A Crash Course in Perl5

Part 5: Data

Zeegee Software Inc.
http://www.zeegee.com/

Terms and Conditions

Road map

- Basics
  - Introduction
  - Perl syntax
  - Basic data types
  - Basic operators
- Patterns
  - Introduction
  - String matching and modifying
  - Pattern variables
- Data structures
  - LISTs and arrays
  - Context
  - Hashes
- Flow control
  - Program structures
  - Subroutines
  - References
  - Error handling
- Data
  - Input and output
  - Binary data
  - Special variables
- Object-oriented programming
  - Modules
  - Objects
  - Inheritance
  - Tying

Input and output

Filehandles

- When you open a file, you give Perl a name by which you will refer to that file in the future. This name is the **filehandle**.

```
open LOG, ">/var/log/test.log";
print LOG "Processing begun\n";
close LOG;
```

- Filehandles are (often) **ordinary text strings**, typically in ALL CAPS.
- Perl predefines the special filehandles `STDIN`, `STDOUT`, and `STDERR`.

Passing filehandles around

- When when you want to pass a filehandle into a user-defined subroutine in Perl, it is best to do so as a *typeglob reference*. Basically, that means prepending a * to the filehandle name... like this:

```
open LOG, ">/var/log/test.log";
message(\*LOG, 'ERR', $errstr);
```

- **Don't worry about what this means yet. Just do it**. You'll run into far fewer problems that way. Think of it as one more piece of bizzare Perl syntax, unique to filehandles.

open(*FILEHANDLE*, *EXPR*)

- Opens *FILEHANDLE* onto file/pipe given by the *EXPR*ession, which may be evaluate to the following...

`filename`	Open *filename* for reading
`<filename`	Open *filename* for reading
`>filename`	Open *filename* for writing, erasing existing contents
`>>filename`	Open *filename* for appending
`|command`	Open pipe for writing: run *command* so that output written to the filehandle is piped into *command*'s stdin
`command|`	Open pipe for reading: run *command* so that output to its stdout may be read from the filehandle
`-`	Open on STDIN (like C's `fdopen(0)`)
`>-`	Open on STDOUT (like C's `fdopen(1)`)

open() (cont'd)

- Putting a **+** in front of **<**, **>**, or **>>**, means that read/write access is requested... beware which form you choose!

```
open READFIRST, "+<ReadThenOverwrite.dat";
open WRITEFIRST, "+>OverwriteThenRead.dat";
```

- To "duplicate" a filehandle, use the form ***&filehandle*** in place of ***filename*** after any of the 6 **<**, **>**, or **>>** forms:

```
# Redirect STDOUT, but save it:
open USER, ">&STDOUT" or die "open: $!";
open STDOUT, ">tmp.out" or die "open: $!";
```

open() (cont'd)

- **open()** returns nonzero on success, undefined otherwise. Always check the return value... a lot can go wrong!
- On failure, check `$!` for the reason:

```
open LOG, "$file" or die "open $file: $!";
```

close(*FILEHANDLE*)

- Close file or pipe associated with *FILEHANDLE*, and reset the input-line counter (`$.`)
- Opening an already-open filehandle causes the existing file to be closed first (but leaves `$.` alone!)
- Closing a pipe waits for the process to finish, and puts execution status into the `$?` variable:

```
open CMD, "| somecommand -i -o -u";
print CMD "Command data\n";
close CMD;
$exit = ($? >> 8);            #  get exit status
die "command failed: $exit" if ($exit != 0);
```

Data / Input and output

print(*FILEHANDLE LIST*)

- Print a *LIST* of strings to the given *FILEHANDLE*
  - If *FILEHANDLE* not given, outputs to the currently-selected filehandle (default: STDOUT)
  - If *LIST* is also not given, outputs the string in `$_`

⚠ There is no comma after the *FILEHANDLE* !

```
print;
print $a, $b, $c;
print @a, $b, @c;
print STDOUT "Hi!\n";
print LOG "ERROR:", $message, "\n";
print LOG $status, ' ', @items;
```

printf(*FILEHANDLE LIST*)

- Print a formatted string to the given *FILEHANDLE* , just like in C.

- First element of the list is the format string, which uses basically the same format directives as in C.

```
printf LOG "Date: %02d/%02d/%04d\n",
      $day, $mon, $yr;
```

- Equivalent to... `print FILEHANDLE sprintf(LIST)`

a string function

select(*FILEHANDLE*)

- Select the given *FILEHANDLE* for output:
  - **write()** or **print()** without a filehandle will now use *this* filehandle
  - Variables which pertain to currently-selected filehandle will now pertain to *this* filehandle
- The previously-selected filehandle is returned

```
open LOG, ">>captains.log";
$oldfh = select LOG;                 # save old
print "Stardate $stardate:\n";       # goes to LOG
print @msgs, "\n";                   # goes to LOG
select $oldfh;                       # restore old
```

