Perl III

File input and output

Sofia Robb

v7 2015

Recap of UNIX I/O

STDIN - Reads in the text you type or from a file using redirection or pipes.

STDOUT - Prints to your screen, but can be redirected to a file or other program in the shell using redirection or pipes.

STDERR Standard error, used for diagnostic messages. Also prints to your screen, and also can be redirected in the shell.

Recap of UNIX I/O

The UNIX way of reading from and writing to files is redirection.

If we use output redirection on the UNIX command line, the standard output goes to a file and we see only the standard error on the screen:

```
$ ls
kubrick.txt

$ cat kubrick.txt
Barry Lyndon
The Shining

$ cat kubrick.txt fincher.txt > films.txt
ls: fincher.txt: No such file or directory
```

Perl I/O

The Perl way of reading from or writing to a file is the function open.

```
open(IN, '<', 'myfile.txt') or die "can't open
myfile.txt: $!\n";
```

open's <u>first argument</u>

open is a function, which takes 3 arguments:

```
open(IN, '<', 'myfile.txt')
```

First argument

1st argument, or the filehandle, can be any name.

Examples:
FASTA, FILE, INFASTA, OUTFASTA, TAB, TEXT

The first argument is a filehandle. Filehandles are how you refer to a file within Perl.

STDOUT and STDERR are filehandles.

When you open a file yourself, you make your own filehandle and give it a name (here, I chose IN).

Filehandles

Reading and writing to the filesystem is very complicated, involving bits, buffers, and memory.

Perl provides a 'handle' to the file and takes care of all the complicated parts for us so we can interact with a file more simply.

Filenames, filehandles, and the data in a file are three different things!!!!

Filename	the name of the file
Filehandle	a way to get access to a file's contents
File contents	the actual data inside the file

open's <u>second</u> argument

```
open(IN, '<', 'myfile.txt')
```

$\uparrow$

Second argument

The second argument is a mode. The modes are borrowed from redirection on the command line.

<	for reading from a file
>	for writing to a file
>>	for appending to a file

open's <u>third</u> argument

```
open(IN, '<', 'myfile.txt')
```

Third argument

The third argument is the name of a file to open. It can either be a literal name:

```
open(IN, '<', 'myfile.txt')
```

or a variable containing a filename:

```
open(IN, '<', $file)
```

Catch errors with `die`

If you're going to read from a file, that file must exist and be readable.

Since it rarely makes sense to continue when it's not possible to read the file, we want the program to stop. We do this with die.

```
  open(IN, '<', 'myfile.txt')
      or die "can't open myfile.txt: $!\n";
```

`open or die` is a Perl idiom. die is a function that exits the program immediately and prints the specified string to STDERR.

Capturing system errors with $!

Perl can also tell us what the filesystem said about why the file couldn't be opened.

`$!` is a special Perl variable that contains error messages from the system. If there was a problem with opening your file, there will be an error message in `$!` , and we can include it in our error string.

```
   or die "can't open $file: $!\n";
```

↑

contains error
message from
the filesystem

Open a file for writing

Open also can be used to open files for writing by using '>' as the second argument to open.

```
my $out = 'out.txt';
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

Now specify that filehandle when you print

```
print OUT "I'm writing to a file!\n";
```

and the output will go into a file instead of the screen:

```
$ perl myprog.pl          ⟵ no redirection on the command line
$ cat out.txt
I'm writing to a file!
```

Open a file for writing

If you open a file for writing and the file doesn't exist, it will be created.

```
$ ls
                          ←— look! no file there!

$ perl myprog.pl
$ ls
out.txt                   ←— out.txt has been created by myprog.pl
```

Be careful! If you open an existing file for writing, you will erase everything inside that file!

Opening multiple files

You can open more than one file in a script — just give them different filehandles.

