NAME

perlfaq5 - Files and Formats

VERSION

version 5.021010

DESCRIPTION

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

How do I flush/unbuffer an output filehandle? Why must I do this?

(contributed by brian d foy)

You might like to read Mark Jason Dominus's "Suffering From Buffering" at <http://perl.plover.com/FAQs/Buffering.html>

Perl normally buffers output so it doesn't make a system call for every bit of output. By saving up output, it makes fewer expensive system calls. For instance, in this little bit of code, you want to print a dot to the screen for every line you process to watch the progress of your program. Instead of seeing a dot for every line, Perl buffers the output and you have a long wait before you see a row of 50 dots all at once:

```
# long wait, then row of dots all at once
while( <> ) {
print ".";
print "\n" unless ++$count % 50;

#... expensive line processing operations
}
```

To get around this, you have to unbuffer the output filehandle, in this case, STDOUT. You can set the special variable $| to a true value (mnemonic: making your filehandles "piping hot"):

```
$|++;

# dot shown immediately
while( <> ) {
print ".";
print "\n" unless ++$count % 50;

#... expensive line processing operations
}
```

The $| is one of the per-filehandle special variables, so each filehandle has its own copy of its value. If you want to merge standard output and standard error for instance, you have to unbuffer each (although STDERR might be unbuffered by default):

```
{
my $previous_default = select(STDOUT); # save previous default
$|++; # autoflush STDOUT
select(STDERR);
$|++; # autoflush STDERR, to be sure
select($previous_default); # restore previous default
}

# now should alternate . and +
while( 1 ) {
sleep 1;
print STDOUT ".";
print STDERR "+";
print STDOUT "\n" unless ++$count % 25;
}
```

Besides the $| special variable, you can use `binmode` to give your filehandle a `:unix` layer, which is unbuffered:

```
binmode( STDOUT, ":unix" );

while( 1 ) {
sleep 1;
print ".";
print "\n" unless ++$count % 50;
}
```

For more information on output layers, see the entries for `binmode` and open in perlfunc(1), and the PerlIO module documentation.

If you are using IO::Handle or one of its subclasses, you can call the `autoflush` method to change the settings of the filehandle:

```
use IO::Handle;
open my( $io_fh ), ">", "output.txt";
$io_fh->autoflush(1);
```

The IO::Handle objects also have a `flush` method. You can flush the buffer any time you want without auto-buffering

```
$io_fh->flush;
```

How do I change, delete, or insert a line in a file, or append to the beginning of a file?

(contributed by brian d foy)

The basic idea of inserting, changing, or deleting a line from a text file involves reading and printing the file to the point you want to make the change, making the change, then reading and printing the rest of the file. Perl doesn't provide random access to lines (especially since the record input separator, $/, is mutable), although modules such as Tie::File can fake it.

A Perl program to do these tasks takes the basic form of opening a file, printing its lines, then closing the file:

```
open my $in, '<', $file or die "Can't read old file: $!";
open my $out, '>', "$file.new" or die "Can't write new file: $!";

while( <$in> ) {
print $out $_;
}

close $out;
```

Within that basic form, add the parts that you need to insert, change, or delete lines.

To prepend lines to the beginning, print those lines before you enter the loop that prints the existing lines.

```
open my $in, '<', $file or die "Can't read old file: $!";
open my $out, '>', "$file.new" or die "Can't write new file: $!";

print $out "# Add this line to the top\n"; # <--- HERE'S THE MAGIC

while( <$in> ) {
print $out $_;
}

close $out;
```

To change existing lines, insert the code to modify the lines inside the `while` loop. In this case, the code finds all lowercased versions of ''perl'' and uppercases them. The happens for every line, so be sure that

you're supposed to do that on every line!

```
open my $in, '<', $file or die "Can't read old file: $!";
open my $out, '>', "$file.new" or die "Can't write new file: $!";

print $out "# Add this line to the top\n";

while( <$in> ) {
s/\b(perl)\b/Perl/g;
print $out $_;
}

close $out;
```

To change only a particular line, the input line number, $., is useful. First read and print the lines up to the one you want to change. Next, read the single line you want to change, change it, and print it. After that, read the rest of the lines and print those:

```
while( <$in> ) { # print the lines before the change
print $out $_;
last if $. == 4; # line number before change
}

my $line = <$in>;
$line =~ s/\b(perl)\b/Perl/g;
print $out $line;

while( <$in> ) { # print the rest of the lines
print $out $_;
}
```

To skip lines, use the looping controls. The next in this example skips comment lines, and the last stops all processing once it encounters either _ _END_ _ or _ _DATA_ _.

```
while( <$in> ) {
next if /^\s+#/; # skip comment lines
last if /^__(END|DATA)__$/; # stop at end of code marker
print $out $_;
}
```

Do the same sort of thing to delete a particular line by using next to skip the lines you don't want to show up in the output. This example skips every fifth line:

```
while( <$in> ) {
next unless $. % 5;
print $out $_;
}
```

If, for some odd reason, you really want to see the whole file at once rather than processing line-by-line, you can slurp it in (as long as you can fit the whole thing in memory!):

```
open my $in, '<', $file or die "Can't read old file: $!"
open my $out, '>', "$file.new" or die "Can't write new file: $!";

my $content = do { local $/; <$in> }; # slurp!

# do your magic here

print $out $content;
```

Modules such as Path::Tiny and Tie::File can help with that too. If you can, however, avoid reading the entire file at once. Perl won't give that memory back to the operating system until the process finishes.

