Chapter 12. File Tests

Earlier, we showed you how to open a filehandle for output. Normally, that creates 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 have not been accessed for a certain amount of time. Perl has a complete set of tests you can use to find out information about files.

File Test Operators

Perl has a set of file test operators that let you get particular information about files. They all take the form of `-X`, where the `X` represents the particular test (and there is a literal `-X` file test operator too, to confuse things a bit). In most cases, these operators return true or false. Although we call these things operators, you'll find their documentation in perlfunc.

NOTE

To get the list, use the command line *perldoc -f -X*. That `-X` is literal and not a command-line switch. It stands in for all the file test operators since you can't use *perldoc* to look them up individually.

Before you start a program that creates a new file, you might want to ensure that the file doesn't already exist so that you don't accidentally overwrite a vital spreadsheet datafile or that important birthday calendar. For this, you can use the `-e` file test, testing a filename for existence:

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

Notice that you don't include `$!` in this `die` message, since you're not reporting that the system refused a request in this case. Here's an example of checking whether a file is being kept up-to-date. In this case, you're testing an already-opened filehandle instead of a string filename. Let's say that your 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, something is wrong. The `-M` file test returns the file modification time in days before the start of the program, which seems like a mouthful until you see how convenient the code is:

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

The third example is more complex. Here, say that disk space is filling up, and rather than buy more disks, you decide to move any large, useless files to a backup. So let's go through our list of files to see which of them are larger than 100 K. But even if a file is large, you shouldn't warehouse it unless it hasn't been accessed in the last 90 days (so we know that it's not used too often). The `-s` file test operator, instead of returning true or false, returns the file size in bytes (and an existing file might have 0 bytes):

```
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;
}
```

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

The file tests all look like a hyphen and a letter, which is the name of the test, followed by either a filename or a filehandle to test. Many of them return a true/false value, but several give something more interesting. See Table 12-1 for the complete list, and read the following explanation to learn more about the special cases.

Table 12-1. 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

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 `isatty()` 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 whether 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.

NOTE

Note 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 an explanation of set-ID programs.

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 whether 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, even 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 nonempty, 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.

On a Unix filesystem there are 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. But if you have a symbolic link pointing to a file, that will report true for both `-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 "Links and Files".)

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 the information about the file except for its contents—see the `stat` system call documentation 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 had been 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 in the morning when you check a file modified 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 even get a negative value like `-1.2`, which means that the file's last-access timestamp is set at about 30 hours in the future! The zero point on this timescale is the moment your program started running, so that value might mean that 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` take a try at telling whether a file is text or binary. But people who know a lot about filesystems know that 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. As you might guess, it sometimes guesses wrong. It's not perfect, but if you just 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—in short, 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.

NOTE

The `IO::Interactive` module might be a better choice for this because the situation is actually a bit more complicated. That module explains this in its documentation.

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 `undef` for an unavailable feature. Usually you'll be able to guess correctly what they'll do.

If you omit the filename or filehandle parameter to a file test (that is, if you have just `-r` or just `-s`, say), the default operand is the file named in `$_`. The `-t` file test is an exception, because that test isn't useful with filenames (they're never TTYs). By default, it tests `STDIN`. So, to test a list of filenames to see which ones are readable, you simply type:

```
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 kilobytes 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 the forward slash. One simple way to prevent this kind of confusion is to put parentheses around the file test:

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

Of course, it's always safe to explicitly give a file test a parameter:

```
my $size_in_k = (-s $filename) / 1024;
```

Testing Several Attributes of the Same File

You can use more than one file test on the same file to create a complex logical condition. Suppose you only want to operate on files that are both readable and writable; you check each attribute and combine them with `and`:

```
if (-r $filename and -w $filename) {
  ... }
```

This is an expensive operation, though. Each time you perform a file test, Perl asks the filesystem for all the information about the file (Perl's actually doing a `stat` each time, which we talk about in the next section). Although you already got that information when you tested `-r`, Perl asks for the same information again so that it can test `-w`. What a waste! This can be a significant performance problem if you're testing many attributes on many files.

Perl has a special shortcut to help you not do so much work. The virtual filehandle `_` (just the underscore) uses the information from the last file lookup that a file test operator performed. Perl only has to look up the file information once now:

```
if (-r $filename and -w _) {
    ... }
```

You don't have to use the file tests next to each other to use `_`. Here we have them in separate `if` conditions:

```
if (-r $filename) {
  print "The file is readable!\n";
}

if (-w _) {
  print "The file is writable!\n";
}
```

You have to watch out that you know what the last file lookup really was, though. If you do something else between the file tests, such as call a subroutine, the last file you looked up might be different. For instance, this example calls the `lookup` subroutine, which has a file test in it. When you return from that subroutine and do another file test, the `_` filehandle isn't for `$filename`, like you expect, but for `$other_filename`:

```
if (-r $filename) {
  print "The file is readable!\n";
}

lookup( $other_filename );

if (-w _) {
  print "The file is writable!\n";
}

sub lookup {
  return -w $_[0];
}
```

Stacked File Test Operators

Prior to Perl 5.10, if you wanted to test several file attributes at the same time, you had to test them individually, even when using the `_` filehandle to save some work. Suppose you want to test if a file was readable and writable at the same time. You have to test if it's readable, then also test if it's writable:

```
if (-r $filename and -w _) {
  print "The file is both readable and writable!\n";
}
```

It's much easier to do this all at once. Starting with Perl 5.10, you can "stack" your file test operators by lining them all up before the filename:

```
use v5.10;
```

```
if (-w -r $filename) {
  print "The file is both readable and writable!\n";
}
```

This stacked example is the same as the previous example with just a change in syntax, although it looks like the file tests are reversed. Perl does the file test nearest the filename first. Normally, this isn't going to matter.

Stacked file tests are especially handy for complex situations. Suppose you want to list all the directories that are readable, writable, executable, and owned by your user. You just need the right set of file tests:

```
use v5.10;

if (-r -w -x -o -d $filename) {
  print "My directory is readable, writable, and executable!\n";
}
```

Stacked file tests aren't good for those tests that return values other than true or false that you would want to use in a comparison. You might think that this next bit of code first tests that it's a directory and then tests that it is less than 512 bytes, but it doesn't:

```
use v5.10;

if (-s -d $filename < 512) {    # WRONG! DON'T DO THIS
  say 'The directory is less than 512 bytes!';
}
```

Rewriting the stacked file tests as the previous notation shows us what is going on. The result of the combination of the file tests becomes the argument for the comparison:

```
if (( -d $filename and -s _ ) < 512) {
  print "The directory is less than 512 bytes!\n";
}
```

When the `-d` returns false, Perl compares that false value to 512. That turns out to be true since false will be 0, which just happens to be less than 512. Instead of worrying about that sort of confusion, you just write it as separate file tests to be nice to the maintenance programmers who come after you:

```
if (-d $filename and -s _ < 512) {
  print "The directory is less than 512 bytes!\n";
}
```

The stat and lstat Functions

While these file tests are fine for testing various attributes regarding a particular file or filehandle, they don't tell you 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, merely call the `stat` function, which returns pretty much everything that the `stat` Unix system call returns (hopefully more than you want to know).

NOTE

On a non-Unix system, both `stat` and `lstat`, as well as the file tests, should return "the closest thing available." 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 documentation for the latest about what to expect on different systems.

The operand to `stat` is a filehandle (including the `_` virtual 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 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) documentation. You should probably look there for the detailed descriptions. But in short, 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 should always 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 contains something like `-rwxr-xr-x`. That information is wrapped up in `$mode`.

`$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 13. 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 telling how many seconds have passed since the *epoch*, an arbitrary starting point for measuring system time.

Some filesystems, such as *ext2*, may disable `atime` as a performance measure.

Invoking `stat` on the name of a symbolic link returns information on what the symbolic link points at, not information about the symbolic link itself (unless the link just 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 `$_`, meaning the underlying `stat` system call will be performed on the file named by the scalar variable `$_`.

NOTE

The `File::stat` module provides a friendlier interface to `stat`.

The localtime Function

When you have a timestamp number (such as the ones from `stat`), it will typically look something like 1454133253. That's not very useful for most humans, unless you need to compare two timestamps by subtracting. You may need to convert it to something human-readable, such as a string like "`Sat Jan 30 00:54:13 2016`." Perl can do that with the `localtime` function in a scalar context:

```
my $timestamp = 1454133253;
my $date = localtime $timestamp;
```

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

```
my($sec, $min, $hour, $mday, $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 in the case of leap years, `365`—for December 31).

There are two related functions that you'll also find useful. The `gmtime` function is just the same as `localtime`, except that it returns the time in Universal Time. If you need the current timestamp number from the system clock, just 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 on manipulating dates and times, see Appendix B for information about some useful modules.

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 are the operators that 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 the bitwise-and operation on 10 (which is `1010` in binary) and 12 (which is `1100`) gives 8 (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 12-1.

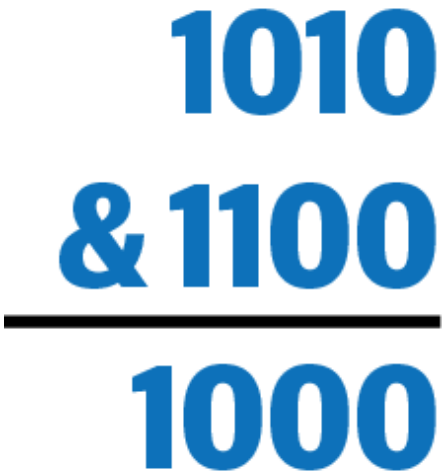

Figure 12-1. Bitwise-and addition

The different bitwise operators and their meanings are shown in Table 12-2.

