Chapter 13. Directory Operations

The files you created in Chapter 12 were generally in the same place as your program. But modern operating systems let you organize files into directories. Perl lets you manipulate these directories directly, in ways that are even fairly portable from one operating system to another.

Perl tries very hard to act the same no matter which system it runs on. Despite that, this chapter certainly shows Perl's preference toward its Unix history. If you are using Windows, you should look at the `Win32` distribution. Those modules provide hooks to the Win32 API.

The Current Working Directory

Your program runs with a *working directory*. This is the default directory for everything your program does.

With the `Cwd` module (part of the Standard Library), you can see what that directory is. Try this program, which we'll call *show_my_cwd*:

```
use v5.10;
use Cwd;
say "The current working directory is ", getcwd();
```

This should be the same directory that you'd get if you ran *pwd* in the Unix shell or *cd* (with no argument) in the Windows command shell. While you're practicing Perl with this book, you're most likely working in the same directory that holds your program.

When you open a file using a *relative path* (one that does not give the complete path from the top of the filesystem tree), Perl interprets that relative path starting at the current directory. Say that your current working directory is */home/fred*. When you run this line of code to read a file, Perl looks for */home/fred/relative/path.txt*:

```
# relative to the current working directory
open my $fh, '<:utf8', 'relative/path.txt'
```

If you aren't using a shell or terminal program, the thing that runs your program might have a different idea about its current working directory. If you can run a program from your editor, that application might have a current working directory that is different from the one in which you've saved the file. Scheduling your program with something like *cron* probably does the same thing.

The current working directory is not necessarily the directory that holds the program. Both of these commands look for *my_program* in the current directory:

```
$ ./show_my_cwd
$ perl show_my_cwd
```

But you can run that program from another directory by giving it the full path to the program:

```
$ /home/fred/show_my_cwd
$ perl /home/fred/show_my_cwd
```

If you put your program in one of the directories your shell will search to find a program, you can run your program without any path hints from any directory you like:

```
$ show_my_cwd
```

NOTE

You can use the `File::Spec` module (part of the Standard Library) to convert between relative and absolute paths.

Changing the Directory

You may not want the current working directory that your program starts with. The `chdir` operator changes the working directory. It's just like the shell's *cd* command:

```
chdir '/etc' or die "cannot chdir to /etc: $!";
```

Because this is a system request, Perl sets the value of `$!` if an error occurs. You should normally check `$!` when `chdir` returns a false value since that indicates that something has not gone as requested.

The working directory is inherited by all processes that Perl starts (we'll talk more about that in Chapter 15). However, the change in working directory cannot affect the process that invoked Perl, such as the shell. You can change the current working directory for the currently running program and set that directory for processes you start, but you can't change it for the process that started your program. You can affect things at your level and below. This isn't a limitation on Perl's part; it's actually a feature of Unix, Windows, and other systems.

If you call `chdir` without an argument, Perl determines your home directory as best as possible and attempts to set the working directory to your home directory, similar to using the Unix *cd* command at the shell without a parameter. This is one of the few places where omitting the parameter doesn't use `$_` as the default. Instead, it looks in the environment variables `$ENV{HOME}` and `$ENV{LOGDIR}`, in that order. If neither is set, it does nothing.

NOTE

Some environments don't set these environment variables for you. The `File::HomeDir` module can help you set the environment variables that `chdir` will examine.

Older Perls let you use the empty string or `undef` (both false) as an argument to `chdir`, but that was deprecated with v5.12. If you want to change to the home directory, don't give `chdir` any argument.

Some shells permit you to use a tilde-prefixed path with *cd* to use another user's home directory as a starting point (like *cd ~fred*). This is a function of the shell, not the operating system, and Perl is calling the operating system directly. Thus, a tilde prefix will not work with `chdir`.

NOTE

You might try the `File::HomeDir` module to get the user's home directory in a mostly portable fashion.

Globbing

Normally, the shell expands any filename patterns on the command line into the matching filenames. This is called *globbing*. For example, if you give a filename pattern of `*.pm` to the Unix *echo* command, the shell expands this list to a list of names that match:

```
$ echo *.pm
barney.pm dino.pm fred.pm wilma.pm
```

The *echo* command doesn't have to know anything about expanding `*.pm`, because the shell has already expanded it. This works even for your Perl programs. Here's a program that simply prints its arguments:

```
foreach $arg (@ARGV) {
  print "one arg is $arg\n";
}
```

When you run this program with a glob as the single argument, the shell expands the glob before it sends the result to your program. Thus, you think you got many arguments:

```
$ perl show-args *.pm
one arg is barney.pm
one arg is dino.pm
one arg is fred.pm
one arg is wilma.pm
```

Note that `show-args` didn't need to know anything about globbing—the names were already expanded in `@ARGV`.

