

Processes, Files, Databases

Beginning Perl Programming
Doug Treder

comics from xkcd.com

Touching the World

- Our programs so far work in a vacuum.
- They can print and get input from the user, and make their own files, but
- They're a bit like the creepy neighbor who never talks to you
- Today's material covers a lot of different areas
- but it's all about interacting with the rest of what's happening on your computer.

File Tests

- Perl has some succinct built-in functions for file testing:
- They start with dashes and expect filenames
- Most of them are simply booleans
- Some others return timestamps for access time or modified time

```
die "oops! that file $filename already exists!"  
if -e $filename;
```

File Tests

- e file exists
- r file is readable by you
- w file is writable by you
- x file is executable by you
- z file exists and is empty (zero size)
- s file exists and returns its size in bytes
- f this is a file
- d this is a directory
- l this is a symbolic link
- T looks like a text file
- B looks like a binary file
- M number of days since file was modified
- A number of days since file was accessed

File Tests

- -M and -A will return fractional days

```
my $log = "/var/log/httpd.error_log";

if (! -f $log ) {
    die "no log file $log found : is website up?";
}

elsif ( -M $log > 2 ) {
    print "Logfile received no entries in 2 days : website is
probably busted";
    my $last = time - ( (-M $log) * 24 * 60 * 60 );
    print "Last entry: "
        . scalar localtime($last) . "\n";
}
```

Running child processes

- system executes other programs on your computer
- system waits until the other program is done before returning
- system only tells you if the other program crashed
- system returns 0 for good (opposite of open!)

Using system

```
# runs date program, but nothing comes back!  
system("date");  
  
# quoted strings work fine  
system("cp $file1 $file2"); # copies file  
  
# backslashes have to be backslashed  
system("open /Applications/Aquamacs\\ Emacs.app")  
  
# system returns 0 for success (didn't crash)  
if (0 != system("open /Applications/TextEdit.app")) {  
    die "failed opening textedit";  
}
```

Using system

- System returns 0 for good, so use 'and' as shortcut

```
sub copy_file {  
    my ($from, $to) = @_;  
    !system("cp $from $to")  
        or die("Failed copying $from to $to");  
}  
  
copy_file("final.pl", "final_copy.pl");  
  
# convenient that the shell can figure out your homedir:  
copy_file("/var/log/access_log", "~/mylog.txt");
```

Using system

- system opens a shell to do command interpolation

```
sub copy_file {  
  my ($from, $to) = @_;  
  system("cp $from $to")  
  and die("Failed copying $from to $to");  
}
```

inconvenient if you're not carefully checking your input
copy_file("iluv", "you; rm -rf /"); # ouch!

WE HAD A FIGHT
LAST NIGHT.



I GUESS SHE'S
STILL MAD.



I WOKE UP TO
FIND SHE'D WRITTEN
A SAPPY LOVE NOTE



TO MY BOOT SECTOR.

OPERATING
SYSTEM
NOT FOUND



Using system

- The multi-argument form of system *doesn't* open a shell, which can be safer:

```
sub copy_file {  
    my ($from, $to) = @_;  
    system("cp", $from, $to)  
        && die("Failed copying $from to $to");  
}  
  
# with multi-arg form, a shell is not created  
# only one program is executed  
  
copy_file("haxored", "you; rm -rf /"); # can't hack me!
```

Getting the output of other programs

- backticks work like system, but return the STDOUT of the program
- backticks throw away the exit value of the program
- backticks can't tell you if the program crashes
- backticks always invoke a shell and always interpolate like double-quotes.
- backticks are dangerous!

Using backticks

```
# runs date program, and grabs its output!
```

```
my $date = `date`;
```

```
# always interpolates like a double-quoted string
```

```
`cp $file1 $file2`; # output is thrown away!
```

```
# backslashes still have to be backslashed
```

```
`open /Applications/Aquamacs\\ Emacs.app`;
```

```
# bad choice: output isn't needed,
```

```
# so you should use system('open', '/Ap...'); instead.
```

```
# Unix shells use 2>&1 to mean merge STDERR and STDOUT
```

```
`open /Applications/TextEdit.app 2>&1`;
```

Environment Variables:

- You can get access to the environment variables your program started with via %ENV
- \$ENV{PATH} : list of places to look for applications
- \$ENV{HOME} : user's home directory
- others are system-specific

opening other programs

- Neither system nor backticks gives you much input over the other program.
- Light the fuse and stand back!
- For more control, Perl allows using open to address child processes input or output
- just as if you were writing or reading a file!

