # The 31-day months
        \s+                            # Whitespace
        (0?[1-9]        |              # 1-9 (01 also allowed)
         [12][0-9]      |              # 10-29
         3[01]                         # 30 and 31
        )
  | (apr|jun|sep|nov)                  # The 30-day months
        \s+
        (0?[1-9]        |              # 1-9 (01 also allowed)
         [12][0-9]      |              # 10-29
         30                            # 30
        )
  | feb                                # February: we don't
                                       # allow leap years
        \s+
        (0?[1-9]        |              # 1-9 (01 also allowed)
         1[0-9]         |              # 10-19
         2[0-8]                        # 20-28
        )
 )
 # Finally, get the year
 (19)?\d{2}
$                                      # Anchor end
/xi
```

Regexp Special Variables

	`$_`	The default variable to match.
	`$`*nn*	Matches the *nn*th parenthesized expression.
`$MATCH`	`$&`	String that matched the last pattern.
`$PREMATCH`	`` $` ``	Text preceding that which matched the last pattern.
`$POSTMATCH`	`$'`	Text following that which matched the last pattern.
`$LAST_PAREN_MATCH`	`$+`	Text of the last parenthesized expression that matched something.

`m//`

`m/`*pattern*`/`*[options]*

- `m/`*pattern*`/` returns true if `$_` matches *pattern*.
- `$var =~ m/`*pattern*`/` returns true if `$var` matches *pattern*.
- `$var !~ m/`*pattern*`/` returns true if `$var` does not match *pattern*.
- You can use any delimiter you like, instead of slashes.
- If you do use slashes, you can omit the `m`.

Options:

`/i`	Case-insensitive matching.
`/g`	Global pattern match: returns all matches in the string. `\G` anchors the spot where the last `m//g` left off.
`/c`	Prevents `/g` from resetting on non-match.
`/m`	This is a multi-line pattern. `\A` and `\Z` match the very beginning and very end of the string, but `^` and `$` match the beginning and end of every line.
`/o`	Only compile the pattern once.
`/s`	Treat the string as a single line: `.` will match newlines.
`/x`	Enable extended regular expressions.

s///

s/*pattern*/*replacement*/*[options]*

- s/*pat*/*text*/ replaces the first instance of *pat* in $_ with *text*.
- $var =~ s/*pat*/*text*/ replaces the first instance of *pat* in $var with *text*.
- You can use $1, $2, $& etc. in the replacement text.
- You can use delimiters other than slashes.
- All of the options for m// work with s///.

```
$string = "ABCdef";
$string =~ s/^(...)/$1($1)/;    # Becomes "ABC(ABC)def"
```

Additional Option:

/e Evaluates *replacement* before substitution. Warning: *replacement* is not parsed at compile-time.

```
# Replace uids with usernames
s/\d+/(getpwuid($&))[0]/eg;
```

`tr///`

`tr/`*searchlist*`/`*replacelist*`/`*[options]*

- `tr/`*search*`/`*replace*`/` replaces all characters in *search* with the corresponding ones from *replace*, in `$_`.

```
$header =~ tr/A-Z/a-z/;     # Convert $header to lowercase
```

- `$var =~ tr/`*search*`/`*replace*`/`: ditto, with `$var` instead of `$_`.
- You can use delimiters other than slashes.
- `y///` is a synonym for `tr///`.

Options:

`/c`	Complement the search list.
`/d`	Delete characters not in *searchlist*.
`/s`	Squash identical characters in the output.

`split` and `join`

`split` */pattern/*, *[string*, *[limit]]*

`join` *string*, *array*

- `split` splits *string* into sections delimited by *pattern*, and returns the resulting list.

```
split /:/, "staff:*:10:arensb, arnie"
        # Returns ("staff", "*", "10", "arensb, arnie")
```

- `join` concatenates the elements of *array*, joined with *string*, and returns the result.

```
join(":", @fields);
```

- Warning: `split //` usually doesn't do what you want.

Functions

- Define a function with `sub` *name* `{` *body* `}`
- Declare a function with `sub` *name*`;` (*i.e.*, omit the body), or just by defining it before it's called.
- `my $var` declares a variable local to the enclosing block.
- Arguments are given in `@_`.
- Return value(s) with `return`.

```
sub dostuff {
        my $x = shift;          # First element of @_
        my $y;                  # Just a 'my' variable
        my @rest = @_;

        print "Hello, @rest\n";
        return $x + $y;
}

&dostuff(0, "friendly", "people");      # Explicit call
dostuff 0, "y'all!";                    # List operator call
```

- `&dostuff;` (no parens) recycles existing `@_`.
- Use `my`, not `local`.

Function Prototypes

- Prototype specifies how the function should be called.
- It's a little picture of the call.

