NAME

perlopentut - tutorial on opening things in Perl

DESCRIPTION

Perl has two simple, built-in ways to open files: the shell way for convenience, and the C way for precision. The shell way also has 2- and 3-argument forms, which have different semantics for handling the filename. The choice is yours.

Open à la shell

Perl's open function was designed to mimic the way command-line redirection in the shell works. Here are some basic examples from the shell:

```
$ myprogram file1 file2 file3
$ myprogram    <  inputfile
$ myprogram    >  outputfile
$ myprogram    >> outputfile
$ myprogram    |  otherprogram
$ otherprogram |  myprogram
```

And here are some more advanced examples:

```
$ otherprogram      | myprogram f1 - f2
$ otherprogram 2>&1 | myprogram -
$ myprogram     <&3
$ myprogram     >&4
```

Programmers accustomed to constructs like those above can take comfort in learning that Perl directly supports these familiar constructs using virtually the same syntax as the shell.

Simple Opens

The open function takes two arguments: the first is a filehandle, and the second is a single string comprising both what to open and how to open it. open returns true when it works, and when it fails, returns a false value and sets the special variable $! to reflect the system error. If the filehandle was previously opened, it will be implicitly closed first.

For example:

```
open(INFO,       "datafile") || die("can't open datafile: $!");
open(INFO,    "<  datafile") || die("can't open datafile: $!");
open(RESULTS,">  runstats") || die("can't open runstats: $!");
open(LOG,     ">> logfile ") || die("can't open logfile:  $!");
```

If you prefer the low-punctuation version, you could write that this way:

```
open INFO,    "<  datafile"  or die "can't open datafile: $!";
open RESULTS,">  runstats"  or die "can't open runstats: $!";
open LOG,     ">> logfile "  or die "can't open logfile:  $!";
```

A few things to notice. First, the leading less-than is optional. If omitted, Perl assumes that you want to open the file for reading.

Note also that the first example uses the || logical operator, and the second uses or, which has lower precedence. Using || in the latter examples would effectively mean

```
open INFO, ( "<  datafile"  || die "can't open datafile: $!" );
```

which is definitely not what you want.

The other important thing to notice is that, just as in the shell, any whitespace before or after the filename is ignored. This is good, because you wouldn't want these to do different things:

```
open INFO,   "<datafile"
open INFO,   "< datafile"
open INFO,   "<  datafile"
```

Ignoring surrounding whitespace also helps for when you read a filename in from a different file, and forget to trim it before opening:

```
$filename = <INFO>;         # oops, \n still there
open(EXTRA, "< $filename") || die "can't open $filename: $!";
```

This is not a bug, but a feature. Because `open` mimics the shell in its style of using redirection arrows to specify how to open the file, it also does so with respect to extra whitespace around the filename itself as well. For accessing files with naughty names, see *Dispelling the Dweomer*.

There is also a 3-argument version of `open`, which lets you put the special redirection characters into their own argument:

```
open( INFO, ">", $datafile ) || die "Can't create $datafile: $!";
```

In this case, the filename to open is the actual string in `$datafile`, so you don't have to worry about `$datafile` containing characters that might influence the open mode, or whitespace at the beginning of the filename that would be absorbed in the 2-argument version. Also, any reduction of unnecessary string interpolation is a good thing.

Indirect Filehandles

`open`'s first argument can be a reference to a filehandle. As of perl 5.6.0, if the argument is uninitialized, Perl will automatically create a filehandle and put a reference to it in the first argument, like so:

```
    open( my $in, $infile )   or die "Couldn't read $infile: $!";
    while ( <$in> ) {
 # do something with $_
    }
    close $in;
```

Indirect filehandles make namespace management easier. Since filehandles are global to the current package, two subroutines trying to open `INFILE` will clash. With two functions opening indirect filehandles like `my $infile`, there's no clash and no need to worry about future conflicts.

