

Making Easy Things Easy & Hard Things Possible

4th Edition
Covers Perl 5.8

Learning Perl



O'REILLY®

*Randal L. Schwartz,
Tom Phoenix & brian d foy*

File Tests

Earlier, we showed how to open a filehandle for output. Normally, that will create a new file, wiping out any existing file with the same name. Perhaps you want to check that there isn't a file by that name. Perhaps you need to know how old a given file is. Or perhaps you want to go through a list of files to find which ones are larger than a certain number of bytes and not accessed for a certain amount of time. Perl has a complete set of tests you can use to find information about files.

File Test Operators

Before we start a program that creates a new file, let's make sure the file doesn't already exist so that we don't accidentally overwrite a vital spreadsheet data file or that important birthday calendar. For this, we use the `-e` file test, testing a filename for existence:

```
die "Oops! A file called '$filename' already exists.\n"
    if -e $filename;
```

We didn't include `#!` in this `die` message since we're not reporting that the system refused a request in this case. Here's an example of checking if a file is being kept up to date. In this case, we're testing an already opened filehandle instead of a string file name. Let's say that our program's configuration file should be updated every week or two. (Maybe it's checking for computer viruses.) If the file hasn't been modified in the past 28 days, then something is wrong:

```
warn "Config file is looking pretty old!\n"
    if -M CONFIG > 28;
```

The third example is more complex. Let's say disk space is filling up; rather than buy more disks, we've decided to move any large, useless files to the backup tapes. So

let's go through our list of files* to see which of them are larger than 100 KB. But even if a file is large, we shouldn't move it to the backup tapes unless it hasn't been accessed in the last 90 days (so we know it's not used too often):†

```
my @original_files = qw/ fred barney betty wilma pebbles dino bamm-bamm /;
my @big_old_files; # The ones we want to put on backup tapes
foreach my $filename (@original_files) {
    push @big_old_files, $filename
        if -s $filename > 100_000 and -A $filename > 90;
}
```

This is the first time that you've seen it, so maybe you noticed that the control variable of the `foreach` loop is a `my` variable. That declares it to have the scope of the loop, so this example should work under `use strict`. Without the `my` keyword, this would be using the global `$filename`.

The file tests look like a hyphen and a letter, which is the name of the test, followed by a filename or a filehandle to test. Many of them return a true/false value, but several give something more interesting. See Table 11-1 for the complete list and read the following discussion to learn more about the special cases.

Table 11-1. File tests and their meanings

File test	Meaning
-r	File or directory is readable by this (effective) user or group
-w	File or directory is writable by this (effective) user or group
-x	File or directory is executable by this (effective) user or group
-o	File or directory is owned by this (effective) user
-R	File or directory is readable by this real user or group
-W	File or directory is writable by this real user or group
-X	File or directory is executable by this real user or group
-O	File or directory is owned by this real user
-e	File or directory name exists
-z	File exists and has zero size (always false for directories)
-s	File or directory exists and has nonzero size (the value is the size in bytes)
-f	Entry is a plain file
-d	Entry is a directory
-l	Entry is a symbolic link
-S	Entry is a socket

* It's more likely that, instead of having the list of files in an array as our example shows, you'll read it directly from the filesystem using a glob or directory handle as we will show in Chapter 12. Since you haven't seen that yet, we'll just start with the list and go from there.

† There's a way to make this example more efficient as you'll see by the end of the chapter.

Table 11-1. File tests and their meanings (continued)

File test	Meaning
-p	Entry is a named pipe (a “fifo”)
-b	Entry is a block-special file (like a mountable disk)
-c	Entry is a character-special file (like an I/O device)
-u	File or directory is setuid
-g	File or directory is setgid
-k	File or directory has the sticky bit set
-t	The filehandle is a TTY (as reported by the <code>isatty()</code> system function; filenames can't be tested by this test)
-T	File looks like a “text” file
-B	File looks like a “binary” file
-M	Modification age (measured in days)
-A	Access age (measured in days)
-C	Inode-modification age (measured in days)

The tests `-r`, `-w`, `-x`, and `-o` tell if the given attribute is true for the effective user or group ID,* which essentially refers to the person who is in charge of running the program.† These tests look at the permission bits on the file to see what is permitted. If your system uses Access Control Lists (ACLs), the tests will use those as well. These tests generally tell if the system would *try* to permit something, but it doesn't mean that it really would be possible. For example, `-w` may be true for a file on a CD-ROM, though you can't write to it, or `-x` may be true on an empty file, which can't truly be executed.

