6

Files and Data

We're starting to write real programs now, and real programs need to be able to read and write files to and from your hard drive. At the moment, all we can do is ask the user for input using `<STDIN>` and print data on the screen using `print`. Pretty simple stuff, yes, but these two ideas actually form the basis of a great deal of the file handling you'll be doing in Perl.

What we want to do in this chapter is extend these techniques into reading from and writing to files, and we'll also look at the other techniques we have for handling files, directories, and data.

Filehandles

First though, let's do some groundwork. When we're dealing with files, we need something that tells Perl which file we're talking about, which allows us to refer to and access a certain file on the disk. We need a label, something that will give us a 'handle' on the file we want to work on. For this reason, the 'something' we want is known as a **filehandle**.

We've actually already seen a filehandle: the **`STDIN`** of `<STDIN>`. This is a filehandle for the special file 'standard input', and whenever we've used the idiom `<STDIN>` to read a line, we've been reading from the standard input file. Standard input is the input provided by a user either directly as we've seen, by typing on the keyboard, or indirectly, through the use of a 'pipe' that (as we'll see) pumps input into the program.

As a counterpart to standard input, there's also standard output: **`STDOUT`**. Conversely, it's the output we provide to a user, which at the moment we're doing by writing to the screen. Every time we've used the function print so far, we've been implicitly using `STDOUT`:

```
print STDOUT "Hello, world.\n";
```

is just the same as our original example in Chapter 1. There's one more 'standard' filehandle: standard error, or **`STDERR`**, which is where we write the error messages when we `die`.

Every program has these three filehandles available, at least at the beginning of the program. To read and write from other files, though, you'll want to open a filehandle of your own. Filehandles are usually one-way: You can't write to the user's keyboard, for instance, or read from his or her screen. Instead, filehandles are open either for reading or for writing, for input or for output. So, here's how you'd open a filehandle for reading:

```
open FH, $filename or die $!;
```

The operator for opening a filehandle is `open`, and it takes two arguments, the first being the name of the filehandle we want to open. Filehandles are slightly different from ordinary variables, and they do not need to be declared with `my`, even if you're using `strict` as you should. It's traditional to use all-capitals for a filehandle to distinguish them from keywords.

The second argument is the file's name – either as a variable, as shown above, or as a string literal, like this:

```
open FH, 'output.log' or die $!;
```

You may specify a full path to a file, but don't forget that if you're on Windows, a backslash in a double-quoted string introduces an escape character. So, for instance, you should say this:

```
open FH, 'c:/test/news.txt' or die $!;
```

rather than:

```
open FH, "c:\test\news.txt" or die $!;
```

as `\t` in a double-quoted string is a tab, and `\n` is a new line. You could also say `"c:\\test\\news.txt"` but that's a little unwieldy. My advice is to make use of the fact that Windows allows forward slashes internally, and forward slashes do not need to be escaped: `"c:/test/news.txt"` should work perfectly fine.

So now we have our filehandle open – or have we? As I mentioned in Chapter 4, the `X or Y` style of conditional is often used for ensuring that operations were successful. Here is the first real example of this.

When you're dealing with something like the file system, it's dangerous to blindly assume that everything you are going to do will succeed. A file may not be present when you expect it to be, a file name you are given may turn out to be a directory, something else may be using the file at the time, and so on. For this reason, you need to check that the `open` did actually succeed. If it didn't, we `die`, and our message is whatever is held in `$!`.

What's `$!`? This is one of Perl's **special variables**, designed to give you a way of getting at various things that Perl wants to tell you. In this case, Perl is passing on an error message from the system, and this error message should tell you why the `open` failed: It's usually something like 'No such file or directory' or 'permission denied'.

There are special variables to tell you what version of Perl you are running, what user you are logged in as on a multi-user system, and so on. Appendix B contains a complete description of Perl's special variables.

So, for instance, if we try and open a file that is actually a directory, this happens:

```
#!/usr/bin/perl
# badopen.plx
use warnings;
use strict;
open BAD, "/temp" or die "We have a problem: $!";
```

```
>perl badopen.plx
Name "main::BAD" used only once: possible typo at badopen.plx line 5
We have a problem: Permission denied at badopen.plx line 5.
>
```

The first line we see is a warning. If we were to finish the program, adding further operations on `BAD` (or get rid of `use warnings`), it wouldn't show up.

You should also note that if the argument you give to `die` does not end with a new line, Perl automatically adds the name of the program and the location that had the problem. If you want to avoid this, always remember to put new lines on the end of everything you `die` with.

Reading Lines

Now that we can open a file, we can then move on to reading the file one line at a time. We do this by replacing the `STDIN` filehandle in `<STDIN>` with our new filehandle, to get `<FH>`. Just as `<STDIN>` reads a single line from the keyboard, `<FH>` reads one line from a filehandle. This <…> construction is called the **diamond operator**, or **readline operator**:

Try It Out : Numbering Lines

We'll use the `<FH>` construct in conjunction with a `while` loop to go through each line in a file. So then, to print a file with line numbers added, you can say something like this:

```
#!/usr/bin/perl
# nl.plx
use warnings;
use strict;

open FILE, "nlexample.txt" or die $!;
my $lineno = 1;

while (<FILE>) {
   print $lineno++;
   print ": $_";
}
```

Now, create the file `nlexample.txt` with the following contents:

```
One day you're going to have to face
   A deep dark truthful mirror,
And it's gonna tell you things that I still
   Love you too much to say.
####### Elvis Costello, Spike, 1988 #######
```

This is what you should see when you run the program:

```
> perl nl.plx
1: One day you're going to have to face
2:      A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:      Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
>
```

How It Works

We begin by opening our file and making sure it was opened correctly:

```
    open FILE, "nlexample.txt" or die $!;
```

Since we're expecting our line numbers to start at one, we'll initialize our counter:

```
    my $lineno = 1;
```

Now we read each line from the file in turn, which we do with a little magic:

```
    while (<FILE>) {
```

This syntax is actually equivalent to:

```
    while (defined ($_ = <FILE>)) {
```

That is, we read a line from a file and assign it to `$_`, and we see whether it is defined. If it is, we do whatever's in the loop. If not, we are probably at the end of the file so we need to come out of the loop. This gives us a nice, easy way of setting `$_` to each line in turn.

As we have a new line, we print out its line number and advance the counter:

```
        print $lineno++;
```

Finally, we print out the line in question:

```
        print ": $_";
```

There's no need to add a newline since we didn't bother `chomp`ing the incoming line. Of course, using a statement modifier, we can make this even more concise:

```
    open FILE, "nlexample.txt" or die $!;
    my $lineno = 1;

    print $lineno++, ": $_" while <FILE>
```

But since we're going to want to expand the capabilities of our program -adding more operations to the body of the loop – we're probably better off with the original format.

Creating Filters

As well as the three standard filehandles, Perl provides a special filehandle called **ARGV**. This reads the names of files from the command line and opens them all, or if there is nothing specified on the command line, it reads from standard input. Actually, the `@ARGV` array holds any text after the program's name on the command line, and `<ARGV>` takes each file in turn. This is often used to create filters, which read in data from one or more files, process it, and produce output on `STDOUT`.

