NAME

perlfaq5 - Files and Formats ($Revision: 1.31 $, $Date: 2004/02/07 04:29:50 $)

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?

Perl does not support truly unbuffered output (except insofar as you can `syswrite(OUT, $char, 1)`), although it does support is "command buffering", in which a physical write is performed after every output command.

The C standard I/O library (stdio) normally buffers characters sent to devices so that there isn't a system call for each byte. In most stdio implementations, the type of output buffering and the size of the buffer varies according to the type of device. Perl's print() and write() functions normally buffer output, while syswrite() bypasses buffering all together.

If you want your output to be sent immediately when you execute print() or write() (for instance, for some network protocols), you must set the handle's autoflush flag. This flag is the Perl variable $| and when it is set to a true value, Perl will flush the handle's buffer after each print() or write(). Setting $| affects buffering only for the currently selected default file handle. You choose this handle with the one argument select() call (see *"$|" in perlvar* and *"select" in perlfunc*).

Use select() to choose the desired handle, then set its per-filehandle variables.

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);
```

Some idioms can handle this in a single statement:

```
select((select(OUTPUT_HANDLE), $| = 1)[0]);

$| = 1, select $_ for select OUTPUT_HANDLE;
```

Some modules offer object-oriented access to handles and their variables, although they may be overkill if this is the only thing you do with them. You can use IO::Handle:

```
use IO::Handle;
open(DEV, ">/dev/printer");   # but is this?
DEV->autoflush(1);
```

or IO::Socket:

```
   use IO::Socket;    # this one is kinda a pipe?
my $sock = IO::Socket::INET->new( 'www.example.com:80' ) ;

   $sock->autoflush();
```

How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?

Use the Tie::File module, which is included in the standard distribution since Perl 5.8.0.

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

One fairly efficient way is to count newlines in the file. The following program uses a feature of tr///, as documented in *perlop*. If your text file doesn't end with a newline, then it's not really a proper text file,

so this may report one fewer line than you expect.

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

This assumes no funny games with newline translations.

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* 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 do I make a temporary file name?

Use the File::Temp module, see *File::Temp* for more information.

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

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

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

  $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;
    $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;
my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMPDIR} || $ENV{TEMP};
my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
sub temp_file {
    local *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 means it's 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 );
```

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 using 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, store that in a scalar variable, and use it as though it were a normal filehandle.

```
use IO::Handle;                     # 5.004 or higher
$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: ";
$got = <$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:

```
@fd = (*STDIN, *STDOUT, *STDERR);
print $fd[1] "Type it: ";                          # WRONG
$got = <$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:

```
$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* has a couple of techniques to make it possible for the intrepid hacker.

How can I write() into a string?

See *"Accessing Formatting Internals" in perlform* for an swrite() function.

How can I output my numbers with commas added?

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{1,3}?          # 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*. Older versions of Perl require that you have a shell installed that groks tildes. Recent perl versions 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 and *then* gives you read-write access:

```
    open(FH, "+> /path/name");  # WRONG (almost always)
```

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

```
    open(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

```
    use Fcntl;
```

To open file for reading:

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

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

```
open(FH, "> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT)       || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666) || die $!;
```

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

```
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT)        || die $!;
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT, 0666)  || die $!;
```

To open file for appending, create if necessary:

```
open(FH, ">> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT)      || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT, 0666) || die $!;
```

To open file for appending, file must exist:

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

To open file for update, file must exist:

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

To open file for update, create file if necessary:

```
sysopen(FH, $path, O_RDWR|O_CREAT)                 || die $!;
sysopen(FH, $path, O_RDWR|O_CREAT, 0666)           || die $!;
```

To open file for update, file must not exist:

```
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT)          || die $!;
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT, 0666)    || die $!;
```

To open a file without blocking, creating if necessary:

```
sysopen(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 the new *perlopentut* if you have it (new for 5.6).

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::KGlob, one that doesn't use the shell to do globbing.

Is there a leak/bug in glob()?

Due to the current implementation on some operating systems, when you use the glob() function or its angle-bracket alias in a scalar context, you may cause a memory leak and/or unpredictable behavior. It's best therefore to use glob() only in list context.

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

Normally perl ignores trailing blanks in filenames, and interprets certain leading characters (or a trailing "|") to mean something special.

The three argument form of open() lets you specify the mode separately from the filename. The open() function treats special mode characters and whitespace in the filename as literals

