Chapter 9. Processing Text with Regular Expressions

You can use regular expressions to change text too. So far we've only shown you how to match a pattern, and now we'll show you how to use patterns to locate the parts of strings that you want to change.

Substitutions with s///

If you think of the `m//` pattern match as being like your word processor's "search" feature, the "search and replace" feature would be Perl's `s///` substitution operator. This simply replaces whatever part of a variable matches the pattern with a replacement string:

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/;  # Replace Barney with Fred
print "$_\n";
```

NOTE

Unlike `m//`, which can match against any string expression, `s///` is modifying data that must therefore be contained in what's known as an `lvalue`. This is nearly always a variable, but it could be anything you can use on the left side of an assignment operator.

If the match fails, nothing happens, and the variable is untouched:

```
# Continuing from previous; $_ has "He's out bowling with Fred tonight."
s/Wilma/Betty/;  # Replace Wilma with Betty (fails)
```

Of course, both the pattern and the replacement string could be more complex. Here, the replacement string uses the first capture variable, `$1`, which is set by the pattern match:

```
s/with (\w+)/against $1's team/;
print "$_\n";  # says "He's out bowling against Fred's team tonight."
```

Here are some other possible substitutions. These are here only as samples; in the real world, it would not be typical to do so many unrelated substitutions in a row:

```
$_ = "green scaly dinosaur";
s/(\w+) (\w+)/$2, $1/;  # Now it's "scaly, green dinosaur"
s/\A/huge, /;           # Now it's "huge, scaly, green dinosaur"
s/,.*een//;             # Empty replacement: Now it's "huge dinosaur"
s/green/red/;           # Failed match: still "huge dinosaur"
s/\w+$/($`!)$&/;        # Now it's "huge (huge !)dinosaur"
s/\s+(!\W+)/$1 /;       # Now it's "huge (huge!) dinosaur"
s/huge/gigantic/;       # Now it's "gigantic (huge!) dinosaur"
```

There's a useful Boolean value from `s///`; it's true if a substitution was successful, otherwise it's false:

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
  print "Successfully replaced fred with wilma!\n";
}
```

Global Replacements with /g

As you may have noticed in a previous example, `s///` will make just one replacement, even if others are possible. Of course, that's just the default. The `/g` modifier tells `s///` to make all possible nonoverlapping replacements; that is, each new match starts looking just beyond the latest replacement:

```
$_ = "home, sweet home!";
s/home/cave/g;
print "$_\n";  # "cave, sweet cave!"
```

A fairly common use of a global replacement is to collapse whitespace; that is, to turn any arbitrary whitespace into a single space:

```
$_ = "Input  data\t may have    extra whitespace.";
s/\s+/ /g;  # Now it says "Input data may have extra whitespace."
```

Once we show collapsing whitespace, everyone wants to know about stripping leading and trailing whitespace. That's easy enough, in two steps:

```
s/\A\s+//;  # Replace leading whitespace with nothing
s/\s+\z//;  # Replace trailing whitespace with nothing
```

We could do that in one step with an alternation and the `/g` modifier, but that turns out to be a bit slower, at least when we wrote this. The regular expression engine is always being tuned, but to learn more about that, you can get *Mastering Regular Expressions* by Jeffrey Friedl and find out what makes regular expressions fast (or slow):

```
s/\A\s+|\s+\z//g;  # Strip leading, trailing whitespace
```

Different Delimiters

Just as you did with `m//` and `qw//`, you can change the delimiters for `s///`. But the substitution uses three delimiter characters, so things are a little different.

With ordinary (nonpaired) characters that don't have a left and right variety, just use three of them, as you did with the forward slash. Here, you use the octothorp (or pound sign) as the delimiter:

```
s#\Ahttps://#http://#;
```

But if you use paired characters, which have a left and right variety, you have to use two pairs: one to hold the pattern and one to hold the replacement string. In this case, the delimiters don't have to be the same kind around the string as they are around the pattern. In fact, the delimiters of the string could even be nonpaired. These are all the same:

```
s{fred}{barney};
s[fred](barney);
s<fred>#barney#;
```