Because it is used so commonly, Perl provides an abbreviation for `<ARGV>`: an empty diamond, or `<>`. We can make our line counter a little more flexible by using this filehandle:

```
#!/usr/bin/perl
# nl2.plx
use warnings;
use strict;

my $lineno = 1;

while (<>) {
   print $lineno++;
   print ": $_";
}
```

Now Perl expects us to give the name of the file on the command line:

```
> perl nl2.plx nlexample.txt
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
>
```

We can actually place a fair number of files on the command line, and they'll all be processed together. For example:

```
> perl nl2.plx nlexample.txt nl2.plx
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
6: #!/usr/bin/perl
7: # nl2.plx
8: use warnings;
9: use strict;
10:
11: my $lineno = 1;
12:
13: while (<>) {
14:     print $lineno++;
15:     print ": $_";
16: }
```

If we need to find out the name of the file we're currently reading, it's stored in the special variable `$ARGV`. We can use this to reset the counter when the file changes.

Try it out : Numbering Lines in Multiple Files

By detecting when `$ARGV` changes, we can reset the counter and display the name of the new file:

```
#!/usr/bin/perl
# nl3.plx
use warnings;
use strict;

my $lineno;
my $current = "";

while (<>) {
   if ($current ne $ARGV) {
      $current = $ARGV;
      print "\n\t\tFile: $ARGV\n\n";
      $lineno=1;
   }

   print $lineno++;
   print ": $_";
}
```

And now we can run this on our example file and itself:

```
> perl nl3.plx nlexample.txt nl3.plx

            File: nlexample.txt

1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######

            File: nl3.plx

1: #!/usr/bin/perl
2: # nl3.plx
3: use warnings;
4: use strict;
5:
6: my $lineno;
7: my $current = "";
8:
9: while (<>) {
10:     if ($current ne $ARGV) {
11:            $current = $ARGV;
12:            print "\n\t\tFile: $ARGV\n\n";
13:            $lineno=1;
14:    }
15:
16:    print $lineno++;
17:    print ": $_";
18: }
>
```

How It Works

This is a technique you'll often see in programming to detect when a variable has changed. `$current` is meant to contain the current value of `$ARGV`. But if it doesn't, `$ARGV` has changed:

```
    if ($current ne $ARGV) {
```

so we set `$current` to what it should be – the new value – so we can catch it again next time:

```
        $current = $ARGV;
```

We then print out the name of the new file, offset by new lines and tabs:

```
        print "\n\t\tFile: $ARGV\n\n";
```

and reset the counter so we start counting the new file from line one again.

```
        $lineno=1;
    }
```

As with most tricks like these, it's actually really simple to code it once you've seen how it's coded. The catch is having to solve problems like these for the first time by yourself.

Reading More than One Line

Sometimes we'll want to read more than one line at once. When you use the diamond operator in a scalar context, as we've been doing so far, it'll provide you with the next line. However, in a list context, it will return all of the remaining lines. For instance, you can read in an entire file like this:

```
open INPUT, "somefile.dat" or die $!;
my @data;
@data = <INPUT>;
```

This is, however, quite memory-intensive. Perl has to store every single line of the file into the array, whereas you may only want to be dealing with one or two of them. Usually, you'll want to step over a file with a `while` loop as before. However, for some things, an array is the easiest way of doing things. For example, how do you print the last five lines in a file?

The problem with reading a line at a time is that you don't know how much text left you've got to read. You can only tell when you run out of data, so you'd have to keep an array of the last five lines read and drop an old line when a new one comes in. You'd do it something like this:

```
#!/usr/bin/perl
# tail.plx
use warnings;
use strict;

open FILE, "gettysburg.txt" or die $!;
my @last5;

while (<FILE>) {
   push @last5, $_; # Add to the end
   shift @last5 if @last5 > 5; # Take from the beginning
}

print "Last five lines:\n", @last5;
```

And that's exactly how you'd do it if you were concerned about memory use on big files. Given a suitably primed `gettysburg.txt`, this is what you'd get:

```
>perl tail.plx
Last five lines:
- that from these honored dead we take increased devotion to that cause for
which they gave the last full measure of devotion - that we here highly resolve
that these dead shall not have died in vain, that this nation under God shall
have a new birth of freedom, and that government of the people, by the people,
for the people shall not perish from the earth.
>
```

However, if memory wasn't a problem, or you knew you were going to be primarily dealing with small files, this would be perfectly sufficient:

```
#!/usr/bin/perl
# tail2.plx
use warnings;
use strict;

open FILE, "gettysburg.txt" or die $!;
my @speech = <FILE>;

print "Last five lines:\n", @speech[-5 ... -1];
```

What's My Line (Separator)?

So far we've been reading in single lines – a series of characters ending in a new line. One of the other things we can do is to alter Perl's definition of what separates a line.

The special variable `$/` is called the 'input record separator'. Usually, it's set to be the newline character, `\n`, and each 'record' is a line. We might say more correctly that `<FILE>` reads a single **record** from the file. Furthermore, `chomp` doesn't just remove a trailing new line – it removes a trailing record separator. However, we can set this separator to whatever we want, and this will change the way Perl sees lines. So if, for instance, our data was defined in terms of paragraphs, rather than lines, we could read one paragraph at a time by changing `$/`.

Try It Out : Fortune Cookie Dispenser

The fortune cookies file for the UNIX `fortune` program – as well as some 'tagline' generators for e-mail and news articles – consist of paragraphs separated by a percent sign on a line of its own, like this:

```
We all agree on the necessity of compromise.  We just can't agree on
when it's necessary to compromise.
    -- Larry Wall
%
All language designers are arrogant.  Goes with the territory...
    -- Larry Wall
%
Oh, get a hold of yourself. Nobody's proposing that we parse English.
    -- Larry Wall
%
Now I'm being shot at from both sides. That means I *must* be right.
     -- Larry Wall
%
```

Save this as `quotes.dat` and then write a program to pick a random quote from the file:

```
#!/usr/bin/perl
# fortune.plx
use warnings;
use strict;

$/ = "\n%\n";

open QUOTES, "quotes.dat" or die $!;
my @file = <QUOTES>;

my $random = rand(@file);
my $fortune = $file[$random];
chomp $fortune;

print $fortune, "\n";
```

This is what you get (or might get – it is random, after all):

```
> perl fortune.plx
Now I'm being shot at from both sides. That means I *must* be right.
   -- Larry Wall
>
```

How It Works

Once we've set our record separator appropriately, most of the work is already done for us. This is how we change it:

```
$/ = "\n%\n";
```

Now a 'line' is everything up to a newline character and then a percent sign on its own and then another new line, and when we read the file into an array, it ends up looking something like this:

```
my @file = (
   "We all agree on the necessity of compromise.  We just can't agree on
when it's necessary to compromise.\n    -- Larry Wall\n%\n",
   "All language designers are arrogant.  Goes with the territory...\n    -- Larry
Wall\n%\n",
   ...
);
```

We want a random line from the file. The operator for this is `rand`:

```
my $random = rand(@file);
my $fortune = $file[$random];
```

`rand` produces a random number between zero and the number given as an argument. What's the argument we give it? As you know, an array in a scalar context gives you the number of elements in the array. `rand` actually generates a fractional number, but when we look it up in the array, as we've seen in Chapter 3, Perl ignores the fractional part. Actually, it's more likely that in existing code you'll see those two statements combined into one, like this:

```
my $fortune = $file[rand @file];
```

Now we have our fortune, but it still has the record separator on the end, so we need to `chomp` to remove it:

```
chomp $fortune ;
```

Finally, we can print it back out, remembering that we need to put a new line on the end:

```
print $fortune, "\n";
```

Reading Paragraphs at a Time

If you set the input record separator, `$/`, to the empty string, "", Perl reads in a paragraph at a time. Paragraphs must be separated by a completely blank line, with no spaces on it at all. Of course, you can use `split` or similar to extract individual lines from each paragraph. This program creates a 'paragraph summary' by printing out the first line of each paragraph in a file:

Try It Out : Paragraph Summariser

We'll use `split` to get at the first line in each paragraph, and we'll number the paragraphs:

```
#!/usr/bin/perl
# summary.plx
use warnings;
use strict;

$/ = "";
my $counter = 1;

while (<>) {
   print $counter++, ":";
   print ((split /\n/, $_)[0]);
   print "\n";
}
```

When run on the beginning of this chapter, it gives the following output:

```
> perl summary.plx chapter6
1:We're starting to write real programs now, and real programs
2:What we want to do in this chapter is extend these techniques
3:First though, let's do some groundwork. When we're dealing
4:We've actually already seen a filehandle: the STDIN of <STDIN>.
5:As a counterpart to standard input, there's also standard
6:Every program has these three filehandles available, at least
>
```

We're assuming here that each line in the paragraph ends with a newline character rather than wrapping around to the next line. In the latter case, our program would return each of the paragraphs in their entirety, because split is being based on `\n`.