Another convenient behavior is that an indirect filehandle automatically closes when it goes out of scope or when you undefine it:

```
    sub firstline {
 open( my $in, shift ) && return scalar <$in>;
 # no close() required
    }
```

Pipe Opens

In C, when you want to open a file using the standard I/O library, you use the `fopen` function, but when opening a pipe, you use the `popen` function. But in the shell, you just use a different redirection character. That's also the case for Perl. The `open` call remains the same--just its argument differs.

If the leading character is a pipe symbol, `open` starts up a new command and opens a write-only filehandle leading into that command. This lets you write into that handle and have what you write show up on that command's standard input. For example:

```
open(PRINTER, "| lpr -Plp1")    || die "can't run lpr: $!";
print PRINTER "stuff\n";
close(PRINTER)                  || die "can't close lpr: $!";
```

If the trailing character is a pipe, you start up a new command and open a read-only filehandle leading out of that command. This lets whatever that command writes to its standard output show up on your handle for reading. For example:

```
open(NET, "netstat -i -n |")    || die "can't fork netstat: $!";
while (<NET>) { }               # do something with input
close(NET)                      || die "can't close netstat: $!";
```

What happens if you try to open a pipe to or from a non-existent command? If possible, Perl will detect the failure and set `$!` as usual. But if the command contains special shell characters, such as > or *, called 'metacharacters', Perl does not execute the command directly. Instead, Perl runs the shell, which then tries to run the command. This means that it's the shell that gets the error indication. In such a case, the `open` call will only indicate failure if Perl can't even run the shell. See *"How can I capture STDERR from an external command?" in perlfaq8* to see how to cope with this. There's also an explanation in *perlipc*.

If you would like to open a bidirectional pipe, the IPC::Open2 library will handle this for you. Check out *"Bidirectional Communication with Another Process" in perlipc*

The Minus File

Again following the lead of the standard shell utilities, Perl's `open` function treats a file whose name is a single minus, "-", in a special way. If you open minus for reading, it really means to access the standard input. If you open minus for writing, it really means to access the standard output.

If minus can be used as the default input or default output, what happens if you open a pipe into or out of minus? What's the default command it would run? The same script as you're currently running! This is actually a stealth `fork` hidden inside an `open` call. See *"Safe Pipe Opens" in perlipc* for details.

Mixing Reads and Writes

It is possible to specify both read and write access. All you do is add a "+" symbol in front of the redirection. But as in the shell, using a less-than on a file never creates a new file; it only opens an existing one. On the other hand, using a greater-than always clobbers (truncates to zero length) an existing file, or creates a brand-new one if there isn't an old one. Adding a "+" for read-write doesn't affect whether it only works on existing files or always clobbers existing ones.

```
open(WTMP, "+< /usr/adm/wtmp")
    || die "can't open /usr/adm/wtmp: $!";

open(SCREEN, "+> lkscreen")
    || die "can't open lkscreen: $!";

open(LOGFILE, "+>> /var/log/applog"
    || die "can't open /var/log/applog: $!";
```

The first one won't create a new file, and the second one will always clobber an old one. The third one will create a new file if necessary and not clobber an old one, and it will allow you to read at any point in the file, but all writes will always go to the end. In short, the first case is substantially more common

than the second and third cases, which are almost always wrong. (If you know C, the plus in Perl's open is historically derived from the one in C's fopen(3S), which it ultimately calls.)

In fact, when it comes to updating a file, unless you're working on a binary file as in the WTMP case above, you probably don't want to use this approach for updating. Instead, Perl's **-i** flag comes to the rescue. The following command takes all the C, C++, or yacc source or header files and changes all their foo's to bar's, leaving the old version in the original filename with a ".orig" tacked on the end:

```
$ perl -i.orig -pe 's/\bfoo\b/bar/g' *.[Cchy]
```

This is a short cut for some renaming games that are really the best way to update textfiles. See the second question in *perlfaq5* for more details.