But sometimes you end up with a pattern like `*.pm` inside your Perl program. Can we expand this pattern into the matching filenames without working very hard? Sure—just use the `glob` operator:

```
my @all_files = glob '*';
my @pm_files = glob '*.pm';
```

Here, `@all_files` gets all the files in the current directory, alphabetically sorted, but doesn't get the files beginning with a period—just like the shell. And `@pm_files` gets the same list you got before by using `*.pm` on the command line.

In fact, anything you can say on the command line, you can also put as the (single) argument to `glob`, including multiple patterns separated by spaces:

```
my @all_files_including_dot = glob '.* *';
```

Here, you include an additional "dot star" parameter to get the filenames that begin with a dot as well as the ones that don't. Please note that the space between these two items inside the quoted string is significant, as it separates two different items you want to glob.

Windows users may be accustomed to using a glob of `*.*` to mean "all files." But that actually means "all files with a dot in their names," even in Perl in Windows.

The reason this works exactly as the shell works is that prior to v5.6, the `glob` operator simply called */bin/csh* behind the scenes to perform the expansion. Because of this, globs were time consuming and could break in large directories, or in some other cases. Conscientious Perl hackers avoided globbing in favor of directory handles, which we will show later in this chapter. However, if you're using a modern version of Perl, you should no longer be concerned about such things.

NOTE

Perl's built-in `glob` isn't your only option. The `File::Glob` module provides other forms that handle edge cases.

An Alternate Syntax for Globbing

Although we use the term *globbing* freely, and we talk about the `glob` operator, you might not see the word `glob` in very many of the programs that use globbing. Why not? Well, a lot of legacy code was written before the Perl developers gave the `glob` operator its name. Instead, it used the angle-bracket syntax, similar to reading from a filehandle:

```
my @all_files = <*>;    # exactly the same as my @all_files = glob "*";
```

Perl interpolates the value between the angle brackets similarly to a double-quoted string, which means that Perl expands variables to their current Perl values before being globbed:

```
my $dir = '/etc';
my @dir_files = <$dir/* $dir/.*>;
```

Here, you fetch all the nondot and dot files from the designated directory because `$dir` has been expanded to its current value.

So, if using angle brackets means both filehandle reading and globbing, how does Perl decide which of the two operators to use? Well, a filehan-

dle has to be a Perl identifier or a variable. So, if the item between the angle brackets is strictly a Perl identifier, it's a filehandle read; otherwise, it's a globbing operation. For example:

```
my @files = <FRED/*>;   # a glob
my @lines = <FRED>;     # a filehandle read
my @lines = <$fred>;    # a filehandle read
my $name = 'FRED';
my @files = <$name/*>;  # a glob
```

The one exception is if the contents are a simple scalar variable (not an element of a hash or array) that's not a filehandle object, then it's an *indirect filehandle read*, where the variable contents give the name of the filehandle you want to read:

```
my $name = 'FRED';
my @lines = <$name>; # an indirect filehandle read of FRED handle
```

The determination of whether it's a glob or a filehandle read happens at compile time, and thus it is independent of the content of the variables.

If you want, you can get the operation of an indirect filehandle read using the `readline` operator, which also makes it clearer:

```
my $name = 'FRED';
my @lines = readline FRED;  # read from FRED
my @lines = readline $name; # read from FRED
```

But Perlers rarely use the `readline` operator, as indirect filehandle reads are uncommon and are generally performed against a simple scalar variable anyway.

Directory Handles

Another way to get a list of names from a given directory is with a *directory handle*. A directory handle looks and acts like a filehandle. You open

it (with `opendir` instead of `open`), you read from it (with `readdir` instead of `readline`), and you close it (with `closedir` instead of `close`). But instead of reading the *contents* of a file, you're reading the *names* of files (and other things) in a directory. For example:

```
my $dir_to_process = '/etc';
opendir my $dh, $dir_to_process or die "Cannot open $dir_to_process: $!";
foreach $file (readdir $dh) {
  print "one file in $dir_to_process is $file\n";
}
closedir $dh;
```

Like filehandles, directory handles are automatically closed at the end of the program or if the directory handle is reopened onto another directory.

You can also use a bareword directory handle, just like you could with a filehandle, but this has the same drawbacks we wrote about earlier:

```
opendir DIR, $dir_to_process
    or die "Cannot open $dir_to_process: $!";
foreach $file (readdir DIR) {
  print "one file in $dir_to_process is $file\n";
}
closedir DIR;
```

This is a lower-level operation and we have to do more of the work ourselves. For example, the names are returned in no particular order. And the list includes all files, not just those matching a particular pattern (like `*.pm` from our globbing examples). So, if you wanted only the *pm*-ending files, you could use a skip-over function inside the loop:

```
while ($name = readdir $dh) {
  next unless $name =~ /\.pm\z/;
  ... more processing ...
}
```

Note here that the syntax is that of a regular expression, not a glob. And if you wanted all the nondot files, you could say that:

```
next if $name =~ /\A\./;
```

Or if you wanted everything but the common dot (current directory) and dot-dot (parent directory) entries, you could explicitly say that:

```
next if $name eq '.' or $name eq '..';
```

Here's another part that gets most people mixed up, so pay close attention. The filenames returned by the `readdir` operator have *no* pathname component. It's just the *name* within the directory. So, instead of */etc/hosts*, you get just *hosts*. And because this is another difference from the globbing operation, it's easy to see how people get confused.