You can also use Perl one-liners to modify a file in-place. The following changes all 'Fred' to 'Barney' in *inFile.txt*, overwriting the file with the new contents. With the -p switch, Perl wraps a `while` loop around the code you specify with -e, and -i turns on in-place editing. The current line is in $_. With -p, Perl automatically prints the value of $_ at the end of the loop. See perlrun(1) for more details.

```
perl -pi -e 's/Fred/Barney/' inFile.txt
```

To make a backup of `inFile.txt`, give -i a file extension to add:

```
perl -pi.bak -e 's/Fred/Barney/' inFile.txt
```

To change only the fifth line, you can add a test checking $., the input line number, then only perform the operation when the test passes:

```
perl -pi -e 's/Fred/Barney/ if $. == 5' inFile.txt
```

To add lines before a certain line, you can add a line (or lines!) before Perl prints $_:

```
perl -pi -e 'print "Put before third line\n" if $. == 3' inFile.txt
```

You can even add a line to the beginning of a file, since the current line prints at the end of the loop:

```
perl -pi -e 'print "Put before first line\n" if $. == 1' inFile.txt
```

To insert a line after one already in the file, use the -n switch. It's just like -p except that it doesn't print $_ at the end of the loop, so you have to do that yourself. In this case, print $_ first, then print the line that you want to add.

```
perl -ni -e 'print; print "Put after fifth line\n" if $. == 5' inFile.txt
```

To delete lines, only print the ones that you want.

```
perl -ni -e 'print if /d/' inFile.txt
```

How do I count the number of lines in a file?

(contributed by brian d foy)

Conceptually, the easiest way to count the lines in a file is to simply read them and count them:

```
my $count = 0;
while( <$fh> ) { $count++; }
```

You don't really have to count them yourself, though, since Perl already does that with the $. variable, which is the current line number from the last filehandle read:

```
1 while( <$fh> );
my $count = $.;
```

If you want to use $., you can reduce it to a simple one-liner, like one of these:

```
% perl -lne '} print $.; {' file

% perl -lne 'END { print $. }' file
```

Those can be rather inefficient though. If they aren't fast enough for you, you might just read chunks of data and count the number of newlines:

```
my $lines = 0;
open my($fh), '<:raw', $filename or die "Can't open $filename: $!";
while( sysread $fh, $buffer, 4096 ) {
$lines += ( $buffer =~ tr/\n// );
}
close $fh;
```

However, that doesn't work if the line ending isn't a newline. You might change that `tr///` to a `s///` so you can count the number of times the input record separator, $/, shows up:

```
my $lines = 0;
open my($fh), '<:raw', $filename or die "Can't open $filename: $!";
while( sysread $fh, $buffer, 4096 ) {
$lines += ( $buffer =~ s|$/||g; );
}
close $fh;
```

If you don't mind shelling out, the `wc` command is usually the fastest, even with the extra interprocess overhead. Ensure that you have an untainted filename though:

```
#!perl -T

$ENV{PATH} = undef;

my $lines;
if( $filename =~ /^([0-9a-z_.]+)\z/ ) {
$lines = `/usr/bin/wc -l $1`
chomp $lines;
}
```

How do I delete the last N lines from a file?

(contributed by brian d foy)

The easiest conceptual solution is to count the lines in the file then start at the beginning and print the number of lines (minus the last N) to a new file.

Most often, the real question is how you can delete the last N lines without making more than one pass over the file, or how to do it without a lot of copying. The easy concept is the hard reality when you might have millions of lines in your file.

One trick is to use File::ReadBackwards, which starts at the end of the file. That module provides an object that wraps the real filehandle to make it easy for you to move around the file. Once you get to the spot you need, you can get the actual filehandle and work with it as normal. In this case, you get the file position at the end of the last line you want to keep and truncate the file to that point:

```
use File::ReadBackwards;

my $filename = 'test.txt';
my $Lines_to_truncate = 2;

my $bw = File::ReadBackwards->new( $filename )
or die "Could not read backwards in [$filename]: $!";

my $lines_from_end = 0;
until( $bw->eof or $lines_from_end == $Lines_to_truncate ) {
print "Got: ", $bw->readline;
$lines_from_end++;
}

truncate( $filename, $bw->tell );
```

The File::ReadBackwards module also has the advantage of setting the input record separator to a regular expression.

You can also use the Tie::File module which lets you access the lines through a tied array. You can use normal array operations to modify your file, including setting the last index and using `splice`.

How can I use Perl's `-i` option from within a program?

`-i` sets the value of Perl's `$^I` variable, which in turn affects the behavior of `<>`; see perlrun(1) for more details. By modifying the appropriate variables directly, you can get the same behavior within a larger

program. For example:

```
# ...
{
local($^I, @ARGV) = ('.orig', glob("*.c"));
while (<>) {
if ($. == 1) {
print "This line should appear at the top of each file\n";
}
s/\b(p)earl\b/${1}erl/i; # Correct typos, preserving case
print;
close ARGV if eof; # Reset $.
}
}
# $^I and @ARGV return to their old values here
```

This block modifies all the `.c` files in the current directory, leaving a backup of the original data from each file in a new `.c.orig` file.

How can I copy a file?

(contributed by brian d foy)

Use the File::Copy module. It comes with Perl and can do a true copy across file systems, and it does its magic in a portable fashion.

```
use File::Copy;

copy( $original, $new_copy ) or die "Copy failed: $!";
```

If you can't use File::Copy, you'll have to do the work yourself: open the original file, open the destination file, then print to the destination file as you read the original. You also have to remember to copy the permissions, owner, and group to the new file.

How do I make a temporary file name?