The `-s` test does return true if the file is non-empty, but it's a special kind of true. It's the length of the file, measured in bytes, which evaluates as true for a nonzero number.

A Unix filesystem‡ has seven types of items, represented by the seven file tests `-f`, `-d`, `-l`, `-S`, `-p`, `-b`, and `-c`. Any item should be one of those. If you have a symbolic link pointing to a file, that will report true for `-f` and `-l`. So if you want to know whether something is a symbolic link, you should generally test that first. (You'll learn more about symbolic links in Chapter 12.)

* The `-o` and `-O` tests relate only to the user ID and not to the group ID.

† For advanced students, the corresponding `-R`, `-W`, `-X`, and `-O` tests use the real user or group ID, which becomes important if your program may be running set-ID. In that case, it's generally the ID of the person who requested running it. See any good book about advanced Unix programming for a discussion of set-ID programs.

‡ This is the case on many non-Unix filesystems but not all of the file tests are meaningful everywhere. For example, you aren't likely to have block special files on your non-Unix system.

The age tests, `-M`, `-A`, and `-C` (yes, they're uppercase) return the number of days since the file was last modified, accessed, or had its inode changed.* (The inode contains all of the information about the file except for its contents.) See the `stat` system call manpage or a good book on Unix internals for details.) This age value is a full floating-point number, so you might get a value of 2.00001 if a file were modified two days and one second ago. These “days” aren't necessarily the same as a human would count. For example, if it's 1:30 A.M. when you check a file modified at about an hour before midnight, the value of `-M` for this file would be around 0.1, even though it was modified “yesterday.”

When checking the age of a file, you might get a negative value like `-1.2`, which means that the file's last access timestamp is set at about thirty hours in the future. The zero point on this timescale is the moment your program started running,† so that value might mean a long-running program was looking at a file that had just been accessed. Or a timestamp could be set (accidentally or intentionally) to a time in the future.

The tests `-T` and `-B` determine if a file is text or binary. But people who know a lot about filesystems know there's no bit (at least in Unix-like operating systems) to indicate that a file is a binary or text file, so how can Perl tell? The answer is that Perl cheats: it opens the file, looks at the first few thousand bytes, and makes an educated guess. If it sees a lot of null bytes, unusual control characters, and bytes with the high bit set, then that looks like a binary file. If there's not much weird stuff, then it looks like text. It sometimes guesses wrong. If a text file has a lot of Swedish or French words (which may have characters represented with the high bit set, as some ISO-8859-something variant, or perhaps even a Unicode version), it may fool Perl into declaring it binary. So it's not perfect, but if you need to separate your source code from compiled files, or HTML files from PNGs, these tests should do the trick.

You'd think that `-T` and `-B` would always disagree since a text file isn't a binary and vice versa, but there are two special cases where they're in complete agreement. If the file doesn't exist, or can't be read, both are false since it's neither a text file nor a binary. Alternatively, if the file is empty, it's an empty text file and an empty binary file at the same time, so they're both true.

The `-t` file test returns true if the given filehandle is a TTY—if it's interactive because it's not a simple file or pipe. When `-t STDIN` returns true, it generally means that you can interactively ask the user questions. If it's false, your program is probably getting input from a file or pipe, rather than a keyboard.

* This information will be somewhat different on non-Unix systems since not all keep track of the same times that Unix does. For example, on some systems, the `ctime` field (which the `-C` test looks at) is the file creation time (which Unix doesn't keep track of), rather than the inode change time. See the `perlport` manpage.

† As recorded in the `$^T` variable, which you could update (with a statement like `$^T = time;`) if you needed to get the ages relative to a different starting time.