Filters

One of the most common uses for open is one you never even notice. When you process the ARGV filehandle using <ARGV>, Perl actually does an implicit open on each file in @ARGV. Thus a program called like this:

```
$ myprogram file1 file2 file3
```

Can have all its files opened and processed one at a time using a construct no more complex than:

```
while (<>) {
    # do something with $_
}
```

If @ARGV is empty when the loop first begins, Perl pretends you've opened up minus, that is, the standard input. In fact, $ARGV, the currently open file during <ARGV> processing, is even set to "-" in these circumstances.

You are welcome to pre-process your @ARGV before starting the loop to make sure it's to your liking. One reason to do this might be to remove command options beginning with a minus. While you can always roll the simple ones by hand, the Getopts modules are good for this:

```
use Getopt::Std;

# -v, -D, -o ARG, sets $opt_v, $opt_D, $opt_o
getopts("vDo:");

# -v, -D, -o ARG, sets $args{v}, $args{D}, $args{o}
getopts("vDo:", \%args);
```

Or the standard Getopt::Long module to permit named arguments:

```
use Getopt::Long;
GetOptions( "verbose"  => \$verbose,        # --verbose
            "Debug"    => \$debug,          # --Debug
            "output=s" => \$output );
 # --output=somestring or --output somestring
```

Another reason for preprocessing arguments is to make an empty argument list default to all files:

```
@ARGV = glob("*") unless @ARGV;
```

You could even filter out all but plain, text files. This is a bit silent, of course, and you might prefer to mention them on the way.

```
@ARGV = grep { -f && -T } @ARGV;
```

If you're using the **-n** or **-p** command-line options, you should put changes to @ARGV in a `BEGIN{}` block.

Remember that a normal `open` has special properties, in that it might call fopen(3S) or it might called popen(3S), depending on what its argument looks like; that's why it's sometimes called "magic open". Here's an example:

```
$pwdinfo = ‘domainname‘ =~ /^(\(none\))?$/
               ? ’< /etc/passwd’
               : ’ypcat passwd |’;

open(PWD, $pwdinfo)
            or die "can’t open $pwdinfo: $!";
```

This sort of thing also comes into play in filter processing. Because <ARGV> processing employs the normal, shell-style Perl `open`, it respects all the special things we've already seen:

```
$ myprogram f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

That program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file.

Yes, this also means that if you have files named "-" (and so on) in your directory, they won't be processed as literal files by `open`. You'll need to pass them as "./-", much as you would for the *rm* program, or you could use `sysopen` as described below.

One of the more interesting applications is to change files of a certain name into pipes. For example, to autoprocess gzipped or compressed files by decompressing them with *gzip*:

```
@ARGV = map { /^\.(gz|Z)$/ ? "gzip -dc $_ |" : $_  } @ARGV;
```

Or, if you have the *GET* program installed from LWP, you can fetch URLs before processing them:

```
@ARGV = map { m#^\w+://# ? "GET $_ |" : $_ } @ARGV;
```

It's not for nothing that this is called magic <ARGV>. Pretty nifty, eh?

Open à la C

If you want the convenience of the shell, then Perl's `open` is definitely the way to go. On the other hand, if you want finer precision than C's simplistic fopen(3S) provides you should look to Perl's `sysopen`, which is a direct hook into the open(2) system call. That does mean it's a bit more involved, but that's the price of precision.

`sysopen` takes 3 (or 4) arguments.

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

The HANDLE argument is a filehandle just as with `open`. The PATH is a literal path, one that doesn't pay attention to any greater-thans or less-thans or pipes or minuses, nor ignore whitespace. If it's there, it's part of the path. The FLAGS argument contains one or more values derived from the Fcntl module that have been or'd together using the bitwise "|" operator. The final argument, the MASK, is optional; if present, it is combined with the user's current umask for the creation mode of the file. You should usually omit this.

Although the traditional values of read-only, write-only, and read-write are 0, 1, and 2 respectively,

this is known not to hold true on some systems. Instead, it's best to load in the appropriate constants first from the Fcntl module, which supplies the following standard flags:

```
O_RDONLY            Read only
O_WRONLY            Write only
O_RDWR              Read and write
O_CREAT             Create the file if it doesn't exist
O_EXCL              Fail if the file already exists
O_APPEND            Append to the file
O_TRUNC             Truncate the file
O_NONBLOCK          Non-blocking access
```

Less common flags that are sometimes available on some operating systems include O_BINARY, O_TEXT, O_SHLOCK, O_EXLOCK, O_DEFER, O_SYNC, O_ASYNC, O_DSYNC, O_RSYNC, O_NOCTTY, O_NDELAY and O_LARGEFILE. Consult your open(2) manpage or its local equivalent for details. (Note: starting from Perl release 5.6 the O_LARGEFILE flag, if available, is automatically added to the sysopen() flags because large files are the default.)

Here's how to use sysopen to emulate the simple open calls we had before. We'll omit the `|| die $!` checks for clarity, but make sure you always check the return values in real code. These aren't quite the same, since open will trim leading and trailing whitespace, but you'll get the idea.

To open a file for reading:

```
open(FH, "< $path");
sysopen(FH, $path, O_RDONLY);
```

To open a file for writing, creating a new file if needed or else truncating an old file:

```
open(FH, "> $path");
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

To open a file for appending, creating one if necessary:

```
open(FH, ">> $path");
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

To open a file for update, where the file must already exist:

```
open(FH, "+< $path");
sysopen(FH, $path, O_RDWR);
```

And here are things you can do with sysopen that you cannot do with a regular open. As you'll see, it's just a matter of controlling the flags in the third argument.

To open a file for writing, creating a new file which must not previously exist:

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

To open a file for appending, where that file must already exist:

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

To open a file for update, creating a new file if necessary:

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

To open a file for update, where that file must not already exist:

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

To open a file without blocking, creating one if necessary:

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK | O_CREAT);
```

Permissions à la mode

If you omit the MASK argument to `sysopen`, Perl uses the octal value 0666. The normal MASK to use for executables and directories should be 0777, and for anything else, 0666.

Why so permissive? Well, it isn't really. The MASK will be modified by your process's current `umask`. A umask is a number representing *disabled* permissions bits; that is, bits that will not be turned on in the created files' permissions field.

For example, if your `umask` were 027, then the 020 part would disable the group from writing, and the 007 part would disable others from reading, writing, or executing. Under these conditions, passing `sysopen` 0666 would create a file with mode 0640, since `0666 & ~027` is 0640.

You should seldom use the MASK argument to `sysopen()`. That takes away the user's freedom to choose what permission new files will have. Denying choice is almost always a bad thing. One exception would be for cases where sensitive or private data is being stored, such as with mail folders, cookie files, and internal temporary files.

Obscure Open Tricks

Re-Opening Files (dups)

Sometimes you already have a filehandle open, and want to make another handle that's a duplicate of the first one. In the shell, we place an ampersand in front of a file descriptor number when doing redirections. For example, `2>&1` makes descriptor 2 (that's STDERR in Perl) be redirected into descriptor 1 (which is usually Perl's STDOUT). The same is essentially true in Perl: a filename that begins with an ampersand is treated instead as a file descriptor if a number, or as a filehandle if a string.

```
open(SAVEOUT, ">&SAVEERR") || die "couldn't dup SAVEERR: $!";
open(MHCONTEXT, "<&4")     || die "couldn't dup fd4: $!";
```

That means that if a function is expecting a filename, but you don't want to give it a filename because you already have the file open, you can just pass the filehandle with a leading ampersand. It's best to use a fully qualified handle though, just in case the function happens to be in a different package:

```
somefunction("&main::LOGFILE");
```

This way if somefunction() is planning on opening its argument, it can just use the already opened handle. This differs from passing a handle, because with a handle, you don't open the file. Here you have something you can pass to open.