Substitution Modifiers

In addition to the `/g` modifier, you can use the `/i`, `/x`, `/m`, and `/s` modifiers that you saw in ordinary pattern matching (the order of modifiers isn't significant):

```
s#wilma#Wilma#gi;  # replace every WiLmA or WILMA with Wilma
s{__END__.*}{}s;   # chop off the end marker and all following lines
```

The Binding Operator

Just as you saw with `m//`, we can choose a different target for `s///` by using the binding operator:

```
$file_name =~ s#\A.*/##s;  # In $file_name, remove any Unix-style path
```

Nondestructive Substitutions

What if you want to have the original and the modified versions of a string at the same time? You could make a copy and work with that:

```
my $original = 'Fred ate 1 rib';
my $copy = $original;
$copy =~ s/\d+ ribs?/10 ribs/;
```

You could also write that as a single statement where you do the assignment and perform the substitution on the result:

```
(my $copy = $original) =~ s/\d+ ribs?/10 ribs/;
```

That can be a bit confusing because many people forget that the result of the assignment is just as good as a string, so it's really `$copy` that gets changed. Perl 5.14 adds a `/r` modifier that changes how this works. Normally the result of a `s///` is the number of substitutions it made, but with the `/r`, it leaves the original string alone and returns a modified copy of it:

```
use v5.14;

my $copy = $original =~ s/\d+ ribs?/10 ribs/r;
```

That looks almost the same as the previous example, just without the parentheses. In this case, though, things happen in reverse order. You do the substitution first and the assignment second. Chapter 2 (and the perlop documentation) has the precedence chart; the `=~` has higher precedence than the `=`.

Case Shifting

It often happens in a substitution that you'll want to make sure that a replacement word is properly capitalized (or not, as the case may be). That's easy to accomplish with Perl, by using some backslash escapes. The `\U` escape forces what follows to all uppercase:

```
$_ = "I saw Barney with Fred.";
s/(fred|barney)/\U$1/gi;  # $_ is now "I saw BARNEY with FRED."
```

Remember all of our cautions in "Choosing a Character Interpretation"!

Similarly, the `\L` escape forces lowercase. Continuing from the previous code:

```
s/(fred|barney)/\L$1/gi;  # $_ is now "I saw barney with fred."
```

By default, these affect the rest of the (replacement) string, or you can turn off case shifting with `\E`:

```
s/(\w+) with (\w+)/\U$2\E with $1/i;  # $_ is now "I saw FRED with barney."
```

When written in lowercase (`\l` and `\u`), they affect only the next character:

```
s/(fred|barney)/\u$1/ig;  # $_ is now "I saw FRED with Barney."
```

You can even stack them up. Using `\u` with `\L` means "all lowercase, but capitalize the first letter":

```
s/(fred|barney)/\u\L$1/ig;  # $_ is now "I saw Fred with Barney."
```

As it happens, although we're covering case shifting in relation to substitutions, these escape sequences are available in any double-quotish string:

```
print "Hello, \L\u$name\E, would you like to play a game?\n";
```

The `\L` and `\u` may appear together in either order. Larry realized that people would sometimes get those two backward, so he made Perl figure out that you want just the first letter capitalized and the rest lowercase.

Not all lowercasing is the same. Normally, you will probably lowercase things to normalize the string before you compare it:

```
my $input  = 'fRed';
my $string = 'FRED';
if( "\L$input" eq "\L$string" ) {
  print "They are the same name\n";
}
```

However, not everything lowercases as you might expect, or there are equivalent forms. The *ß* is equivalent to *ss*, but:

```
use utf8;
```

```
my $input  = 'Steinerstraße';
my $string = 'STEINERSTRASSE';
if ( "\L$input" eq "\L$string" ) {      # doesn't work!
  print "They are the same name\n";
}
```

Those two don't match in Perl, even though they do match logically. Perl's lowercasing doesn't know the Unicode rules. If you have v5.16 or later and want proper Unicode case folding, you can use `\F` (for "foldcase") escape:

```
use v5.16;

my $input  = 'Steinerstraße';
my $string = 'STEINERSTRASSE';
if ( "\F$input" eq "\F$string" ) {      # works
  print "They are the same name\n";
}
```

NOTE

The new case-folding feature does not help with strings such as `İstanbul` where the lowercase version of `I` is a dotted-i, but the combining dot also shows up, like `i`. You can use the `Unicode::Casing` module for something more sophisticated.