How It Works

This time we're reading from files specified on the command line, so we use the diamond operator. We start by putting the input record separator into paragraph mode:

```
$/ = "";
```

For every paragraph we read in, we print a new number, then get the first line of the paragraph:

```
print ((split /\n/, $_)[0]);
```

First we split the paragraph into lines, by `split`ting around a newline character. Since `split` just produces a list, we can take the first element of this list in the same way as any other.

Reading Entire Files

Finally, you may want to read a whole file into a single string. You could do this easily enough using `join`, but Perl provides another special value of `$/` for this. If we want to say that there is no record separator, we set `$/` to the undefined value. So, for instance, to read the whole of the above quotes file into a variable, we do this:

```
$/ = undef;
open QUOTES, "quotes.dat" or die $!;
my $file = <QUOTES>;
```

You may also see the form `undef $/` doing the same job: the `undef` operator gives a variable the undefined value.

Writing to Files

We've been using the `print` operator to print a list to standard output. We'll also use a different form of the `print` operator to print to a file. However, as we mentioned above, files are usually open either for reading *or* for writing – not both. We've been opening files and reading from them, but how do we open them for writing?

Opening a File for Writing

We actually use a syntax that's used by the shell for writing to files. In Windows and UNIX, if we want to put standard output into a file, we add the operator **>** and the file name to the end of the command. For example, saying something like this:

```
> perl summary.plx chapter6 > summary6
```

will create a file called `summary6`, which contains the following text:

```
1:We're starting to write real programs now, and real programs
2:What we want to do in this chapter is extend these techniques
3:First though, let's do some groundwork. When we're dealing
4:We've actually already seen a filehandle: the STDIN of <STDIN>.
5:As a counterpart to standard input, there's also standard
6:Every program has these three filehandles available, at least
```

Now, to open a file for writing, we do this:

```
open FH, "> $filename" or die $!;
```

This will either create a new file or completely wipe out the contents of an already existing file and let us start writing from the beginning. Don't use this on files you want to keep! If we want to add things to the end of an existing file, use two arrows, like this:

```
open FH, ">> $filename" or die $!;
```

There's no easy way of adding or changing text at the beginning or middle of a file. The typical way to do this is to read in the original and write the changed data to another file. We'll see shortly how this is done.

Similarly, you can redirect data to a program's standard *input* by using the left arrow, like this:

```
>perl summary.plx < chapter6.txt
```

As you've probably guessed, this means you can open files for input by saying:

```
open FH, "< $filename";
```

This is exactly the same as `open FH, $filename`; as we've used previously; it's just a little more explicit.

Writing on a Filehandle

We're now ready to write the file, which we'll do by using a special form of the `print` operator. Normally, to print things out from the screen, we say this:

```
print list;
```

When we want to write to a file, we'll use this instead:

```
print FH list;
```

So, for instance, here's one way of copying a file:

Try It Out : File Copying

We'll read in a file one line at a time, writing each line onto the new file:

```
#!/usr/bin/perl
# copy.plx
use warnings;
use strict;

my $source = shift @ARGV;
my $destination = shift @ARGV;
```

```
open IN, $source or die "Can't read source file $source: $!\n";
open OUT, ">$destination" or die "Can't write on file $destination: $!\n";

print "Copying $source to $destination\n";

while (<IN>) {
   print OUT $_;
}
```

Now there isn't much to see in this program, but let's run it anyway:

```
> perl copy.plx gettysburg.txt speech.txt
Copying gettysburg.txt to speech.txt
>
```

How It Works

We get the name of the file to copy from the command line:

```
my $source = shift @ARGV;
my $destination = shift @ARGV;
```

The command line arguments to our program are in the `@ARGV` array, as we saw in Chapter 4, and we use `shift` (which pops the top element of an array into a variable) to get an element out. We could quite easily have said this:

```
my $source = $ARGV[0];
my $destination = $ARGV[1];
```

However, `shift` is slightly more common. Next, open our two files:

```
open IN, $source or die "Can't read source file $source: $!\n";
open OUT, "> $destination" or die "Can't write on file $destination: $!\n";
```

The first of those lines should be familiar. The second, meanwhile, adds the arrow to show we want to write on that file. It's a double-quoted string so, as always, the destination file name is interpolated. Notice that we're taking care to check if the files can be opened for reading and writing; it is essential to let the user know if, for example, they do not have permission to access a certain file, or the file does not exist. There's never really good reason not to do this.

The copying procedure is simple enough: read a line from the source file, and then write it to the destination:

```
while (<IN>) {
   print OUT $_;
}
```

`<IN>` returns a list of as many lines as it can in list context. So the `while` loop steps through this list, copies each line to memory and `print`ing to the destination file `OUT`, one at a time for each cycle. So why don't we just say:

```
print OUT <IN>;
```

The trouble is, that's not very memory conscious. Perl would have to read in the *whole* file at once in order to construct the list and only then pass it out to `print`. For small files, this is fine. On the other hand, if we thought we could get away with reading the whole file into memory at one go, we also could do it this way:

```
$/ = undef;
print OUT <IN>;
```

This will read the whole file as a single entry, which is faster for sure, since Perl won't have to think about separating each line and building up a list, but still only suited to small files. Since we want to allow for large files, too, we'll stick with our original technique.

Let's see another example. This time, instead of writing the file straight out, we'll sort the lines in it first. In this case, we can't avoid reading in every line into memory. We need to have all the lines in an array or something similar. Let's see how we'd go about doing this.

Try It Out : File Sorter

If you've ever needed to sort the lines in a file, this is for you. The program works in three stages:

- First, open the files that the user specifies.
- Next, read in the file and sort it.
- Finally, write the sorted lines out.

Here's the full listing:

```
#!/usr/bin/perl
# sort.plx
use warnings;
use strict;

my $input = shift;
my $output = shift;
open INPUT, $input or die "Couldn't open file $input: $!\n";
open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";

my @file = <INPUT>;
@file = sort @file;

print OUTPUT @file;
```

Now if we have the following file, `sortme.txt`:

```
Well, I finally found someone to turn me upside-down
And nail my feet up where my head should be
If they had a king of fools then I could wear that crown
And you can all die laughing, because I'd wear it proudly
```

We can run our program like this:

```
>perl sort.plx sortme.txt sorted.txt
>
```

And we'll end up with a file, `sorted.txt`:

```
And nail my feet up where my head should be
And you can all die laughing, because I'd wear it proudly
If they had a king of fools then I could wear that crown
Well, I finally found someone to turn me upside-down
```

How It Works

The first stage, that of opening the files, is very similar to what we did before, with one small change:

```
my $input = shift;
my $output = shift;
open INPUT, $input or die "Couldn't open file $input: $!\n";
open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";
```

We don't tell Perl which array to `shift`, so it assumes we want `@ARGV`, which is just as well, because in this case, we do!