```
open FILE, "<", "  file  ";  # filename is "   file   "
open FILE, ">", ">file";     # filename is ">file"
```

It may be a lot clearer to use sysopen(), though:

```
    use Fcntl;
    $badpath = "<<<something really wicked   ";
    sysopen (FH, $badpath, O_WRONLY | O_CREAT | O_TRUNC)
or die "can't open $badpath: $!";
```

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.

Newer versions of File::Copy export a move() function.

How can I lock a file?

Perl's builtin flock() function (see *perlfunc* 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* 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 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* 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(LCK, "> 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(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 subdesirable.

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(FH, "numfile", O_RDWR|O_CREAT)   or die "can’t open numfile:
$!";
    flock(FH, LOCK_EX)      or die "can’t flock numfile: $!";
    $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 -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 above code.

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 systems 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.

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:

```
    $RECSIZE = 220; # size of record, in bytes
    $recno   = 37;  # which record to update
    open(FH, "+<somewhere") || die "can't update somewhere: $!";
    seek(FH, $recno * $RECSIZE, 0);
    read(FH, $record, $RECSIZE) == $RECSIZE || 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 **-M**, **-A**, 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* 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:

```
    $write_secs = (stat($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;
    $date_string = ctime(stat($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* 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";
    }
    $timestamp = shift;
    ($atime, $mtime) = (stat($timestamp))[8,9];
    utime $atime, $mtime, @ARGV;
```

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

Note that utime() currently doesn't work correctly with Win95/NT ports. A bug has been reported. Check it carefully before using utime() on those platforms.

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 $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

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

You can use the File::Slurp module to do it in one step.

```
 use File::Slurp;

 $all_of_it = read_file($filename); # entire file in scalar
    @all_lines = read_file($filename); # one line perl element
```

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

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

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

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

actually accesses the corresponding line in the file.

You can read the entire filehandle contents into a scalar.

```
    {
 local(*INPUT, $/);
 open (INPUT, $file)  || die "can't open $file: $!";
 $var = <INPUT>;
    }
```

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

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

For ordinary files you can also use the read function.

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

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

How can I read in a file by paragraphs?

Use the `$/` variable (see *perlvar* 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) {
 my $got;
 print "gimme: ";
 $got = getone();
 print "--> $got\n";
    }
    exit;

    BEGIN {
 use POSIX qw(:termios_h);

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

 $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(TTY, "</dev/tty");
    print "Gimme a char: ";
    ReadMode "raw";
    $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?

First try

```
seek(GWFILE, 0, 1);
```

The statement `seek(GWFILE, 0, 1)` doesn't change the current position, but it does clear the end-of-file condition on the handle, so that the next <GWFILE> 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(GWFILE); <GWFILE>; $curpos = tell(GWFILE)) {
```

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

If this still doesn't work, look into the POSIX module. POSIX defines the clearerr() method, which can remove the end of file condition on a filehandle. The method: read until end of file, clearerr(), read some more. Lather, rinse, repeat.

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

How do I dup() a filehandle in Perl?

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

```
open(LOG, ">>/foo/logfile");
open(STDERR, ">&LOG");
```

Or even with a literal numeric descriptor:

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

Note that "<&STDIN" makes a copy, but "<&=STDIN" make 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?

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';
$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():

```
   {
local *F;
open F, "<&=$fd" or die "Cannot reopen fd=$fd: $!";
close F;
   }
```

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?

Here's an 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?

Saying

```
print "@lines\n";
```

joins together the elements of `@lines` with a space between them. If `@lines` were `("little", "fluffy", "clouds")` then the above statement would print

```
little fluffy clouds
```

but if each element of `@lines` was a line of text, ending a newline character `("little\n", "fluffy\n", "clouds\n")` then it would print:

```
little
 fluffy
 clouds
```

If your array contains lines, just print them:

```
print @lines;
```

AUTHOR AND COPYRIGHT