The <> operator

- In a **scalar** context, the `<>` operator reads and returns a single line from a filehandle:

```
open LS, "ls -l |" or die "open: $!";
while (defined($line = <LS>)) {
      print $line;
}
close LS;
```

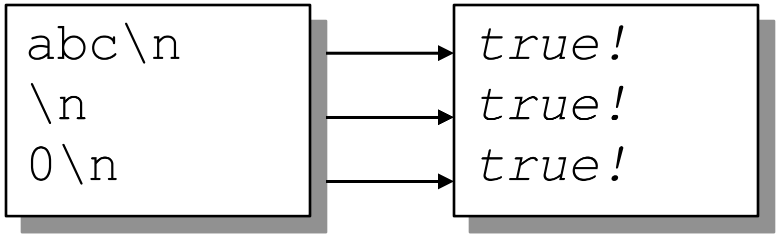
⚠ The newline character is *not* removed automatically from the end of the line!

- Returns `undef` on end of file, so loops are easy!

Data / Input and output / <>

Why test <> for defined?

- Since the newline isn't removed, it appears that all lines of a valid text file would evaluate true, and EOF is false:



- So why bother to test <> for defined()? Why not...

```
while ($line = <LS>) { ... }
```

- Answer: if your text file happens to end in a line consisting of **a single 0 and no newline**, the above loop will quit without processing that last line!

<> inside "while"

- The `<>` operator is a good friend of **while**... if a **while** test consists only of the `<>` invocation, the value is automagically put in `$_` and tested for being defined:

```
while (<LS>) { ... }
```

```
while (defined($_ = <LS>)) { ... }
```

- These are all equivalent, and pass STDIN to STDOUT:

```
while (defined($_ = <STDIN>)) { print; }
while (<STDIN>)               { print; }
for (; <STDIN>; )             { print; }
print while defined($_ = <STDIN>);
print while <STDIN>;
```

Using <> in list context

- If the `<>` operator is used in a **list** context, **a list consisting of *all* the input lines is returned**, one line per list element:

```
open UNSORTED, "unsorted.dat";
@sorted = sort <UNSORTED>;
close UNSORTED;
```

- It's easy to chew up memory this way, so use with *extreme* care!

<> and the "null filehandle"

- The null filehandle <> can be used to emulate the behavior of **sed** and **awk**, and to create standard Unix "filters":

```
while (<>) {
        #  process current line
}
```

```
@ARGV or unshift(@ARGV, '-');
while ($ARGV = shift @ARGV) {
        open(ARGV, $ARGV);
        while (<ARGV>) {
                #  process current line
        }
}
```

not *exactly* the same, but almost

Other things inside <>

- If the string inside the <> is a **scalar variable**, then that scalar contains the name of the actual filehandle to read:

```
$fh = 'STDIN';
while (<$fh>) { ... }
```

- If the string inside the <> is not a filehandle, it is interpreted as a filename pattern to be globbed. The "lines" returned are the matching filenames:

```
chmod 0644, <*.c>;
```

But use **readdir()** instead... it's more efficient and reliable

chop(*VAR*) / chomp(*VAR*)

- **chop()** chops off the last character of a string and returns the character. It was once used to remove the newline at the end of an input line.
- **chomp()** is safer: it removes the **input record separator** (usually a newline), and *only* if the string *actually ends* in that separator.
- Without arguments, both **chop** and **chomp** work on `$_`

```
while (<LS>) {
        chomp;
        print "Next line: <$_>\n";
}
```

$. ($INPUT_LINE_NUMBER)

- Current input line number of the last filehandle that was read.

```
while (<STDIN>) {
    /^\s/ and print "Leading space on line $.\n";
}
```

- Read-only
- **Mnemonic** (Larry's)**:** many Unix programs use . for the current line number.
- **Mnemonic** (mine): not the input line itself... just the line *number*, period!

$/ ($INPUT_RECORD_SEPARATOR)

- Boundary on which the <> operator will read "records"
  - Defaults to `"\n"`, so normal "records" are single lines
  - If set to special `""`, will split input stream on 2 or more consecutive blank lines (*not* the same as `"\n\n"`!)
  - If undefined, input stream is not split at all. *Use with extreme care... this can really chew up memory!*

```
undef $/;
$everything = <STDIN>;   # slurp in entire input stream!
```

- **Mnemonic:** / delimits boundaries when quoting poetry

Data / Input and output / Special variables

$| ($OUTPUT_AUTOFLUSH)

- Set to nonzero to force a flush on the currently-selected filehandle after every write()/print(). Default is 0.
- Useful when sending output to a pipe, where you don't want to have to deal with buffering.

```
open PIPE, "| program";
$oldfh = select PIPE; $| = 1; select $oldfh;
print PIPE "Send this now!";
print PIPE "Send THIS now!";
```

- **Mnemonic:** when you want your pipes to be piping hot

File test operators

- Perl provides many *sh*-like unary operators for testing files, all of the form (*-X filename*). Here are just a few...

```
(-x $file) or die;
```

`-r`	is file readable by euid/egid?
`-w`	is file writable by euid/egid?
`-x`	is file executable by euid/egid?
`-o`	is file owned by euid/egid?
`-M`	age of file in days when script began

`-e`	does file exist?	`-s`	file size
`-f`	is file a plain file?	`-d`	is file a directory?
`-l`	is file a symbolic link?	`-S`	is file a socket?

