3 Useful Perl idioms

What this chapter covers:

- Simple and complex sorts
- The Orcish manoeuvre and the Schwartzian and Guttman-Rosler transforms
- Database Interface and database driver modules
- Benchmarking
- Command line scripts

There are a number of Perl idioms that will be useful in many data munging programs. Rather than introduce them in the text when they are first met, we will discuss them all here.

3.1 Sorting

Sorting is one of the most common tasks that you will carry out when data munging. As you would expect, Perl makes sorting very easy for you, but there are a few niceties that we'll come to later in this section.

3.1.1 Simple sorts

Perl has a built-in `sort` function which will handle simple sorts very easily. The syntax for the `sort` function is as follows:

```
@out = sort @in;
```

This takes the elements of the list `@in`, sorts them lexically, and returns them in array `@out`. This is the simplest scenario. Normally you will want something more complex, so `sort` takes another argument which allows you to define the sort that you want to perform. This argument is either the name of a subroutine or a block of Perl code (enclosed in braces). For example, to sort data numerically[1] you would write code like this:

```
@out = sort numerically @in;
```

and a subroutine called `numerically` which would look like this:

```
sub numerically {
  return $a <=> $b;
}
```

There are a couple of things to notice in this subroutine. First, there are two special variables, `$a` and `$b`, which are used in the subroutine. Each time Perl calls the subroutine, these variables are set to two of the values in the source array. Your subroutine should compare these two values and return a value that indicates which of the elements should come first in the sorted list. You should return `-1` if `$a` comes before `$b`, `1` if `$b` comes before `$a`, and `0` if they are the same. The other thing to notice is the `<=>` operator which takes two values and returns `-1`, `0`, or `1`, depending on which value is numerically larger. This function, therefore, compares the two values and returns the values required by `sort`. If you wanted to sort the list in

[1] Rather than lexically, where 100 comes before 2.

descending numerical order, you would simply have to reverse the order of the comparison of $a and $b like so:

```
sub desc_numerically {
  return $b <=> $a;
}
```

Another way of handling this is to sort the data in ascending order and reverse the list using Perl's built-in reverse function like this:

```
@out = reverse sort numerically @in;
```

There is also another operator, cmp, which returns the same values but orders the elements lexically. The original default sort is therefore equivalent to:

```
@out = sort lexically @in;

sub lexically {
  return $a cmp $b;
}
```

3.1.2 Complex sorts

Sorts as simple as the ones we've discussed so far are generally not written using the subroutine syntax that we have used above. In these cases, the block syntax is used. In the block syntax, Perl code is placed between the sort function and the input list. This code must still work on $a and $b and must obey the same rules about what it returns. The sorts that we have discussed above can therefore be rewritten like this:

```
@out = sort { $a <=> $b } @in;
@out = sort { $b <=> $a } @in; # or @out = reverse sort { $a <=> $b } @in
@out = sort { $a cmp $b } @in;
```

The subroutine syntax can, however, be used to produce quite complex sort criteria. Imagine that you have an array of hashes where each hash has two keys, forename and surname, and you want to sort the list like a telephone book (i.e., surname first and then forename). You could write code like this:

```
my @out = sort namesort @in;

sub namesort {
  return $a->{surname} cmp $b->{surname}
    || $a->{forename} cmp $b->{forename};
}
```

Note that we make good use of the "short circuit" functionality of the Perl || operator. Only if the surnames are the same and the first comparison returns 0 is the second comparison evaluated.

We can, of course, mix numeric comparisons with lexical comparisons and even reverse the order on some comparisons. If our hash also contains a key for age, the following code will resolve two identical names by putting the older person first.

```
my @out = sort namesort @in;

sub namesort {
  return $a->{surname} cmp $b->{surname}
    || $a->{forename} cmp $b->{forename}
    || $b->{age} <=> $a->{age};
}
```

This default sort mechanism is implemented using a Quicksort algorithm. In this type of sort, each element of the list is compared with at least one other element in order to determine the correct sequence. This is an efficient method if each comparison is relatively cheap; however, there are circumstances where you are sorting on a value which is calculated from the element. In these situations, recalculating the value each time can have a detrimental effect on performance. There are a number of methods available to minimize this effect and we will now discuss some of the best ones.

