
8

File Contents

*The most brilliant decision in all of Unix
was the choice of a single character
for the newline sequence.*

—Mike O'Dell, only half jokingly

8.0. Introduction

Before the Unix Revolution, every kind of data source and destination was inherently different. Getting two programs merely to understand each other required heavy wizardry and the occasional sacrifice of a virgin stack of punch cards to an itinerant mainframe repairman. This computational Tower of Babel made programmers dream of quitting the field to take up a less painful hobby, like autoflagellation.

These days, such cruel and unusual programming is largely behind us. Modern operating systems work hard to provide the illusion that I/O devices, network connections, process control information, other programs, the system console, and even users' terminals are all abstract streams of bytes called *files*. This lets you easily write programs that don't care where their input came from or where their output goes.

Because programs read and write via byte streams of simple text, every program can communicate with every other program. It is difficult to overstate the power and elegance of this approach. No longer dependent upon troglodyte gnomes with secret tomes of JCL (or COM) incantations, users can now create custom tools from smaller ones by using simple command-line I/O redirection, pipelines, and backticks.

Treating files as unstructured byte streams necessarily governs what you can do with them. You can read and write sequential, fixed-size blocks of data at any location in the file, increasing its size if you write past the current end. Perl uses

the standard C I/O library to implement reading and writing of variable-length records like lines, paragraphs, and words.

What can't you do to an unstructured file? Because you can't insert or delete bytes anywhere but at end of file, you can't change the length of, insert, or delete records. An exception is the last record, which you can delete by truncating the file to the end of the previous record. For other modifications, you need to use a temporary file or work with a copy of the file in memory. If you need to do this a lot, a database system may be a better solution than a raw file (see Chapter 14, *Database Access*).

The most common files are text files, and the most common operations on text files are reading and writing lines. Use `<FH>` (or the internal function implementing it, `readline`) to read lines, and use `print` to write them. These functions can also be used to read or write any record that has a specific record separator. Lines are simply records that end in `"\n"`.

The `<FH>` operator returns `undef` on error or when end of the file is reached, so use it in loops like this:

```
while (defined ($line = <DATAFILE>)) {
    chomp $line;
    $size = length $line;
    print "$size\n";           # output size of line
}
```

Because this is a common operation and that's a lot to type, Perl gives it a shorthand notation. This shorthand reads lines into `$_` instead of `$line`. Many other string operations use `$_` as a default value to operate on, so this is more useful than it may appear at first:

```
while (<DATAFILE>) {
    chomp;
    print length, "\n";       # output size of line
}
```

Call `<FH>` in scalar context to read the next line. Call it in list context to read all remaining lines:

```
@lines = <DATAFILE>;
```

Each time `<FH>` reads a record from a filehandle, it increments the special variable `$.` (the "current input record number"). This variable is only reset when `close` is called explicitly, which means that it's not reset when you reopen an already opened filehandle.

Another special variable is `$/`, the input record separator. It is set to `"\n"`, the default end-of-line marker. You can set it to any string you like, for instance `"\0"` to read null-terminated records. Read paragraphs by setting `$/` to the empty string,

" ". This is almost like setting `$/` to `"\n\n"`, in that blank lines function as record separators, but `" "` treats two or more consecutive empty lines as a single record separator, whereas `"\n\n"` returns empty records when more than two consecutive empty lines are read. Undefine `$/` to read the rest of the file as one scalar:

```
undef $/;
$whole_file = <FILE>;          # 'slurp' mode
```

The `-0` option to Perl lets you set `$/` from the command line:

```
% perl -040 -e '$word = <>; print "First word is $word\n";'
```

The digits after `-0` are the octal value of the single character that `$/` is to be set to. If you specify an illegal value (e.g., with `-0777`) Perl will set `$/` to `undef`. If you specify `-00`, Perl will set `$/` to `" "`. The limit of a single octal value means you can't set `$/` to a multibyte string, for instance, `"%\n"` to read *fortune* files. Instead, you must use a `BEGIN` block:

```
% perl -ne 'BEGIN { $/="%\n" } chomp; print if /Unix/i' fortune.dat
```

Use `print` to write a line or any other data. The `print` function writes its arguments one after another and doesn't automatically add a line or record terminator by default.

```
print HANDLE "One", "two", "three"; # "Onetwothree"
print "Baa baa black sheep.\n";    # Sent to default output handle
```

There is no comma between the filehandle and the data to print. If you put a comma in there, Perl gives the error message `"No comma allowed after filehandle"`. The default output handle is `STDOUT`. Change it with the `select` function. (See the introduction to Chapter 7, *File Access*.)

All systems use the virtual `"\n"` to represent a line terminator, called a *newline*. There is no such thing as a newline character. It is an illusion that the operating system, device drivers, C libraries, and Perl all conspire to preserve. Sometimes, this changes the number of characters in the strings you read and write. The conspiracy is revealed in Recipe 8.11.

Use the `read` function to read a fixed-length record. It takes three arguments: a filehandle, a scalar variable, and the number of bytes to read. It returns `undef` if an error occurred or else the number of bytes read. To write a fixed-length record, just use `print`.

```
$rv = read(HANDLE, $buffer, 4096)
      or die "Couldn't read from HANDLE : $!\n";
# $rv is the number of bytes read,
# $buffer holds the data read
```

The `truncate` function changes the length of a file, which can be specified as a filehandle or as a filename. It returns true if the file was successfully truncated, false otherwise:

```
truncate(HANDLE, $length)
    or die "Couldn't truncate: $!\n";
truncate("/tmp/$$.pid", $length)
    or die "Couldn't truncate: $!\n";
```

Each filehandle keeps track of where it is in the file. Reads and writes occur from this point, unless you've specified the `O_APPEND` flag (see Recipe 7.1). Fetch the file position for a filehandle with `tell`, and set it with `seek`. Because the stdio library rewrites data to preserve the illusion that `"\n"` is the line terminator, you cannot portably `seek` to offsets calculated by counting characters. Instead, only `seek` to offsets returned by `tell`.

```
$pos = tell(DATAFILE);
print "I'm $pos bytes from the start of DATAFILE.\n";
```

The `seek` function takes three arguments: the filehandle, the offset (in bytes) to go to, and a numeric argument indicating how to interpret the offset. 0 indicates an offset from the start of the file (the kind of value returned by `tell`); 1, an offset from the current location (a negative number means move backwards in the file, a positive number means move forward); and 2, an offset from end of file.

```
seek(LOGFILE, 0, 2)           or die "Couldn't seek to the end: $!\n";
seek(DATAFILE, $pos, 0)      or die "Couldn't seek to $pos: $!\n";
seek(OUT, -20, 1)           or die "Couldn't seek back 20 bytes: $!\n";
```

So far we've been describing buffered I/O. That is, `<FH>`, `print`, `read`, `seek`, and `tell` are all operations that use buffers for speed. Perl also provides unbuffered I/O operations: `sysread`, `syswrite`, and `sysseek`, all discussed in Chapter 7.