If you have one of those tricky, newfangled I/O objects that the C++ folks are raving about, then this doesn't work because those aren't a proper filehandle in the native Perl sense. You'll have to use fileno() to pull out the proper descriptor number, assuming you can:

```
use IO::Socket;
$handle = IO::Socket::INET->new("www.perl.com:80");
$fd = $handle->fileno;
somefunction("&$fd");  # not an indirect function call
```

It can be easier (and certainly will be faster) just to use real filehandles though:

```
use IO::Socket;
local *REMOTE = IO::Socket::INET->new("www.perl.com:80");
die "can't connect" unless defined(fileno(REMOTE));
somefunction("&main::REMOTE");
```

If the filehandle or descriptor number is preceded not just with a simple "&" but rather with a "&=" combination, then Perl will not create a completely new descriptor opened to the same place using the dup(2) system call. Instead, it will just make something of an alias to the existing one using the fdopen(3S) library call This is slightly more parsimonious of systems resources, although this is less a concern these days. Here's an example of that:

```
$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fd")   or die "couldn't fdopen $fd: $!";
```

If you're using magic <ARGV>, you could even pass in as a command line argument in @ARGV something like "<&=$MHCONTEXTFD", but we've never seen anyone actually do this.

Dispelling the Dweomer

Perl is more of a DWIMmer language than something like Java--where DWIM is an acronym for "do what I mean". But this principle sometimes leads to more hidden magic than one knows what to do with. In this way, Perl is also filled with *dweomer*, an obscure word meaning an enchantment. Sometimes, Perl's DWIMmer is just too much like dweomer for comfort.

If magic open is a bit too magical for you, you don't have to turn to sysopen. To open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace. Leading whitespace is protected by inserting a "./" in front of a filename that starts with whitespace. Trailing whitespace is protected by appending an ASCII NUL byte ("\0") at the end of the string.

```
$file =~ s#^(\s)#./$1#;
open(FH, "< $file\0")   || die "can't open $file: $!";
```

This assumes, of course, that your system considers dot the current working directory, slash the directory separator, and disallows ASCII NULs within a valid filename. Most systems follow these conventions, including all POSIX systems as well as proprietary Microsoft systems. The only vaguely popular system that doesn't work this way is the "Classic" Macintosh system, which uses a colon where the rest of us use a slash. Maybe sysopen isn't such a bad idea after all.

If you want to use <ARGV> processing in a totally boring and non-magical way, you could do this first:

```
#   "Sam sat on the ground and put his head in his hands.
#   'I wish I had never come here, and I don't want to see
#   no more magic,' he said, and fell silent."
for (@ARGV) {
    s#^([^./])#./$1#;
    $_ .= "\0";
}
while (<>) {
    # now process $_
}
```

But be warned that users will not appreciate being unable to use "-" to mean standard input, per the standard convention.

Paths as Opens

You've probably noticed how Perl's `warn` and `die` functions can produce messages like:

```
Some warning at scriptname line 29, <FH> line 7.
```

That's because you opened a filehandle FH, and had read in seven records from it. But what was the name of the file, rather than the handle?

If you aren't running with `strict refs`, or if you've turned them off temporarily, then all you have to do is this:

```
open($path, "< $path") || die "can't open $path: $!";
while (<$path>) {
    # whatever
}
```

Since you're using the pathname of the file as its handle, you'll get warnings more like

```
Some warning at scriptname line 29, </etc/motd> line 7.
```

Single Argument Open

Remember how we said that Perl's open took two arguments? That was a passive prevarication. You see, it can also take just one argument. If and only if the variable is a global variable, not a lexical, you can pass `open` just one argument, the filehandle, and it will get the path from the global scalar variable of the same name.

```
$FILE = "/etc/motd";
open FILE or die "can't open $FILE: $!";
while (<FILE>) {
    # whatever
}
```

Why is this here? Someone has to cater to the hysterical porpoises. It's something that's been in Perl since the very beginning, if not before.