3.1.3 The Orcish Manoeuvre

One simple way to minimize the effect of calculating the sort value multiple times is to cache the results of each calculation so that we only have to carry out each calculation once. This is the basis of the *Orcish Manoeuvre* (a pun on “or cache”) devised by Joseph Hall. In this method, the results of previous calculations are stored in a hash. The basic code would look like this:

```
my %key_cache;

my @out = sort orcish @in;

sub orcish {

  return ($key_cache{$a} ||= get_sort_key($a))
    <=> ($key_cache{$b} ||= get_sort_key($b));
}

sub get_sort_key {

  # Code that takes the list element and returns
  # the part that you want to sort on
}
```

There is a lot going on here so it’s worth looking at it in some detail.

The hash `%key_cache` is used to store the precalculated sort keys.

The function `orcish` carries out the sort, but for each element, before calculating the sort key, it checks to see if the key has already been calculated, in which case it will be stored in `%key_cache`. It makes use of Perl's `||=` operator to make the code more streamlined. The code

```
$key_cache{$a} ||= get_sort_key($a)
```

can be expanded to

```
$key_cache{$a} = $key_cache{$a} || get_sort_key($a)
```

The net effect of this code is that if `$key_cache{$a}` doesn't already exist then `get_sort_key` is called to calculate it and the result is stored in `$key_cache{$a}`. The same procedure is carried out for `$b` and the two results are then compared using <=> (this could just as easily be `cmp` if you need a lexical comparison).

Depending on how expensive your `get_sort_key` function is, this method can greatly increase your performance in sorting large lists.

3.1.4 *Schwartzian transform*

Another way of avoiding recalculating the sort keys a number of times is to use the Schwartzian transform. This was named after Randal L. Schwartz, a well-known member of the Perl community and author of a number of Perl books, who was the first person to post a message using this technique to the comp.lang.perl.misc newsgroup.

In the Schwartzian transform we precalculate all of the sort keys before we begin the actual sort.

As an example, let's go back to our list of CDs. If you remember, we finally decided that we would read the data file into an array of hashes, where each hash contained the details of each CD. Figure 3.1 is a slightly simplified diagram of the `@CDs` array (each hash has only two fields).

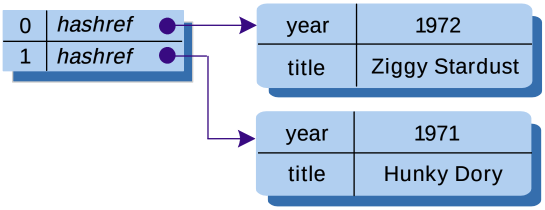

Figure 3.1
The unsorted array of CD hashes

Suppose that now we want to produce a list of CDs arranged in order of release date. The naïve way to write this using `sort` would be like this:

```
my @CDs_sorted_by_year = sort { $a->{year} <=> $b->{year} } @CDs;
```

We could then iterate across the sorted array and print out whatever fields of the hash were of interest to us.

As you can see, to get to the sort key (the release date) we have to go through a hash reference to get to that hash itself. Hash lookup is a reasonably expensive operation in Perl and we'd be better off if we could avoid having to look up each element a number of times.

Let's introduce an intermediate array. Each element of the array will be a reference to a two-element array. The first element will be the year and the second element will be a reference to the original hash. We can create this list very easily using `map`.

```
my @CD_and_year = map { [$_->{year}, $_] } @CDs;
```

Figure 3.2 shows what this new array would look like.