The `sysread` and `syswrite` functions are different from their `<FH>` and `print` counterparts. They both take a filehandle to act on, a scalar variable to either read into or write out from, and the number of bytes to read or write. They can also take an optional fourth argument, the offset in the scalar variable to start reading or writing at:

```
$written = syswrite(DATAFILE, $mystring, length($mystring));
die "syswrite failed: $!\n" unless $written == length($mystring);
$read = sysread(INFILE, $block, 256, 5);
warn "only read $read bytes, not 256" if 256 != $read;
```

The `syswrite` call sends the contents of `$mystring` to `DATAFILE`. The `sysread` call reads 256 bytes from `INFILE` and stores them 5 characters into `$block`, leaving its first 5 characters intact. Both `sysread` and `syswrite` return the number of bytes transferred, which could be different than the amount of data you were

attempting to transfer. Maybe the file didn't have all the data you thought it did, so you got a short read. Maybe the filesystem that the file lives on filled up. Maybe your process was interrupted part of the way through the write. Stdio takes care of finishing the transfer in cases of interruption, but if you use the `sysread` and `syswrite` calls, you must do it yourself. See Recipe 9.3 for an example of this.

The `sysseek` function doubles as an unbuffered replacement for both `seek` and `tell`. It takes the same arguments as `seek`, but it returns either the new position if successful or `undef` on error. To find the current position within the file:

```
$pos = sysseek(HANDLE, 0, 1);      # don't change position
die "Couldn't sysseek: $!\n" unless defined $pos;
```

These are the basic operations available to you. The art and craft of programming lies in using these basic operations to solve complex problems like finding the number of lines in a file, reversing the order of lines in a file, randomly selecting a line from a file, building an index for a file, and so on.

8.1. Reading Lines with Continuation Characters

Problem

You have a file with long lines split over two or more lines, with backslashes to indicate that a continuation line follows. You want to rejoin those split lines. Makefiles, shell scripts, and many other scripting or configuration languages let you break a long line into several shorter ones in this fashion.

Solution

Build up the complete lines one at a time until reaching one without a backslash:

```
while (defined($line = <FH>) ) {
    chomp $line;
    if ($line =~ s/\\$//) {
        $line .= <FH>;
        redo unless eof(FH);
    }
    # process full record in $line here
}
```

Discussion

Here's an example input file:

```
DISTFILES = $(DIST_COMMON) $(SOURCES) $(HEADERS) \
            $(TEXINFOS) $(INFOS) $(MANS) $(DATA)
```

```

DEP_DISTFILES = $(DIST_COMMON) $(SOURCES) $(HEADERS) \
                $(TEXINFOS) $(INFO_DEPS) $(MANS) $(DATA) \
                $(EXTRA_DIST)

```

You'd like to process that file with the escaped newlines ignored. That way the first record would in this case be the first two lines, the second record the next three lines, etc.

Here's how the algorithm works. The `while` loop reads lines, which may or may not be complete records—they might end in backslash (and a newline). The substitution operator `s///` tries to remove a trailing backslash. If the substitution fails, we've found a line without a backslash at the end. Otherwise, read another record, concatenate it onto the accumulating `$line` variable, and use `redo` to jump back to just inside the opening brace of the `while` loop. This lands us back on the `chomp`.

A common problem with files in this format is invisible blanks between the backslash and end of line. It would be more forgiving if the substitute were like this:

```

if ($line =~ s/\\s*$/) {
    # as before
}

```

Unfortunately, even if your program is forgiving, others doubtlessly aren't. Just remember to be liberal in what you accept and conservative in what you produce.

See Also

The `chomp` function in *perlfunc(1)* and in Chapter 3 of *Programming Perl*; the `redo` keyword in the “Loop Control” sections of *perlsyn(1)* and Chapter 2 of *Programming Perl*

8.2. Counting Lines (or Paragraphs or Records) in a File

Problem

You need to compute the number of lines in a file.

Solution

Many systems have a `wc` program to count lines in a file:

```

$count = `wc -l < $file`;
die "wc failed: $?" if $?;
chomp($count);

```

You could also open the file and read line-by-line until the end, counting lines as you go:

```
open(FILE, "< $file") or die "can't open $file: $!";
$count++ while <FILE>;
# $count now holds the number of lines read
```

Here's the fastest solution, assuming your line terminator really *is* "\n":

```
$count += tr/\n/\n/ while sysread(FILE, $_, 2 ** 16);
```

Discussion

Although you can use `-s $file` to determine the file size file in bytes, you generally cannot use it to derive a line count. See the Introduction to Chapter 9, *Directories*, for more on `-s`.

If you can't or don't want to call another program to do your dirty work, you can emulate `wc` by opening up and reading the file yourself:

```
open(FILE, "< $file") or die "can't open $file: $!";
$count++ while <FILE>;
# $count now holds the number of lines read
```

Another way of writing this is:

```
open(FILE, "< $file") or die "can't open $file: $!";
for ($count=0; <FILE>; $count++) { }
```

If you're not reading from any other files, you don't need the `$count` variable in this case. The special variable `$.` holds the number of lines read since a filehandle was last explicitly closed:

```
1 while <FILE>;
$count = $.;
```

This reads all the records in the file and discards them.

To count paragraphs, set the global input record separator variable `$/` to the empty string ("") before reading to make `<>` read a paragraph at a time.

```
$/ = ''; # enable paragraph mode for all reads
open(FILE, $file) or die "can't open $file: $!";
1 while <FILE>;
$para_count = $.;
```

See Also

Your system's `wc(1)` manpage; the `$/` entry in `perlvar(1)`, and in the "Special Variables" section of Chapter 2 of *Programming Perl*; the Introduction to Chapter 9

8.3. Processing Every Word in a File

Problem

You need to do something to every word in a file, similar to the `foreach` function of *csb*.

Solution

Either `split` each line on whitespace:

```
while (<>) {
    for $chunk (split) {
        # do something with $chunk
    }
}
```

Or use the `m//g` operator to pull out one chunk at a time:

```
while (<>) {
    while ( /(\w[\w'-]*)/g ) {
        # do something with $1
    }
}
```

Discussion

Decide what you mean by “word.” Sometimes you want anything but whitespace, sometimes you only want program identifiers, and sometimes you want English words. Your definition governs which regular expression to use.

The preceding two approaches work differently. Patterns are used in the first approach to decide what is *not* a word. In the second, they’re used to decide what *is* a word.

With these techniques, it’s easy to make a word frequency counter. Use a hash to store how many times each word has been seen:

```
# Make a word frequency count
%seen = ();
while (<>) {
    while ( /(\w['\w-]*)/g ) {
        $seen{lc $1}++;
    }
}

# output hash in a descending numeric sort of its values
foreach $word ( sort { $seen{$b} <=> $seen{$a} } keys %seen) {
    printf "%5d %s\n", $seen{$word}, $word;
}
```

To make the example program count line frequency instead of word frequency, omit the second `while` loop and do `$seen{lc $_}++` instead:

```
# Line frequency count
%seen = ();
while (<>) {
    $seen{lc $_}++;
}
foreach $line ( sort { $seen{$b} <=> $seen{$a} } keys %seen ) {
    printf "%5d %s", $seen{$line}, $line;
}
```

Odd things that may need to be considered as words include “M.I.T.”, “Micro\$oft”, “o’clock”, “49ers”, “street-wise”, “and/or”, “&”, “c/o”, “St.”, “Tschüß”, and “Niño”. Bear this in mind when you choosing a pattern to match. The last two require you to place a `use locale` in your program and then employ `\w` for a word character in the current locale.

