NAME

perlopentut - simple recipes for opening files and pipes in Perl

DESCRIPTION

Whenever you do I/O on a file in Perl, you do so through what in Perl is called a **filehandle**. A filehandle is an internal name for an external file. It is the job of the `open` function to make the association between the internal name and the external name, and it is the job of the `close` function to break that association.

For your convenience, Perl sets up a few special filehandles that are already open when you run. These include `STDIN`, `STDOUT`, `STDERR`, and `ARGV`. Since those are pre-opened, you can use them right away without having to go to the trouble of opening them yourself:

```
print STDERR "This is a debugging message.\n";

print STDOUT "Please enter something: ";
$response = <STDIN> // die "how come no input?";
print STDOUT "Thank you!\n";

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

As you see from those examples, `STDOUT` and `STDERR` are output handles, and `STDIN` and `ARGV` are input handles. They are in all capital letters because they are reserved to Perl, much like the `@ARGV` array and the `%ENV` hash are. Their external associations were set up by your shell.

You will need to open every other filehandle on your own. Although there are many variants, the most common way to call Perl's open() function is with three arguments and one return value:

OK = open(*HANDLE*, *MODE*, *PATHNAME*)

Where:

OK

will be some defined value if the open succeeds, but `undef` if it fails;

HANDLE

should be an undefined scalar variable to be filled in by the `open` function if it succeeds;

MODE

is the access mode and the encoding format to open the file with;

PATHNAME

is the external name of the file you want opened.

Most of the complexity of the `open` function lies in the many possible values that the *MODE* parameter can take on.

One last thing before we show you how to open files: opening files does not (usually) automatically lock them in Perl. See *perlfaq5* for how to lock.

Opening Text Files

Opening Text Files for Reading

If you want to read from a text file, first open it in read-only mode like this:

```
my $filename = "/some/path/to/a/textfile/goes/here";
my $encoding = ":encoding(UTF-8)";
my $handle   = undef;     # this will be filled in on success
```

```
open($handle, "< $encoding", $filename)
    || die "$0: can't open $filename for reading: $!";
```

As with the shell, in Perl the "<" is used to open the file in read-only mode. If it succeeds, Perl allocates a brand new filehandle for you and fills in your previously undefined `$handle` argument with a reference to that handle.

Now you may use functions like `readline`, `read`, `getc`, and `sysread` on that handle. Probably the most common input function is the one that looks like an operator:

```
$line = readline($handle);
$line = <$handle>;          # same thing
```

Because the `readline` function returns `undef` at end of file or upon error, you will sometimes see it used this way:

```
$line = <$handle>;
if (defined $line) {
    # do something with $line
}
else {
    # $line is not valid, so skip it
}
```

You can also just quickly `die` on an undefined value this way:

```
$line = <$handle> // die "no input found";
```

However, if hitting EOF is an expected and normal event, you do not want to exit simply because you have run out of input. Instead, you probably just want to exit an input loop. You can then test to see if an actual error has caused the loop to terminate, and act accordingly:

```
while (<$handle>) {
    # do something with data in $_
}
if ($!) {
    die "unexpected error while reading from $filename: $!";
}
```

A Note on Encodings: Having to specify the text encoding every time might seem a bit of a bother. To set up a default encoding for `open` so that you don't have to supply it each time, you can use the `open` pragma:

```
use open qw< :encoding(UTF-8) >;
```

Once you've done that, you can safely omit the encoding part of the open mode:

```
open($handle, "<", $filename)
    || die "$0: can't open $filename for reading: $!";
```

But never use the bare "<" without having set up a default encoding first. Otherwise, Perl cannot know which of the many, many, many possible flavors of text file you have, and Perl will have no idea how to correctly map the data in your file into actual characters it can work with. Other common encoding formats including `"ASCII"`, `"ISO-8859-1"`, `"ISO-8859-15"`, `"Windows-1252"`, `"MacRoman"`, and even `"UTF-16LE"`. See *perlunitut* for more about encodings.

Opening Text Files for Writing

When you want to write to a file, you first have to decide what to do about any existing contents of that file. You have two basic choices here: to preserve or to clobber.

If you want to preserve any existing contents, then you want to open the file in append mode. As in the shell, in Perl you use ">>" to open an existing file in append mode. ">>" creates the file if it does not already exist.