Getting the file sorted is a simple matter of reading it into an array and calling `sort` on the array:

```
my @file = <INPUT>;
@file = sort @file;
```

In fact, we could just say `my @file = sort <INPUT>;` and that would be slightly more efficient. Perl would only have to throw the list around once.

Finally, we write the sorted array out:

```
print OUTPUT @file;
```

We could even do all this in one line, without using an array:

```
print OUTPUT sort <INPUT>;
```

This is arguably the most efficient solution, and you might think it's relatively easy to understand. What are we doing after all? We're printing the sorted input file on the output file. But it's the least extensible way of writing it. We can't change any of the stages when it's written like that.

What could we change? Well, remember that there are at least two ways to sort things: `sort` usually does an ASCII-order sort, but this doesn't help us when we're sorting columns of numbers. To do that, we need to use the numeric comparison operator, `<=>`, when we're sorting. As we saw in Chapter 3, the syntax would be something like this:

```
@sorted = sort { $a <=> $b } @unsorted;
```

Let's now extend our sort program to optionally sort numerically. Add the following lines:

```
#!/usr/bin/perl
# sort2.plx
use warnings;
use strict;

my $numeric = 0;
my $input = shift;
if ($input eq "-n") {
   $numeric = 1;
   $input = shift;
}
my $output = shift;

open INPUT, $input or die "Couldn't open file $input: $!\n";
open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";

my @file = <INPUT>;
if ($numeric) {
   @file = sort { $a <=> $b } @file;
} else {
   @file = sort @file;
}

print OUTPUT @file;
```

What have we done? We've declared a flag, `$numeric`, which will tell us whether or not we're to do a numeric sort. If the first thing we see on the command line after our program's name is the string `-n`, then we're doing a numeric sort, and so we set our flag. Now that we've dealt with the `-n`, the input and output are the next two things on the command line. So we have to `shift` again.

Now that we've read the file in, we can choose which way we want to sort it: either normally, if `-n` was not given, or numerically if it was. If we have a file containing a list of numbers, called `sortnum.txt`, we can see the difference between the two methods:

>perl sort2.plx sortnum.txt sorted.txt

will write

```
121
1324515
13461
7446
576124
```

to the file `sorted.txt`, while:

>perl sort2.plx –n sortnum.txt sorted.txt

gives us:

```
121
7446
13461
576124
1324515
```

Try expanding the one-line version of `sort.plx` to match that.

Accessing Filehandles

Before we leave this program, there's one more thing we should do. One piece of programming design UNIX encourages is that it's better to string together lots of little things than deal with a huge program. Houses are built from individual bricks, not single lumps of rock. This is a design principle that's useful everywhere, not just on UNIX, and so let's try and make use of it here.

UNIX invented the use of **pipes** to connect programs. Perl supports these, and we'll see how they work later on. To make use of them, though, our program must be able to read lines from the standard input and put out sorted lines to the standard output in the event that no parameters were specified. Let's modify our program to do this:

Try It Out : Sort As A Filter

To see how many parameters have been passed, we'll test to see if `$input` and `$output` are defined after we shift them:

```
#!/usr/bin/perl
# sort3.plx
use warnings;
use strict;

my $numeric = 0;
my $input = shift;
if (defined $input and $input eq "-n") {
   $numeric = 1;
   $input = shift;
}
my $output = shift;

if (defined $input) {
   open INPUT, $input or die "Couldn't open file $input: $!\n";
} else {
   *INPUT = *STDIN;
}

if (defined $output) {
   open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";
} else {
   *OUTPUT = *STDOUT;
}

my @file = <INPUT>;
if ($numeric) {
   @file = sort { $a <=> $b } @file;
} else {
   @file = sort @file;
}

print OUTPUT @file;
```

This time, we'll give no parameters but instead pass data on the command-line using the left arrow:

```
> perl sort.plx < sortme.txt
And nail my feet up where my head should be
And you can all die laughing, because I'd wear it proudly
If they had a king of fools then I could wear that crown
Well, I finally found someone to turn me upside-down
>
```

As you can see, the data ends up on standard output. But how?

How It Works

The key magic occurs in the following lines:

```
if (defined $input) {
   open INPUT, $input or die "Couldn't open file $input: $!\n";
} else {
   *INPUT = *STDIN;
}
```

If there's an input file name defined, we use that. Otherwise, we do this strange thing with the stars. What we're doing is telling Perl that `INPUT` should be the same filehandle as standard input. If we wanted to set array `@a` to be the same as array `@b`, we'd say `@a = @b;` With filehandles, we can't just say `INPUT = STDIN;` we have to put a star before their names. From now on, everything that is read from `INPUT` will actually be taken from `STDIN`. Similarly, everything that is written to `OUTPUT` goes to `STDOUT`:

What the star – or, to give it its proper name, the glob – does is actually very subtle: `*a = *b` makes everything called `a` – that is `$a`, `@a`, `%a`, and the filehandle called `a` – into an alias for everything called `b`. This is more than just setting them to the same value – it makes them the same thing. Now everything that alters `$a` also alters `$b` and vice versa. That's why it's a good convention to keep filehandles purely upper-case. That keeps them distinct from other variables, meaning you won't inadvertently alias two variables.

The reason we have to do this to manipulate filehandles is because there isn't a 'type symbol' for them as there is for scalars, arrays, and hashes. This is now seen as a mistake, but there's little we can do about it at this stage.

Writing Binary Data

So far, we've been dealing primarily with text files: speeches, Perl programs, and so on. When we get to data that's designed for computers to read – binary files – things change somewhat. The first problem is the newline character, `\n. This` is actually nothing more than a convenient fiction, allowing you to denote a new line using ASCII symbols on whatever operating system you're working with. In truth, different operating systems have differing ideas about what a newline really is when written to a file.

On UNIX, it really is `\n` – character number 10 in the ASCII sequence. When a Macintosh reads a file, the lines are separated by character 13 in the ASCII sequence, which you can generate by saying `\r`. A Macintosh version of Perl, then, will convert `\r` on the disk to `\n` in the program when reading in from a file, then write `\r` to the disk in place of `\n` when writing out to a file.

Windows, on the other hand, is different again. The DOS family of operating systems use `\r\n` on the disk to represent a new line. Therefore Perl has to silently drop or insert `\r` in the relevant places to make it look as if you're dealing with `\n` all the time.

When you're dealing with text, this is exactly what you want to happen. Perl's idea of a newline needs to correspond with the native operating system's idea of a newline – whatever that may be. However, with binary files, where every byte is important, you don't want Perl fiddling with the data just because it looks like the end of a line of text. You want those `\r`s to stay where they are!

Worse still, on DOS, Windows, and friends, character 26 is seen as the end of a file. Perl will stop reading once it sees this character, regardless of whether there's any data to follow.

To get a round both these problems, you need to tell Perl when you're reading from and writing to binary files, so that it can compensate. You can do this by using the `binmode` operator:

```
binmode FILEHANDLE;
```

To ensure your files are read and written correctly, **always** use `binmode` on binary files, **never** on text files.