See Also

The `split` function in *perlfunc(1)* and in Chapter 3 of *Programming Perl*; Recipe 6.3; Recipe 6.23

8.4. Reading a File Backwards by Line or Paragraph

Problem

You want to process each line or paragraph of a text file in reverse.

Solution

Read all lines into an array, then process that array from the end to the start:

```
@lines = <FILE>;
while ($line = pop @lines) {
    # do something with $line
}
```

Or store an array of lines in reverse order:

```
@lines = reverse <FILE>;
foreach $line (@lines) {
    # do something with $line
}
```

Discussion

The limitations of file access mentioned in this chapter's Introduction prevent you from reading a line at a time starting from the end. You must read the lines into memory, then process them in reverse order. Needless to say, this requires at least as much available memory as the size of the file.

The first technique moves through the array of lines, in reverse order. This *destructively* processes the array, popping an element off the end of the array each time through the loop. We could do it non-destructively with:

```
for ($i = $#lines; $i != -1; $i--) {
    $line = $lines[$i];
}
```

The second approach generates an array of lines that is already in reverse order. This array can then be processed non-destructively. We get the reversed lines because the assignment to `@lines` forces list context on `reverse`, which in turn forces it on `<FILE>`. `<>` in a list context returns a list of all lines in the file.

These approaches are easily extended to paragraphs just by changing `$/`:

```
# this enclosing block keeps local $/ temporary
{
    local $/ = '';
    @paragraphs = reverse <FILE>;
}

foreach $paragraph (@paragraphs) {
    # do something
}
```

See Also

The `reverse` function in *perlfunc(1)* and in Chapter 3 of *Programming Perl*; the `$/` entry in *perlvar(1)*, and in the “Special Variables” section of Chapter 2 of *Programming Perl*; Recipe 4.10; Recipe 1.6

8.5. Trailing a Growing File

Problem

You want to read from a continually growing file, but the read fails when you reach the (current) end of file.

Solution

Read until the end of file. Sleep, clear the EOF flag, and read some more. Repeat until interrupted. To clear the EOF flag, either use `seek`:

```
for (;;) {
    while (<FH>) { .... }
    sleep $SOMETIME;
    seek(FH, 0, 1);
}
```

or the `IO::Handle` module's `clearerr` method:

```
use IO::Seekable;

for (;;) {
    while (<FH>) { .... }
    sleep $SOMETIME;
    FH->clearerr();
}
```

Discussion

When you read to the end of a file, an internal flag is set that prevents further reading. The most direct way to clear this flag is the `clearerr` method, if supported: it's in the `IO::Handle` and `FileHandle` modules.

```
$naptime = 1;

use IO::Handle;
open (LOGFILE, "/tmp/logfile") or die "can't open /tmp/logfile: $!";
for (;;) {
    while (<LOGFILE>) { print }      # or appropriate processing
    sleep $naptime;
    LOGFILE->clearerr();            # clear stdio error flag
}
```

If that simple approach doesn't work on your system, you may need to use `seek`. The `seek` code given above tries to move zero bytes from the current position, which nearly always works. It doesn't change the current position, but it should clear the end-of-file condition on the handle so that the next `<LOGFILE>` picks up new data.

If that still doesn't work (e.g., it relies on features of your C library's (so-called) standard I/O implementation), then you may need to use the following `seek` code, which remembers the old file position explicitly and returns there directly.

```
for (;;) {
    for ($curpos = tell(LOGFILE); <LOGFILE>; $curpos = tell(LOGFILE)) {
        # process $_ here
    }
}
```

```

    sleep $naptime;
    seek(LOGFILE, $curpos, 0); # seek to where we had been
}

```

On some kinds of filesystems, the file could be removed while you are reading it. If so, there's probably little reason to continue checking whether it grows. To make the program exit in that case, `stat` the handle and make sure its link count (the third field in the return list) hasn't gone to 0:

```
exit if (stat(LOGFILE))[3] == 0
```

If you're using the `File::stat` module, you could write that more readably as:

```

use File::stat;
exit if stat(*LOGFILE)->nlink == 0;

```

See Also

The `seek` function in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; your system's *tail*(1) and *stdio*(3) manpages

8.6. Picking a Random Line from a File

Problem

You want to return a random line from a file.

Solution

Use `rand` and `$.` (the current line number) to decide which line to print:

```

srand;
rand($.) < 1 && ($line = $_) while <>;
# $line is the random line

```

Discussion

This is a beautiful example of a solution that may not be obvious. We read every line in the file but don't have to store them all in memory. This is great for large files. Each line has a 1 in N (where N is the number of lines read so far) chance of being selected.

Here's a replacement for *fortune* using this algorithm:

```

$/ = "%%\n";
$data = '/usr/share/games/fortunes';
srand;
rand($.) < 1 && ($adage = $_) while <>;
print $adage;

```

If you know line offsets (for instance, you've created an index) and the number of lines, you can randomly select a line and jump to its offset in the file. But you usually don't have such an index.

Here's a more rigorous explanation of how the algorithm works. The function call `rand ($.)` picks a random number between 0 and the current line number. Therefore, you have a one in N chance, that is, $\frac{1}{N}$, of keeping the Nth line. Therefore you've a 100% chance of keeping the first line, a 50% chance of keeping the second, a 33% chance of keeping the third, and so on. The question is whether this is fair for all N, where N is any positive integer.

First, some concrete examples, then abstract ones.

Obviously, a file with one line (N=1) is fair: you always keep the first line because $\frac{1}{1} = 100\%$, making it fair for files of 1 line. For a file with two lines, N=2. You always keep the first line; then when reaching the second line, you have a 50% chance of keeping it. Thus, both lines have an equal chance of being selected, which shows that N=2 is fair. For a file with three lines, N=3. You have a one-third chance, 33%, of keeping that third line. That leaves a two-thirds chance of retaining one of the first two out of the three lines. But we've already shown that for those first two lines there's a 50-50 chance of selecting either one. 50 percent of two-thirds is one-third. Thus, you have a one-third chance of selecting each of the three lines of the file.

In the general case, a file of N+1 lines will choose the last line $\frac{1}{N+1}$ times and one of the previous N lines $\frac{N}{N+1}$ times. Dividing $\frac{N}{N+1}$ by N leaves us with $\frac{1}{N+1}$ for each the first N lines in our N+1 line file, and also $\frac{1}{N+1}$ for line number N+1. The algorithm is therefore fair for all N, where N is a positive integer.

We've managed to choose fairly a random line from a file with speed directly proportional to the size of the file, but using no more memory than it takes to hold the longest line, even in the worst case.

See Also

The `$.` entry in *perlvar*(1) and in the "Special Variables" section of Chapter 2 of *Programming Perl*; Recipe 2.7; Recipe 2.8

8.7. Randomizing All Lines

Problem

You want to copy a file and randomly reorder its lines.