```
my $handle   = undef;
my $filename = "/some/path/to/a/textfile/goes/here";
my $encoding = ":encoding(UTF-8)";

open($handle, ">> $encoding", $filename)
    || die "$0: can't open $filename for appending: $!";
```

Now you can write to that filehandle using any of `print`, `printf`, `say`, `write`, or `syswrite`.

As noted above, if the file does not already exist, then the append-mode open will create it for you. But if the file does already exist, its contents are safe from harm because you will be adding your new text past the end of the old text.

On the other hand, sometimes you want to clobber whatever might already be there. To empty out a file before you start writing to it, you can open it in write-only mode:

```
my $handle   = undef;
my $filename = "/some/path/to/a/textfile/goes/here";
my $encoding = ":encoding(UTF-8)";

open($handle, "> $encoding", $filename)
    || die "$0: can't open $filename in write-open mode: $!";
```

Here again Perl works just like the shell in that the ">" clobbers an existing file.

As with the append mode, when you open a file in write-only mode, you can now write to that filehandle using any of `print`, `printf`, `say`, `write`, or `syswrite`.

What about read-write mode? You should probably pretend it doesn't exist, because opening text files in read-write mode is unlikely to do what you would like. See *perlfaq5* for details.

Opening Binary Files

If the file to be opened contains binary data instead of text characters, then the `MODE` argument to `open` is a little different. Instead of specifying the encoding, you tell Perl that your data are in raw bytes.

```
my $filename = "/some/path/to/a/binary/file/goes/here";
my $encoding = ":raw :bytes"
my $handle   = undef;     # this will be filled in on success
```

And then open as before, choosing "<", ">>", or ">" as needed:

```
open($handle, "< $encoding", $filename)
    || die "$0: can't open $filename for reading: $!";

open($handle, ">> $encoding", $filename)
    || die "$0: can't open $filename for appending: $!";
```

```
open($handle, "> $encoding", $filename)
    || die "$0: can't open $filename in write-open mode: $!";
```

Alternately, you can change to binary mode on an existing handle this way:

```
binmode($handle)    || die "cannot binmode handle";
```

This is especially handy for the handles that Perl has already opened for you.

```
binmode(STDIN)      || die "cannot binmode STDIN";
binmode(STDOUT)     || die "cannot binmode STDOUT";
```

You can also pass `binmode` an explicit encoding to change it on the fly. This isn't exactly "binary" mode, but we still use `binmode` to do it:

```
binmode(STDIN,  ":encoding(MacRoman)") || die "cannot binmode STDIN";
binmode(STDOUT, ":encoding(UTF-8)")    || die "cannot binmode STDOUT";
```

Once you have your binary file properly opened in the right mode, you can use all the same Perl I/O functions as you used on text files. However, you may wish to use the fixed-size `read` instead of the variable-sized `readline` for your input.

Here's an example of how to copy a binary file:

```
my $BUFSIZ   = 64 * (2 ** 10);
my $name_in  = "/some/input/file";
my $name_out = "/some/output/flie";

my($in_fh, $out_fh, $buffer);

open($in_fh,  "<", $name_in)
    || die "$0: cannot open $name_in for reading: $!";
open($out_fh, ">", $name_out)
    || die "$0: cannot open $name_out for writing: $!";

for my $fh ($in_fh, $out_fh)  {
    binmode($fh)              || die "binmode failed";
}

while (read($in_fh, $buffer, $BUFSIZ)) {
    unless (print $out_fh $buffer) {
        die "couldn't write to $name_out: $!";
    }
}

close($in_fh)       || die "couldn't close $name_in: $!";
close($out_fh)      || die "couldn't close $name_out: $!";
```

Opening Pipes

To be announced.

Low-level File Opens via sysopen

To be announced. Or deleted.

SEE ALSO

To be announced.

AUTHOR and COPYRIGHT