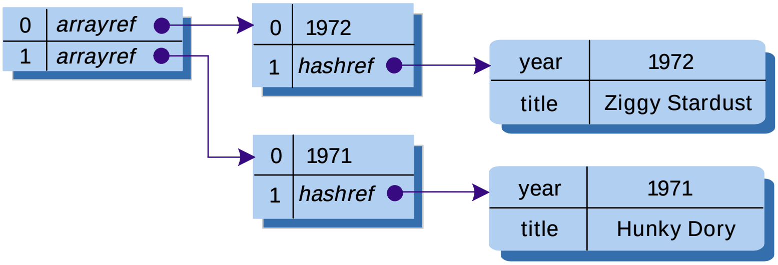

Figure 3.2 **`@CD_and_year` contains references to a two element array**

The year field in each hash has been extracted only once, which will save us a lot of time. We can now sort our new array on the first element of the array. Array lookup is much faster than hash lookup. The code to carry out this sort looks like this:

```
my @sorted_CD_and_year = sort { $a->[0] <=> $b->[0] } @CD_and_year;
```

Figure 3.3 shows this new array.

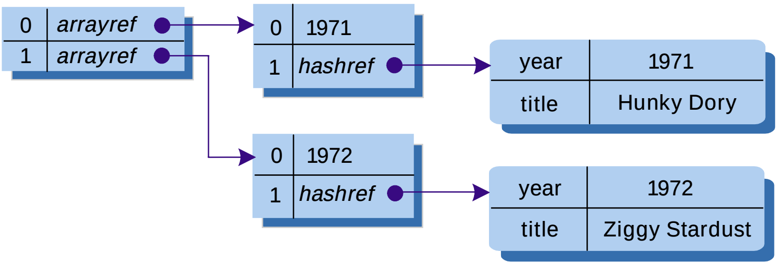

Figure 3.3 **`@sorted_CD_and_year` is `@CD_and_year` sorted by the first element of the array**

Now in `@sorted_CD_and_year` we have an array of references to arrays. The important thing, however, is that the array is ordered by year. In fact, we only need the second element of each of these arrays, because that is a reference to our original hash. Using `map` it is simple to strip out the parts that we need.

```
my @CDs_sorted_by_year = map { $_->[1] } @sorted_CD_and_year;
```

Figure 3.4 shows what this array would look like.

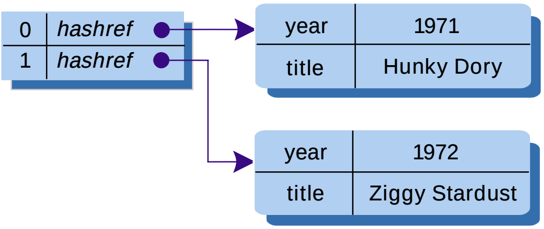

Figure 3.4
`@CDs_sorted_by_year` contains just the hash references from `@sorted_CD_and_year`

Let's put those three stages together.

```
my @CD_and_year = map { [$_, $_->{year}] } @CDs;
my @sorted_CD_and_year = sort { $a->[1] <=> $b->[1] } @CD_and_year;
my @CDs_sorted_by_year = map { $_->[0] } @sorted_CD_and_year;
```

That, in a nutshell, is the Schwartzian transform—a sort surrounded by two `map`s. There is one more piece of tidying up that we can do. As each of the `map`s and the `sort` take an array as input and return an array we can chain all of these transformations together in one statement and lose both of the intermediate arrays.

```
my @CDs_sorted_by_year = map { $_->[0] }
                         sort { $a->[1] <=> $b->[1] }
                         map { [$_, $_->{year}] } @CDs;
```

If this doesn't look quite like what we had before, try tracing it through in reverse. Our original array (`@CDs`) is passed in at the bottom. It goes through the `map` that dereferences the hash, then the `sort`, and finally the last `map`.

The chaining together of multiple list processing functions, where the output of the first map becomes the input to the sort and so on, is very similar to the I/O pipes that we saw when looking at the UNIX filter model earlier.

