

File Handling with Perl

By Nathan Torkington

Introduction

This month's column talks about some Perl idioms for working with files and the data inside them. Using these idioms, you will be able to construct shorter and more reliable programs than those you might currently be writing.

Opening Files with Open()

The usual idiom for opening files is:

```
open(FH, "< $filename") or die "Couldn't open $filename for reading: $!";
```

The `open()` function normally takes two arguments. The first is a filehandle, a bare word that will be used from then on to refer to the open file. The second is a mixture of filename and "mode" (how the file is to be opened). `open()` returns true if the file was opened, false otherwise. We test this condition with "or."

The mode in the code above is indicated by the less-than character. If the file doesn't exist, `open()` will return false. In this instance, you will be able to read from the filehandle, but not write to it.

There are other modes. A greater-than character indicates writing. If the file doesn't exist, it will be created. If the file did exist, it will be truncated and any previous data in it will be lost. You can write to the filehandle, but you can't read from it.

create the file if it doesn't exist

```
open(FH, "> $filename") or die "Couldn't open $filename for writing: $!";
```

Appending (indicated by two greater-than characters) also creates the file if it doesn't exist, but this mode doesn't lose any existing data. Like the "<" or "reading" mode, you can only write to the filehandle (all writes are appended to the end of the file). Attempts to read from it will generate a runtime error.

```
open(FH, ">> $filename") or die "Couldn't open $filename for appending: $!";
```

You can both read and write by using the "+<" mode. You can move around in the file with the `tell()` function, and find out where you are with the `seek()` function. If the file doesn't exist, it will be created. If the file had data, the data will not be truncated.

If you want to truncate, either call `truncate()` yourself, or use the "+>" mode:

```
open(FH, "+> $filename") or die "Couldn't open $filename for reading and writing: $!";
```

Notice how "+<" and "+>" are different: both can be read from or written to, but the former inherits the gentle semantics of read-only filehandles, while the latter has the destructive semantics of write-only filehandles.

Errors

How can this fail? Any number of ways: the directory may not exist, the file may not be writable by you, your program may be out of filehandles, etc. You should always check the results of your system calls (like `open()` and `sysopen()`) to find out whether they succeeded.

File Handling with Perl

By Nathan Torkington

When you check for errors, usually with "or die()", you should note these particulars. First, be sure to mention the system call that failed ("open"). Secondly, mention the filename in order to locate the error more easily when attempting to fix it. Thirdly, note your method for opening the file ("for writing," "for appending"). Fourthly, provide the operating system's error message (contained in \$!) so the user of your program has some idea why the file couldn't be opened.

Sometimes we merge the first and third:

```
or die "unable to append to $filename: $!";
```

If you write the filename in full in both the open() and the error message, you run the risk of changing the open() but leaving the error message out of date and incorrect:

the following has a bogus error message

```
open(FH, "</var/run/file.pid") or die "Can't open /var/log/file.pod for writing : $!";
```

For more control over how the files are opened, what previous content they may contain and the creation of new files if they didn't already exist, use the sysopen() function:

```
use Fcntl;
```

```
sysopen(FH, $filename, O_RDWR|O_CREAT, 0666) or die "Can't open $filename for reading/writing/creating : $!";
```

The function sysopen() takes four arguments. The first is a filehandle just like open(). The second argument is the filename without any mode information. The third argument is the mode, a number created by logically OR-ing together constants provided by the Fcntl module. The fourth (and only optional) argument is the octal permissions value (leave it as 0666 for datafiles or 0777 for programs). sysopen() returns true if the file could be opened, false if the open function failed.

Unlike open(), sysopen() provides no simple shorthands for specifying the mode. Instead, you must join together some constants. Also unlike open(), each mode constant has a single meaning. Only by OR-ing them together can you get more than one set of behaviours:

O_RDONLY Read-only

O_WRONLY Write-only

O_RDWR Reading and writing

O_APPEND Writes go to the end of the file

O_TRUNC Truncate the file if it existed

O_CREAT Create the file if it didn't exist

O_EXCL Error if the file already existed (used with O_CREAT)

Use sysopen() when you want to be cautious. For instance, if you want to append an existing log file, but not create one if it's not already there, you would write:

```
sysopen(LOG, "/var/log/myprog.log", O_APPEND, 0666) or die "Can't open /var/log/myprog.log for appending: $!";
```

File Handling with Perl

By Nathan Torkington

There's one easy way to read from filehandles: the `<FH>` operator. In scalar context, it returns the next record from the file or `undef` on error. We can use it to read a single line into a variable:

```
$line = <FH>;  
die "Unexpected end-of-file" unless defined $line;
```

In a loop, we could write this:

```
while (defined ($record = <FH>)) {# long-winded  
# $record is set to each record in the file, one at a time  
}
```

Because we do this a lot, we often use this shorthand to put the record into `$_` instead of `$record`:

```
while (<FH>) {  
# $_ is set to each record in the file, one at a time  
}
```

In Perl 5.004_04, we can say:

```
while ($record = <FH>) {  
# $record is set to each record in the file, one at a time  
}
```

and the `defined()` is automatically put in for us. In versions of Perl before 5.004_04, this command gave a warning. You can find which version of Perl you're running with this command on the command-line:

```
perl -v
```

Once we have a record, we normally want to chop off the record separator (the newline by default):

```
chomp($record);
```

Version 4 of Perl only had the `chop()` operator, which removed the last character of the string regardless of what that character was. `chomp()` is not quite so destructive: it only removes line separators, and only if there was a line separator there. Use `chomp()` instead of `chop()`, if you're taking the line separator.