Solution

Read all lines into an array, shuffle the array using the algorithm from Recipe 4.17, and write the shuffled lines back out:

```
# assumes the &shuffle sub from Chapter 4
while (<INPUT>) {
    push(@lines, $_);
}
@reordered = shuffle(@lines);
foreach (@reordered) {
    print OUTPUT $_;
}
```

Discussion

The easiest approach is to read all lines into memory and shuffle them there. Because you don't know where lines start in the file, you can't just shuffle a list of line numbers and then extract the lines in the order they'll appear in the shuffled file. Even if you *did* know their starts, it would probably still be slower because you'd be seeking around in the file instead of simply reading it from start to finish.

See Also

Recipe 2.7; Recipe 2.8; Recipe 4.17

8.8. Reading a Particular Line in a File

Problem

You want to extract a single line from a file.

Solution

The simplest solution is to read the lines until you get to the one you want:

```
# looking for line number $DESIRED_LINE_NUMBER
$. = 0;
do { $LINE = <HANDLE> } until $. == $DESIRED_LINE_NUMBER || eof;
```

If you are going to be doing this a lot and the file fits into memory, read the file into an array:

```
@lines = <HANDLE>;
$LINE = $lines[$DESIRED_LINE_NUMBER];
```

If you will be retrieving lines by number often and the file doesn't fit into memory, build a byte-address index to let you `seek` directly to the start of the line:

```

# usage: build_index(*DATA_HANDLE, *INDEX_HANDLE)
sub build_index {
    my $data_file = shift;
    my $index_file = shift;
    my $offset = 0;

    while (<$data_file> {
        print $index_file pack("N", $offset);
        $offset = tell($data_file);
    }
}

# usage: line_with_index(*DATA_HANDLE, *INDEX_HANDLE, $LINE_NUMBER)
# returns line or undef if LINE_NUMBER was out of range
sub line_with_index {
    my $data_file = shift;
    my $index_file = shift;
    my $line_number = shift;

    my $size;           # size of an index entry
    my $i_offset;      # offset into the index of the entry
    my $entry;         # index entry
    my $d_offset;      # offset into the data file

    $size = length(pack("N", 0));
    $i_offset = $size * ($line_number-1);
    seek($index_file, $i_offset, 0) or return;
    read($index_file, $entry, $size);
    $d_offset = unpack("N", $entry);
    seek($data_file, $d_offset, 0);
    return scalar(<$data_file>);
}

# usage:
open(FILE, "< $file")           or die "Can't open $file for reading: $!\n";
open(INDEX, "+>$file.idx")     or die "Can't open $file.idx for read/write: $!\n";
build_index(*FILE, *INDEX);
$line = line_with_index(*FILE, *INDEX, $seeking);

```

If you have the `DB_File` module, its `DB_RECNO` access method ties an array to a file, one line per array element:

```

use DB_File;
use Fcntl;

$tie = tie(@lines, $FILE, O_RDWR, 0666, $DB_RECNO) or die
    "Cannot open file $FILE: $!\n";
# extract it
$line = $lines[$sought-1];

```

Discussion

Each strategy has different features, useful in different circumstances. The linear access approach is easy to write and best for short files. The index method gives quick two-step lookup, but requires that the index be pre-built, so it is best when the file being indexed doesn't change often compared to the number of lookups. The DB_File mechanism has some initial overhead, but subsequent accesses are much faster than with linear access, so use it for long files that are accessed more than once and are accessed out of order.

It is important to know whether you're counting lines from 0 or 1. The `$.` variable is 1 after the first line is read, so count from 1 when using linear access. The index mechanism uses lots of offsets, so count from 0. DB_File treats the file's records as an array indexed from 0, so count lines from 0.

Here are three different implementations of the same program, *print_line*. The program takes two arguments, a filename, and a line number to extract.

The version in Example 8-1 simply reads lines until it finds the one it's looking for.

Example 8-1. *print_line-v1*

```
#!/usr/bin/perl -w
# print_line-v1 - linear style

@ARGV == 2 or die "usage: print_line FILENAME LINE_NUMBER\n";

($filename, $line_number) = @ARGV;
open(INFILE, "< $filename") or die "Can't open $filename for reading: !\n";
while (<INFILE>) {
    $line = $_;
    last if $. == $line_number;
}
if ($. != $line_number) {
    die "Didn't find line $line_number in $filename\n";
}
print;
```

The index version in Example 8-2 must build an index. For many lookups, you could build the index once and then use it for all subsequent lookups:

Example 8-2. *print_line-v2*

```
#!/usr/bin/perl -w
# print_line-v2 - index style
# build_index and line_with_index from above
@ARGV == 2 or
    die "usage: print_line FILENAME LINE_NUMBER";

($filename, $line_number) = @ARGV;
open(ORIG, "< $filename")
```

Example 8-2. *print_line-v2* (continued)

```

    or die "Can't open $filename for reading: $!";

# open the index and build it if necessary
# there's a race condition here: two copies of this
# program can notice there's no index for the file and
# try to build one. This would be easily solved with
# locking
$indexname = "$filename.index";
sysopen(IDX, $indexname, O_CREAT|O_RDWR)
    or die "Can't open $indexname for read/write: $!";
build_index(*ORIG, *IDX) if -z $indexname; # XXX: race unless lock

$line = line_with_index(*ORIG, *IDX, $line_number);
die "Didn't find line $line_number in $filename" unless defined $line;
print $line;

```

The DB_File version in Example 8-3 is indistinguishable from magic.

Example 8-3. *print_line-v3*

```

#!/usr/bin/perl -w
# print_line-v3 - DB_File style
use DB_File;
use Fcntl;

@ARGV == 2 or
    die "usage: print_line FILENAME LINE_NUMBER\n";

($filename, $line_number) = @ARGV;
$tie = tie(@lines, "DB_File", $filename, O_RDWR, 0666, $DB_RECNO)
    or die "Cannot open file $filename: $!\n";

unless ($line_number < $tie->length) {
    die "Didn't find line $line_number in $filename\n"
}

print $lines[$line_number-1];          # easy, eh?

```

See Also

The documentation for the standard DB_File module (also in Chapter 7 of *Programming Perl*); the tie function in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; the entry on \$. in *perlvar*(1) and in the “Special Variables” section of Chapter 2 of *Programming Perl*

8.9. Processing Variable-Length Text Fields

Problem

You want to extract variable length fields from your input.

Solution

Use `split` with a pattern matching the field separators.

```
# given $RECORD with field separated by PATTERN,
# extract @FIELDS.
@FIELDS = split(/PATTERN/, $RECORD);
```

Discussion

The `split` function takes up to three arguments: `PATTERN`, `EXPRESSION`, and `LIMIT`. The `LIMIT` parameter is the maximum number of fields to split into. (If the input contains more fields, they are returned unsplit in the final list element.) If `LIMIT` is omitted, all fields (except any final empty ones) are returned. `EXPRESSION` gives the string value to split. If `EXPRESSION` is omitted, `$_` is split. `PATTERN` is a pattern matching the field separator. If `PATTERN` is omitted, contiguous stretches of whitespace are used as the field separator and leading empty fields are silently discarded.