If you don't need to know the name of the file, you can use `open()` with `undef` in place of the file name. In Perl 5.8 or later, the `open()` function creates an anonymous temporary file:

```
open my $tmp, '+>', undef or die $!;
```

Otherwise, you can use the File::Temp module.

```
use File::Temp qw/ tempfile tempdir /;

my $dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );

# or if you don't need to know the filename

my $fh = tempfile( DIR => $dir );
```

The File::Temp has been a standard module since Perl 5.6.1. If you don't have a modern enough Perl installed, use the `new_tmpfile` class method from the IO::File module to get a filehandle opened for reading and writing. Use it if you don't need to know the file's name:

```
use IO::File;
my $fh = IO::File->new_tmpfile()
or die "Unable to make new temporary file: $!";
```

If you're committed to creating a temporary file by hand, use the process ID and/or the current time-value. If you need to have many temporary files in one process, use a counter:

```
    BEGIN {
    use Fcntl;
    use File::Spec;
    my $temp_dir = File::Spec->tmpdir();
    my $file_base = sprintf "%d-%d-0000", $$, time;
    my $base_name = File::Spec->catfile($temp_dir, $file_base);

    sub temp_file {
    my $fh;
    my $count = 0;
    until( defined(fileno($fh)) || $count++ > 100 ) {
    $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
    # O_EXCL is required for security reasons.
    sysopen $fh, $base_name, O_WRONLY|O_EXCL|O_CREAT;
    }

    if( defined fileno($fh) ) {
    return ($fh, $base_name);
    }
    else {
    return ();
    }
    }
    }
```

How can I manipulate fixed-record-length files?

The most efficient way is using *pack()* and *unpack()*. This is faster than using *substr()* when taking many, many strings. It is slower for just a few.

Here is a sample chunk of code to break up and put back together again some fixed-format input lines, in this case from the output of a normal, Berkeley-style ps:

```
    # sample input line:
    # 15158 p5 T 0:00 perl /home/tchrist/scripts/now-what
    my $PS_T = 'A6 A4 A7 A5 A*';
    open my $ps, '-|', 'ps';
    print scalar <$ps>;
    my @fields = qw( pid tt stat time command );
    while (<$ps>) {
    my %process;
    @process{@fields} = unpack($PS_T, $_);
    for my $field ( @fields ) {
    print "$field: <$process{$field}>\n";
    }
    print 'line=', pack($PS_T, @process{@fields} ), "\n";
    }
```

We've used a hash slice in order to easily handle the fields of each row. Storing the keys in an array makes it easy to operate on them as a group or loop over them with `for`. It also avoids polluting the program with global variables and using symbolic references.

How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?

As of perl5.6, *open()* autovivifies file and directory handles as references if you pass it an uninitialized scalar variable. You can then pass these references just like any other scalar, and use them in the place of named handles.

```
open my $fh, $file_name;

open local $fh, $file_name;

print $fh "Hello World!\n";

process_file( $fh );
```

If you like, you can store these filehandles in an array or a hash. If you access them directly, they aren't simple scalars and you need to give `print` a little help by placing the filehandle reference in braces. Perl can only figure it out on its own when the filehandle reference is a simple scalar.

```
my @fhs = ( $fh1, $fh2, $fh3 );

for( $i = 0; $i <= $#fhs; $i++ ) {
print {$fhs[$i]} "just another Perl answer, \n";
}
```

Before perl5.6, you had to deal with various typeglob idioms which you may see in older code.

```
open FILE, "> $filename";
process_typeglob( *FILE );
process_reference( \*FILE );

sub process_typeglob { local *FH = shift; print FH "Typeglob!" }
sub process_reference { local $fh = shift; print $fh "Reference!" }
```

If you want to create many anonymous handles, you should check out the Symbol or IO::Handle modules.

How can I use a filehandle indirectly?

An indirect filehandle is the use of something other than a symbol in a place that a filehandle is expected. Here are ways to get indirect filehandles:

```
$fh = SOME_FH; # bareword is strict-subs hostile
$fh = "SOME_FH"; # strict-refs hostile; same package only
$fh = *SOME_FH; # typeglob
$fh = \*SOME_FH; # ref to typeglob (bless-able)
$fh = *SOME_FH{IO}; # blessed IO::Handle from *SOME_FH typeglob
```

Or, you can use the `new` method from one of the IO::* modules to create an anonymous filehandle and store that in a scalar variable.

```
use IO::Handle; # 5.004 or higher
my $fh = IO::Handle->new();
```

Then use any of those as you would a normal filehandle. Anywhere that Perl is expecting a filehandle, an indirect filehandle may be used instead. An indirect filehandle is just a scalar variable that contains a filehandle. Functions like `print`, `open`, `seek`, or the `<FH>` diamond operator will accept either a named filehandle or a scalar variable containing one:

```
($ifh, $ofh, $efh) = (*STDIN, *STDOUT, *STDERR);
print $ofh "Type it: ";
my $got = <$ifh>
print $efh "What was that: $got";
```

If you're passing a filehandle to a function, you can write the function in two ways:

```
sub accept_fh {
my $fh = shift;
print $fh "Sending to indirect filehandle\n";
}
```

Or it can localize a typeglob and use the filehandle directly:

```
sub accept_fh {
local *FH = shift;
print FH "Sending to localized filehandle\n";
}
```

Both styles work with either objects or typeglobs of real filehandles. (They might also work with strings under some circumstances, but this is risky.)

```
accept_fh(*STDOUT);
accept_fh($handle);
```

In the examples above, we assigned the filehandle to a scalar variable before using it. That is because only simple scalar variables, not expressions or subscripts of hashes or arrays, can be used with built-ins like `print`, `printf`, or the diamond operator. Using something other than a simple scalar variable as a filehandle is illegal and won't even compile:

```
my @fd = (*STDIN, *STDOUT, *STDERR);
print $fd[1] "Type it: "; # WRONG
my $got = <$fd[0]> # WRONG
print $fd[2] "What was that: $got"; # WRONG
```

With `print` and `printf`, you get around this by using a block and an expression where you would place the filehandle:

```
print { $fd[1] } "funny stuff\n";
printf { $fd[1] } "Pity the poor %x.\n", 3_735_928_559;
# Pity the poor deadbeef.
```

That block is a proper block like any other, so you can put more complicated code there. This sends the message out to one of two places:

```
my $ok = -x "/bin/cat";
print { $ok ? $fd[1] : $fd[2] } "cat stat $ok\n";
print { $fd[ 1+ ($ok || 0) ] } "cat stat $ok\n";
```

This approach of treating `print` and `printf` like object methods calls doesn't work for the diamond operator. That's because it's a real operator, not just a function with a comma-less argument. Assuming you've been storing typeglobs in your structure as we did above, you can use the built-in function named `readline` to read a record just as `<>` does. Given the initialization shown above for `@fd`, this would work, but only because *readline()* requires a typeglob. It doesn't work with objects or strings, which might be a bug we haven't fixed yet.

```
$got = readline($fd[0]);
```

Let it be noted that the flakiness of indirect filehandles is not related to whether they're strings, typeglobs, objects, or anything else. It's the syntax of the fundamental operators. Playing the object game doesn't help you at all here.