```
sub dostuff($$@) {
        my $x = shift;          # First element of @_
        my $y = shift;          # Next element of @_
        my @rest = @_;

        print "Hello, @rest\n";
        return $x + $y;
}

dostuff 1;                      # Wrong: causes error.
dostuff 1, 2, "world";          # Okay.
dostuff 3, @somearray;          # Okay.
```

$ Indicates a scalar argument.
@ Indicates an array argument.
% Indicates a hash argument.
; Separates mandatory from optional arguments.
* Indicates a glob (usually a filehandle).

Built-In Functions

`chr` *num*	Converts ASCII value to character.
`ord` *expr*	Converts character to ASCII value.
`chmod` *nnn*, *files...*	Changes the mode of *files*.
`chown` *uuu*, *ggg*, *files...*	Changes the owner of *files*.
`die` *message*	Prints *message* and exits.
`warn` *message*	Prints *message* like `die`, but doesn't exit.
`exit` *code*	Exits the program with the given exit status.
`exec` *prog[, arg...]*	`execvp()`s *prog*.
`system` *list*	Executes a program and waits for it to exit.
`grep` *expr*, *list*	Returns the elements of *list* for which *expr* is true.
`map` *expr*, *list*	Returns the result of applying *expr* to the elements of *list*
`sort` *expr list*	Sorts the elements of *list* according to *expr*.
`splice` *array*, *offset*, *[length, list]]*	Generalized list-substitution function.
`uc` *expr*, `lc` *expr*	Converts *expr* to upper/lower case.
`ucfirst` *expr*, `lcfirst` *expr*	Converts the first character of *expr* to upper/lower case.
`int` *expr*	Returns *expr* as an integer.

Built-In Functions

`hex` *expr*	Returns the value of hex number *expr*.
`oct` *expr*	Returns the value of octal (or hex) number *expr*.
`pack` *template*, *list*	Converts a list of values to a binary string.
`unpack` *template*, *expr*	Parses a binary string into a list of values.
`length` *expr*	Returns the string length of *expr*.
`printf` *[FILEHANDLE] format*, *list*	C's `printf()`.
`sprintf` *[FILEHANDLE] format*, *list*	C's `sprintf()`.
`reverse` *list*	Reverses an array or string.
`select` *[FILEHANDLE]*	Sets or returns the default filehandle.
`substr` *expr*, *offset*, *[len]*	Returns a substring of *expr*.
`sysopen` *filehandle*, *filename*, *mode*, *[perm]*	`open(2)`.
`sysread` *filehandle*, *var*, *len*, *[offset]*	`read(2)`.
`syswrite` *filehandle*, *var*, *len*, *[offset]*	`write(2)`.
`study` *var*	Optimize pattern-matching on *var*.
`index` *str*, *substr [, pos]*	Returns the first occurrence of *substr* in *str*.

Built-In Functions

`rindex` *str*, *substr* [, *pos*]	Returns the last occurrence of *substr* in *str*.
`pos` *var*	Where did $*scalar* `=~ m//g` leave off?
`vec` *expr*, *offset*, *bits*	Manipulate bit-fields.
`getlogin`	Looks up the current username in */etc/utmp*.
`glob` *expr*	Same as <*file pattern*>.
`reset` *[expr]*	Resets certain matches, or variables beginning with *expr*.

Unix Functions

abs
accept
atan2
bind
chdir
chroot
connect
cos
crypt
exp
fcntl
fileno
flock
fork
getc
getgrent
getgrgid
getgrnam
gethostbyaddr
gethostbyname
gethostent
getnetbyaddr
getnetbyname
getnetent
getpeername
getpgrp
getppid
getpriority
getprotobyname
getprotobynumber
getprotoent
getpwent
getpwnam
getpwuid
getservbyname
getservbyport
getservent
getsockname
getsockopt
gmtime
ioctl
kill
link
listen
log
lstat
mkdir
msgctl
msgget
msgrcv
msgsnd
opendir
pipe
rand
readdir
readlink
recv
rename
rewinddir
rmdir
seek
seekdir
select
semctl
semget
semop
send
setpgrp
setpriority
setsockopt
shmctl
shmget
shmread
shmwrite
shutdown
sin
sleep
socket
socketpair
sprintf
sqrt
srand
stat
symlink
syscall
tell
telldir
time
times
truncate
umask
unlink
utime
wait
waitpid