opening other programs

- Use > for writing a file
- Use >> for appending a file
- Use < for reading a file
- Use | - to write to a *process*
- Use -| to read from a *process*

Writing to Child Process

- We always prefer three-argument open:

```
open my $mail_fh, "|-", "/usr/bin/mail dtreder"  
  or die "can't send mail : $!";  
  
print $mail_fh "WHASSUP!"      # Subject Line  
  or die "failed writing subject :$!";  
  
for (1..3) { # mail body  
  print $mail_fh "Buy small cap stock DTRED!\n";  
}  
  
close $mail_fh or die "failed sending mail : $!"; # mail is sent!
```

Reading from Process

- We always prefer three-argument open:

```
open my $log_fh, "-|",  
    "tail -f /var/log/httpd.access_log"  
    or die "can't read logs : $!";  
  
my $hits_from_google = 0;  
while( my $line = <$log_fh> ) {  
    if ($line =~ /google/) {  
        $hits_from_google ++;  
    }  
}  
close $log_fh or die "can't close logs : $!";
```

Inter Process Communication

Technique	IN	OUT	ERR	Success
Backticks		Returned	Terminal	Ignored
system()		Terminal	Terminal	Returned
open with -		Write To	Terminal	Check open/ close
open with -	Write to		Terminal	Check open/ close
IPC::Open2	Write to	Capture	Terminal	Check open/ close
IPC::Open3	Write to	Capture	Capture	Check open/ close

Inter Process Communication

- if you need to both read and write to the same process (feed it data and also see what it outputs)
- Or if you want separate filehandles for STDOUT and STDERR
- Check out `IPC::open2` or `IPC::open3`
- These come with Perl's standard distribution

using modules

- There's at least three ways built-in ways to load external code (also known as libraries)
- `do "filename.pl"` compiles and runs `filename.pl`
- returns whatever that code returns.
- Sometimes useful as configuration written in perl
- Useless from other languages; dangerous

Using 'do' for config files

```
config.pl:
```

```
dir => "/home/dtreder/var",  
force => 0,  
verbose => 11,  
top => 50,  
color => undef
```

```
prog.pl:
```

```
my %config = do 'config.pl';  
  
if (! %config) {  
    if ($!) {  
        die "failed reading: $!";  
    }  
    elsif ($@) {  
        die "error:  $@";  
    }  
}
```

Modules vs Libraries

- Some great libraries (modules) come with Perl
- More available via CPAN
- Load modules with 'use' and give the package name
- Modules sometimes have features or imports that you specify with the use statement in a list:

```
use strict;  
use File::Temp;  
use File::Basename qw(basename); # provides basename function  
use CGI::Carp qw(fatalsToBrowser); # turns on this feature
```

File::Spec Module

- File::Spec gives you functions to work with file paths and directories independent of OS.
- Whether your paths have \ or / or something else!
- Makes your code more portable
- Has an 'OO' interface : use File::Spec-> to call its functions.

File::Spec

```
use File::Spec;

my $cwd = File::Spec->curdir;

print "Your dir : $cwd\n";
print "Root dir : " . File::Spec->rootdir . "\n";

my $path = File::Spec->catfile($cwd, "file.txt");

my ($vol, $dir, $file) = File::Spec->splitpath($path);
print "Volume : $vol Dir: $dir File: $file\n";

print "Local path: " . File::Spec->abs2rel($path);

print "Absolute path: " . File::Spec->rel2abs($cwd);
```

File::Temp

- Gives you access to a temporary file or directory
- Handles cleaning up automatically
- Prevents your program from colliding with itself or other programs
- Opens the temporary file's filehandle for you

```
use File::Temp qw(tempfile tempdir);

my ($temp_fh, $filename) = tempfile();

# put tempfiles here: will be cleaned up on exit
my $tempdir = tempdir(CLEANUP => 1);
```

Carp

- Carp gives better error messages
- croak and cluck are die and warn, but against the caller
- confess is like die, but gives a full stack trace!
- Carp comes with Perl!

```
use Carp ('croak', 'confess');  
  
sub do_me {  
    croak "You must give me arguments!" unless @_;
```

santabot

TO COMPLETE YOUR WEB REGISTRATION, PLEASE PROVE
THAT YOU'RE HUMAN:

WHEN LITTLEFOOT'S MOTHER DIED IN THE ORIGINAL
'LAND BEFORE TIME,' DID YOU FEEL SAD?

- YES
- NO

(BOTS: NO LYING)

That's all folks!


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.



HUH?

SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.

AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.



MY GOD

IT'S FULL OF 'car's'

THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.

TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.



NO, IT'S NOT.



IT'S NOT?

I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.