If your input field separator isn't a fixed string, you might want `split` to return the field separators as well as the data by using parentheses in `PATTERN` to save the field separators. For instance:

```
split(/[+-]/, "3+5-2");
```

returns the values:

```
(3, '+', 5, '-', 2)
```

To split colon-separated records in the style of the `/etc/passwd` file, use:

```
@fields = split(/:/, $RECORD);
```

The classic application of `split` is whitespace-separated records:

```
@fields = split(/\s+/, $RECORD);
```

If `$RECORD` started with whitespace, this last use of `split` would have put an empty string into the first element of `@fields` because `split` would consider the record to have an initial empty field. If you didn't want this, you could use this special form of `split`:

```
@fields = split(" ", $RECORD);
```

This behaves like `split` with a pattern of `/\s+/,` but ignores leading whitespace.

When the record separator can appear in the record, you have a problem. The usual solution is to escape occurrences of the record separator in records by prefixing them with a backslash. See Recipe 1.13.

See Also

The `split` function in *perlfunc*(1) and in Chapter 3 of *Programming Perl*

8.10. Removing the Last Line of a File

Problem

You'd like to remove the last line from a file.

Solution

Read the file a line at a time and keep track of the byte address of the last line you've seen. When you've exhausted the file, truncate to the last address you saved:

```
open (FH, "+< $file")           or die "can't update $file: $!";
while ( <FH> ) {
    $addr = tell(FH) unless eof(FH);
}
truncate(FH, $addr)             or die "can't truncate $file: $!";
```

Discussion

This is much more efficient than reading the file into memory all at once, since it only holds one line at a time. Although you still have to grope your way through the whole file, you can use this program on files larger than available memory.

See Also

The `open` and `binmode` functions in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; your system's *open*(2) and *fopen*(3) manpages

8.11. Processing Binary Files

Problem

Your system distinguishes between text and binary files. How do you?

Solution

Use the `binmode` function on the filehandle:

```
binmode(HANDLE);
```

Discussion

Not everyone agrees what constitutes a line in a text file, because one person's textual character set is another's binary gibberish. Even when everyone is using ASCII instead of EBCDIC, Rad50, or Unicode, discrepancies arise.

As mentioned in the Introduction, there is no such thing as a newline character. It is purely virtual, a figment of the operating system, standard libraries, device drivers, and Perl.

Under Unix or Plan9, a `"\n"` represents the physical sequence `"\cJ"` (the Perl double-quote escape for Ctrl-J), a linefeed. However, on a terminal that's not in raw mode, an Enter key generates an incoming `"\cM"` (a carriage return) which turns into `"\cJ"`, whereas an outgoing `"\cJ"` turns into `"\cM\cJ"`. This strangeness doesn't happen with normal files, just terminal devices, and it is handled strictly by the device driver.

On a Mac, a `"\n"` is usually represented by `"\cM"`; just to make life interesting (and because the standard requires that `"\n"` and `"\r"` be different), a `"\r"` represents a `"\cJ"`. This is exactly the opposite of the way that Unix, Plan9, VMS, CP/M, or nearly anyone else does it. So, Mac programmers writing files for other systems or talking over a network have to be careful. If you send out `"\n"`, you'll deliver a `"\cM"`, and no `"\cJ"` will be seen. Most network services prefer to receive and send `"\cM\cJ"` as a line terminator, but most accept merely a `"\cJ"`.

Under VMS, DOS, or their derivatives, a `"\n"` represents `"\cJ"`, similar to Unix and Plan9. From the perspective of a tty, Unix and DOS behave identically: a user who hits Enter generates a `"\cM"`, but this arrives at the program as a `"\n"`, which is `"\cJ"`. A `"\n"` (that's a `"\cJ"`, remember) sent to a terminal shows up as a `"\cM\cJ"`.

These strange conversions happen to Windows files as well. A DOS text file actually physically contains two characters at the end of every line, `"\cM\cJ"`. The last block in the file has a `"\cZ"` to indicate where the text stops. When you write a line like `"bad news\n"` on those systems, the file contains `"bad news\cM\cJ"`, just as if it were a terminal.

When you read a line on such systems, it's even stranger. The file itself contains `"bad news\cM\cJ"`, a 10-byte string. When you read it in, your program gets nothing but `"bad news\n"`, where that `"\n"` is the virtual newline character, that is, a linefeed (`"\cJ"`). That means to get rid of it, a single `chop` or `chomp` will do.

But your poor program has been tricked into thinking it's only read nine bytes from the file. If you were to read 10 such lines, you would appear to have read just 90 bytes into the file, but in fact would be at position 100. That's why the `tell` function must always be used to determine your location. You can't infer your position just by counting what you've read.

This legacy of the old CP/M filesystem, whose equivalent of a Unix inode stored only block counts and not file sizes, has frustrated programmers for decades, and no end is in sight. Because DOS is compatible with CP/M file formats, Windows with DOS, and NT with Windows, the sins of the fathers have truly been visited unto the children of the fourth generation.

You can circumvent the single `"\n"` terminator by telling Perl (and the operating system) that you're working with binary data. The `binmode` function indicates that data read or written through the given filehandle should not be mangled the way a text file would likely be on those systems.

```
$gifname = "picture.gif";
open(GIF, $gifname)      or die "can't open $gifname: $!";

binmode(GIF);           # now DOS won't mangle binary input from GIF
binmode(STDOUT);       # now DOS won't mangle binary output to STDOUT

while (read(GIF, $buff, 8 * 2**10)) {
    print STDOUT $buff;
}
```

Calling `binmode` on systems that don't make this distinction (including Unix, the Mac, and Plan 9) is harmless. Inappropriately doing so (such as on a text file) on systems that do (including MVS, VMS, and DOS, regardless of its GUI) can mangle your files.

If you're *not* using `binmode`, the data you read using `stdio` (`<>`) will automatically have the native system's line terminator changed to `"\n"`, even if you change `$/`. Similarly, any `"\n"` you `print` to the filehandle will be turned into the native line terminator. See this chapter's Introduction for more details.

If you want to get what was on the disk, byte for byte, you should set `binmode` if you're on one of the odd systems listed above. Then, of course, you also have to set `$/` to the real record separator if you want to use `<>` on it.

See Also

The `open` and `binmode` functions in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; your system's *open*(2) and *fopen*(3) manpages

8.12. Using Random-Access I/O

Problem

You have to read a binary record from the middle of a large file but don't want to read a record at a time to get there.

Solution

Once you know the record's size, multiply it by the record number to get the byte address, and then seek to that byte address and read the record:

```
$ADDRESS = $RECSIZE * $RECNO;
seek(FH, $ADDRESS, 0) or die "seek:$!";
read(FH, $BUFFER, $RECSIZE);
```

Discussion

The Solution assumes the first record has a RECNO of 0. If you're counting from one, use:

```
$ADDRESS = $RECSIZE * ($RECNO-1);
```

This won't work on a text file unless all lines are the same length. This is rarely the case.

See Also

The `seek` function in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; Recipe 8.13

8.13. Updating a Random-Access File

Problem

You want to read in an old record from a binary file, change its values, and write back the record.