So you need to patch up the name to get the full name:

```
opendir my $somedir, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir $somedir) {
  next if $name =~ /\A\./;           # skip over dot files
  $name = "$dirname/$name";          # patch up the path
  next unless -f $name and -r $name; # only readable files
  ...
}
```

For portability, you might want to use the `File::Spec::Functions` module that knows how to construct paths appropriate for the local system:

```
use File::Spec::Functions;

opendir my $somedir, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir $somedir) {
  next if $name =~ /\A\./;            # skip over dot files
  $name = catfile( $dirname, $name ); # patch up the path
  next unless -f $name and -r $name;  # only readable files
  ...
}
```

NOTE

The `Path::Class` module is a nicer interface to the same thing, but it doesn't come with Perl.

Without the patch, the file tests would have been checking files in the current directory rather than in the directory named in `$dirname` . This is the single most common mistake when using directory handles.

Manipulating Files and Directories

Perl is commonly used to wrangle files and directories. Because Perl grew up in a Unix environment and still spends most of its time there, most of the description in this chapter may seem Unix centric. But the nice thing is that to whatever degree possible, Perl works exactly the same way on non-Unix systems.

Removing Files

Most of the time, you make files so that the data can stay around for a while. But when the data has outlived its usefulness, it's time to make the file go away. At the Unix shell level, you type an *rm* command to remove a file or files:

```
$ rm slate bedrock lava
```

In Perl, you use the `unlink` operator with a list of the files that you want to remove:

```
unlink 'slate', 'bedrock', 'lava';

unlink qw(slate bedrock lava);
```

This sends the three named files away to bit heaven, never to be seen again.

The link is between a filename and something stored on the disk, but some filesystems allow multiple "hard" links to the data. The data is freed once all of those links disappear. `unlink` dissociates a file entry from the data. If that happens to be the last link, the filesystem can reuse that space.

Now, since `unlink` takes a list and the `glob` function returns a list, you can combine the two to delete many files at once:

```
unlink glob '*.o';
```

This is similar to `rm *.o` at the shell, except that you didn't have to fire off a separate `rm` process. So you can make those important files go away that much faster!

The return value from `unlink` tells you how many files have been successfully deleted. So, back to the first example, you can check its success:

```
my $successful = unlink "slate", "bedrock", "lava";
print "I deleted $successful file(s) just now\n";
```

Sure, if this number is `3`, you know it removed all the files, and if it's `0`, it removed none of them. But what if it's `1` or `2`? Well, there's no clue which ones had problems. If you need to know, do them one at a time in a loop:

```
foreach my $file (qw(slate bedrock lava)) {
  unlink $file or warn "failed on $file: $!\n";
}
```

Here, each file being deleted one at a time means the return value will be `0` (failed) or `1` (succeeded), which happens to look like a nice Boolean value, controlling the execution of `warn`. Using `or warn` is similar to `or`

`die`, except that it's not fatal, of course (as we said back in Chapter 5). In this case, you put the newline on the end of the message to warn because it's not a bug in *your* program that causes the message.

When a particular `unlink` fails, Perl sets the `$!` variable to something related to the operating system error, which you can include in the message. This makes sense to use only when you're checking one filename at a time because the next operating system failed request resets the variable. You can't remove a directory with `unlink`, just like you can't remove a directory with the simple `rm` invocation either. Look for the `rmdir` function coming up shortly for that.

Now, here's a little-known Unix fact. It turns out that you can have a file that you can't read, you can't write, you can't execute—maybe you don't even own the file—but you can still delete that file. The permission to unlink a file doesn't depend on the permission bits on the file itself; it's the permission bits on the directory that contains the file that matter.

We mention this because it's normal for a beginning Perl programmer, in the course of trying out `unlink`, to make a file, to `chmod` it to `0` (so that it's not readable or writable), and then to see whether this makes `unlink` fail. But instead it vanishes without so much as a whimper. If you really want to see a failed `unlink`, though, just try to remove */etc/hosts* or a similar system file. Since that's a file controlled by the system administrator, you won't be able to remove it.