Don't worry if you don't know what some of the other file tests mean—if you've never heard of them, you won't be needing them. But if you're curious, get a good book about programming for Unix. (On non-Unix systems, these tests all try to give results analogous to what they do on Unix, or give `undef` for an unavailable feature. Usually, you'll be able to guess what they'll do.)

If you omit the filename or filehandle parameter to a file test (that is, if you have `-r` or just `-s`), the default operand is the file named in `$_`.^{*} So, to test a list of filenames to see which ones are readable, you type the following:

```
foreach (@lots_of_filenames) {  
    print "$_ is readable\n" if -r; # same as -r $_  
}
```

But if you omit the parameter, be careful that whatever follows the file test doesn't look like it could be a parameter. For example, if you wanted to find out the size of a file in KB rather than in bytes, you might be tempted to divide the result of `-s` by 1000 (or 1024), like this:

```
# The filename is in $_  
my $size_in_K = -s / 1000; # Oops!
```

When the Perl parser sees the slash, it doesn't think about division. Since it's looking for the optional operand for `-s`, it sees what looks like the start of a regular expression in forward slashes. To prevent this confusion, put parentheses around the file test:

```
my $size_in_k = (-s) / 1024; # Uses $_ by default
```

Explicitly giving a file test a parameter is safer.

The `stat` and `lstat` Functions

Though these file tests are fine for testing various attributes regarding a particular file or filehandle, they don't tell the whole story. For example, there's no file test that returns the number of links to a file or the owner's user ID (uid). To get at the remaining information about a file, call the `stat` function, which returns pretty much everything that the `stat` Unix system call returns (and more than you want to know).[†] The operand to `stat` is a filehandle or an expression that evaluates to a filename. The return value is either the empty list indicating that the `stat` failed (usually because

^{*} The `-t` file test is an exception since that test isn't useful with filenames (they're never TTYs). By default, it tests `STDIN`.

[†] On a non-Unix system, `stat` and `lstat`, as well as the file tests, should return "the closest thing available." For example, a system that doesn't have user IDs (that is, a system that has just one "user," in the Unix sense) might return zero for the user and group IDs as if the only user is the system administrator. If `stat` or `lstat` fails, it will return an empty list. If the system call underlying a file test fails (or isn't available on the given system), that test will generally return `undef`. See the `perlport` manpage for the latest about what to expect on different systems.

the file doesn't exist), or a 13-element list of numbers, most easily described using the following list of scalar variables:

```
my($dev, $ino, $mode, $nlink, $uid, $gid, $rdev,  
   $size, $atime, $mtime, $ctime, $blksize, $blocks)  
  = stat($filename);
```

The names here refer to the parts of the `stat` structure, described in detail in the `stat(2)` manpage. You should look there for the detailed descriptions. Here's a quick summary of the important ones:

\$dev and \$ino

The device number and inode number of the file. Together, they make up a "license plate" for the file. Even if it has more than one name (hard link), the combination of device and inode numbers will be unique.

\$mode

The set of permission bits for the file and some other bits. If you've ever used the Unix command `ls -l` to get a detailed (long) file listing, you'll see that each line of output starts with something like `-rwxr-xr-x`. The nine letters and hyphens of file permissions* correspond to the nine least significant bits of `$mode`, which would give the octal number `0755` in this case. The other bits, beyond the lowest nine, indicate other details about the file. If you need to work with the mode, you'll want to use the bitwise operators covered later in this chapter.

\$nlink

The number of (hard) links to the file or directory. This is the number of true names that the item has. This number is always 2 or more for directories and (usually) 1 for files. You'll see more about this when we talk about creating links to files in Chapter 12. In the listing from `ls -l`, this is the number just after the permission bits string.

\$uid and \$gid

The numeric user ID and group ID showing the file's ownership.

\$size

The size in bytes, as returned by the `-s` file test.

\$atime, \$mtime, and \$ctime

The three timestamps, but here they're represented in the system's timestamp format: a 32-bit number telling how many seconds have passed since the *Epoch*, which is an arbitrary starting point for measuring system time. On Unix systems and some others, the Epoch is the beginning of 1970 at midnight Universal Time, but the Epoch is different on some machines. There's more information later in this chapter on turning that timestamp number into something useful.