The Schwartzian transform can be used anywhere that you want to sort a list of data structures by one of the data values contained within it, but that's not all it can do. Here's a three-line script that prints out our CD file (read in through `STDIN`), sorted by the recording label.

```
print map { $_->[0] }
      sort { $a->[1] cmp $b->[1] }
      map { [$_, (split /\t/)[2]] } <STDIN>;
```

3.1.5 The Guttman-Rosler transform

At the 1999 Perl Conference, Uri Guttman and Larry Rosler presented a paper on sorting with Perl. It covered all of the techniques discussed so far and went a step further, by introducing the concept of the *packed-default* sort. They started from two premises:

1. Eliminating any hash or array dereferencing would speed up the sort.
2. The default lexical sort (without any sort subroutine or block) is the fastest.

The resulting method is an interesting variation on the Schwartzian transform. Instead of transforming each element of the list into a two element list (the sort key and the original data) and sorting on the first element of this list, Guttman and Rosler suggest converting each element of the original list into a string with the sort key at the beginning and the original element at the end. A list containing these strings can then be sorted lexically and the original data is extracted from the sorted list.

The example that they use in their paper is that of sorting IP addresses. First they convert each element to a string in which each part of the IP address is converted (using `pack`) to the character represented by that value in ASCII. This four-character string has the original data appended to the end:

```
my @strings = map { pack('C4', /(\d+)\.(\d+)\.(\d+)\.(\d+)/) . $_ } @IPs;
```

then the strings are lexically sorted using the default `sort` mechanism

```
my @sorted_strings = sort @strings
```

and finally the original data is extracted.

```
my @sorted @IPs = map { substr($_, 4) } @sorted_strings;
```

Rewriting this to make it look more like the Schwartzian transform, we get this:

```
my @sorted_IPs = map { substr($_, 4) }
                 sort
                 map { pack('C4', /(\d+)\.(\d+)\.(\d+)\.(\d+)/) . $_ } @IPs;
```

This type of sort needs a bit more thought than the other methods that we have considered in order to create a suitable string for sorting; however, it can pay great dividends in the amount of performance improvement that you can see.

3.1.6 Choosing a sort technique

If you are having performance problems with a program that contains a complex sort, then it is quite possible that using one of the techniques from this section will speed up the script. It is, however, possible that your script could get slower. Each of the techniques will improve the actual sort time, but they all have an overhead which means that your sort will need to be quite large before you see any improvement.

When selecting a sort technique to use, it is important that you use the benchmarking methods, discussed in section 3.4, to work out which technique is most appropriate. Of course, if your script is only going to run once, then spending half a day benchmarking sorts for the purpose of shaving five seconds off the runtime isn't much of a gain.

This section has only really started to discuss the subject of sorting in Perl. If you'd like to know more, Guttman and Rosler's paper is a very good place to start. You can find it online at http://www.hpl.hp.com/personal/Larry_Rosler/sort/.

3.2 Database Interface (DBI)

As discussed in chapter 1, a common source or sink for data is a database. For many years Perl has had mechanisms that enable it to talk to various database systems. For example, if you wanted to exchange data with an Oracle database you would use `oraperl` and if you had to communicate with a Sybase database you would use `sybperl`. Modules were also available to talk to many other popular database systems.

Most of these database access modules were a thin Perl wrapper around the programming APIs that were already provided by the database vendors. The mechanisms for talking to the various databases were all doing largely the same thing, but they were doing it in completely incompatible ways.

This has all changed in recent years with the introduction of the generic Perl Database Interface (DBI) module. This module was designed and written by Tim Bunce (the author and maintainer of `oraperl`). It allows a program to connect to any of the supported database systems and read and write data using exactly the same syntax. The only change required to connect to a different database system is to change one string that is passed to the DBI connect function. It does this by using different database driver (DBD) modules. These are all named `DBD::<db_name>`. You will need to obtain the DBD module for whichever database you are using separately from the main DBI module.