Renaming Files

Giving an existing file a new name is simple with the `rename` function:

```
rename 'old', 'new';
```

This is similar to the Unix `mv` command, taking a file named *old* and giving it the name *new* in the same directory. You can even move things around:

```
rename 'over_there/some/place/some_file', 'some_file';
```

Some people like to use the fat arrow that you saw in Chapter 6 ("The Big Arrow"), so they remind themselves which way the rename happens:

```
rename 'over_there/some/place/some_file' => 'some_file';
```

This moves a file called `some_file` from another directory into the current directory, provided the user running the program has the appropriate permissions and you aren't trying to copy files to another disk partition. This merely renames the file entry; it doesn't move any data.

Like most functions that request something of the operating system, `rename` returns false if it fails, and sets `$!` with the operating system error, so you can (and often should) use `or die` (or `or warn`) to report this to the user.

One frequent question is how to batch-rename a list of files, perhaps from those that end with *.old* to the same name with *.new*. Here's how to do it nicely in Perl:

```
foreach my $file (glob "*.old") {
  my $newfile = $file;
  $newfile =~ s/\.old\z/.new/;
  if (-e $newfile) {
    warn "can't rename $file to $newfile: $newfile exists\n";
  } elsif (rename $file => $newfile) {
    # success, do nothing
  } else {
    warn "rename $file to $newfile failed: $!\n";
  }
}
```

The check for the existence of `$newfile` is needed because `rename` will happily rename a file right over the top of an existing file, presuming the user has permission to remove the destination filename. You put the check in so that it's less likely that you'll lose information this way. Of

course, if you *wanted* to replace existing files like *wilma.new*, you wouldn't bother testing with `-e` first.

Those first two lines inside the loop can be combined (and often are) to simply:

```
(my $newfile = $file) =~ s/\.old\z/.new/;
```

This works to declare `$newfile`, copy its initial value from `$file`, then modify `$newfile` with the substitution. You can read this as "transform `$file` to `$newfile` using this replacement on the right." And yes, because of precedence, those parentheses are required.

That's a bit easier in Perl 5.14 with the `/r` flag to the `s///` operator. This line looks almost the same but lacks the parentheses:

```
use v5.14;

my $newfile = $file =~ s/\.old\z/.new/r;
```

Also, some programmers seeing this substitution for the first time wonder why the backslash is needed on the left but not on the right. The two sides aren't symmetrical: the left part of a substitution is a regular expression, and the right part is a double-quoted string. So you use the pattern `/\.old$/` to mean "`.old` anchored at the end of the string" (anchored at the end because you don't want to rename the *first* occurrence of `.old` in a file called *betty.old.old*), but on the right you can simply write `.new` to make the replacement.

Links and Files

To understand more about what's going on with files and directories, it helps to understand the Unix model of files and directories, even if your non-Unix system doesn't work in exactly this way. As usual, there's more to the story than we're able to explain here, so check any good book on Unix internal details if you need the full story.

A *mounted volume* is a hard disk drive (or something else that works more or less like that, such as a disk partition, a solid state device, a floppy disk, a CD-ROM, or a DVD-ROM). It may contain any number of files and directories. Each file is stored in a numbered *inode*, which we can think of as a particular piece of disk real estate. One file might be stored in inode 613, while another is in inode 7033.

To locate a particular file, though, you look it up in a directory. A directory is a special kind of file maintained by the system. Essentially, it is a table of filenames and their inode numbers. Along with the other things in the directory, there are always two special directory entries. One is . (called "dot"), which is the name of that very directory; and the other is .. ("dot-dot"), which is the directory one step higher in the hierarchy (i.e., the directory's parent directory). Figure 13-1 provides an illustration of two inodes. One is for a file called *chicken* and the other is Barney's directory of poems, */home/barney/poems*, which contains that file. The file is stored in inode 613, while the directory is stored in inode 919. (The directory's own name, *poems*, doesn't appear in the illustration, because it's stored in another directory.) The directory contains entries for three files (including *chicken*) and two directories (one of which is the reference back to the directory itself, in inode 919), along with each item's inode number.