How can I set up a footer format to be used with *write()***?**
There's no builtin way to do this, but perlform(1) has a couple of techniques to make it possible for the intrepid hacker.

How can I *write()* **into a string?**
(contributed by brian d foy)

If you want to `write` into a string, you just have to <open> a filehandle to a string, which Perl has been able to do since Perl 5.6:

```
open FH, '>', \my $string;
write( FH );
```

Since you want to be a good programmer, you probably want to use a lexical filehandle, even though formats are designed to work with bareword filehandles since the default format names take the filehandle name. However, you can control this with some Perl special per-filehandle variables: `$^`, which names the

top-of-page format, and `$~` which shows the line format. You have to change the default filehandle to set these variables:

```
open my($fh), '>', \my $string;

{ # set per-filehandle variables
my $old_fh = select( $fh );
$~ = 'ANIMAL';
$^ = 'ANIMAL_TOP';
select( $old_fh );
}

format ANIMAL_TOP =
ID Type Name
.

format ANIMAL =
@## @<<< @<<<<<<<<<<<<
$id, $type, $name
.
```

Although write can work with lexical or package variables, whatever variables you use have to scope in the format. That most likely means you'll want to localize some package variables:

```
{
local( $id, $type, $name ) = qw( 12 cat Buster );
write( $fh );
}

print $string;
```

There are also some tricks that you can play with `formline` and the accumulator variable `$^A`, but you lose a lot of the value of formats since `formline` won't handle paging and so on. You end up reimplementing formats when you use them.

How can I open a filehandle to a string?

(contributed by Peter J. Holzer, hjp-usenet2@hjp.at)

Since Perl 5.8.0 a file handle referring to a string can be created by calling open with a reference to that string instead of the filename. This file handle can then be used to read from or write to the string:

```
open(my $fh, '>', \$string) or die "Could not open string for writing";
print $fh "foo\n";
print $fh "bar\n"; # $string now contains "foo\nbar\n"

open(my $fh, '<', \$string) or die "Could not open string for reading";
my $x = <$fh>; # $x now contains "foo\n"
```

With older versions of Perl, the IO::String module provides similar functionality.

How can I output my numbers with commas added?

(contributed by brian d foy and Benjamin Goldberg)

You can use Number::Format to separate places in a number. It handles locale information for those of you who want to insert full stops instead (or anything else that they want to use, really).

This subroutine will add commas to your number:

```
sub commify {
local $_ = shift;
1 while s/^([-+]?\d+)(\d{3})/$1,$2/;
return $_;
}
```

This regex from Benjamin Goldberg will add commas to numbers:

```
s/(^[-+]?\d+?(?=(?>(?:\d{3})+)(?!\d))|\G\d{3}(?=\d))/$1,/g;
```

It is easier to see with comments:

```
s/(
^[-+]? # beginning of number.
\d+? # first digits before first comma
(?= # followed by, (but not included in the match) :
(?>(?:\d{3})+) # some positive multiple of three digits.
(?!\d) # an *exact* multiple, not x * 3 + 1 or whatever.
)
| # or:
\G\d{3} # after the last group, get three digits
(?=\d) # but they have to have more digits after them.
)/$1,/xg;
```

How can I translate tildes (˜) in a filename?

Use the <> (glob()) operator, documented in perlfunc. Versions of Perl older than 5.6 require that you have a shell installed that groks tildes. Later versions of Perl have this feature built in. The File::KGlob module (available from CPAN) gives more portable glob functionality.

Within Perl, you may use this directly:

```
$filename =˜ s{
^ ˜ # find a leading tilde
( # save this in $1
[^/] # a non-slash character
* # repeated 0 or more times (0 means me)
)
}{
$1
? (getpwnam($1))[7]
: ( $ENV{HOME} || $ENV{LOGDIR} )
}ex;
```

How come when I open a file read-write it wipes it out?

Because you're using something like this, which truncates the file *then* gives you read-write access:

```
open my $fh, '+>', '/path/name'; # WRONG (almost always)
```

Whoops. You should instead use this, which will fail if the file doesn't exist:

```
open my $fh, '+<', '/path/name'; # open for update
```

Using ">" always clobbers or creates. Using "<" never does either. The "+" doesn't change this.

Here are examples of many kinds of file opens. Those using sysopen all assume that you've pulled in the constants from Fcntl:

```
use Fcntl;
```

To open file for reading:

```
open my $fh, '<', $path or die $!;
sysopen my $fh, $path, O_RDONLY or die $!;
```

To open file for writing, create new file if needed or else truncate old file:

```
open my $fh, '>', $path or die $!;
sysopen my $fh, $path, O_WRONLY|O_TRUNC|O_CREAT or die $!;
sysopen my $fh, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666 or die $!;
```

To open file for writing, create new file, file must not exist:

```
sysopen my $fh, $path, O_WRONLY|O_EXCL|O_CREAT or die $!;
sysopen my $fh, $path, O_WRONLY|O_EXCL|O_CREAT, 0666 or die $!;
```

