Perl III

File I/O, more on system calls

Dave Messina

File I/O

I/O stands for input/output.

It's how get computer programs talk to the rest of the world.

Perl has magic

Perl has a magic way that makes it super easy to get data from files and into your program.

It looks like this: <>

<>

<> will:

read filenames that are arguments on the command line

open each file in turn

read each line from the file

<>

```
#!/usr/bin/perl
# how to read a file with <>
use warnings;
use strict;

while (my $line = <>) {
    chomp $line;
    print "Here's a line: ", $line, "\n";
}
```

Sidebar: 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 file, the first thing you always want to do is `chomp`.

<>

Let's make a file and read from it.
We'll call it myfile.txt

```
% perl read_from_file.pl myfile.txt
```

And now we're giving the name myfile.txt as a command-line argument to our Perl script.

<> line count

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

```
my $line_count;
while (my $line = <>) {
    chomp $line;
    $line_count++;
}
say "There are $line_count lines";
```

Sidebar: increment operators

Yesterday we learned several numeric operators. Here are a couple more common ones:

++ the increment operator

```
my $x = 1;
$x++;          # add 1 to $x

# exactly the same as
$x = $x + 1;
```

Sidebar: decrement operators

-- the decrement operator

```
my $x = 1;
$x--;          # subtract 1 from $x

# exactly the same as
$x = $x - 1;
```

<> line count

With ++, we're counting each time we go through the loop.

```
my $line_count;
while (my $line = <>) {
    chomp $line;
    $line_count++;
}
say "There are $line_count lines";
```

<> multiple files

If there is more than one argument, each one is opened and read completely, one after the other.

```
% perl read_from_file.pl myfile.txt another.txt
```

So let's create another file and try it.

<> mistakes

Remember how yesterday we had command-line arguments that were numbers?

Does Perl know that the arguments are files?

```
% perl read_from_file.pl 2 9
```

Let's try it and see what happens.

the input loop

Let's step back for a moment and think about why <> works. What is `while`? What is it testing?

```
my $line_count;
while (my $line = <>) {
    chomp $line;
    $line_count++;
}
say "There are $line_count lines";
```

the input loop

What exactly is going on on this line?

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

The <> is a function.
It returns a line of input.
We assign that line to a variable, `$line`.
While tests that assignment for truth:
"Can we assign a value to `$line`?"

the input loop

If there is another line in the file, the answer is "yes, we can, it's TRUE."

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.

the input loop

Once we've exited the loop, the say statement gets executed.

```
my $line_count;
while (my $line = <>) {
    chomp $line;
    $line_count++;
}
say "There are $line_count lines";
```

the input loop

To summarize:

The while loop will read one line of text after another. At the end of input, the <> operator returns undef and the while loop terminates.

Remember that even blank lines in a file are TRUE, because they consist of a single newline character.

STDOUT and STDERR

Every Perl script by default has two places it knows where to write to:

STDOUT and STDERR

STDOUT and STDERR

STDOUT

Standard output, used to write data out. Initially connected to the terminal, but can be redirected to a file or other program from the shell using redirection or pipes.

STDOUT and STDERR

STDERR

Standard error, used for diagnostic messages. Initially connected to the terminal.

STDOUT and STDERR

You've actually been usually STDOUT all along. It's the default place where your program's output goes.

When you use say or print, you're actually writing to STDOUT.

These are equivalent:

```
say "Well, how did I get here?";
say STDOUT "Well, how did I get here?";
```

STDOUT and STDERR

But you can also specify other places to write to.

Like STDERR:

```
say STDOUT "You may ask yourself:";
say STDERR "Well, how did I get here?";
```

STDOUT and STDERR

At first it looks exactly the same as STDOUT, but if we use output redirection on the command line, we can see that the output is actually going to a different place:

```
$ perl test.pl > output.txt
Well, how did I get here?
```

open for reading

<> is great, but often you want to read from a specific file. You can do that using open.

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

open

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

Let's break this down into pieces:

```
my $file = shift;
```

reads the filename from the command line.

open

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

open is a function, which is taking 3 arguments:

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

open

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

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

open

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

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

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

or a variable containing a filename:

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

Where can you go for more information on open?

open or die

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

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

Why `or`? What is being tested for truth?

open — $!

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

$! 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.

Let's try it.

open for writing

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

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

Now specify that filehandle when you say or print:

```
say FILE "I'm writing to a file!";
```

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

open

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

```
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";
```

open

To read from a filehandle line by line, you put the name of the filehandle inside <>, 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: $line\n";
}
```

a quick word on `system`

We saw yesterday that there were two ways of executing a command line from within Perl:

```
# with system
system("sort $file");

# or with backticks
`sort $file`;
```

a quick word on `system`

With backticks, you can capture the output from the command into a variable:

```
open(OUT, '>', 'sorted.txt') or die "error:$!";
my $sorted_output = `sort $file`;
print OUT "sorted output:\n", $sorted_output;
```