These case-shifting operators are also available as the functions `lc`, `uc`, `fc`, `lcfirst`, and `ucfirst`:

```
my $start   = "Fred";
my $uncapp  = lcfirst( $start );    # fred
my $uppered = uc( $uncapp );        # FRED
my $lowered = lc( $uppered );       # fred
my $capped  = ucfirst( $lowered );  # Fred
my $folded  = fc( $capped );        # fred
```

Metaquoting

There's another escape that's similar to case shifting. The `\Q` quotes any metacharacter in the string. Suppose you start with this pattern where you want to remove literal parentheses before a name:

```
if ( s/(((Fred/Fred/ ) {       # Does not compile!
  print "Removed parens\n";
}
```

You have to quote those characters to make them literal parentheses:

```
if ( s/\(\(\(Fred/Fred/ ) {    # Compiles, but messy!
  print "Removed parens\n";
}
```

That can be annoying. All of those backslashes clutter the pattern. You can use `\Q`, which quotes everything after it. This makes the pattern a bit cleaner:

```
if ( s/\Q(((Fred/Fred/ ) {     # Less messy
  print "Removed parens\n";
}
```

If you want the quoting to apply to only part of the pattern, you can use the `\E`:

```
if ( s/\Q(((\E(Fred)/$1/ ) {   # Even less messy
  print "Cleansed $1\n";
}
```

This is useful if you want to interpolate a variable that you want to be literal characters. Since the interpolation happens, the `\Q` applies to the value in the variable:

```
if ( s/\Q$prefix\E(Fred)/$1/ ) {    # Compiles!
  print "Cleansed $1\n";
}
```

You can do the same thing ahead of time with the `quotemeta` function:

```
my $prefix = quotemeta( $input_pattern );
if ( s/$prefix(Fred)/$1/ ) {    # Compiles!
  print "Cleansed $1\n";
}
```

The split Operator

Another operator that uses regular expressions is `split`, which breaks up a string according to a pattern. This is useful for tab-separated data, or colon-separated, whitespace-separated, or *anything*-separated data, really. So long as you can specify the separator with a regular expression (and generally, it's a simple regular expression), you can use `split`. It looks like this:

```
my @fields = split /separator/, $string;
```

NOTE

Comma-separated values (CSV) files are a pain to do with `split`; you're better off getting the `Text::CSV_XS` module from CPAN.

The `split` drags the pattern through a string and returns a list of fields (substrings) that were separated by the separators. Whenever the pattern matches, that's the end of one field and the start of the next. So, anything that matches the pattern never shows up in the returned fields. Here's a typical `split` pattern, splitting on colons:

```
my @fields = split /:/, "abc:def:g:h";  # gives ("abc", "def", "g", "h")
```

You could even have an empty field, if there were two delimiters together:

```
my @fields = split /:/, "abc:def::g:h";  # gives ("abc", "def", "", "g", "h")
```

Here's a rule that seems odd at first, but it rarely causes problems—leading empty fields are always returned, but trailing empty fields are discarded:

```
my @fields = split /:/, ":::a:b:c:::";  # gives ("", "", "", "a", "b", "c")
```

If you want the trailing empty fields, give `split` a third argument of `-1`:

```
my @fields = split /:/, ":::a:b:c:::", -1;  # gives
("", "", "", "a", "b", "c", "", "", "")
```

It's also common to `split` on whitespace, using `/\s+/` as the pattern. Under that pattern, all whitespace runs are equivalent to a single space:

```
my $some_input = "This  is a \t        test.\n";
my @args = split /\s+/, $some_input;  # ("This", "is", "a", "test.")
```

The default for `split` is to break up `$_` on whitespace:

```
my @fields = split;  # like split /\s+/, $_;
```

This is almost the same as using `/\s+/` as the pattern, except that in this special case a leading empty field is suppressed—so, if the line starts with whitespace, you won't see an empty field at the start of the list. If you'd like to get the same behavior when splitting another string on whitespace, just use a single space in place of the pattern: `split ' ', $other_string`. Using a space instead of the pattern is a special kind of `split`.