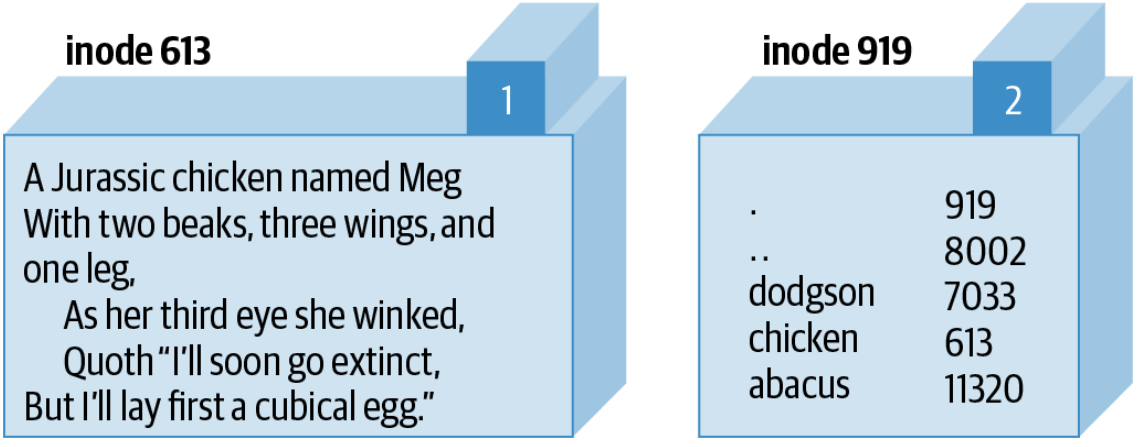

Figure 13-1. The chicken before the egg

When it's time to make a new file in a given directory, the system adds an entry with the file's name and the number of a new inode. How can the system tell that a particular inode is available, though? Each inode holds a number called its *link count*. The link count is always 0 if the inode isn't

listed in any directory, so any inode with a link count of 0 is available for new file storage. When the inode is added to a directory, the link count is incremented; when the listing is removed, the link count is decremented. For the file *chicken* illustrated in Figure 13-1, the inode count of `1` is shown in the box above the inode's data.

But some inodes have more than one listing. For example, you've already seen that each directory entry includes ., which points back to that directory's own inode. So the link count for a directory should always be at least two: its listing in its parent directory and its listing in itself. In addition, if it has subdirectories, each of those will add a link, since each will contain .. . In Figure 13-1, the directory's inode count of `2` is shown in the box above its data. A link count is the number of true names for the inode. Could an ordinary file inode have more than one listing in the directory? It certainly could. Suppose that, working in the directory shown in the figure, Barney uses the Perl `link` function to create a new link:

```
link 'chicken', 'egg'
  or warn "can't link chicken to egg: $!";
```

This is similar to typing `ln chicken egg` at the Unix shell prompt. If `link` succeeds, it returns true. If it fails, it returns false and sets `$!`, which Barney is checking in the error message. After this runs, the name *egg* is another name for the file *chicken*, and vice versa; neither name is "more real" than the other, and (as you may have guessed) it would take some detective work to find out which came first. Figure 13-2 shows a picture of the new situation, where there are two links to inode 613.

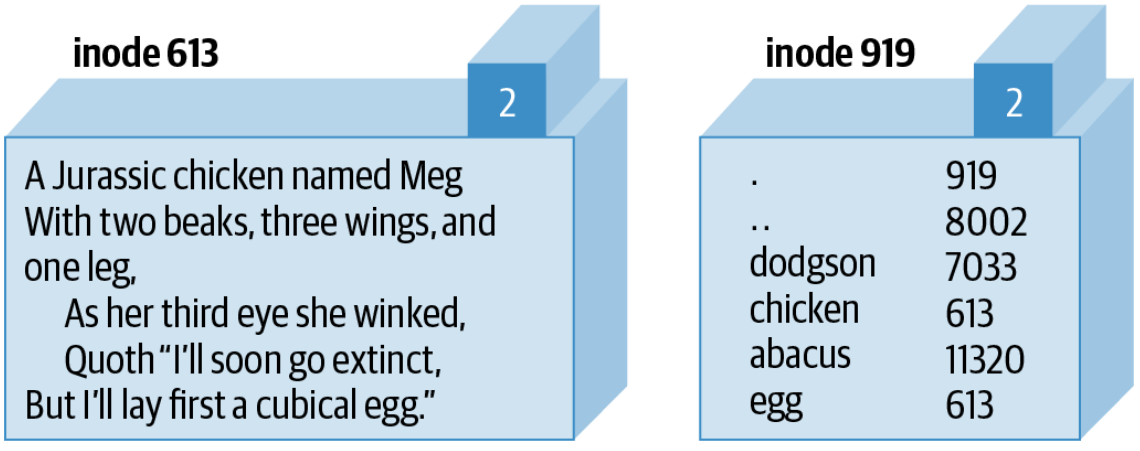

Figure 13-2. The egg is linked to the chicken

These two filenames are thus talking about the same place on the disk. If the file *chicken* holds 200 bytes of data, *egg* holds the same 200 bytes, for a total of 200 bytes (since it's really just one file with two names). If Barney appends a new line of text to the file *egg*, that line will also appear at the end of *chicken*. Now, if Barney were to accidentally (or intentionally) delete *chicken*, that data would not be lost—it's still available under the name *egg*. And vice versa: if he were to delete *egg*, he would still have *chicken*. Of course, if he were to delete both of them, the data would be lost. There's another rule about the links in directory listings: the inode numbers in a given directory listing all refer to inodes on that same mounted volume. This rule ensures that if you move the physical medium (a thumb drive, perhaps) to another machine, all the directories stick together with their files. That's why you can use `rename` to move a file from one directory to another, but only if both directories are on the same filesystem (mounted volume). If they were on different disks, the system would have to relocate the inode's data, which is too complex an operation for a simple system call.