```
my $in  = 'in.txt';
my $out = 'out.txt';
open(IN,  '<', $in ) or die "can't open $in: $!\n";
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

Open files from user input

Instead of hardcoding filenames inside your program, you can read them in from the command line:

```
my $in  = shift;
my $out = shift;
open(IN,  '<', $in ) or die "can't open $in: $!\n";
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

On the command line, you'd type this:

```
$ perl test.pl myinfile.txt myoutfile.txt
```

Open files from user input

Command line

```
$ perl test.pl myinfile.txt myoutfile.txt
```

Inside our Perl program

```
my $in  = shift;
my $out = shift;
open(IN,  '<', $in ) or die "can't open $in: $!\n";
open(OUT, '>', $out) or die "can't open $out: $!\n";
```

Which are the filehandles and which are the filenames?

`myinfile.txt` and `myoutfile.txt` are filenames.
`IN` and `OUT` are filehandles.
`$in` and `$out` are variables containing the filenames.

<> to get contents out of a file

Perl reads files one line at a time.

To read a line from a file, you put the filehandle inside <>, like this:

```
my $in  = 'in.txt';
open(IN, '<', $in ) or die "can't open $in: $!\n";


print "This is the first line from the file $in:
\n";
my $line = <IN>;
print $line;
```

<> to get contents out of a file

This code reads the first two lines from a file:

```
my $in  = "in.txt";
open(IN, '<', $in ) or die "can't open $in: $!\n";

print "This is the first line from the file $in:\n";
my $line = <IN>;
print $line;
print "This is the 2nd line from the file $in:\n";
$line = <IN>;
print $line;
```

<> to get contents out of a file

Most files have lots of lines, and we often want to read all the lines in a file one by one. We can do that using a while loop.

To read from a filehandle line by line, put

```
my $line = <IN>
```

into a while loop, like this:

```
my $in  = shift;
open(IN, '<', $in) or die "can't open $in: $!\n";

while (my $line = <IN>) {
    chomp $line;
    print "This line is from the file $in:\n";
    print $line\n";
}
```

Removing newlines with `chomp`

`chomp` removes the newline from the end of a string (if there is a newline).

```
my $string = "hey there!\n";
print "my string is: ", $string, "\n";
chomp $string;
print "after chomp : ", $string, "\n";
```

When you read a line from a file, the first thing you always want to do is chomp.

Counting lines in a file

Let's do something more interesting than printing the line back out. Let's count how many lines there are in the file.

```perl
my $line_count;
while (my $line = <IN>) {
    chomp $line;
    $line_count = $line_count + 1;
}
print "There are $line_count lines\n";
```

Why we read a file with `while`

Let's step back for a moment and think about why this works. What exactly is going on on this line?

```
    while (my $line = <IN>) {
```

`<IN>` returns a line from a file.
We assign that line to a variable, `$line`.
`while` tests that assignment for truth:
"Can we assign a value to `$line` ?"

If we've hit the end of the file, there are no more lines to read, and so the answer is "no", or FALSE.
When the expression in parentheses is false, we exit the loop.

What happens if the input file contains a blank line?

Example Sequence File Parser

HDAC.nt

```
CTTTTTTAAATTTAGTAAAATGGAAACCCCAGTTCGTAAGAAAGTGTGCTATTATTATGATGGAGACATT
GGGAATTATTATTATGGCCAGGGTCATCCCATGAAACCTCACCGTATTCGCATGACGCACAGTTTACTTT
TAAATTATGGCCTATATAGAAAGATGGAAGTATACAAGCCTTCAAAGGCTACTGCAGACGACATGACTAT
GTTTCATACTGATGAGTATATCCAATTTCTTCAGAGAATCCATCCCGATAATATGCACGAATACAACAAA
GAAATGCAAAGGTTCAACGTTGGAGAAGATTGTCCTGTATTTGATGGGTTATTCGAATTTTGTCAATTAT
```

Notice that the sequence is not contiguous. There are new lines at the end of each line.

```
my $in  = shift;
open(IN, '<', $in) or die "can't open $in: $!\n";
my $seq = '';
while (my $line = <IN>) {
    chomp $line;
    $seq .= $line;
}
print $seq , "\n";
```