Playing with STDIN and STDOUT

One clever move with STDOUT is to explicitly close it when you're done with the program.

```
END { close(STDOUT) || die "can't close stdout: $!" }
```

If you don't do this, and your program fills up the disk partition due to a command line redirection, it won't report the error exit with a failure status.

You don't have to accept the STDIN and STDOUT you were given. You are welcome to reopen them if you'd like.

```
    open(STDIN, "< datafile")
 || die "can't open datafile: $!";

    open(STDOUT, "> output")
 || die "can't open output: $!";
```

And then these can be accessed directly or passed on to subprocesses. This makes it look as though the program were initially invoked with those redirections from the command line.

It's probably more interesting to connect these to pipes. For example:

```
    $pager = $ENV{PAGER} || "(less || more)";
    open(STDOUT, "| $pager")
 || die "can't fork a pager: $!";
```

This makes it appear as though your program were called with its stdout already piped into your pager. You can also use this kind of thing in conjunction with an implicit fork to yourself. You might do this if you would rather handle the post processing in your own program, just in a different process:

```
    head(100);
    while (<>) {
        print;
    }

    sub head {
        my $lines = shift || 20;
        return if $pid = open(STDOUT, "|-");       # return if parent
        die "cannot fork: $!" unless defined $pid;
        while (<STDIN>) {
            last if --$lines < 0;
            print;
        }
        exit;
    }
```

This technique can be applied to repeatedly push as many filters on your output stream as you wish.

Other I/O Issues

These topics aren't really arguments related to `open` or `sysopen`, but they do affect what you do with your open files.

Opening Non-File Files

When is a file not a file? Well, you could say when it exists but isn't a plain file. We'll check whether it's a symbolic link first, just in case.

```
    if (-l $file || ! -f _) {
        print "$file is not a plain file\n";
    }
```

What other kinds of files are there than, well, files? Directories, symbolic links, named pipes, Unix-domain sockets, and block and character devices. Those are all files, too--just not *plain* files. This isn't the same issue as being a text file. Not all text files are plain files. Not all plain files are text files. That's why there are separate `-f` and `-T` file tests.

To open a directory, you should use the `opendir` function, then process it with `readdir`, carefully restoring the directory name if necessary:

```
    opendir(DIR, $dirname) or die "can't opendir $dirname: $!";
    while (defined($file = readdir(DIR))) {
        # do something with "$dirname/$file"
    }
    closedir(DIR);
```

If you want to process directories recursively, it's better to use the File::Find module. For example, this prints out all files recursively and adds a slash to their names if the file is a directory.

```
    @ARGV = qw(.) unless @ARGV;
```

```
use File::Find;
find sub { print $File::Find::name, -d && '/', "\n" }, @ARGV;
```

This finds all bogus symbolic links beneath a particular directory:

```
find sub { print "$File::Find::name\n" if -l && !-e }, $dir;
```

As you see, with symbolic links, you can just pretend that it is what it points to. Or, if you want to know *what* it points to, then `readlink` is called for:

```
if (-l $file) {
    if (defined($whither = readlink($file))) {
        print "$file points to $whither\n";
    } else {
        print "$file points nowhere: $!\n";
    }
}
```

Opening Named Pipes

Named pipes are a different matter. You pretend they're regular files, but their opens will normally block until there is both a reader and a writer. You can read more about them in *"Named Pipes" in perlipc*. Unix-domain sockets are rather different beasts as well; they're described in *"Unix-Domain TCP Clients and Servers" in perlipc*.