Binary data

read(*FILEHANDLE, SCALAR, LENGTH*)

- Reads LENGTH bytes from the given FILEHANDLE, and puts them into SCALAR.
- Returns actual number of bytes read, or undefined on error.

```
#  Read a stream of 54-byte records...
while (!eof(STDIN)) {
     (read(STDIN, $buf, 54) == 54) or
          die "couldn't get record!";
     #  ...current record is in $buf...
}
```

- Buffered: can be intermixed with <>.

write()

⚠ GOTCHA! Unfortunately, **write() is not the counterpart of read() you were expecting it to be**. It does something else entirely.

- To write an arbitrary number of bytes to a filehandle from a scalar, just use **print()**... with **substr()** if you need to:

```
#  Write a stream of 54-byte records...
while (1) {
        #  ...current record is in $buf; print first 54 bytes:
        print STDOUT substr($buf, 0, 54);
}
```

seek(*FILEHANDLE*, *POS*, *WHENCE*)

- Randomly positions the file pointer for FILEHANDLE, like **fseek()** in stdio. It is positioned POS bytes from WHENCE, as follows...

Integer WHENCE	POSIX WHENCE	Means to position pointer to...
0	SEEK_SET	POS bytes after start of file
1	SEEK_CUR	POS bytes after current position
2	SEEK_END	POS bytes after end of file

```
#  Read bytes 100 through 199 inclusive...
open DATA, $datafile or die "open: $!";
seek DATA, 100, 0;
read DATA, $data, 100;
```

pack(*TEMPLATE*, *LIST*)

- Kind of like **sprintf()**... takes a LIST of values and packs them into a single scalar, using the characters in TEMPLATE to determine how each value is to be packaged.

```
$s = pack('cccc', 65,66,67,68);   # "ABCD"
$s = pack('c4',   65,66,67,68);   # same
$s = pack('ccxcc',65,66,67,68);   # "AB\0CD"

$s = pack('a5', "cat");           # "cat\0\0"
$s = pack('A5', "cat");           # "cat  "
$s = pack('aa', "cat", "dog");    # "cd"

$n = pack('S',1);     # little-endian: "\1\0"
                      # big-endian:    "\0\1"
```

unpack(*TEMPLATE*, *EXPR*)

- Reverse of **pack()**... takes an EXPR evaluating to a scalar and unpacks it into a list of values, using the characters in TEMPLATE to determine how each value is to be unpacked.
- The TEMPLATE has the same format as in **pack()**.

```
@A = unpack('cccc', "ABCD");      # (65,66,67,68)
@A = unpack('c4', "ABCD");        # same
```

Special variables

$$ ($PROCESS_ID)

- The process number of the Perl running this script.

```
print "My pid = $$\n";
```

- **Mnemonic** (Larry's)**:** same as sh/csh
- **Mnemonic** (mine)**:** earning $$ is a painful process

$0 ($PROGRAM_NAME)

- The name of the file containing the Perl script being executed.
- Assigning to $0 modifies the area that the *ps* program sees.
- **Mnemonic** (Larry's): same as sh/ksh
- **Mnemonic** (mine): Oh... *that's* your name!

$] ($PERL_VERSION)

- In a **string** context, the string printed out when you say "`perl -v`".
- In a **numeric** context, returns *version + patchlevel/1000*

```
warn "No checksumming!\n" if $] < 3.019;
```

- **Mnemonic:** is this version of Perl in the right bracket?

Data / Special variables

@ARGV

- The command-line arguments intended for the script
- Equivalent to argv[1..n] in C

⚠ $ARGV[0] is *not* the program name: it's the first argument! Use $0 to get the program name.

Data / Special variables

%ENV

- Hash representing the environment.
- Access it to perform a *getenv()*:

```
$homedir = $ENV{'HOME'};
```

- Modify it to perform a *putenv()*, which will affect the environment for the current process and any child processes:

```
$ENV{'PATH'} .= ':/usr/special:/usr/games';
system("someprog -a");
```

Data / Special variables

%SIG

- Used to set signal handlers:

```
sub handler {
   die "Caught a signal: shutting down";
}

$SIG{'INT'} = 'handler';         #  old style
$SIG{'HUP'} = \&handler;         #  preferred
```

- Also used to set some internal hooks...

```
$SIG{__WARN__} = \&warning_handler;
$SIG{__DIE__}  = \&fatal_error_handler;
```