And yet another restriction on links is that they can't make new names for directories. That's because the directories are arranged in a hierarchy. If you were able to change that, utility programs like *find* and *pwd* could easily become lost trying to find their way around the filesystem.

So you can't add links to directories, and they can't cross from one mounted volume to another. Fortunately, there's a way to get around these restrictions on links, by using a new and different kind of link: a *symbolic link*. A symbolic link (also called a *soft link* to distinguish it from the true or *hard links* that we've been talking about up to now) is a special entry in a directory that tells the system to look elsewhere. Let's say that Barney (working in the same directory of poems as before) creates a symbolic link with Perl's `symlink` function, like this:

```
symlink 'dodgson', 'carroll'
  or warn "can't symlink dodgson to carroll: $!";
```

This is similar to what would happen if Barney used the command *ln -s dodgson carroll* from the shell. Figure 13-3 shows a picture of the result,

including the poem in inode 7033.

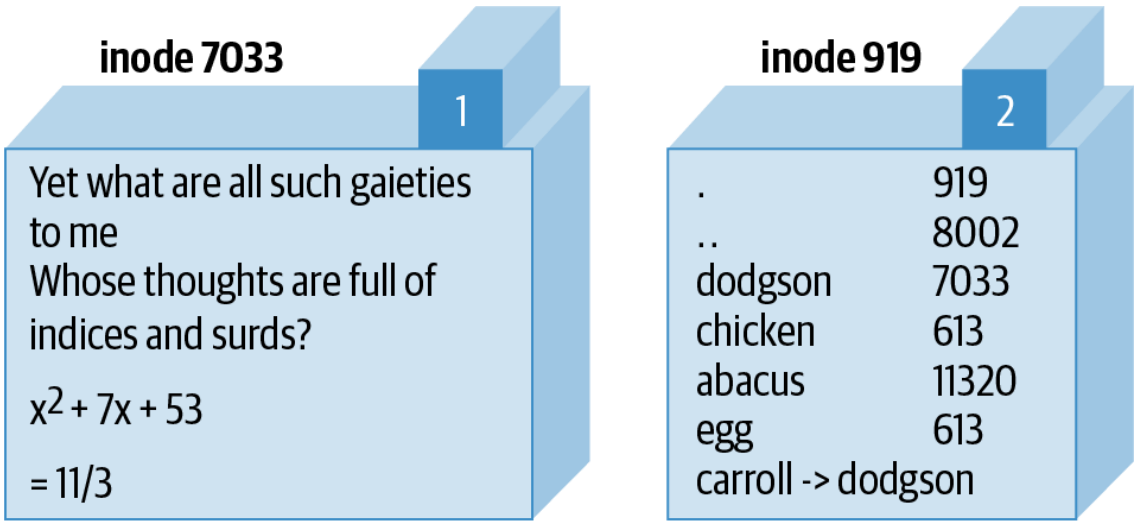

Figure 13-3. A symlink to inode 7033

Now if Barney chooses to read */home/barney/poems/carroll*, he gets the same data as if he had opened */home/barney/poems/dodgson* because the system follows the symbolic link automatically. But that new name isn't the "real" name of the file because (as you can see in the diagram) the link count on inode 7033 is still just one. That's because the symbolic link simply tells the system, "If you got here looking for *carroll*, now you want to go off to find something called *dodgson* instead."

A symbolic link can freely cross mounted filesystems or provide a new name for a directory, unlike a hard link. In fact, a symbolic link could point to any filename, one in this directory or in another one—or even to a file that doesn't exist! But that also means that a soft link can't keep data from being lost as a hard link can, since the symlink doesn't contribute to the link count. If Barney were to delete *dodgson*, the system would no longer be able to follow the soft link. Even though there would still be an entry called *carroll*, trying to read from it would give an error like `file not found`. The file test `-l 'carroll'` would report true, but `-e 'carroll'` would be false: it's a symlink, but its target doesn't exist. Deleting *carroll* would merely remove the symlink, of course.

Since a soft link could point to a file that doesn't yet exist, it could be used when creating a file as well. Barney has most of his files in his home directory, */home/barney*, but he also needs frequent access to a directory

with a long name that is difficult to type: */usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin*. So he sets up a symlink named */home/barney/my_stuff*, which points to that long name, and now it's easy for him to get to it. If he creates a file (from his home directory) called *my_stuff/bowling*, that file's real name is */usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin/bowling*. Next week, when the system administrator moves these files of Barney's to */usr/local/opt/internal/httpd/www-dev/users/staging/barney/cgi-bin*, Barney just repoints the one symlink, and now he and all of his programs can still find his files with ease.