Generally, the patterns you use for `split` are as simple as the ones you see here. But if the pattern becomes more complex, be sure to avoid using capturing parentheses in the pattern since these trigger the (usually) unwanted "separator retention mode" (see the perlfunc documentation for details). Use the noncapturing parentheses, `(?:)`, in `split` if you need to group things.

The join Function

The `join` function doesn't use patterns; it performs the opposite function of `split`: `split` breaks up a string into a number of pieces, and `join` glues together a bunch of pieces to make a single string. The `join` function looks like this:

```
my $result = join $glue, @pieces;
```

The first argument to `join` is the glue, which may be any string. The remaining arguments are a list of pieces. `join` puts the glue string between the pieces and returns the resulting string:

```
my $x = join ":", 4, 6, 8, 10, 12;  # $x is "4:6:8:10:12"
```

In that example, you have five items, so there are only four colons. That is, there are four pieces of glue. The glue shows up only between the pieces, never before or after them. So, there will be one fewer piece of glue than the number of items in the list.

This means there may be no glue at all if the list doesn't have at least two elements:

```
my $y = join "foo", "bar";       # gives just "bar", since no foo glue is neede
my @empty;                       # empty array
my $empty = join "baz", @empty;  # no items, so it's an empty string
```

Using `$x` from earlier, you can break up a string and put it back together with a different delimiter:

```
my @values = split /:/, $x;  # @values is (4, 6, 8, 10, 12)
my $z = join "-", @values;   # $z is "4-6-8-10-12"
```

Although `split` and `join` work well together, don't forget the first argument to `join` is always a string, not a pattern.

m// in List Context

When you use `split`, the pattern specifies the separator: the part that isn't the useful data. Sometimes it's easier to specify what you want to keep.

When a pattern match (`m//`) is used in a list context, the return value is a list of the capture variables created in the match, or an empty list if the match failed:

```
$_ = "Hello there, neighbor!";
my ($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

This makes it easy to give the match variables easy-to-use names, and these names may persist past the next pattern match. (Note also that, because there's no `=~` in that code, the pattern matches against `$_` by default.)

NOTE

Perl v5.26 adds the special `@{^CAPTURE}` array variable that holds all the capture variables. The first element is the same as `$&` (the entire match); the rest of the elements align with the capture buffer numbers.

The `/g` modifier that you first saw on `s///` also works with `m//`, which lets it match at more than one place in a string. In this case, a pattern with a pair of parentheses will return a capture from each time it matches:

```
my $text = "Fred dropped a 5 ton granite block on Mr. Slate";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
# Result: Fred dropped a ton granite block on Mr Slate
```

This is like using `split` "inside out": instead of specifying what we want to remove, we specify what we want to keep.

In fact, if there is more than one pair of parentheses, each match may return more than one string. Let's say that we have a string that we want to read into a hash, something like this:

```
my $text = "Barney Rubble Fred Flintstone Wilma Flintstone";
my %last_name = ($text =~ /(\w+)\s+(\w+)/g);
```

Each time the pattern matches, it returns a pair of captures. Those pairs of values then become the key-value pairs in the newly created hash.

More Powerful Regular Expressions

After already reading three (almost!) chapters about regular expressions, you know that they're a powerful feature in the core of Perl. But there are even more features that the Perl developers have added; you'll see some of the most important ones in this section. At the same time, you'll see a little more about the internal operation of the regular expression engine.

Nongreedy Quantifiers

So far Perl's quantifiers have been greedy. They match as much text as they can using the leftmost longest rule. Sometimes this means that you match too much.

Consider this example where you want to replace a name between tags with an all uppercase version:

```
my $text = '<b>Fred</b> and <b>Barney</b>';
$text =~ s|<b>(.*)</b>|<b>\U$1\E</b>|g;
print "$text\n";
```

It doesn't work:

```
<b>FRED</B> AND <B>BARNEY</b>
```

What happened? You tried to do a global match and expected two matches? How many actually matched?