* The first character in that string isn't a permission bit. It indicates the type of entry: a hyphen for an ordinary file, `d` for directory, or `l` for symbolic link, among others. The `ls` command determines this from the other bits past the least significant nine.

Invoking `stat` on the name of a symbolic link returns information on what the symbolic link points at and not information about the symbolic link itself unless the link happens to be pointing at nothing currently accessible. If you need the (mostly useless) information about the symbolic link itself, use `lstat` rather than `stat` (which returns the same information in the same order). If the operand isn't a symbolic link, `lstat` returns the same things that `stat` would.

Like the file tests, the operand of `stat` or `lstat` defaults to `$_`, meaning the underlying `stat` system call will be performed on the file named by the scalar variable `$_`.

The localtime Function

When you have a timestamp number (such as the ones from `stat`), it will typically look something like 1180630098. That won't help you, unless you need to compare two timestamps by subtracting. You may need to convert it to something human-readable, such as a string like "Thu May 31 09:48:18 2007". Perl can do that with the `localtime` function in a scalar context:

```
my $timestamp = 1180630098;
my $date = localtime $timestamp;
```

In a list context, `localtime` returns a list of numbers, several of which may not be what you'd expect:

```
my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst)
= localtime $timestamp;
```

The `$mon` is a month number, ranging from 0 to 11, which is handy as an index into an array of month names. The `$year` is the number of years since 1900, oddly enough, so add 1900 to get the real year number. The `$wday` ranges from 0 (for Sunday) through 6 (for Saturday), and the `$yday` is the day-of-the-year (ranging from 0 for January 1, through 364 or 365 for December 31).

Two related functions are also useful. The `gmtime` function is the same as `localtime`, except that it returns the time in Universal Time (what we once called Greenwich Mean Time). If you need the current timestamp number from the system clock, use the `time` function. Both `localtime` and `gmtime` default to using the current time value if you don't supply a parameter:

```
my $now = gmtime; # Get the current universal timestamp as a string
```

For more information on manipulating date and time information, see the information about some useful modules in Appendix R.

Bitwise Operators

When you need to work with numbers bit by bit, as when working with the mode bits returned by `stat`, you'll need to use the bitwise operators. These operators perform

binary math operations on values. The bitwise-and operator (&) reports which bits are set in the left argument *and* in the right argument. For example, the expression `10 & 12` has the value 8. The bitwise-and needs to have a one-bit in both operands to produce a one-bit in the result. That means that the logical-and operation on ten (which is 1010 in binary) and twelve (which is 1100) gives eight (which is 1000, with a one-bit only where the left operand has a one-bit *and* the right operand also has a one-bit). See Figure 11-1.

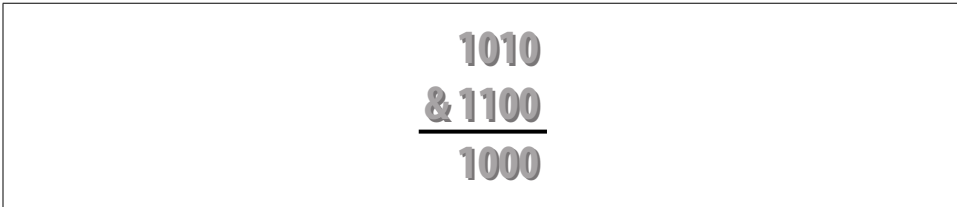


Figure 11-1. Bitwise-and addition

The different bitwise operators and their meanings are shown in Table 11-2.