To open file for appending, create if necessary:

```
open my $fh, '>>' $path or die $!;
sysopen my $fh, $path, O_WRONLY|O_APPEND|O_CREAT or die $!;
sysopen my $fh, $path, O_WRONLY|O_APPEND|O_CREAT, 0666 or die $!;
```

To open file for appending, file must exist:

```
sysopen my $fh, $path, O_WRONLY|O_APPEND or die $!;
```

To open file for update, file must exist:

```
open my $fh, '+<', $path or die $!;
sysopen my $fh, $path, O_RDWR or die $!;
```

To open file for update, create file if necessary:

```
sysopen my $fh, $path, O_RDWR|O_CREAT or die $!;
sysopen my $fh, $path, O_RDWR|O_CREAT, 0666 or die $!;
```

To open file for update, file must not exist:

```
sysopen my $fh, $path, O_RDWR|O_EXCL|O_CREAT or die $!;
sysopen my $fh, $path, O_RDWR|O_EXCL|O_CREAT, 0666 or die $!;
```

To open a file without blocking, creating if necessary:

```
sysopen my $fh, '/foo/somefile', O_WRONLY|O_NDELAY|O_CREAT
or die "can't open /foo/somefile: $!":
```

Be warned that neither creation nor deletion of files is guaranteed to be an atomic operation over NFS. That is, two processes might both successfully create or unlink the same file! Therefore O_EXCL isn't as exclusive as you might wish.

See also perlopentut.

Why do I sometimes get an "Argument list too long" when I use <*>?

The <> operator performs a globbing operation (see above). In Perl versions earlier than v5.6.0, the internal *glob()* operator forks *csh(1)* to do the actual glob expansion, but csh can't handle more than 127 items and so gives the error message `Argument list too long`. People who installed tcsh as csh won't have this problem, but their users may be surprised by it.

To get around this, either upgrade to Perl v5.6.0 or later, do the glob yourself with *readdir()* and patterns, or use a module like File::Glob, one that doesn't use the shell to do globbing.

How can I open a file with a leading ">" or trailing blanks?

(contributed by Brian McCauley)

The special two-argument form of Perl's *open()* function ignores trailing blanks in filenames and infers the mode from certain leading characters (or a trailing "|"). In older versions of Perl this was the only version of *open()* and so it is prevalent in old code and books.

Unless you have a particular reason to use the two-argument form you should use the three-argument form of *open()* which does not treat any characters in the filename as special.

```
open my $fh, "<", " file "; # filename is " file "
open my $fh, ">", ">file"; # filename is ">file"
```

How can I reliably rename a file?

If your operating system supports a proper *mv(1)* utility or its functional equivalent, this works:

```
rename($old, $new) or system("mv", $old, $new);
```

It may be more portable to use the File::Copy module instead. You just copy to the new file to the new name (checking return values), then delete the old one. This isn't really the same semantically as a `rename()`, which preserves meta-information like permissions, timestamps, inode info, etc.

How can I lock a file?

Perl's builtin *flock()* function (see perlfunc(1) for details) will call *flock(2)* if that exists, *fcntl(2)* if it doesn't (on perl version 5.004 and later), and *lockf(3)* if neither of the two previous system calls exists. On some systems, it may even use a different form of native locking. Here are some gotchas with Perl's *flock()*:

1. Produces a fatal error if none of the three system calls (or their close equivalent) exists.

2. *lockf(3)* does not provide shared locking, and requires that the filehandle be open for writing (or appending, or read/writing).

3. Some versions of *flock()* can't lock files over a network (e.g. on NFS file systems), so you'd need to force the use of *fcntl(2)* when you build Perl. But even this is dubious at best. See the flock entry of perlfunc(1) and the *INSTALL* file in the source distribution for information on building Perl to do this.

   Two potentially non-obvious but traditional flock semantics are that it waits indefinitely until the lock is granted, and that its locks are *merely advisory*. Such discretionary locks are more flexible, but offer fewer guarantees. This means that files locked with *flock()* may be modified by programs that do not also use *flock()*. Cars that stop for red lights get on well with each other, but not with cars that don't stop for red lights. See the perlport(1) manpage, your port's specific documentation, or your system-specific local manpages for details. It's best to assume traditional behavior if you're writing portable programs. (If you're not, you should as always feel perfectly free to write for your own system's idiosyncrasies (sometimes called "features"). Slavish adherence to portability concerns shouldn't get in the way of your getting your job done.)

   For more information on file locking, see also "File Locking" in perlopentut(1) if you have it (new for 5.6).

Why can't I just open(FH, ">file.lock")?

A common bit of code **NOT TO USE** is this:

```
sleep(3)
while -e 'file.lock'; # PLEASE DO NOT USE
 open my $lock, '>', 'file.lock'; # THIS BROKEN CODE
```

This is a classic race condition: you take two steps to do something which must be done in one. That's why computer hardware provides an atomic test-and-set instruction. In theory, this "ought" to work:

```
sysopen my $fh, "file.lock", O_WRONLY|O_EXCL|O_CREAT
 or die "can't open file.lock: $!";
```

except that lamentably, file creation (and deletion) is not atomic over NFS, so this won't work (at least, not every time) over the net. Various schemes involving *link()* have been suggested, but these tend to involve busy-wait, which is also less than desirable.

I still don't get locking. I just want to increment the number in the file. How can I do this?

Didn't anyone ever tell you web-page hit counters were useless? They don't count number of hits, they're a waste of time, and they serve only to stroke the writer's vanity. It's better to pick a random number; they're more realistic.

Anyway, this is what you can do if you can't help yourself.

```
use Fcntl qw(:DEFAULT :flock);
sysopen my $fh, "numfile", O_RDWR|O_CREAT or die "can't open numfile: $!";
flock $fh, LOCK_EX or die "can't flock numfile: $!";
my $num = <$fh> || 0;
seek $fh, 0, 0 or die "can't rewind numfile: $!";
truncate $fh, 0 or die "can't truncate numfile: $!";
(print $fh $num+1, "\n") or die "can't write numfile: $!";
close $fh or die "can't close numfile: $!";
```

Here's a much better web-page hit counter:

```
$hits = int( (time() - 850_000_000) / rand(1_000) );
```

If the count doesn't impress your friends, then the code might. :-)

All I want to do is append a small amount of text to the end of a file. Do I still have to use locking?