When it comes to opening devices, it can be easy and it can be tricky. We'll assume that if you're opening up a block device, you know what you're doing. The character devices are more interesting. These are typically used for modems, mice, and some kinds of printers. This is described in *"How do I read and write the serial port?" in perlfaq8* It's often enough to open them carefully:

```
    sysopen(TTYIN, "/dev/ttyS1", O_RDWR | O_NDELAY | O_NOCTTY)
  # (O_NOCTTY no longer needed on POSIX systems)
      or die "can't open /dev/ttyS1: $!";
    open(TTYOUT, "+>&TTYIN")
      or die "can't dup TTYIN: $!";

    $ofh = select(TTYOUT); $| = 1; select($ofh);

    print TTYOUT "+++at\015";
    $answer = <TTYIN>;
```

With descriptors that you haven't opened using `sysopen`, such as sockets, you can set them to be non-blocking using `fcntl`:

```
use Fcntl;
my $old_flags = fcntl($handle, F_GETFL, 0)
    or die "can't get flags: $!";
fcntl($handle, F_SETFL, $old_flags | O_NONBLOCK)
    or die "can't set non blocking: $!";
```

Rather than losing yourself in a morass of twisting, turning `ioctl`s, all dissimilar, if you're going to manipulate ttys, it's best to make calls out to the stty(1) program if you have it, or else use the portable POSIX interface. To figure this all out, you'll need to read the termios(3) manpage, which describes the POSIX interface to tty devices, and then *POSIX*, which describes Perl's interface to POSIX. There are also some high-level modules on CPAN that can help you with these games. Check out Term::ReadKey and Term::ReadLine.

Opening Sockets

What else can you open? To open a connection using sockets, you won't use one of Perl's two open functions. See *"Sockets: Client/Server Communication" in perlipc* for that. Here's an example. Once you have it, you can use FH as a bidirectional filehandle.

```
use IO::Socket;
local *FH = IO::Socket::INET->new("www.perl.com:80");
```

For opening up a URL, the LWP modules from CPAN are just what the doctor ordered. There's no filehandle interface, but it's still easy to get the contents of a document:

```
use LWP::Simple;
$doc = get('http://www.linpro.no/lwp/');
```

Binary Files

On certain legacy systems with what could charitably be called terminally convoluted (some would say broken) I/O models, a file isn't a file--at least, not with respect to the C standard I/O library. On these old systems whose libraries (but not kernels) distinguish between text and binary streams, to get files to behave properly you'll have to bend over backwards to avoid nasty problems. On such infelicitous systems, sockets and pipes are already opened in binary mode, and there is currently no way to turn that off. With files, you have more options.

Another option is to use the `binmode` function on the appropriate handles before doing regular I/O on them:

```
binmode(STDIN);
binmode(STDOUT);
while (<STDIN>) { print }
```

Passing `sysopen` a non-standard flag option will also open the file in binary mode on those systems that support it. This is the equivalent of opening the file normally, then calling `binmode` on the handle.

```
sysopen(BINDAT, "records.data", O_RDWR | O_BINARY)
    || die "can't open records.data: $!";
```

Now you can use `read` and `print` on that handle without worrying about the non-standard system I/O library breaking your data. It's not a pretty picture, but then, legacy systems seldom are. CP/M will be with us until the end of days, and after.

On systems with exotic I/O systems, it turns out that, astonishingly enough, even unbuffered I/O using `sysread` and `syswrite` might do sneaky data mutilation behind your back.

```
while (sysread(WHENCE, $buf, 1024)) {
    syswrite(WHITHER, $buf, length($buf));
}
```

Depending on the vicissitudes of your runtime system, even these calls may need `binmode` or `O_BINARY` first. Systems known to be free of such difficulties include Unix, the Mac OS, Plan 9, and Inferno.

File Locking

In a multitasking environment, you may need to be careful not to collide with other processes who want to do I/O on the same files as you are working on. You'll often need shared or exclusive locks on files for reading and writing respectively. You might just pretend that only exclusive locks exist.

Never use the existence of a file `-e $file` as a locking indication, because there is a race condition

between the test for the existence of the file and its creation. It's possible for another process to create a file in the slice of time between your existence check and your attempt to create the file. Atomicity is critical.

Perl's most portable locking interface is via the `flock` function, whose simplicity is emulated on systems that don't directly support it such as SysV or Windows. The underlying semantics may affect how it all works, so you should learn how `flock` is implemented on your system's port of Perl.