Table 11-2. Bitwise operators

Expression	Meaning
<code>10 & 12</code>	Bitwise-and; which bits are true in both operands (this gives 8)
<code>10 12</code>	Bitwise-or; which bits are true in one operand or the other (this gives 14)
<code>10 ^ 12</code>	Bitwise-xor; which bits are true in one operand or the other but not both (this gives 6)
<code>6 << 2</code>	Bitwise shift left; shift the left operand the number of bits shown by the right operand, adding zero-bits at the least-significant places (this gives 24)
<code>25 >> 2</code>	Bitwise shift right; shift the left operand the number of bits shown by the right operand, discarding the least-significant bits (this gives 6)
<code>~ 10</code>	Bitwise negation, also called unary bit complement; return the number with the opposite bit for each bit in the operand (this gives 0xFFFFFFFF5, but see the text)

So, here's an example of some things you could do with the `$mode` returned by `stat`. The results of these bit manipulations could be useful with `chmod`, which you'll see in Chapter 12:

```
# $mode is the mode value returned from a stat of CONFIG
warn "Hey, the configuration file is world-writable!\n"
  if $mode & 0002;                               # configuration security problem
my $classical_mode = 0777 & $mode;              # mask off extra high-bits
my $_plus_x = $classical_mode | 0100;           # turn one bit on
my $_go_minus_r = $classical_mode & (~ 0044);  # turn two bits off
```

Using Bitstrings

All of the bitwise operators can work with bitstrings, as well as with integers. If the operands are integers, the result will be an integer. (The integer will be at least a 32-bit integer but may be larger if your machine supports that. That is, if you have a 64-bit machine, `~10` may give the 64-bit result `0xFFFFFFFFFFFF5`, rather than the 32-bit result `0xFFFFFFF5`.)

But if any operand of a bitwise operator is a string, Perl will perform the operation on that bitstring. That is, `"\xAA" | "\x55"` will give the string `"\xFF"`. Note that these values are single-byte strings and the result is a byte with all eight bits set. Bitstrings may be arbitrarily long.

This is one of the few places where Perl distinguishes between strings and numbers. See the `perlop` manpage for more information on using bitwise operators on strings.

Using the Special Underscore Filehandle

Every time you use `stat`, `lstat`, or a file test in a program, Perl has to go out to the system to ask for a stat buffer on the file (that is, the return buffer from the `stat` system call). That means if you want to know if a file is readable and writable, you'll ask the system twice for the same information, which isn't likely to change in a nonhostile environment.

This looks like a waste of time,* and can be avoided. Doing a file test, `stat`, or `lstat` on the special `_` filehandle (the operand being a single underscore) tells Perl to use whatever happens to be lounging around in memory from the previous file test, `stat`, or `lstat` function, rather than going out to the operating system again. Sometimes this is dangerous: a subroutine call can invoke `stat` without your knowledge, blowing your buffer away. If you're careful, you can save yourself a few unneeded system calls, thereby making your program faster. Here's that example of finding files to put on the backup tapes again, using the new tricks you've learned:

```
my @original_files = qw/ fred barney betty wilma pebbles dino bamm-bamm /;
my @big_old_files;           # The ones we want to put on backup tapes
foreach (@original_files) {
    push @big_old_files, $_
        if (-s) > 100_000 and -A _ > 90;    # More efficient than before
}
```

We used the default of `$_` for the first test; this is as more efficient (except perhaps for the programmer), and it gets the data from the operating system. The second test uses the magic `_` filehandle. For this test, the data left around after getting the file's size are used, which are what we want.

* Because it is. Asking the system for information is relatively slow.

Testing the `_filehandle` is different from allowing the operand of a file test, `stat`, or `lstat` to default to testing `$_`. Using `$_` would be a fresh test each time on the current file named by the contents of `$_`, but using `_` saves the trouble of calling the system again. Here is another case where similar names were chosen for radically different functions.

Exercises

See Appendix Q for answers to the following exercises:

1. [15] Make a program that takes a list of files named on the command line and reports for each one whether it's readable, writable, executable, or doesn't exist. (Hint: It may be helpful to have a function that will do all of the file tests for one file at a time.) What does it report about a file which has been `chmod`'ed to 0? (That is, if you're on a Unix system, use the command `chmod 0 some_file` to mark that file as neither being readable, writable, nor executable.) In most shells, use a star as the argument to mean all of the normal files in the current directory. That is, you could type something like `./ex11-1 *` to ask the program for the attributes of many files at once.
2. [10] Make a program to identify the oldest file named on the command line and report its age in days. What does it do if the list is empty (that is, if no files are mentioned on the command line)?