If you are on a system that correctly implements `flock` and you use the example appending code from "perldoc(1) -f flock" everything will be OK even if the OS you are on doesn't implement append mode correctly (if such a system exists). So if you are happy to restrict yourself to OSs that implement `flock` (and that's not really much of a restriction) then that is what you should do.

If you know you are only going to use a system that does correctly implement appending (i.e. not Win32) then you can omit the `seek` from the code in the previous answer.

If you know you are only writing code to run on an OS and filesystem that does implement append mode correctly (a local filesystem on a modern Unix for example), and you keep the file in block-buffered mode and you write less than one buffer-full of output between each manual flushing of the buffer then each bufferload is almost guaranteed to be written to the end of the file in one chunk without getting intermingled with anyone else's output. You can also use the `syswrite` function which is simply a wrapper around your system's write(2) system call.

There is still a small theoretical chance that a signal will interrupt the system-level `write()` operation before completion. There is also a possibility that some STDIO implementations may call multiple system level `write()`s even if the buffer was empty to start. There may be some systems where this probability is reduced to zero, and this is not a concern when using `:perlio` instead of your system's STDIO.

How do I randomly update a binary file?

If you're just trying to patch a binary, in many cases something as simple as this works:

```
perl -i -pe 's{window manager}{window mangler}g' /usr/bin/emacs
```

However, if you have fixed sized records, then you might do something more like this:

```
my $RECSIZE = 220; # size of record, in bytes
my $recno = 37; # which record to update
open my $fh, '+<', 'somewhere' or die "can't update somewhere: $!";
seek $fh, $recno * $RECSIZE, 0;
read $fh, $record, $RECSIZE == $RECSIZE or die "can't read record $recno: $!";
# munge the record
seek $fh, -$RECSIZE, 1;
print $fh $record;
close $fh;
```

Locking and error checking are left as an exercise for the reader. Don't forget them or you'll be quite sorry.

How do I get a file's timestamp in perl?

If you want to retrieve the time at which the file was last read, written, or had its meta-data (owner, etc) changed, you use the **-A**, **-M**, or **-C** file test operations as documented in perlfunc. These retrieve the age of the file (measured against the start-time of your program) in days as a floating point number. Some platforms may not have all of these times. See perlport(1) for details. To retrieve the "raw" time in seconds since the epoch, you would call the stat function, then use `localtime()`, `gmtime()`, or `POSIX::strftime()` to convert this into human-readable form.

Here's an example:

```
my $write_secs = (stat($file))[9];
printf "file %s updated at %s\n", $file,
scalar localtime($write_secs);
```

If you prefer something more legible, use the File::stat module (part of the standard distribution in version 5.004 and later):

```
# error checking left as an exercise for reader.
use File::stat;
use Time::localtime;
my $date_string = ctime(stat($file)->mtime);
print "file $file updated at $date_string\n";
```

The *POSIX::strftime()* approach has the benefit of being, in theory, independent of the current locale. See perllocale(1) for details.

How do I set a file's timestamp in perl?

You use the *utime()* function documented in "utime" in perlfunc. By way of example, here's a little program that copies the read and write times from its first argument to all the rest of them.

```
if (@ARGV < 2) {
die "usage: cptimes timestamp_file other_files ...\n";
}
my $timestamp = shift;
my($atime, $mtime) = (stat($timestamp))[8,9];
utime $atime, $mtime, @ARGV;
```

Error checking is, as usual, left as an exercise for the reader.

The perldoc(1) for utime also has an example that has the same effect as *touch(1)* on files that *already exist*.

Certain file systems have a limited ability to store the times on a file at the expected level of precision. For example, the FAT and HPFS filesystem are unable to create dates on files with a finer granularity than two seconds. This is a limitation of the filesystems, not of *utime()*.

How do I print to more than one file at once?

To connect one filehandle to several output filehandles, you can use the IO::Tee or Tie::FileHandle::Multiplex modules.

If you only have to do this once, you can print individually to each filehandle.

```
for my $fh ($fh1, $fh2, $fh3) { print $fh "whatever\n" }
```

How can I read in an entire file all at once?

The customary Perl approach for processing all the lines in a file is to do so one line at a time:

```
open my $input, '<', $file or die "can't open $file: $!";
while (<$input>) {
chomp;
# do something with $_
}
close $input or die "can't close $file: $!";
```

This is tremendously more efficient than reading the entire file into memory as an array of lines and then processing it one element at a time, which is often — if not almost always — the wrong approach. Whenever you see someone do this:

```
my @lines = <INPUT>;
```

You should think long and hard about why you need everything loaded at once. It's just not a scalable solution.

If you "mmap" the file with the File::Map module from CPAN, you can virtually load the entire file into a

string without actually storing it in memory:

```
use File::Map qw(map_file);

map_file my $string, $filename;
```

Once mapped, you can treat $string as you would any other string. Since you don't necessarily have to load the data, mmap-ing can be very fast and may not increase your memory footprint.

You might also find it more fun to use the standard Tie::File module, or the DB_File module's $DB_RECNO bindings, which allow you to tie an array to a file so that accessing an element of the array actually accesses the corresponding line in the file.