Solution

After reading the old record, pack up the updated values, seek to the previous address, and write it back.

```
use Fcntl;                                     # for SEEK_SET and SEEK_CUR

$ADDRESS = $RECSIZE * $RECNO;
```

```

seek(FH, $ADDRESS, SEEK_SET)          or die "Seeking: $!";
read(FH, $BUFFER, $RECSIZE) == $RECSIZE
                                       or die "Reading: $!";
@FIELDS = unpack($FORMAT, $BUFFER);
# update fields, then
$BUFFER = pack($FORMAT, @FIELDS);
seek(FH, -$RECSIZE, SEEK_CUR)        or die "Seeking: $!";
print FH $BUFFER;
close FH                              or die "Closing: $!";

```

Discussion

You don't have to use anything fancier than `print` in Perl to output a record. Remember that the opposite of `read` is not `write` but `print`, although oddly enough, the opposite of `sysread` actually is `syswrite`. (`split` and `join` are opposites, but there's no `speak` to match `listen`, no `resurrect` for `kill`, and no `curse` for `bless`.)

The example program shown in Example 8-4, *weekearly*, takes one argument: the user whose record you want to backdate by a week. (Of course, in practice, you wouldn't really want to (nor be able to!) mess with the system accounting files.) This program requires write access to the file to be updated, since it opens the file in update mode. After fetching and altering the record, it packs it up again, skips backwards in the file one record, and writes it out.

Example 8-4. *weekearly*

```

#!/usr/bin/perl
# weekearly -- set someone's login date back a week
use User::pwent;
use IO::Seekable;

$typedef = 'L A12 A16';          # linux fmt; sunos is "L A8 A16"
$sizeof = length(pack($typedef, ()));
$user    = shift(@ARGV) || $ENV{USER} || $ENV{LOGNAME};

$address = getpwnam($user)->uid * $sizeof;

open (LASTLOG, "+</var/log/lastlog")
    or die "can't update /usr/adm/lastlog: $!";
seek(LASTLOG, $address, SEEK_SET)
    or die "seek failed: $!";
read(LASTLOG, $buffer, $sizeof) == $sizeof
    or die "read failed: $!";

($time, $line, $host) = unpack($typedef, $buffer);
$time    -= 24 * 7 * 60 * 60;          # back-date a week
$buffer = pack($typedef, $time, $line, $time);

seek(LASTLOG, -$sizeof, SEEK_CUR)    # backup one record
    or die "seek failed: $!";
print LASTLOG $record;

```

Example 8-4. weekearly (continued)

```
close(LASTLOG)
    or die "close failed: $!";
```

See Also

The `open`, `seek`, `read`, `pack`, and `unpack` functions in the *perlfunc(1)* and in Chapter 3 of *Programming Perl*; Recipe 8.12; Recipe 8.14

8.14. Reading a String from a Binary File

Problem

You want to read a NUL-terminated string from a file, starting at a particular address.

Solution

Set `$/` to an ASCII NUL, and read the string with `<>`:

```
$old_rs = $/;                # save old $/
$/ = "\0";                  # NULL
seek(FH, $addr, SEEK_SET)   or die "Seek error: $!\n";
$string = <FH>;              # read string
chomp $string;              # remove NULL
$/ = $old_rs;               # restore old $/
```

You can use `local` to save and restore `$/` if you want:

```
{
    local $/ = "\0";
    # ...
}                                # $/ is automatically restored
```

Discussion

The example program shown in Example 8-5, *bgets*, accepts a filename and one or more byte addresses as arguments. Decimal, octal, or hexadecimal addresses may be specified. For each address, the program reads and prints the NULL- or EOF-terminated string at that position:

Example 8-5. bgets

```
#!/usr/bin/perl
# bgets - get a string from an address in a binary file
use IO::Seekable;
($file, @addr) = @ARGV          or die "usage: $0 addr ...";
open(FH, $file)                 or die "cannot open $file: $!";
$/ = "\000";
```

Example 8-5. *bgets* (continued)

```
foreach $addr (@addrs) {
    $addr = oct $addr if $addr =~ /^0/;
    seek(FH, $addr, SEEK_SET)
        or die "can't seek to $addr in $file: $!";
    printf qq{%#x %#o %d "%s"\n}, $addr, $addr, $addr, scalar <>;
}
```

Here's a simple implementation of the Unix *strings* program:

Example 8-6. *strings*

```
#!/usr/bin/perl
# strings - pull strings out of a binary file
$/ = "\0";
while (<>) {
    while (/([\040-\176\s]{4,})/g) {
        print $1, "\n";
    }
}
```

See Also

The `seek`, `getc`, and `ord` functions in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; the discussion of `qq//` in the “Quote and Quote-Like Operators” section of the *perlop*(1) manpage, and in the “Pick Your Own Quotes” section of Chapter 2 of *Programming Perl*

8.15. Reading Fixed-Length Records

Problem

You want to read a file whose records have a fixed length.

Solution

Use `read` and `unpack`:

```
# $RECORDSIZE is the length of a record, in bytes.
# $TEMPLATE is the unpack template for the record
# FILE is the file to read from
# @FIELDS is an array, one element per field

until ( eof(FILE) ) {
    read(FILE, $record, $RECORDSIZE) == $RECORDSIZE
        or die "short read\n";
    @FIELDS = unpack($TEMPLATE, $record);
}
```

Discussion

Because the file in question is not a text file, you can't use `<FH>` or `IO::` modules' `getline` method to read in records. Instead, you must simply `read` a particular number of bytes into a buffer. This buffer then contains one record's data, which you decode using `unpack` with the right format.

For binary data, the catch is often determining the right format. If you're reading data written by a C program, this can mean peeking at C include files or manpages describing the structure layout, and this requires knowledge of C. It also requires that you become unnaturally chummy with your C compiler, because otherwise it's hard to predict field padding and alignment (such as the `x2` in the format used in Recipe 8.18). If you're lucky enough to be on a Berkeley Unix system or a system supporting `gcc`, then you may be able to use the `c2pb` tool distributed with Perl to cajole your C compiler into helping you with this.

The `tailwtmp` program at the end of this chapter uses the format described in `utmp(5)` under Linux and works on its `/var/log/wtmp` and `/var/run/utmp` files. Once you commit to working in a binary format, machine dependencies creep in fast. It probably won't work unaltered on your system, but the procedure is still illustrative. Here is the relevant layout from the C include file on Linux:

```
#define UT_LINESIZE      12
#define UT_NAMESIZE     8
#define UT_HOSTSIZE     16

struct utmp {
    short ut_type;           /* s for short, must be padded */
    pid_t ut_pid;          /* i for integer */
    char ut_line[UT_LINESIZE]; /* A12 for 12-char string */
    char ut_id[2];         /* A2, but need x2 for alignment */
    time_t ut_time;       /* l for long */
    char ut_user[UT_NAMESIZE]; /* A8 for 8-char string */
    char ut_host[UT_HOSTSIZE]; /* A16 for 16-char string */
    long ut_addr;         /* l for long */
};
```

Once you figure out the binary layout, feed that (in this case, "`s x2 i A12 A2 x2 l A8 A16 l`") to `pack` with an empty field list to determine the record's size. Remember to check the return value of `read` when you read in your record to make sure you got back the number of bytes you asked for.

If your records are text strings, use the `"a"` or `"A"` unpack templates.