Table 12-2. Examples of bitwise operations

Expression	Meaning
10 & 12	Bitwise-and—which bits are true in both operands (this gives `8`)
10 | 12	Bitwise-or—which bits are true in one operand or the other (this gives `14`)
10 ^ 12	Bitwise-xor—which bits are true in one operand or the other but not both (this gives `6`)
6 << 2	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`)
25 >> 2	Bitwise shift right—shift the left operand the number of bits shown by the right operand, discarding the least-significant bits (this gives `6`)
~ 10	Bitwise negation, also called unary bit complement, returns the number with the opposite bit for each bit in the operand (this gives `0xFFFFFFF5`, 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 13:

```
# $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 $u_plus_x = $classical_mode | 0100;          # turn one bit on
my $go_minus_r = $classical_mode & (~ 0044);    # turn two bits off
```

Using Bitstrings

All the bitwise operators can work with bitstrings as well as with integers. If either operand is an integer, the result will be an integer. (The integer will be at least a 32-bit integer, but it may be larger if your machine supports that. That is, if you have a 64-bit machine, `~10` may give the 64-bit result, `0xFFFFFFFFFFFFFFF5`, rather than the 32-bit result, `0xFFFFFFF5`.)

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

This is one of the very few places where Perl distinguishes between strings and numbers. You can run into problems if you think you are doing number operations but give one of these operators two strings. Perl v5.22 adds a feature to fix this, but first you should understand the problem.

If Perl thinks either of the operands is a number, it does a numeric operation. Consider this code where you have `$number_str` that looks like a number but it's quoted like a string. So far, Perl thinks that's a string because we haven't done anything with it yet:

```
use v5.10;

my $number     = 137;
my $number_str = '137';
my $string     = 'Amelia';

say "number_str & string:  ", $number_str & $string;
say "number & string:      ", $number & $string;
say "number & number_str:  ", $number & $number_str;
say "number_str & string:  ", $number_str & $string;
```

Notice that the first and last `say` statements are the same. You haven't explicitly done anything to change the variables, and if you printed their values you'd see what you expected. But then why is this output so weird?

```
number_str & string:  ¿!%
number & string:      0
```

```
number & number_str:  137
number_str & string:  0
```

The first line of the output shows some gobbledygook from `'137' & 'Amelia'`. That's a string operation because both sides are strings.

The second output line shows that `137 & 'Amelia'` is zero. Since one of the operands is a number, Perl converts the other operand's value to its number form, which is 0. No bits are set in 0, so there are no bits set in common. The result is zero.

The same thing happens for the third line. The string `'137'` converts to the number 137, which is the same as the other argument. Since all of their set bits are the same, `137` is the answer.

Now it gets weird. When you redo the same operation you did for the first line of output you get a different answer! You didn't explicitly do anything to either value, but along the way Perl had to convert the values of `$number_str` and `$string` to a numeric form. When it did that, it secretly stored the result in case it had to do that again. When Perl does the last operation, it looks at the variables and sees that they have a numeric version, concludes that they are both numbers, and does a numeric bit operation. The numeric value of `$string` is `0` like the previous time we used it, so the answer is zero.

NOTE

Perl has the idea of a *dualvar*. A scalar can have separate numeric and string values at the same time. In most cases this is not a problem, and in some cases it's useful. For instance, the system error variable `$!` is a human-meaningful message as a string, but the system error number as a number. See the `Scalar::Util` module.

Perl v5.22 adds an experimental feature (Appendix D) to solve part of this problem. When you use an operator, you want to know what it's going to do no matter what path your operands took to get there. If you want a numeric bitwise operation, the `bitwise` feature makes the bitwise operations treat all operands as numbers:

```
use v5.22.0;
use feature qw(bitwise);
no warnings qw(experimental::bitwise);

my $number     = 137;
my $number_str = '137';
my $string     = 'Amelia';

say "number_str & string:  ", $number_str & $string;
say "number & string:      ", $number     & $string;
say "number & number_str:  ", $number     & $number_str;
say "number_str & string:  ", $number_str & $string;
```

No gobbledygook in the first line of the output. Even though both operands are strings, Perl treats them as numbers:

```
number_str & string:  0
number & string:      0
number & number_str:  137
number_str & string:  0
```

If you wanted string bit operations instead, the `bitwise` feature adds new operators that look like the bitwise operators with a `.` after them:

```
use v5.22.0;
use feature qw(bitwise);
no warnings qw(experimental::bitwise);

my $number     = 137;
my $number_str = '137';
my $string     = 'Amelia';

say "number_str &. string:  ", $number_str &. $string;
say "number &. string:      ", $number     &. $string;
say "number &. number_str:  ", $number     &. $number_str;
say "number_str &. string:  ", $number_str &. $string;
```

Now each of those is a string operation and each operand is converted to a string. The only result that seems to make sense is the one in the third line since it has `'137'` on each side of the `&.` operation:

```
number_str &. string:  ¿!%
number &. string:      ¿!%
number &. number_str:  137
number_str &. string:  ¿!%
```

Exercises

See "Answers to Chapter 12 Exercises" for answers to these 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 the file tests for one file at a time.) What does it report about a file that 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 being neither readable, writable, nor executable.) In most shells, use a star as the argument to mean all the normal files in the current directory. That is, you could type something like `./ex12-1 *` 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)?
3. [10] Make a program that uses stacked file test operators to list all files named on the command line that are readable, writable, and owned by you.