If you want to load the entire file, you can use the Path::Tiny module to do it in one simple and efficient step:

```
use Path::Tiny;

my $all_of_it = path($filename)->slurp; # entire file in scalar
my @all_lines = path($filename)->lines; # one line per element
```

Or you can read the entire file contents into a scalar like this:

```
my $var;
{
local $/;
open my $fh, '<', $file or die "can't open $file: $!";
$var = <$fh>;
}
```

That temporarily undefs your record separator, and will automatically close the file at block exit. If the file is already open, just use this:

```
my $var = do { local $/; <$fh> };
```

You can also use a localized @ARGV to eliminate the open:

```
my $var = do { local( @ARGV, $/ ) = $file; <> };
```

For ordinary files you can also use the read function.

```
read( $fh, $var, -s $fh );
```

That third argument tests the byte size of the data on the $fh filehandle and reads that many bytes into the buffer $var.

How can I read in a file by paragraphs?

Use the $/ variable (see perlvar(1) for details). You can either set it to "" to eliminate empty paragraphs ("abc\n\n\n\ndef", for instance, gets treated as two paragraphs and not three), or "\n\n" to accept empty paragraphs.

Note that a blank line must have no blanks in it. Thus "fred\n \nstuff\n\n" is one paragraph, but "fred\n\nstuff\n\n" is two.

How can I read a single character from a file? From the keyboard?

You can use the builtin getc() function for most filehandles, but it won't (easily) work on a terminal device. For STDIN, either use the Term::ReadKey module from CPAN or use the sample code in ''getc'' in perlfunc.

If your system supports the portable operating system programming interface (POSIX), you can use the following code, which you'll note turns off echo processing as well.

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
print "gimme: ";
my $got = getone();
print "--> $got\n";
}
exit;

BEGIN {
use POSIX qw(:termios_h);

my ($term, $oterm, $echo, $noecho, $fd_stdin);

my $fd_stdin = fileno(STDIN);

$term = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm = $term->getlflag();

$echo = ECHO | ECHOK | ICANON;
$noecho = $oterm & ~$echo;

sub cbreak {
$term->setlflag($noecho);
$term->setcc(VTIME, 1);
$term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
$term->setlflag($oterm);
$term->setcc(VTIME, 0);
$term->setattr($fd_stdin, TCSANOW);
}

sub getone {
my $key = '';
cbreak();
sysread(STDIN, $key, 1);
cooked();
return $key;
}
}

END { cooked() }
```

The Term::ReadKey module from CPAN may be easier to use. Recent versions include also support for non-portable systems as well.

```
use Term::ReadKey;
open my $tty, '<', '/dev/tty';
print "Gimme a char: ";
ReadMode "raw";
my $key = ReadKey 0, $tty;
ReadMode "normal";
printf "\nYou said %s, char number %03d\n",
$key, ord $key;
```

How can I tell whether there's a character waiting on a filehandle?

The very first thing you should do is look into getting the Term::ReadKey extension from CPAN. As we mentioned earlier, it now even has limited support for non-portable (read: not open systems, closed, proprietary, not POSIX, not Unix, etc.) systems.

You should also check out the Frequently Asked Questions list in comp.unix.* for things like this: the answer is essentially the same. It's very system-dependent. Here's one solution that works on BSD systems:

```
sub key_ready {
my($rin, $nfd);
vec($rin, fileno(STDIN), 1) = 1;
return $nfd = select($rin,undef,undef,0);
}
```

If you want to find out how many characters are waiting, there's also the FIONREAD ioctl call to be looked at. The *h2ph* tool that comes with Perl tries to convert C include files to Perl code, which can be `required`. FIONREAD ends up defined as a function in the *sys/ioctl.ph* file:

```
require 'sys/ioctl.ph';

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size) or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

If *h2ph* wasn't installed or doesn't work for you, you can *grep* the include files by hand:

```
% grep FIONREAD /usr/include/*/*
/usr/include/asm/ioctls.h:#define FIONREAD 0x541B
```

Or write a small C program using the editor of champions:

```
% cat > fionread.c
#include <sys/ioctl.h>
main() {
printf("%#08x\n", FIONREAD);
}
^D
% cc -o fionread fionread.c
% ./fionread
0x4004667f
```

And then hard-code it, leaving porting as an exercise to your successor.

```
$FIONREAD = 0x4004667f; # XXX: opsys dependent

$size = pack("L", 0);
ioctl(FH, $FIONREAD, $size) or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

FIONREAD requires a filehandle connected to a stream, meaning that sockets, pipes, and tty devices work, but *not* files.

How do I do a `tail -f` in perl?

IO::Handle File::Tail clearerr" First try

```
seek($gw_fh, 0, 1);
```

The statement `seek($gw_fh, 0, 1)` doesn't change the current position, but it does clear the end-of-file condition on the handle, so that the next `<$gw_fh>` makes Perl try again to read something.

If that doesn't work (it relies on features of your stdio implementation), then you need something more like this:

```
for (;;) {
for ($curpos = tell($gw_fh); <$gw_fh>; $curpos =tell($gw_fh)) {
# search for some stuff and put it into files
}
# sleep for a while
seek($gw_fh, $curpos, 0); # seek to where we had been
}
```

If this still doesn't work, look into the `clearerr` method from IO::Handle, which resets the error and end-of-file states on the handle.

There's also a File::Tail module from CPAN.

How do I *dup()* a filehandle in Perl?

If you check "open" in perlfunc(1), you'll see that several of the ways to call *open()* should do the trick. For example:

```
open my $log, '>>', '/foo/logfile';
open STDERR, '>&', $log;
```

Or even with a literal numeric descriptor:

```
my $fd = $ENV{MHCONTEXTFD};
open $mhcontext, "<&=$fd"; # like fdopen(3S)
```

Note that "<&STDIN" makes a copy, but "<&=STDIN" makes an alias. That means if you close an aliased handle, all aliases become inaccessible. This is not true with a copied one.

Error checking, as always, has been left as an exercise for the reader.

How do I close a file descriptor by number?

If, for some reason, you have a file descriptor instead of a filehandle (perhaps you used POSIX::open you can use the `close()` function from the POSIX module:

```
use POSIX ();