Fixed-length records are useful in that the n th record begins at byte offset `SIZE * (n-1)` in the file, where `SIZE` is the size of a single record. See the indexing code in Recipe 8.8 for an example of this.

See Also

The `unpack`, `pack`, and `read` functions in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; Recipe 1.1

8.16. Reading Configuration Files

Problem

You want to allow the users of your program to change its behavior through configuration files.

Solution

Either process a file in trivial VAR=VALUE format, setting a hash key-value pair for each setting:

```
while (<CONFIG>) {
    chomp;                # no newline
    s/#.*//;             # no comments
    s/^\s+//;            # no leading white
    s/\s+$//;           # no trailing white
    next unless length;  # anything left?
    my ($var, $value) = split(/\s*=\s*/, $_, 2);
    $User_Preferences{$var} = $value;
}
```

Or better yet, treat the config file as full Perl code:

```
do "$ENV{HOME}/.progrc";
```

Discussion

The first solution lets you read in config files in a trivial format like this (comments and blank lines are allowed):

```
# set class C net
NETMASK = 255.255.255.0
MTU     = 296

DEVICE  = cua1
RATE   = 115200
MODE   = adaptive
```

After you're done, you can pull in a setting by something like `$User_Preferences{"RATE"}` to find the value 115200. If you wanted the config file to directly

set a variable in your program using that name, instead of assigning to the hash, do this:

```
no strict 'refs';
$$var = $value;
```

and the `$RATE` variable would contain 115200.

The second solution uses `do` to pull in raw Perl code directly. When used with an expression instead of a block, `do` interprets the expression as a filename. This is nearly identical to using `require`, but without risk of taking a fatal exception. In the second format, the config file would look like:

```
# set class C net
$NETMASK = '255.255.255.0';
$MTU      = 0x128;
# Brent, please turn on the modem
$DEVICE   = 'cua1';
$RATE     = 115_200;
$MODE     = 'adaptive';
```

If you don't see the point of having extra punctuation and live code, consider this: you can have all of Perl at your disposal. You can now add arbitrary logic and tests to your simple assignments:

```
if ($DEVICE =~ /1$/) {
    $RATE = 28_800;
} else {
    $RATE = 115_200;
}
```

Many programs support system and personal configuration files. If you want the user's choices to override the system ones, load the user file second:

```
$APPPDFLT = "/usr/local/share/myprog";

do "$APPPDFLT/sysconfig.pl";
do "$ENV{HOME}/.myprogrc";
```

If you want to ignore the system config file when the user has their own, test the return value of the `do`.

```
do "$APPPDFLT/sysconfig.pl"
or
do "$ENV{HOME}/.myprogrc";
```

You might wonder what context those files will be executed under. They will be in the same package that `do` itself was compiled into. Typically you'll direct users to set particular variables, which, being unqualified globals, will end up in the current package. If you'd prefer unqualified variables go into a particular package, do this:

```
{ package Settings; do "$ENV{HOME}/.myprogrc" }
```

As with a file read in using `require` or `use`, those read in using `do` count as a separate and unrelated lexical scope. That means the configuration file can't access its caller's lexical (`my`) variables, nor can the caller find any such variables that might have been set in the file. It also means that the user's code isn't held accountable to a pragma like `use strict` or `use integer` that may be in effect in the caller.

If you don't want clean partitioning of variable visibility, you can get the config file's code executed in your own lexical scope. If you have a `cat` program or its technical equivalent handy, you could write yourself a hand-rolled `do`:

```
eval `cat $ENV{HOME}/.myprogrc`;
```

We've never actually seen anyone (except Larry) use that approach in production code.

For one thing, `do` is a lot easier to type. Also, it respects the `@INC` path, which is normally searched if a full path is not specified, but, unlike using a `require`, no implicit error checking happens under `do`. This means you don't have to wrap it in an `eval` to catch exceptions that would otherwise cause your program to die, because `do` already functions as an `eval`.

You can still check for errors on your own if you'd like:

```
$file = "someprog.pl";
unless ($return = do $file) {
    warn "couldn't parse $file: $@"      if $@;
    warn "couldn't do $file: $!"        unless defined $return;
    warn "couldn't run $file"          unless $return;
}
```

This is much simpler for the programmer to source in code than it would be to invent and then parse a complicated, new syntax. It's also much easier on the user than forcing them to learn the syntax rules of yet another configuration file. Even better, you give the user access to a powerful algorithmic programming language.

One reasonable concern is security. How do you know that the file hasn't been tampered with by someone other than the user? The traditional approach here is to do nothing, trusting the directory and file permissions. Nine times out of ten, this is also the right approach. Most projects just aren't worth being that paranoid over. For those that are, see the next recipe.

See Also

The `eval` and `require` functions in *perlfunc*(1) and in Chapter 3 of *Programming Perl*; Recipe 8.17; Recipe 10.12

8.17. Testing a File for Trustworthiness

Problem

You want to read from a file, perhaps because it has configuration information. You only want to use the file if it can't be written to (or perhaps not even be read from) by anyone else than its owner.

Solution

Use the `stat` call to retrieve ownership and file permissions information. You can use the built-in version, which returns a list:

```
( $dev, $ino, $mode, $nlink,
  $uid, $gid, $rdev, $size,
  $atime, $mtime, $ctime,
  $blksize, $blocks ) = stat($filename)
  or die "no $filename: $!";

$mode &= 07777;          # discard file type info
```

Or you can use the by-name interface in:

```
$info = stat($filename)    or die "no $filename: $!";
if ($info->uid == 0) {
    print "Superuser owns $filename\n";
}
if ($info->atime > $info->mtime) {
    print "$filename has been read since it was written.\n";
}
```

Description

Usually you trust users to set file permissions as they wish. If they want others to read their files, or even to write to them, that's their business. Applications like editors, mailers, and shells are often more discerning, though, refusing to evaluate code in configuration files if anyone but the owner can write to them. This helps avoid Trojan horses attacks. Security-minded programs like *ftp* and *rlogin* may even reject config files that can be read by anyone but their owner.

If the file is writable by someone other than the owner or is owned by someone other than the current user or the superuser, it shouldn't be trusted. To figure out file ownership and permissions, the `stat` function is used. The following function returns true if the file is deemed safe and false otherwise. If the `stat` fails, `undef` is returned.

```
use File::stat;

sub is_safe {
```

```

my $path = shift;
my $info = stat($path);
return unless $info;

# owner neither superuser nor me
# the real uid is in stored in the $< variable
if (($info->uid != 0) && ($info->uid != $<)) {
    return 0;
}

# check whether group or other can write file.
# use 066 to detect either reading or writing
if ($info->mode & 022) { # someone else can write this
    return 0 unless -d _; # non-directories aren't safe
    # but directories with the sticky bit (01000) are
    return 0 unless $info->mode & 01000;
}
return 1;
}

```

A directory is considered safe even if others can write to it, provided that its mode 01000 (owner delete only) bit is set.