Selecting a Filehandle

Normally, when you `print`, the data goes to the `STDOUT` filehandle. To send it somewhere else, you say `print FILEHANDLE ...;` However, if you're sending a lot of data to a file, you might not want to have to give the filehandle every time, it would be useful if it were possible to change the default filehandle. You can do this very simply by `select`ing the filehandle:

```
select FILEHANDLE;
```

This will change the default location for `print` to `FILEHANDLE`. Remember to set `STDOUT` back when you're done. A good use of this is to optionally send your program's output to a log file instead of the screen:

Try It Out : Selecting A Log File

This program does very little of interest; however, it does it using a log file. We'll use `select` to control where its output should go.

```
#!/usr/bin/perl
#logfile.plx
use warnings;
use strict;

my $logging = "screen";    # Change this to "file" to send the log to a file!

if ($logging eq "file") {
   open LOG, "> output.log" or die $!;
   select LOG;
}

print "Program started: ", scalar localtime, "\n";
sleep 30;
print "Program finished: ", scalar localtime, "\n";

select STDOUT;
```

As it is, the program will print something like this:

```
> perl logfile.plx
Program started: Sun Apr 22 14:17:07 2000
Program finished: Sun Apr 22 14:17:37 2000
>
```

However, if we change line 6 to this:

```
my $logging = "file";
```

we apparently get no output at all:

```
> perl logfile.plx
>
```

However, we'll find the same style output in the file `output.log`. How?

How It Works

Since the value of `$logging` has changed, it's reasonable to assume that the difference is due to something acting on `$logging` – Perl is nice and deterministic like that. So, sure enough, on line 8, `$logging` gets examined:

```
if ($logging eq "file") {
```

If `$logging` has the value `file`, which it does now:

```
open LOG, "> output.log" or die $!;
```

We open a filehandle for writing, on the file `output.log`:

```
select LOG;
```

Then we `select` that filehandle. Now any `print` statements that don't specify which filehandle to print to go out on `LOG`. If we wanted to write on standard output from now on, we'd have to write:

```
print STDOUT "This goes to the screen.\n";
```

Or, alternatively, we could `select` standard output again:

```
select STDOUT;
```

How did we get Perl to print out the time? The key is in this line:

```
print "Program started: ", scalar localtime, "\n";
```

`localtime` is a function that returns the current time in the local time zone. Ordinarily, it returns a list like this:

```
($sec, $min, $hour,
$day_of_month,
$month_minus_one,
$year_minus_nineteen_hundred,
$day_of_week,
$day_of_year,
$is_this_daylight_savings_time)
```

Right now it would return:

53, 47, 14, (It's 14:47:53.)
22, (It's the 22nd)
3, (April is the third month of the year, counting from the zeroth)
100, (It's the year 2000.)
6, (It's a Saturday. Sunday is the first day of the week, day zero.)
112, (It's the 112th day of the year, counting from the zeroth. January the first is day zero.)
0 (It's not daylight savings time right now.)

Always be careful when dealing with `localtime`. Hopefully by now you see the merit in counting from zero when you're dealing with computers, but it can sometimes catch you out – the month of the year, day of the week and day of the year start from zero, but the day of the month starts from one.

Thankfully, it's now a lot harder to imagine that the fifth element returned is the year. Last year `(localtime)[5]` returned `99`, which some foolhardy programmers assumed was the last two digits of the year. Fortunately, Perl turned out to be perfectly Y2K compliant, unfortunately, those programmers weren't. `(localtime)[5]` is (and has always been) the year minus 1900. If you find this weird and inconsistent, you can blame it all on the fact that Perl bases its idea on how to represent time from C, which first perpetrated this insanity.

In scalar context however, `localtime` provides a much easier value to deal with: It's a string representing the current time in a form designed for human consumption. This allows us to easily produce timestamps to mark when operations happened. However, we must remember that since `print` takes a list, we need to explicitly tell `localtime` to be in scalar context in order to force it to return this string.

Buffering

Try this little program:

```
#!/usr/bin/perl
#time.plx
use warnings;
use strict;

for (1...20) {
   print ".";
   sleep 1;
}
print "\n";
```

You'd probably expect it to print twenty dots, leaving a second's gap between each one – on Windows with ActiveState Perl, that's exactly what it does. However, this is something of an exception. On most other operating systems, you'll have to wait for twenty seconds first, before it prints all twenty at once.

So what's going on? Operating systems often won't actually write something to (or read something from) a filehandle until the end of the line. This is to save doing a lot of short, repetitious read/write operations. Instead, they keep everything you've written queued up in a buffer and access the filehandle once only.

However, you can tell Perl to stop the OS doing this, by modifying the special variable `$|`. If this is set to zero, which it usually is, Perl will tell the operating system to use output buffering if possible. If it's set to one, Perl turns off buffering for the currently selected filehandle.

So, to make our program steadily print out dots – as you might do to show progress on a long operation – we just need to set `$|` to 1 before we do our printing:

```
#!/usr/bin/perl
#time2.plx
use warnings;
use strict;

$| = 1;
for (1...20) {
   print ".";
   sleep 1;
}
print "\n";
```

If you need to turn off buffering when writing to a file, be sure to `select` the appropriate filehandle before changing `$|`, possibly selecting `STDOUT` again when you've done so.

Permissions

Before going on, let's look briefly at the issue of file permissions. If you use UNIX or other multi-user systems, you'll almost certainly be familiar with the very specific access controls that can be imposed, determining who's allowed to do what with any given file or directory. It's most likely that a file or directory on such a system will have at least three sets of permissions for each of three sets of users:

- The file owner,
- The group with which the owner is associated, and
- Everyone else on the system.

Each of these can have 'read', 'write' and 'execute' permissions assigned. You may have seen these displayed along with other file information as a sequence like:

```
drwxrwxrwx
```

which denotes full access on a directory (denoted by the prefix '`d`') for all users or:

```
-rwx--x---
```

which denotes a file (prefix '`-`') to which the owner has full access, members of their group can execute (but not read or modify), and everyone else has no access at all.

> *In fact, the subtleties of permission hierarchies mean that it's not always quite this clear cut. For example, a UNIX file without public 'write' permissions can actually be deleted by any user at all if the file's parent directory has granted them the relevant permission. Take care.*

Perl gives us the function `umask(`*`expr`*`)`, which we can use to set the permission bits to be used when we create a file. The expression it will expect is a three digit octal number, representing the state of the nine flags we've seen. If we consider these as bits in a binary number, we can interpret our second example above as:

```
111001000
```

which breaks down groupwise as:

```
(111) (001) (000)
```

and in octal as:

```
710
```

We can therefore specify `umask(0710);` and subsequent files will be created with any permissions it has been specifically given `AND`ed with the `umask` value. In a nutshell, by setting the umask value, we have set the default permissions for all files or directories on top of which other permissions can be set.

In general, it's a good idea to set the `umask` to `0666` for creating regular files. If you work backwards from the file, you realize that this equates to giving everyone read and write access to the file but no-one execute permissions. Likewise, it's a fairly safe bet to set `umask` to `0777` – full control for everyone – for the creation of directories and, of course, executable files.

Opening Pipes

`open` can be used for more than just plain old files. You can read data from and send data to programs as well. Anything that can read from or write to standard output can talk directly to Perl via a **pipe**.

Pipes were invented by a man called Doug MacIlroy for the UNIX operating system and were soon carried over to other operating systems. They're one of those things that sound amazingly obvious once someone else has thought of it:

A pipe is something that connects two filehandles together.

That's it. Usually, you'll be connecting the standard output of one program to the standard input of another.