File locking *does not* lock out another process that would like to do I/O. A file lock only locks out others trying to get a lock, not processes trying to do I/O. Because locks are advisory, if one process uses locking and another doesn't, all bets are off.

By default, the `flock` call will block until a lock is granted. A request for a shared lock will be granted as soon as there is no exclusive locker. A request for an exclusive lock will be granted as soon as there is no locker of any kind. Locks are on file descriptors, not file names. You can't lock a file until you open it, and you can't hold on to a lock once the file has been closed.

Here's how to get a blocking shared lock on a file, typically used for reading:

```
    use 5.004;
    use Fcntl qw(:DEFAULT :flock);
    open(FH, "< filename")  or die "can't open filename: $!";
    flock(FH, LOCK_SH)      or die "can't lock filename: $!";
    # now read from FH
```

You can get a non-blocking lock by using `LOCK_NB`.

```
    flock(FH, LOCK_SH | LOCK_NB)
        or die "can't lock filename: $!";
```

This can be useful for producing more user-friendly behaviour by warning if you're going to be blocking:

```
    use 5.004;
    use Fcntl qw(:DEFAULT :flock);
    open(FH, "< filename")  or die "can't open filename: $!";
    unless (flock(FH, LOCK_SH | LOCK_NB)) {
 $| = 1;
 print "Waiting for lock...";
 flock(FH, LOCK_SH)  or die "can't lock filename: $!";
 print "got it.\n"
    }
    # now read from FH
```

To get an exclusive lock, typically used for writing, you have to be careful. We `sysopen` the file so it can be locked before it gets emptied. You can get a nonblocking version using `LOCK_EX | LOCK_NB` .

```
    use 5.004;
    use Fcntl qw(:DEFAULT :flock);
    sysopen(FH, "filename", O_WRONLY | O_CREAT)
        or die "can't open filename: $!";
    flock(FH, LOCK_EX)
        or die "can't lock filename: $!";
    truncate(FH, 0)
        or die "can't truncate filename: $!";
    # now write to FH
```

Finally, due to the uncounted millions who cannot be dissuaded from wasting cycles on useless vanity devices called hit counters, here's how to increment a number in a file safely:

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "numfile", O_RDWR | O_CREAT)
    or die "can't open numfile: $!";
# autoflush FH
$ofh = select(FH); $| = 1; select ($ofh);
flock(FH, LOCK_EX)
    or die "can't write-lock numfile: $!";

$num = <FH> || 0;
seek(FH, 0, 0)
    or die "can't rewind numfile : $!";
print FH $num+1, "\n"
    or die "can't write numfile: $!";

truncate(FH, tell(FH))
    or die "can't truncate numfile: $!";
close(FH)
    or die "can't close numfile: $!";
```

IO Layers

In Perl 5.8.0 a new I/O framework called "PerlIO" was introduced. This is a new "plumbing" for all the I/O happening in Perl; for the most part everything will work just as it did, but PerlIO also brought in some new features such as the ability to think of I/O as "layers". One I/O layer may in addition to just moving the data also do transformations on the data. Such transformations may include compression and decompression, encryption and decryption, and transforming between various character encodings.

Full discussion about the features of PerlIO is out of scope for this tutorial, but here is how to recognize the layers being used:

- The three-(or more)-argument form of `open` is being used and the second argument contains something else in addition to the usual `'<'`, `'>'`, `'>>'`, `'|'` and their variants, for example:

  ```
  open(my $fh, "<:utf8", $fn);
  ```

- The two-argument form of `binmode` is being used, for example

  ```
  binmode($fh, ":encoding(utf16)");
  ```

For more detailed discussion about PerlIO see *PerlIO*; for more detailed discussion about Unicode and I/O see *perluniintro*.

SEE ALSO

The `open` and `sysopen` functions in perlfunc(1); the system open(2), dup(2), fopen(3), and fdopen(3) manpages; the POSIX documentation.

AUTHOR and COPYRIGHT

domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

HISTORY

First release: Sat Jan 9 08:09:11 MST 1999