Careful programmers also ensure that no enclosing directory is writable. This is due to systems with the “chown giveaway” problem in which any user can give away a file they own and make it owned by someone else. The following function handles that by using the `is_safe` function to check every enclosing directory up to the root if it detects that you have the `chown` problem, for which it queries the `POSIX::sysconf`. If you don't have an unrestricted version of `chown`, the `is_very_safe` subroutine just calls `is_safe`. If you do have the problem, it walks up the filesystem tree until it reaches the root.

```

use Cwd;
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
sub is_very_safe {
    my $path = shift;
    return is_safe($path) if sysconf(_PC_CHOWN_RESTRICTED);
    $path = getcwd() . '/' . $path if $path !~ m{^/};
    do {
        return unless is_safe($path);
        $path =~ s#([^/]+|/)$##; # dirname
        $path =~ s#/$## if length($path) > 1; # last slash
    } while length $path;

    return 1;
}

```

To use this in a program, try something like this:

```

$file = "$ENV{HOME}/.myprogrc";
readconfig($file) if is_safe($file);

```

This has potential for a race condition, because it's presumed that the hypothetical `readconfig` function will open the file. Between the time when `is_safe` checks the file's stats and when `readconfig` opens it, something wicked could theoretically occur. To avoid this, pass `is_safe` the already open filehandle, which is set up to handle this:

```
$file = "$ENV{HOME}/.myprogrc";
if (open(FILE, "< $file")) {
    readconfig(*FILE) if is_safe(*FILE);
}
```

You would still have to arrange for `readconfig` to accept a filehandle instead of a filename, though.

8.18. Program: *tailwtmp*

Every time a user logs into or out of a Unix system, a record is added to the *wtmp* file. You can't use the normal *tail* program on it because it's in binary format. The *tailwtmp* program in Example 8-7 knows the format of the binary file and shows every new record as it appears. You'll have to adjust the `pack` format for your own system.

Example 8-7. tailwtmp

```
#!/usr/bin/perl
# tailwtmp - watch for logins and logouts;
# uses linux utmp structure, from utmp(5)
$typedef = 's x2 i A12 A4 l A8 A16 l';
$sizeof = length pack($typedef, ( ));
use IO::File;
open(WTMP, '/var/log/wtmp') or die "can't open /var/log/wtmp: $!";
seek(WTMP, 0, SEEK_END);
for (;;) {
    while (read(WTMP, $buffer, $sizeof) == $sizeof) {
        ($type, $pid, $line, $id, $time, $user, $host, $addr)
            = unpack($typedef, $buffer);
        next unless $user && ord($user) && $time;
        printf "%1d %-8s %-12s %2s %-24s %-16s %5d %08x\n",
            $type, $user, $line, $id, scalar(localtime($time)),
            $host, $pid, $addr;
    }
    for ($size = -s WTMP; $size == -s WTMP; sleep 1) {}
    WTMP->clearerr();
}
```

8.19. Program: *tctee*

Not all systems support the classic *tee* program for splitting output pipes to multiple destinations. This command sends the output from *someprog* to */tmp/output* and to the mail pipe beyond.

```
% someprog | tee /tmp/output | Mail -s 'check this' user@host.org
```

This program helps not only users who aren't on Unix systems and don't have a regular *tee*. It also helps those who are, because it offers features not found on other version of *tee*.

The four flag arguments are *-i* to ignore interrupts, *-a* to append to output files, *-u* for unbuffered output, and *-n* to omit copying the output on to standard out.

Because this program uses Perl's magic open, you can specify pipes as well as files.

```
% someprog | tctee f1 "|cat -n" f2 ">>f3"
```

That sends the output from *someprog* to the files *f1* and *f2*, appends it to *f3*, sends a copy to the program *cat -n*, and also produces the stream on standard output.

The program in Example 8-8 is one of many venerable Perl programs written nearly a decade ago that still runs perfectly well. If written from scratch now, we'd probably use `strict`, `warnings`, and ten to thirty thousand lines of modules. But if it ain't broke . . .

Example 8-8. tctee

```
#!/usr/bin/perl
# tctee - clone that groks process tees
# perl3 compatible, or better.

while ($ARGV[0] =~ /^-(.+)/ && (shift, ($_ = $1), 1)) {
    next if /^$/;
    s/i// && (++$ignore_ints, redo);
    s/a// && (++$append, redo);
    s/u// && (++$unbuffer, redo);
    s/n// && (++$nostdout, redo);
    die "usage tee [-aiun] [filenames] ...\n";
}

if ($ignore_ints) {
    for $sig ('INT', 'TERM', 'HUP', 'QUIT') { $SIG{$sig} = 'IGNORE'; }
}

$SIG{'PIPE'} = 'PLUMBER';
$mode = $append ? '>>' : '>';
$fh = 'FH000';

unless ($nostdout) {
    %fh = ('STDOUT', 'standard output'); # always go to stdout
}

$| = 1 if $unbuffer;

for (@ARGV) {
    if (!open($fh, (/^[^>|]/ && $mode) . $_)) {
```

Example 8-8. *tctee* (continued)

```

        warn "$0: cannot open $_: $!\n"; # like sun's; i prefer die
        $status++;
        next;
    }
    select((select($fh), $| = 1)[0]) if $unbuffer;
    $fh{$fh++} = $_;
}

while (<STDIN>) {
    for $fh (keys %fh) {
        print $fh $_;
    }
}

for $fh (keys %fh) {
    next if close($fh) || !defined $fh{$fh};
    warn "$0: couldnt close $fh{$fh}: $!\n";
    $status++;
}

exit $status;

sub PLUMBER {
    warn "$0: pipe to \"${fh{$fh}}\" broke!\n";
    $status++;
    delete $fh{$fh};
}

```

8.20. Program: *laston*

When you log in to a Unix system, it tells you when you last logged in. That information is stored in a binary file called *lastlog*. Each user has their own record; UID 8 is at record 8, UID 239 at record 239, and so on. To find out when a given user last logged in, convert their login name to a number, seek to their record in that file, read, and unpack. Doing so with shell tools is very hard, but it's very easy with the *laston* program. Here's an example:

```

% laston gnat
gnat UID 314 at Mon May 25 08:32:52 1998 on tty0 from below.perl.com

```

The program in Example 8-9 is much newer than the *tctee* program in Example 8-8, but it's less portable. It uses the Linux binary layout of the *lastlog* file. You'll have to change this for other systems.

Example 8-9. *laston*

```

#!/usr/bin/perl
# laston - find out when given user last logged on
use User::pwent;
use IO::Seekable qw(SEEK_SET);

```

Example 8-9. laston (continued)

```
open (LASTLOG, "/var/log/lastlog") or die "can't open /usr/adm/lastlog: $!";

$typedef = 'L A12 A16'; # linux fmt; sunos is "L A8 A16"
$sizeof = length(pack($typedef, ()));

for $user (@ARGV) {
    $U = ($user =~ /^\/\d+$/) ? getpwuid($user) : getpwnam($user);
    unless ($U) { warn "no such uid $user\n"; next; }
    seek(LASTLOG, $U->uid * $sizeof, SEEK_SET) or die "seek failed: $!";
    read(LASTLOG, $buffer, $sizeof) == $sizeof or next;
    ($time, $line, $host) = unpack($typedef, $buffer);
    printf "%-8s UID %5d %s%s%s\n", $U->name, $U->uid,
        $time ? ("at " . localtime($time)) : "never logged in",
        $line && " on $line",
        $host && " from $host";
}
}
```