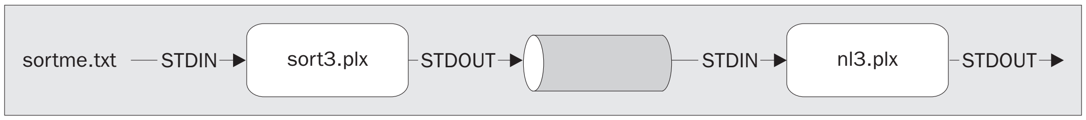
For instance, we've written two filters in this chapter: one to number lines in a file and one to sort files. Let's see what happens when we connect them together:

```
> perl sort3.plx < sortme.txt | perl nl3.plx

            File: -

1: And nail my feet up where my head should be
2: And you can all die laughing, because I'd wear it proudly
3: If they had a king of fools then I could wear that crown
4: Well, I finally found someone to turn me upside-down
>
```

That bar in the middle is the pipe. Here's a diagram of what's going on:



The pipe turns the standard output of `sort3.plx` into input for `nl3.plx`.

While pipes are usually used for gluing programs together on the shell command line, exactly as we've just done above, we can use them in Perl to read from and write to programs.

Piping In

To read the output of a program, simply use `open` and the name of the program (with any command line you want to give it), and put a pipe at the end. For instance, the program `lynx` is a command-line web browser, available via http://lynx.browser.org/. If I say `lynx -source` http://www.perl.com/, `lynx` gets the HTML source to the page and sends it to standard output. I can pick this up from Perl using a pipe.

If you're using Windows, you may need to modify your global path settings – the list of directory paths in which Windows will look for perl, or lynx, or any other executable that you want to call without specifying it's location. It's only because `PATH` contains `C:\Perl\bin\` that we can say:

```
>perl <filename>
```

without saying anything about where `perl.exe` lives. On Windows 9x you can edit the default value of PATH inside the file `autoexec.bat`, which you'll find in the root directory. On Windows 2000, you'll find this under Start Menu|Program Files|Administrative Tools|Computer Management – call up Properties for the local machine, and it's on the Advanced tab, in Environment Variables.

Simply add the full path of the directory into which you've installed `lynx.exe`, separated from previous entries (you should see `C:\Perl\bin\` there already) by a semicolon. Mine now looks like this:

```
C:\PERL\BIN\;C:\PERL\BIN;C:\WINDOWS;C:\WINDOWS\COMMAND;C:\LYNX\DIST\
```

One simpler alternative is to enter this at the DOS command line:

```
set PATH=%PATH%;<add directory path to lynx.exe here>
```

This has the benefit of being quicker. It's also safer, as any modification you make like this is local to the current DOS shell, but that means you'll have to do it again next shell around…

You may still find that lynx still won't run from outside it's own directory and gives you a message like:

```
Configuration file ./lynx.cfg is not available.
```

You can get round this problem by copying the relevant file from the lynx directory into your current one. It's a bit of a fudge, but it does the trick:

Try it out : Perl headline

```
#!/usr/bin/perl
# headline.plx
# Display the www.perl.com top story.
use warnings;
use strict;

open LYNX, "lynx -source http://www.perl.com/ |" or die "Can't open lynx: $!";

# Define $_ and skip through LYNX until a line containing "standard.def"
$_ = "";
$_ = <LYNX> until /standard\.def/;

# The headline is in the following line:
my $head = <LYNX>;

# Extract "Headline" from "<A HREF=something>Headline</a>..."
$head =~ m|^<A HREF=[^>]+>(.*?)</a>|i;

print "Today's www.perl.com headline: $1\n";
```

Run today, this tells me:

```
>perl headline.plx
Today's www.perl.com headline: What's New in 5.6.0.
>
```

Note that this program will work with the layout of www.perl.com *at the time of writing. If the site's layout changes, it might not work in the future.*

How It Works

The important thing, for our purposes, is the pipe:

```
open LYNX, "lynx -source http://www.perl.com/ |" or die "Can't open lynx: $!";
```

What it's saying is that, instead of a file on the disk, the filehandle `LYNX` should read from the standard output of the command `lynx -source http://www.perl.com`. The pipe symbol | at the end of the string tells Perl to run the command and collect the output. The effect is just the same as if we'd had `lynx` write the output to a file and then had Perl read in that file. Each line we read from `LYNX` is the next line of source in the output.

Let's now have a look at how we extracted the headline from the source.

The site is laid out in a standard format, and the headline is on the line following the text "standard.def". So we can happily keep getting new lines until we come across one matching that text:

```
$_ = <LYNX> until /standard\.def/;
```

Note that we have to assign the new line to `$_` ourselves. The assignment to `$_` is only done automatically when you say something like `while(<FILEHANDLE>)`.

The headline is in the next line, so we get that:

```
my $head = <LYNX>;
```

The line containing the headline will look something like this:

```
<A HREF="http://www.perl.com/pub/2000/05/...">Perl used in wombat sexing</A>
```

To retrieve the headline from the middle, we use a regular expression. Generally speaking, reading HTML with a regular expression is a really bad idea, as `perlfaq9` explains. HTML tags are far more complex than just "start at an open bracket and end with a close bracket". That definition would fail spectacularly with tags in comments, tags split over multiple lines, or tags containing a close bracket symbol as part of a quoted string. It's a much harder problem than it first appears, due to the scope of the HTML language.

To read HTML to any degree of accuracy, you need to use an extension module like `HTML::Parser`. However, when the scope of the problem is as limited as the one we're faced with, we can get away with taking a few liberties.

We know that the piece of HTML in question is a single line. We know that the tag we're looking for starts at the beginning of the line and that there are no close brackets within it. So, our regular expression finds `"<A HREF="` at the beginning of the line. After that, we read anything that's not a closing bracket, followed by a closing bracket.

Next, we want our headline: This is the smallest amount of text that will be directly followed by `</A>`. Since there's a forward slash in what we're trying to match, we use alternate delimiters to make the expression more understandable. As we're using alternate delimiters, we need to put an `m` on the front to make it clear that this is a match:

```
$head =~ m|^<A HREF=[^>]+>(.*?)</A>|;
```

We could have said: `$head =~ /^<A HREF=[^>]+>(.*?)<\/A>/;` backslashing the forward slash to avoid it being treated as the end of the regular expression, but that would have been unnecessarily confusing. This is exactly the sort of situation that alternate delimiters were provided for, so we're right to make the most of them.

Why do we use `[^>]+` instead of `.*` or similar? Consider what would happen if there were two stories on the line:

```
<A HREF="http://www.perl.com/...">Perl is really cool</A><A HREF="...">Story 2</A>
```

`<A HREF=.*>` matches **as much as possible** before a close bracket. In this case, the most it can get before a close bracket would be to match everything up until just before Story 2, and we'd miss the main headline altogether. This is because `.` means everything, and everything includes a closing bracket. By saying [^>]+ we're making it clear that there can be no closing brackets in the text we're matching.

Piping Out

As well as reading data in from external programs, we can write out to the standard input of another program. For instance, we could send mail out by writing to a program like `sendmail`, or we could be generating output that we'd like to have sorted before it gets to the user. We'll deal with the second example because, while it's easy enough to collect the data into an array and sort it ourselves before writing it out, we know we have a sorting program handy. After all, we wrote one a few pages ago!

Try It Out : Taking Inventory

Things hide in the kitchen cabinet. Tins of tomatoes can lurk unseen for weeks and months, springing to vision only after I've bought another can. Every so often, then, I need to investigate the cabinets and take inventory to enumerate my baked beans and root out reticent ravioli. The following program can help me do that:

```
#!/usr/bin/perl
# inventory.plx
use warnings;
use strict;

my %inventory;
print "Enter individual items, followed by a new line.\n";
print "Enter a blank line to finish.\n";
while (1) {
   my $item = <STDIN>;
   chomp $item;
   last unless $item;
   $inventory{lc $item}++;
}

open(SORT, "| perl sort.plx") or *SORT = *STDOUT;
select *SORT;
while (my ($item, $quantity) = each %inventory) {
   if ($quantity > 1) {
      $item =~ s/^(\w+)\b/$1s/ unless $item =~ /^\w+s\b/;
   }
   print "$item: $quantity\n";
}
```

Now let's take stock:

```
>perl inventory.plx
Enter individual items, followed by a new line.
Enter a blank line to finish.
jar of jam
loaf of bread
tin of tuna
packet of pancake mix
tin of tomatos
tin of tuna
packet of pasta
clove of garlic
packet of pasta

clove of garlic: 1
jar of jam: 1
loaf of bread: 1
packet of pancake mix: 1
packets of pasta: 2
tin of tomatos: 1
tins of tuna: 2
```

As you can see, we get back a sorted list of totals.