```
my $text = '<b>Fred</b> and <b>Barney</b>';
my $match_count = $text =~ s|<b>(.*)</b>|\U$1|g;
print "$match_count: $text\n";
```

You see that there was only one match:

```
1: FRED</B> AND <B>BARNEY
```

Since the `.*` is greedy, it matched everything from the first `<b>` to the *last* `</b>`. You wanted it to match from a `<b>` to the *next* `</b>`. This is one of the problems with parsing HTML with regular expressions.

NOTE

Most Perlers will tell you that you can't parse HTML with regexes, but that's more about your skill and attention to detail than Perl's ability. Tom Christiansen shows that you can in an answer on StackOverflow.

You don't want the `.*` to match the longest portion it can. You want it to match just enough. If you add a `?` after any quantifier, that quantifier stops matching when it first finds the part after the quantifier:

```
my $text = '<b>Fred</b> and <b>Barney</b>';
my $match_count = $text =~ s|<b>(.*?)</b>|\U$1|g; # nongreedy
print "$match_count: $text\n";
```

Now it makes two matches and only the names are capitalized:

```
2: FRED and BARNEY
```

Instead of matching to the end of the string and backtracking until it finds a way to match the rest of the pattern, the regex keeps checking that it hasn't run into the next part of the pattern (see Table 9-1).

Table 9-1. Regular expression quantifiers with the nongreedy modifier

Metacharacter	Number to match
`??`	Least of zero or one match
`*?`	Zero or more, as few as possible
`+?`	One or more, as few as possible
`{3,}?`	At least three, but as few as possible
`{3,5}?`	At least three, as many as five, but as few as possible

Fancier Word Boundaries

The `\b` matches between a "word" character and a non-"word" character. As you saw in Chapter 7, Perl's idea of a word is a bit different than ours. Suppose that you wanted to capitalize the first letter of each word in a string. You might think that you could substitute anything right after a word boundary:

```
my $string = "This doesn't capitalize correctly.";
$string =~ s/\b(\w)/\U$1/g;
print "$string\n";
```

By Perl's definition, that apostrophe is a word boundary even though it's in the middle of a word (well, a contraction of two words, but still):

```
This Doesn'T Capitalize Correctly.
```

NOTE

Unicode specifies levels of regular expression compliance in Unicode Technical Report #18, which includes sophisticated boundary assertions. Perl aims to be the most Unicode-compliant language out there.

Perl v5.22 added a new sort of word boundary based on the Unicode definition. That definition looks farther around the position to make a better guess about where a word might start or end. The new boundary syntax builds on `\b` by adding curly braces to denote the sort of boundary:

```
use v5.22;

my $string = "this doesn't capitalize correctly.";
$string =~ s/\b{wb}(\w)/\U$1/g;
print "$string\n";
```

The `\b{wb}` is smart enough to recognize that the `t` after the apostrophe is not the start of a new word:

```
This Doesn't Capitalize Correctly.
```

The rules it uses are a bit convoluted and they aren't perfect, but they do better than the old `\b`.

There's also a new sentence boundary in v5.22. The `\b{sb}` uses a set of rules to guess if punctuation is at the end of a sentence or is internal punctuation, such as in "Mr. Flintstone."

That's not enough, though. Perl v5.24 adds a line boundary, which indicates a good place to break text so that you don't wrap a line in the middle of a word, at inappropriate punctuation, or at a nonbreaking space. The `\b{lb}` knows where to insert the newlines:

```
$string =~ s/(.{50,75}\b{lb})/$1\n/g;
```

Like the other fancy word boundaries, this one guesses based on heuristics. In some cases it might not break where you think it should.

Matching Multiple-Line Text

Classic regular expressions were used to match just single lines of text. But since Perl can work with strings of any length, Perl's patterns can

match multiple lines of text as easily as single lines. Of course, you have to include an expression that holds more than one line of text. Here's a string that's four lines long:

```
$_ = "I'm much better\nthan Barney is\nat bowling,\nWilma.\n";
```

Now, the anchors `^` and `$` are normally anchors for the start and end of the whole string (Chapter 8). But the `/m` regular expression option lets them match at internal newlines as well (think `m` for multiple lines). This makes them anchors for the start and end of each *line* rather than the whole string. So this pattern can match:

```
print "Found 'wilma' at start of line\n" if /^wilma\b/im;
```

Similarly, you could do a substitution on each line in a multiline string. Here, we read an entire file into one variable, then add the file's name as a prefix at the start of each line:

```
open FILE, $filename
  or die "Can't open '$filename': $!";
my $lines = join '', <FILE>;
$lines =~ s/^/$filename: /gm;
```

Updating Many Files

The most common way of programmatically updating a text file is by writing an entirely new file that looks similar to the old one but making whatever changes we need as we go along. As you'll see, this technique gives nearly the same result as updating the file itself, but it has some beneficial side effects as well.

In this example, suppose you have hundreds of files with a similar format. One of them is *fred03.dat*, and it's full of lines like these:

```
Program name: granite
Author: Gilbert Bates
Company: RockSoft
```

```
Department: R&D
Phone: +1 503 555-0095
Date: Tues March 9, 2004
Version: 2.1
Size: 21k
Status: Final beta
```

You need to fix this file so that it has some different information. Here's roughly what this one should look like when you're done:

```
Program name: granite
Author: Randal L. Schwartz
Company: RockSoft
Department: R&D
Date: June 12, 2008 6:38 pm
Version: 2.1
Size: 21k
Status: Final beta
```

In short, you need to make three changes. The name of the `Author` should be changed; the `Date` should be updated to today's date; and the `Phone` should be removed completely. And you have to make these changes in hundreds of similar files as well.

Perl supports a way of in-place editing of files with a little extra help from the diamond operator (<>). Here's a program to do what you want, although it may not be obvious how it works at first. This program's only new feature is the special variable `$^I` ; ignore that for now, and we'll come back to it:

```
#!/usr/bin/perl -w

use strict;

chomp(my $date = `date`);
$^I = ".bak";

while (<>) {
  s/\AAuthor:.*/Author: Randal L. Schwartz/;
  s/\APhone:.*\n//;
```

```
    s/\ADate:.*/Date: $date/;
    print;
  }
```

Since you need today's date, the program starts by using the system *date* command. A better way to get the date (in a slightly different format) would almost surely be to use Perl's own `localtime` function in a scalar context:

```
  my $date = localtime;
```

The next line sets `$^I`, but keep ignoring that for the moment.

The list of files for the diamond operator here is coming from the command line. The main loop reads, updates, and prints one line at a time. With what you know so far, that means that you'll dump all of the files' newly modified contents to your terminal, scrolling furiously past your eyes, without the files being changed at all. But stick with us. Note that the second substitution can replace the entire line containing the phone number with an empty string—leaving not even a newline—so when that's printed, nothing comes out, and it's as if the `Phone` never existed. Most input lines won't match any of the three patterns, and those will be unchanged in the output.

So this result is close to what you want, except that we haven't shown you how the updated information gets back out onto the disk. The answer is in the variable `$^I`. By default it's `undef`, and everything is normal. But when it's set to some string, it makes the diamond operator (`<>`) even more magical than usual.

You already know about much of the diamond's magic—it will automatically open and close a series of files for you, or read from the standard-input stream if there aren't any filenames given. But when there's a string in `$^I`, that string is used as a backup filename's extension. Let's see that in action.

Let's say it's time for the diamond to open our file *fred03.dat*. It opens it like before, but now it renames it, calling it *fred03.dat.bak*. You've still got

the same file open, but now it has a different name on the disk. Next, the diamond creates a new file and gives it the name *fred03.dat*. That's OK; you weren't using that name anymore. And now the diamond selects the new file as the default for output so that anything that we print will go into that file. Now the `while` loop will read a line from the old file, update that, and print it out to the new file. This program can update thousands of files in a few seconds on a typical machine. Pretty powerful, huh?

NOTE

The diamond also tries to duplicate the original file's permission and ownership settings as much as possible. See the documentation for your particular system for details.

Once the program has finished, what does the user see? The user says, "Ah, I see what happened! Perl edited my file *fred03.dat*, making the changes I needed, and saved me a copy of the original in the backup file *fred03.dat.bak* just to be helpful!" But you now know the truth: Perl didn't really edit any file. It made a modified copy, said "Abracadabra!", and switched the files around while you were watching sparks come out of the magic wand. Tricky.

Some folks use a tilde (~) as the value for `$^I` , since that resembles what *emacs* does for backup files. Another possible value for `$^I` is the empty string. This enables in-place editing, but doesn't save the original data in a backup file. Since a small typo in your pattern could wipe out all of the old data though, using the empty string is recommended only if you want to find out how good your backup tapes are. It's easy enough to delete the backup files when you're done. And when something goes wrong and you need to rename the backup files to their original names, you'll be glad that you know how to use Perl to do that (we'll show you an example in "Renaming Files").

In-Place Editing from the Command Line

A program like the example from the previous section is fairly easy to write. But Larry decided it wasn't easy enough.

Imagine that you need to update hundreds of files that have the misspelling `Randall` instead of the one-`l` name `Randal`. You could write a program like the one in the previous section. Or you could do it all with a one-line program, right on the command line:

```
$ perl -p -i.bak -w -e 's/Randall/Randal/g' fred*.dat
```

Perl has a whole slew of command-line options you can use to build a complete program in a few keystrokes. Let's see what these few do (and see perlrun for the rest).

Starting the command with *perl* does something like putting `#!/usr/bin/perl` at the top of a file: it says to use the program *perl* to process what follows.

The `-p` option tells Perl to write a program for you. It's not much of a program, though; it looks something like this:

```
while (<>) {
  print;
}
```

If you want even less, you could use `-n` instead; that leaves out the automatic `print` statement, so you can print only what you wish. (Fans of *awk* will recognize `-p` and `-n`.) Again, it's not much of a program, but it's pretty good for the price of a few keystrokes.

The next option is `-i.bak`, which you might have guessed sets `$^I` to `".bak"` before the program starts. If you don't want a backup file, you can use `-i` alone, with no extension. If you don't want a spare parachute, you can leave the airplane with just one.

You've seen `-w` before—it turns on warnings.

The `-e` option says "executable code follows." That means that the `s/Randall/Randal/g` string is treated as Perl code. Since you've already got a `while` loop (from the `-p` option), this code is put inside the loop, before the `print`. For technical reasons, the last semicolon in the `-e` code is optional. But if you have more than one `-e`, and thus more than one chunk of code, you can safely omit only the semicolon at the end of the last one.

The last command-line parameter is `fred*.dat`, which says that `@ARGV` should hold the list of filenames that match that filename pattern. Put the pieces all together, and it's as if you had written a program like this, and put it to work on all of those `fred*.dat` files:

```
#!/usr/bin/perl -w

$^I = ".bak";

while (<>) {
  s/Randall/Randal/g;
  print;
}
```

Compare this program to the one you used in the previous section. It's pretty similar. These command-line options are pretty handy, aren't they?

Exercises

See "Answers to Chapter 9 Exercises" for answers to these exercises:

1. [7] Make a pattern that will match three consecutive copies of whatever is currently contained in `$what`. That is, if `$what` is `fred`, your pattern should match `fredfredfred`. If `$what` is `fred|barney`, your pattern should match `fredfredbarney` or `barneyfredfred` or `barneybarneybarney` or many other variations. (Hint: you should set `$what` at the top of the pattern test program with a statement like `my $what = 'fred|barney';`.)
2. [12] Write a program that makes a modified copy of a text file. In the copy, every string `Fred` (case-insensitive) should be replaced with

`Larry`. (So `Manfred Mann` should become `ManLarry Mann`.) The input filename should be given on the command line (don't ask the user!), and the output filename should be the corresponding filename ending with `.out`.

3. [8] Modify the previous program to change every `Fred` to `Wilma` and every `Wilma` to `Fred`. Now, input like `fred&wilma` should look like `Wilma&Fred` in the output.
4. [10] Extra-credit exercise: write a program to add a copyright line to all of your exercise answers so far, by placing a line like:

```
## Copyright (C) 20XX by Yours Truly
```

   in the file immediately after the "shebang" line. You should edit the files "in place," keeping a backup. Presume that the program will be invoked with the filenames to edit already on the command line.
5. [15] Extra extra-credit exercise: modify the previous program so that it doesn't edit the files that already contain the copyright line. As a hint on that, you might need to know that the name of the file being read by the diamond operator is in `$ARGV`.