POSIX::close( $fd );
```

This should rarely be necessary, as the Perl `close()` function is to be used for things that Perl opened itself, even if it was a dup of a numeric descriptor as with MHCONTEXT above. But if you really have to, you may be able to do this:

```
require 'sys/syscall.ph';
my $rc = syscall(SYS_close(), $fd + 0); # must force numeric
die "can't sysclose $fd: $!" unless $rc == -1;
```

Or, just use the fdopen(3S) feature of `open()`:

```
{
open my $fh, "<&=$fd" or die "Cannot reopen fd=$fd: $!";
close $fh;
}
```

Why can't I use "C:\temp\foo" in DOS paths? Why doesn't 'C:\temp\foo.exe' work?

Whoops! You just put a tab and a formfeed into that filename! Remember that within double quoted strings ("like\this"), the backslash is an escape character. The full list of these is in "Quote and Quote-like Operators" in perlop. Unsurprisingly, you don't have a file called "c:(tab)emp(formfeed)oo" or "c:(tab)emp(formfeed)oo.exe" on your legacy DOS filesystem.

Either single-quote your strings, or (preferably) use forward slashes. Since all DOS and Windows versions since something like MS-DOS 2.0 or so have treated / and \ the same in a path, you might as well use the one that doesn't clash with Perl — or the POSIX shell, ANSI C and C++, awk, Tcl, Java, or Python, just to mention a few. POSIX paths are more portable, too.

Why doesn't glob("*.*") get all the files?

Because even on non-Unix ports, Perl's glob function follows standard Unix globbing semantics. You'll need `glob("*")` to get all (non-hidden) files. This makes *glob()* portable even to legacy systems. Your port may include proprietary globbing functions as well. Check its documentation for details.

Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn't this a bug in Perl?

This is elaborately and painstakingly described in the *file-dir-perms* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz>

The executive summary: learn how your filesystem works. The permissions on a file say what can happen to the data in that file. The permissions on a directory say what can happen to the list of files in that directory. If you delete a file, you're removing its name from the directory (so the operation depends on the permissions of the directory, not of the file). If you try to write to the file, the permissions of the file govern whether you're allowed to.

How do I select a random line from a file?

Short of loading the file into a database or pre-indexing the lines in the file, there are a couple of things that you can do.

Here's a reservoir-sampling algorithm from the Camel Book:

```
srand;
rand($.) < 1 && ($line = $_) while <>;
```

This has a significant advantage in space over reading the whole file in. You can find a proof of this method in *The Art of Computer Programming*, Volume 2, Section 3.4.2, by Donald E. Knuth.

You can use the File::Random module which provides a function for that algorithm:

```
use File::Random qw/random_line/;
my $line = random_line($filename);
```

Another way is to use the Tie::File module, which treats the entire file as an array. Simply access a random array element.

Why do I get weird spaces when I print an array of lines?

(contributed by brian d foy)

If you are seeing spaces between the elements of your array when you print the array, you are probably interpolating the array in double quotes:

```
my @animals = qw(camel llama alpaca vicuna);
print "animals are: @animals\n";
```

It's the double quotes, not the `print`, doing this. Whenever you interpolate an array in a double quote context, Perl joins the elements with spaces (or whatever is in `$"`, which is a space by default):

```
animals are: camel llama alpaca vicuna
```

This is different than printing the array without the interpolation:

```
my @animals = qw(camel llama alpaca vicuna);
print "animals are: ", @animals, "\n";
```

Now the output doesn't have the spaces between the elements because the elements of `@animals` simply

become part of the list to `print`:

```
 animals are: camelllamaalpacavicuna
```

You might notice this when each of the elements of `@array` end with a newline. You expect to print one element per line, but notice that every line after the first is indented:

```
 this is a line
 this is another line
 this is the third line
```

That extra space comes from the interpolation of the array. If you don't want to put anything between your array elements, don't use the array in double quotes. You can send it to print without them:

```
 print @lines;
```

How do I traverse a directory tree?

(contributed by brian d foy)

The File::Find module, which comes with Perl, does all of the hard work to traverse a directory structure. It comes with Perl. You simply call the `find` subroutine with a callback subroutine and the directories you want to traverse:

```
 use File::Find;

 find( \&wanted, @directories );

 sub wanted {
 # full path in $File::Find::name
 # just filename in $_
 ... do whatever you want to do ...
 }
```

The File::Find::Closures, which you can download from CPAN, provides many ready-to-use subroutines that you can use with File::Find.

The File::Finder, which you can download from CPAN, can help you create the callback subroutine using something closer to the syntax of the `find` command-line utility:

```
 use File::Find;
 use File::Finder;

 my $deep_dirs = File::Finder->depth->type('d')->ls->exec('rmdir','{}');

 find( $deep_dirs->as_options, @places );
```

The File::Find::Rule module, which you can download from CPAN, has a similar interface, but does the traversal for you too:

```
 use File::Find::Rule;

 my @files = File::Find::Rule->file()
 ->name( '*.pm' )
 ->in( @INC );
```

How do I delete a directory tree?

(contributed by brian d foy)

If you have an empty directory, you can use Perl's built-in `rmdir`. If the directory is not empty (so, no files or subdirectories), you either have to empty it yourself (a lot of work) or use a module to help you.

The File::Path module, which comes with Perl, has a `remove_tree` which can take care of all of the hard work for you:

```
use File::Path qw(remove_tree);

remove_tree( @directories );
```

The File::Path module also has a legacy interface to the older `rmtree` subroutine.

How do I copy an entire directory?

(contributed by Shlomi Fish)

To do the equivalent of `cp -R` (i.e. copy an entire directory tree recursively) in portable Perl, you'll either need to write something yourself or find a good CPAN module such as File::Copy::Recursive.

AUTHOR AND COPYRIGHT

Copyright (c) 1997-2010 Tom Christiansen, Nathan Torkington, and other authors as noted. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.