How It Works

Whenever you're counting how many of each items you have in a list, you should immediately think about hashes. Here we use a hash to key each item to the quantity; each time we see another one of those items, we add to the quantity in the hash:

```
while (1) {
   my $item = <STDIN>;
   chomp $item;
   last unless $item;
   $inventory{lc $item}++;
}
```

The only way this infinite loop will end is if `$item` contains nothing after being chomped – it was nothing more than a new line.

To ensure that the capitalization of our item isn't significant, we use the operator `lc` to return a lower-case version of the item. Otherwise, "Tin of beans", "TIN OF BEANS" and "tin of beans" would be treated as three totally separate items, instead of three examples of the same thing. By forcing them into lower case, we remove the difference.

The `lc` operator returns the string it was given, but with upper-case characters turned into lower case. So `print lc("FuNnY StRiNg");` should give you the output 'funny string'. There's also a `uc` operator that returns an upper-cased version of the string, so `print uc("FuNnY StRiNg");` will output 'FUNNY STRING'.

Next, we open our pipe. We're going to pass data from our program to another, external program. If you look up at the pipe diagrams above, you'll see that the data flows from left to right. Therefore, we want to put the command to run that external program on the right-hand side of the pipe:

```
open(SORT, "| perl sort.plx") or *SORT = *STDOUT;
```

If we can't successfully open the pipe – the program wasn't found or we couldn't execute Perl – we alias `SORT` to `STDOUT` to get an unsorted version.

Now we can print the data out:

```
while (my ($item, $quantity) = each %inventory) {
```

We use `each` to get each key/value pair from the hash, as explained in chapter 3.

```
   if ($quantity > 1) {
      $item =~ s/(\w+)/$1s/ unless $item =~ /\w+s\b/;
   }
```

This will make the output a little more presentable. If there is more than one of the current item, the name should be pluralized, unless it already ends in an 's'. `\w+` will get the first word in the string, and we add an 's' after it.

This is a relatively crude method for pluralizing English words, If you want to do it properly, there's a module on CPAN called `Lingua::EN::Inflect` that will do the trick.

```
   print "$item: $quantity\n";
```

Last of all, we print this out. It's actually going to the SORT filehandle, because that's the one that's currently selected – that filehandle is, in turn, connected to the sort program.

File Tests

So far, we've just been reading and writing files, and `die`ing if anything bad happens. For small programs, this is usually adequate, but if we want to use files in the context of a larger application, we should really check their status before we try and open them and, if necessary, take preventative measures. For instance, we may want to warn the user if a file we wish to overwrite already exists, giving them a chance to specify a different file. We'll also want to ensure that, for instance, we're not trying to read a directory as if it was a file.

This sort of programming – anticipating the consequences of future actions – is called defensive programming. Just like defensive driving, you assume that everything is out to get you. Files will not exist or not be writeable when you need them, users will specify things inaccurately, and so on. Properly anticipating, diagnosing, and working around these areas is the mark of a top class programmer.

Perl provides us with **file tests**, which allow us to check various characteristics of files. These act as logical operators, and return a true or false value. For instance, to check if a file exists, we write this:

```
if (-e "somefile.dat") {...}
```

The test is `-e`, and it takes a file name as its argument. Just like `open`, this file name can also be specified from a variable. You can just as validly say:

```
if (-e $filename) {...}
```

where `$filename` contains the name of the file you want to check.

For a complete list of file tests, see Appendix C. The table below shows the most common ones:

Test	**Meaning**
`-e`	True if the file exists.
`-f`	True if the file is a plain file – not a directory.
`-d`	True if the file is a directory.
`-z`	True if the file has zero size.
`-s`	True if the file has nonzero size – returns size of file in bytes.
`-r`	True if the file is readable by you.
`-w`	True if the file is writable by you.
`-x`	True if the file is executable by you.
`-o`	True if the file is owned by you.

The last four tests will only make complete sense on operating systems for which files have meaningful permissions, such as UNIX and Windows NT. If this isn't the case, they'll frequently *all* return true (assuming the file or directory exists). So, for instance, if we're going to write to a file, we should check to see whether the file already exists, and if so, what we should do about it.

Note that on systems that don't use permissions comprehensively, `-w` is the most likely of the last four tests to have any significance, testing for Read-only *status. On Windows 9x, this can be found (and modified) on the* General *tab of the file's* Properties *window:*

Try It Out : Paranoid File Writing

This program does all it can to find a safe place to write a file:

```
#!/usr/bin/perl
# filetest1.plx
use warnings;
use strict;
```

```
my $target;
while (1) {
   print "What file should I write on? ";
   $target = <STDIN>;
   chomp $target;
   if (-d $target) {
      print "No, $target is a directory.\n";
      next;
   }
   if (-e $target) {
      print "File already exists. What should I do?\n";
      print "(Enter 'r' to write to a different name, ";
      print "'o' to overwrite or\n";
      print "'b' to back up to $target.old)\n";
      my $choice = <STDIN>;
      chomp $choice;
      if ($choice eq "r") {
         next;
      } elsif ($choice eq "o") {
         unless (-o $target) {
            print "Can't overwrite $target, it's not yours.\n";
            next;
         }
         unless (-w $target) {
            print "Can't overwrite $target: $!\n";
            next;
         }
      } elsif ($choice eq "b") {
         if ( rename($target,$target.".old") ) {
            print "OK, moved $target to $target.old\n";
         } else {
            print "Couldn't rename file: $!\n";
            next;
         }
      } else {
         print "I didn't understand that answer.\n";
         next;
      }
   }
   last if open OUTPUT, "> $target";
   print "I couldn't write on $target: $!\n";
   # and round we go again.
}
print OUTPUT "Congratulations.\n";
print "Wrote to file $target\n";
```

So, after all that, let's see how it copes, first of all with a text file that doesn't exist:

```
> perl filetest1.plx
What file should I write on? test.txt
Wrote to file test.txt
>
```

Seems OK. What about if I 'accidentally' give it the name of a directory? Or give it a file that already exists? Or give it a response it's not prepared for?

```
> perl filetest1.plx
What file should I write on? work
No, work is a directory.
What file should I write on? filetest1.plx
File already exists. What should I do?
(Enter 'r' to write to a different name, 'o' to overwrite or
'b' to back up to filetest1.plx.old)
r
What file should I write on? test.txt
File already exists. What should I do?
(Enter 'r' to write to a different name, 'o' to overwrite or
'b' to back up to test.txt.old)
g
I didn't understand that answer.
What file should I write on? test.txt
File already exists. What should I do?
(Enter 'r' to write to a different name, 'o' to overwrite or
'b' to back up to test.txt.old)
b
OK, moved test.txt to test.txt.old
Wrote to file test.txt
>
```

How It Works

The main program takes place inside an infinite loop. The only way we can exit the loop is via the `last` statement at the bottom:

```
    last if open OUTPUT, "> $target";
```

That `last` will only happen if we're happy with the file name and the computer can successfully open the file. In order to be happy with the file name, though, we have a gauntlet of tests to run:

```
        if (-d $target) {
```

We need to first see whether or not what has been specified is actually a directory. If it is, we don't want to go any further, so we go back and get another file name from the user:

```
            print "No, $target is a directory.\n";
            next;
```

We print a message and then use `next` to take us back to the top of the loop.