It's normal for either */usr/bin/perl* or */usr/local/bin/perl* (or both) to be symbolic links to the true Perl binary on your system. This makes it easy to switch to a new version of Perl. Say you're the system administrator, and you've built the new Perl. Of course, your older version is still running and you don't want to disrupt anything. When you're ready for the switch, you simply move a symlink or two, and now every program that begins with *#!/usr/bin/perl* will automatically use the new version. In the unlikely case that there's some problem, it's a simple thing to replace the old symlinks and have the older Perl running the show again. (But like any good admin, you notified your users to test their code with the new */usr/bin/perl-7.2* well in advance of the switch, and you told them they can keep using the older one during the next month's grace period by changing their programs' first lines to *#!/usr/bin/perl-6.1*, if they need to.)

Perhaps surprisingly, both hard and soft links are very useful. Many non-Unix operating systems have neither, and the lack is sorely felt. On some non-Unix systems, symbolic links may be implemented as a "shortcut" or an "alias"—check the perlport documentation for the latest details.

To find out where a symbolic link is pointing, use the `readlink` function. This will tell you where the symlink leads, or it will return `undef` if its argument wasn't a symlink:

```
my $where = readlink 'carroll';             # Gives "dodgson"

my $perl = readlink '/usr/local/bin/perl';  # Maybe tells where perl is
```

You can remove either kind of link with `unlink`—and now you see where that operation gets its name. `unlink` simply removes the directory entry associated with the given filename, decrementing the link count and thus possibly freeing the inode.

Making and Removing Directories

Making a directory inside an existing directory is easy. Just invoke the `mkdir` function:

```
mkdir 'fred', 0755 or warn "Cannot make fred directory: $!";
```

Again, true means success, and Perl sets `$!` on failure.

But what's that second parameter, `0755`? That's the initial permission setting on the newly created directory (you can always change it later). The value here is specified as an octal because the value will be interpreted as a Unix permission value, which has a meaning based on groups of three bits each, and octal values represent that nicely. Yes, even in Windows or MacPerl, you still need to know a little about Unix permission values to use the `mkdir` function. Mode `0755` is a good one to use because it gives you full permission, but lets everyone else have read access but no permission to change anything.

The `mkdir` function doesn't require you to specify this value in octal—it's just looking for a numeric value (either a literal or a calculation). But unless you can quickly figure that `0755` octal is `493` decimal in your head, it's probably easier to let Perl calculate that. And if you accidentally leave off the leading zero, you get `755` decimal, which is `1363` octal, a strange permission combination indeed.

As you saw earlier (in Chapter 2), a string value being used as a number is never interpreted as octal, even if it starts with a leading zero. So this doesn't work:

```
my $name = "fred";
my $permissions = "0755";  # danger...this isn't working
mkdir $name, $permissions;
```

Oops, you just created a directory with the bizarre `01363` permissions because `0755` was treated as a decimal. To fix that, use the `oct()` function, which forces octal interpretation of a string whether or not there's a leading zero:

```
mkdir $name, oct($permissions);
```

Of course, if you are specifying the permission value directly within the program, just use a number instead of a string. The need for the extra `oct()` function shows up most often when the value comes from user input. For example, suppose you take the arguments from the command line:

```
my ($name, $perm) = @ARGV;  # first two args are name, permissions
mkdir $name, oct($perm) or die "cannot create $name: $!";
```

The value here for `$perm` is initially interpreted as a string, and thus the `oct()` function interprets the common octal representation properly.

To remove empty directories, use the `rmdir` function in a manner similar to the `unlink` function, although it can only remove one directory per call:

```
foreach my $dir (qw(fred barney betty)) {
  rmdir $dir or warn "cannot rmdir $dir: $!\n";
}
```

The `rmdir` operator fails for nonempty directories. As a first pass, you can attempt to delete the contents of the directory with `unlink`, then try to remove what should now be an empty directory. For example, suppose you need a place to write many temporary files during the execution of a program:

```
my $temp_dir = "/tmp/scratch_$$";       # based on process ID; see the text
mkdir $temp_dir, 0700 or die "cannot create $temp_dir: $!";
...
# use $temp_dir as location of all temporary files
...
unlink glob "$temp_dir/* $temp_dir/.*"; # delete contents of $temp_dir
rmdir $temp_dir;                        # delete now-empty directory
```

NOTE

If you really need to create temporary directories or files, check out the `File::Temp` module, which comes with Perl.

The initial temporary directory name includes the current process ID, which is unique for every running process and is accessed with the `$$` variable (similar to the shell). You do this to avoid colliding with any other processes, as long as they also include their process IDs as part of their pathnames. (In fact, it's common to use the program's name as well as the process ID, so if the program is called *quarry*, the directory would probably be something like */tmp/quarry_$$*.)

At the end of the program, that last `unlink` should remove all the files in this temporary directory, and then the `rmdir` function can delete the now-empty directory. However, if you've created subdirectories under that directory, the `unlink` operator fails on those, and the `rmdir` also fails. For a more robust solution, check out the `remove_tree` function provided by the `File::Path` module of the standard distribution.