Reading Multiple Records

If you call `<FH>` in list context, it returns the remaining records in the file. If you were at the end of file, an empty list is returned:

```
@records = <FH>;  
if (@records) {  
print "There were ", scalar(@records), " records read.n";  
}
```

File Handling with Perl

By Nathan Torkington

We could do the assignment and test in one step:

```
if (@records = <FH>) {  
    print "There were ", scalar(@records), " records read.n";  
}
```

chomp() also works on a list of values:

```
@records = <FH>;  
chomp(@records);
```

You can chomp any expression, so you can write this in one step:

```
chomp(@records = <FH>);
```

What is a Record?

The default meaning for "record" is "line". This definition of record is controlled by the `$/` variable, which holds the input record separator. By default it is the string "n" because newline characters (by definition!) separate lines.

You can change this to anything you want. For instance:

```
$/ = ";";  
$record = <FH>; # read next semicolon-separated record
```

The two other interesting values of `$/` are the empty string ("") and undef.

Reading Paragraphs

Writing `$/ = ""` instructs Perl to read paragraphs, where a paragraph is any chunk of text that ends in two or more newlines. This is different from just setting `$/` to "nn," which would read paragraphs as text followed by only two newlines. In this case, a problem can arise when you have more than one consecutive blank lines, for example "textnnnn." You can interpret this as either one paragraph ("text") or two ("text", terminated by two newlines, and an empty paragraph terminated by two newlines).

The second interpretation is rarely useful when reading text. If you're reading paragraphs where sometimes there are two lines and sometimes there are twenty, you don't want to have to filter out the "blank" paragraphs:

```
$/ = "nn";  
while (<FH>) {  
    chomp;  
    next unless length; # skip empty paragraphs  
    # ...  
}
```

File Handling with Perl

By Nathan Torkington

Instead, you can set `$/` to `undef`, which reads paragraphs that are terminated by two or more newlines:

```
undef $/;
while (<FH>) {
  chomp;
  # ...
}
```

Reading the Entire File

The other interesting value of `$/` is `undef`. This value tells Perl that a read command should return the rest of the file as a single string:

```
undef $/;
$file = <FH>;
```

Because changing `$/` affects every read, not just the next one, you normally want to limit the operation's effects by localizing it. Editing as such leads to the following idiom for reading the contents of a filehandle into a single string:

```
{
  local $/ = undef;
  $file = <FH>;
}
```

Remember: Perl's variables can be as big as you want. Although your file cannot exceed the limits of your virtual memory, as we've seen, you can still read as much data as you want.

Regexps for Files

Once you have a variable containing the whole string, you can use regular expressions to work with the entire file not just a chunk of it. Two useful regular expression flags are `/s` and `/m`.

By default, Perl's regular expressions are tuned to working with lines. You can say:

```
undef $/;
$line = <FH>;
if ($line =~ /(b.*grass)$/) {
  print "found $1n";
}
```

If we feed this program a file containing:

```
browngress
bluegrass
```

we'll get this as output:

```
found bluegrass
```

File Handling with Perl

By Nathan Torkington

It didn't find "browngress" because the \$ finds its match only at the end of the string (or the newline before the end of the string). If we want to make "^" and "\$" match lines inside a string containing many lines, we can use the /m ("multiline") option:

```
if ($line =~ /(b.*grass)$/m) {}
```

Now the program says:

```
found browngress
```

Similarly, the period is designed not to match newlines:

```
while (<FH>) {  
  if (/19(.*)$/) {  
    if ($1 < 20) {  
      $year = 2000+$1;  
    } else {  
      $year = 1900+$1;  
    }  
  }  
}
```

If we read "1981" from a file, \$_ will contain "1981n". The period in the regular expression matches the "8" and the "1", but not the "n". This works well here – the newline isn't part of the date.

If we've got a string that contains many lines, however, we might want to pull bigger chunks out of it. These chunks might cross line boundaries. To do this, we can use the /s option to treat the line as a single line of text and make the period match a newline:

```
if (m{<B>(.*?)</B>}s) {  
  print "Found bold text: $1n";  
}
```

Here I've used {} to mark the start and end of the regular expression instead of slashes. Because of this, I have to tell Perl that I'm matching, hence the leading "m". The trailing "s" is the single-line regexp option. You can (and may often do) combine the /s and /m options:

```
if (m{^<FONT COLOR="red">(.*?)</FONT>}sm) {  
  # ...  
}
```

Summary

There are two ways to open files: open() is quick and easy, while sysopen() is more powerful and complex. You can read a record with the <FH> operator, and the \$/ variable lets you control just what a record is. Don't forget the /s and /m regular expression flags if you choose to read lots of lines into a single string.

File Handling with Perl

By Nathan Torkington

More Reading

The subject of opening and closing files is dealt with in Chapter 7 of "Perl Cookbook" (O'Reilly and Associates). Be sure to read the `open()` and `sysopen()` entries in the `perlfunc` manpage (in Chapter 3 of "Programming Perl", also from O'Reilly). Regular expressions are described in the `perlre` manpage (the "Regular expressions" section of Chapter 2 of Programming Perl), and the Cookbook's chapter 6 has a recipe called "Greedy and Non-Greedy Matches" that will explain the `".*?"` I used in the final examples.

Join me next month when I'll talk more about pulling files apart, extracting data from records and building summaries of that data. See you then!