Next, we check to see whether or not the file already exists. If so, we ask the user what we should do about this.

```
      if (-e $target) {
         print "File already exists. What should I do?\n";
         print "(Enter 'r' to write to a different name, ";
         print "'o' to overwrite or\n";
         print "'b' to back up to $target.old\n";
         my $choice = <STDIN>;
         chomp $choice;
```

If they want a different file, we merely go back to the top of the loop:

```
         if ($choice eq "r") {
            next;
```

If they want us to overwrite the file, we see if this is likely to be possible:

```
         } elsif ($choice eq "o") {
```

First, we see if they actually own the file; it's unlikely they'll be allowed to overwrite a file that they do not own.

```
            unless (-o $target) {
               print "Can't overwrite $target, it's not yours.\n";
               next;
            }
```

Next we check to see if there are any other reasons why we can't write on the file, and if there are, we report them and go around for another file name:

```
            unless (-w $target) {
               print "Can't overwrite $target: $!\n";
               next;
            }
```

If they want to back up the file, that is, rename the existing file to a new name, we see if this is possible:

```
         } elsif ($choice eq "b") {
```

The `rename` operator renames a file; it takes two arguments: the current file name and the new name.

```
            if ( rename($target,$target.".old") ) {
               print "OK, moved $target to $target.old\n";
            } else {
```

If we couldn't rename the file, we explain why and start from the beginning again:

```
               print "Couldn't rename file: $!\n";
               next;
            }
```

Otherwise, they said something we weren't prepared for:

```
          } else {
             print "I didn't understand that answer.\n";
             next;
          }
```

You may think this program is excessively paranoid, after all, it's 50 lines just to print a message on a file. In fact, it isn't paranoid enough: it doesn't check to see whether the backup file already exists before renaming the currently existing file. This just goes to show you can never be too careful when dealing with the operating system. Later, we'll see how to turn big blocks of code like this into reusable elements so we don't have to copy that lot out every time we want to safely write to a file.

Directories

As well as files, we can use Perl to examine directories on the disk. There are two major ways to look at the contents of a directory:

Globbing

If you're used to using the command shell, you may well be used to the concept of a **glob**. It's a little like a regular expression, in that it's a way of matching file names. However, the rules for globs are much simpler. In a glob, `*` matches any amount of text.

So, if I were in a directory containing files: `00INDEX 3com.c 3com.txt perl mail Mail`

- `*` would match everything.
- `3*` would match `3com.c` and `3com.txt`.
- `?ail` would match `mail` and `Mail`.
- `*l` would match `perl`, `mail` and `Mail`.

We can do this kind of globbing in Perl: the glob operator takes a string and returns the matching files:

```
#!/usr/bin/perl
# glob.plx
use warnings;
use strict;

my @files = glob("*l");
print "Matched *l : @files\n";
```

```
>perl glob.plx
perl mail Mail
>
```

To get all the files in a directory, you would say `my @files = glob("*");`

Reading Directories

That's the simple way. For more flexibility, you can read files in a directory just like lines in a file. Instead of using `open`, you use `opendir`. Instead of getting a filehandle, you get a **directory handle**:

```
opendir DH, "." or die "Couldn't open the current directory: $!";
```

Now to read each file in the directory, we use `readdir` on the directory handle:

Try It Out : Examining A Directory

This program lists the contents of the current directory and uses filetests to examine each file.

```
#!/usr/bin/perl
# directory.plx
use warnings;
use strict;

print "Contents of the current directory:\n";
opendir DH, "." or die "Couldn't open the current directory: $!";
while ($_ = readdir(DH)) {
        next if $_ eq "." or $_ eq "..";
        print $_, " " x (30-length($_));
        print "d" if -d $_;
        print "r" if -r _;
        print "w" if -w _;
        print "x" if -x _;
        print "o" if -o _;
        print "\t";
        print -s _ if -r _ and -f _;
        print "\n";
}
```

Part of its output looks like this:

```
>perl directory.plx
Contents of the current directory:
…
directory.plx          rwxo    449
filetest1.plx          rwxo    1199
inventory.plx          rwxo    515
mail                   drwxo
nl.plx                 rwxo    240
todo.log               rwo     3583
...
>
```

The number at the end is the size of the file in bytes; as for the letters, 'd' shows that this is a directory, 'r' stands for readable, 'w' for writable, 'x' for executable, and 'o' shows that I am the owner.

How It Works

As we've seen on the previous page, once we've opened our directory handle, we can read from it. We read one file name at a time into `$_`, and while there's still some information there, we examine it more closely:

```
while ($_ = readdir(DH)) {
```

The files `.` and `..` are special directories on DOS and UNIX, referring to the current and parent directories, respectively. We skip these in our program:

```
        next if $_ eq "." or $_ eq "..";
```

We then print out the name of each file, followed by some spaces. The length of the file name plus the number of spaces will always add up to thirty, so we have nicely arranged columns:

```
        print $_, " " x (30-length($_));
```

First we test to see if the file is a directory, using the ordinary filetests we saw above:

```
        print "d" if -d $_;
```

No, this isn't a typo: I do mean _ and not `$_` here. Just as `$_` is the default value for some operations, such as `print`, _ is the default filehandle for filetests. It actually refers to the last file explicitly tested. Since we tested `$_` above, we can use _ for as long as we're referring to the same file:

```
        print "r" if -r _;
        print "w" if -w _;
```

When Perl does a filetest, it actually looks up all the data at once – ownership, readability, writeability and so on; this is called a stat *of the file. _ tells Perl not to do another stat, but to use the data from the previous one. As such, it's more efficient that stat-ing the file each time.*

Finally, we print out the file's size. This is only possible if we can read the file and only useful if the file is not a directory:

```
        print -s _ if -r _ and -f _;
```

Summary

Files give our data permanence by allowing us to store it on the disk. It's no good having the best accounting program in the world, if it loses all your accounts every time the computer is switched off. What we've seen here are the fundamentals of getting data in and out of Perl. In our chapter on Databases, we'll see more practical examples of how to read structured files into Perl data structures and write them out again.

Files are accessed through filehandles. To begin with, we have standard input, standard output, and standard error. We can open other filehandles, either for reading or for writing, with the `open` operator, and we must always remember to check what happened to the `open` call.

The diamond operator `<FILEHANDLE>` reads a line in from the specified filehandle. We can control the definition of a line by altering the value of the record separator, held in special variable `$/`.

Writing to a file is done with the `print` operator. Normally, this writes to standard output, so the filehandle must be specified. Alternatively, you may `select` another filehandle as the recipient of `print`'s output.

Pipes can be used to talk to programs outside of Perl. We can read in and write out data to them as if we were looking at the screen or typing on the keyboard. We can also use them as filters to modify our data on the way in or out of a program.

Filetests can be used to check the status of a file in various ways, and we've seen an example of using filetests to ensure that there are no surprises when we're reading or writing a file.

Finally, we've seen how to read files from directories using the `opendir` and `readdir` operators.

Exercises

1. Write a program that can search for a specified string within all the files in a given directory.
2. Modify the file backup facility in filetest1.plx so that it checks to see if a backup already exists before renaming the currently existing file. When a backup does exist, the user should be asked to confirm that they want to overwrite it. If not, they should be returned to the original query.