Modifying Permissions

The Unix `chmod` command changes the permissions on a file or directory. Similarly, Perl has the `chmod` function to perform this task:

```
chmod 0755, 'fred', 'barney';
```

As with many of the operating system interface functions, `chmod` returns the number of items successfully altered, and when used with a single argument, sets `$!` in a sensible way for error messages when it fails. The first parameter is the Unix permission value (even for non-Unix versions of Perl). For the same reasons we presented earlier in describing `mkdir`, this value is usually specified in octal.

Symbolic permissions (like `+x` or `go=u-w`) accepted by the Unix *chmod* command are not valid for the `chmod` function.

NOTE

The `File::chmod` module from CPAN can upgrade the `chmod` operator to understand symbolic mode values.

Changing Ownership

If the operating system permits it, you may change the ownership and group membership of a list of files (or filehandles) with the `chown` function. The user and group are both changed at once, and both have to be the numeric user ID and group ID values. For example:

```
my $user  = 1004;
my $group = 100;
chown $user, $group, glob '*.o';
```

What if you have a username like `merlyn` instead of the number? Simple. Just call the `getpwnam` function to translate the name into a number, and the corresponding `getgrnam` to translate the group name into its number:

```
defined(my $user = getpwnam 'merlyn') or die 'bad user';
defined(my $group = getgrnam 'users') or die 'bad group';
chown $user, $group, glob '/home/merlyn/*';
```

The `defined` function verifies that the return value is not `undef`, which will be returned if the requested user or group is not valid.

The `chown` function returns the number of files affected, and it sets `$!` on error.

Changing Timestamps

In those rare cases when you want to lie to other programs about when a file was most recently modified or accessed, you can use the `utime` function to fudge the books a bit. The first two arguments give the new access time and modification time, while the remaining arguments are the list of filenames to alter those timestamps. The times are specified in internal timestamp format (the same type of values returned from the `stat` function that we mentioned in "The stat and lstat Functions").

One convenient value to use for the timestamps is "right now," returned in the proper format by the `time` function. To update all the files in the current directory to look like they were modified a day ago, but accessed just now, you could simply do this:

```
my $now = time;
my $ago = $now - 24 * 60 * 60;  # seconds per day
utime $now, $ago, glob '*';     # set access to now, mod to a day ago
```

Of course, nothing stops you from creating a file that is arbitrarily stamped far in the future or past (within the limits of the Unix timestamp values of 1970 to 2038, or whatever your non-Unix system uses, unless you have 64-bit timestamps). Maybe you could use this to create a directory where you keep your notes for that time-travel novel you're writing.

The third timestamp (the `ctime` value) is always set to "now" whenever anything alters a file, so there's no way to set it (it would have to be reset to "now" after you set it) with the `utime` function. That's because its primary purpose is for incremental backups: if the file's `ctime` is newer than the date on the backup tape, it's time to back it up again.

Exercises

The programs here are potentially dangerous! Be careful to test them in a mostly empty directory to make it difficult to accidentally delete something useful.

See "Answers to Chapter 13 Exercises" for answers to these exercises:

1. [12] Write a program to ask the user for a directory name, then change to that directory. If the user enters a line with nothing but whitespace, change to their home directory as a default. After changing, list the ordinary directory contents (not the items whose names begin with a dot) in alphabetical order. (Hint: will that be easier to do with a directory handle or with a glob?) If the directory change doesn't succeed, just alert the user—but don't try showing the contents.
2. [4] Modify the program to include all files, not just the ones that don't begin with a dot.
3. [5] If you used a directory handle for the previous exercise, rewrite it to use a glob. Or if you used a glob, try it now with a directory handle.
4. [6] Write a program that works like *rm*, deleting any files named on the command line. (You don't need to handle any of the options of *rm*.)
5. [10] Write a program that works like *mv*, renaming the first command-line argument to the second command-line argument. (You don't need to handle any of the options of *mv* or additional arguments.) Remember to allow for the destination to be a directory; if it is, use the same original basename in the new directory.
6. [7] If your operating system supports it, write a program that works like *ln*, making a hard link from the first command-line argument to the second. (You don't need to handle options of *ln* or more arguments.) If your system doesn't have hard links, just print out a message telling which operation you would perform if it were available. Hint: this program has something in common with the previous one—recognizing that could save you time in coding.
7. [7] If your operating system supports it, fix up the program from the previous exercise to allow an optional *-s* switch before the other arguments to indicate that you want to make a soft link instead of a hard

link. (Even if you don't have hard links, see whether you can at least make soft links with this program.)

8. [7] If your operating system supports it, write a program to find any symbolic links in the current directory and print out their values (like *ls -l* would: `name -> value`).