3.2.1 Sample DBI program

A sample DBI program to read data from a database would look like this:

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

use strict;
use DBI;

```

```
 6: my $user = 'dave';
 7: my $pass = 'secret';
 8: my $dbh = DBI->connect('dbi:mysql:testdb', $user, $pass,
 9:                        {RaiseError => 1})
10:   || die "Connect failed: $DBI::errstr";
11:
12: my $sth = $dbh->prepare('select col1, col2, col3 from my_table')
13:
14: $sth->execute;
15:
16: my @row;
17: while (@row = $sth->fetchrow_array) {
18:   print join("\t", @row), "\n";
19: }
20:
21: $sth->finish;
22: $dbh->disconnect;
```

While this is a very simple DBI program, it demonstrates a number of important DBI concepts and it is worth examining line by line.

Line 1 points to the Perl interpreter. Notice the use of the `-w` flag.

Line 3 switches on the `strict` pragma.

Line 4 brings in the `DBI.pm` module. This allows us to use the DBI functions.

Lines 6 and 7 define a username and password that we will use to connect to the database. Obviously, in a real program you probably wouldn't want to have a password written in a script in plain text.

Line 8 connects us to the database. In this case we are connecting to a database running MySQL. This free database program is very popular for web systems. This is the only line that would need to change if we were connecting to a different database system. The `connect` function takes a number of parameters which can vary depending on the database to which you are connecting. The first parameter is a connection string. This changes its precise meaning for different databases, but it is always a colon-separated string. The first part is the string `dbi` and the second part is always the name of the database system[2] that we are connecting to. In this case the string `mysql` tells DBI that we will be talking to a MySQL database, and it should therefore load the `DBD::mysql` module. The third section of the connection string in this case is the particular database that we want to connect to. Many database systems (including MySQL) can store many different databases on the same database server. In this case we want to connect to a database called testdb. The second and third parameters are valid usernames and passwords for connecting to this database.

[2] Or, more accurately, the name of the DBD module that we are using to connect to the database.

The fourth parameter to DBI->connect is a reference to a hash containing various configuration options. In this example we switch on the RaiseError option, which will automatically generate a fatal run-time error if a database error occurs.

The DBI->connect function returns a database handle, which can then be used to access other DBI functions. If there is an error, the function returns undef. In the sample program we check for this and, if there is a problem, the program dies after printing the value of the variable $DBI::errstr which contains the most recent database error message.

Line 12 prepares an SQL statement for execution against the database. It does this by calling the DBI function prepare. This function returns a statement handle which can be used to access another set of DBI functions—those that deal with executing queries on the database and reading and writing data. This handle is undefined if there is an error preparing the statement.

Line 14 executes the statement and dies if there is an error.

Line 16 defines an array variable which will hold each row of data returned from the database in turn.

Lines 17 to 19 define a loop which receives each row from the database query and prints it out. On line 17 we call fetchrow_array which returns a list containing one element for each of the columns in the next row of the result set. When the result set has all been returned, the next call to fetchrow_array will return the value undef.

Line 18 prints out the current row with a tab character between each element.

Lines 21 and 22 call functions that reclaim the memory used for the database and statement handles. This memory will be reclaimed automatically when the variables go out of scope, but it is tidier to clean up yourself.

This has been a very quick overview of using the DBI. There are a number of other functions and the most useful ones are listed in appendix A. More detailed documentation comes with the DBI module and your chosen DBD modules.

3.3 Data::Dumper

As your data structures get more and more complex it will become more and more useful to have an easy way to see what they look like. A very convenient way to do this is by using the Data::Dumper module which comes as a standard part of the Perl distribution. Data::Dumper takes one or more variables and produces a "stringified" version of the data contained in the variables.

We'll see many examples of Data::Dumper throughout the book but, as an example, let's use it to get a dump of the CD data structure that we built in the previous chapter. The data structure was built up using code like this:

```
my @CDs;
my @attrs = qw(artist title label year);
  while (<STDIN>) {
    chomp;
    my %rec;
    @rec{@attrs} = split /\t/;
    push @CDs, \%rec;
  }
```

In order to use `Data::Dumper` we just need to add a `use Data::Dumper` statement and a call to the `Dumper` function like this:

```
use Data::Dumper;
my @CDs;

my @attrs = qw(artist title label year);
while (<STDIN>) {
  chomp;
  my %rec;
  @rec{@attrs} = split /\t/;
  push @CDs, \%rec;
}

print Dumper(\@CDs);
```

Running this program using our CD files as input produces the following output:

```
$VAR1 = [
           {
            'artist' => 'Bragg, Billy',
            'title' => 'Workers\' Playtime',
            'year' => '1987',
            'label' => 'Cooking Vinyl'
          },
          {
            'artist' => 'Bragg, Billy',
            'title' => 'Mermaid Avenue',
            'year' => '1998',
            'label' => 'EMI'
          },
          {
            'artist' => 'Black, Mary',
            'title' => 'The Holy Ground',
            'year' => '1993',
            'label' => 'Grapevine'
          },
          {
            'artist' => 'Black, Mary',
            'title' => 'Circus',
            'year' => '1996',
            'label' => 'Grapevine'
          },
     {
      'artist' => 'Bowie, David',
      'title' => 'Hunky Dory',
      'year' => '1971',
      'label' => 'RCA'
     },
     {
      'artist' => 'Bowie, David',
      'title' => 'Earthling',
      'year' => '1998',
      'label' => 'EMI'
     }
    ];
```

This is a very understandable representation of our data structure.

Notice that we passed a reference to our array rather than the array itself. This is because `Dumper` expects a list of variables as arguments so, if we had passed an array, it would have processed each element of the array individually and produced output for each of them. By passing a reference we forced it to treat our array as a single object.

3.4 Benchmarking

When choosing between various ways to implement a task in Perl, it will often be useful to know which option is the quickest. Perl provides a module called Benchmark that makes it easy to get this data. This module contains a number of functions (see the documentation for details) but the most useful for comparing the performance of different pieces of code is called `timethese`. This function takes a number of pieces of code, runs them each a number of times, and returns the time that each piece of code took to run. You should, therefore, break your options down into separate functions which all do the same thing in different ways and pass these functions to `timethese`. For example, there are four ways to put the value of a variable into the middle of a fixed string. You can interpolate the variable directly within the string

```
$str = "The value is $x (or thereabouts)";
```

or join a list of values

```
$str = join '', 'The value is ', $x, ' (or thereabouts)';
```

or concatenate the values

```
$s = 'The value is ' . $x . ' (or thereabouts)';
```

or, finally, use `sprintf`.

```
$str = sprintf 'The value is %s (or thereabouts)', $x;
```

In order to calculate which of these methods is the fastest, you would write a script like this:

```
#!/usr/bin/perl -w
use strict;
use Benchmark qw(timethese);

my $x = 'x' x 100;

sub using_concat {
  my $str = 'x is ' . $x . ' (or thereabouts)';
}

sub using_join {
  my $str = join '', 'x is ', $x, ' (or thereabouts)';
}

sub using_interp {
  my $str = "x is $x (or thereabouts)";
}

sub using_sprintf {
  my $str = sprintf("x is %s (or thereabouts)", $x);
}

timethese (1E6, {
   'concat'  => \&using_concat,
   'join'    => \&using_join,
   'interp'  => \&using_interp,
   'sprintf' => \&using_sprintf,
});
```

On my current computer,[3] running this script gives the following output:

```
Benchmark: timing 1000000 iterations of concat, interp, join, sprintf …
   concat:  8 wallclock secs ( 7.36 usr +  0.00 sys =  7.36 CPU) @ 135869.57/s (n=1000000)
   interp:  8 wallclock secs ( 6.92 usr + -0.00 sys =  6.92 CPU) @ 144508.67/s (n=1000000)
     join:  9 wallclock secs ( 8.38 usr +  0.03 sys =  8.41 CPU) @ 118906.06/s (n=1000000)
   sprintf: 12 wallclock secs (11.14 usr +  0.02 sys = 11.16 CPU) @ 89605.73/s
   (n=1000000)
```

What does this mean? Looking at the script, we can see that we call the function `timethese`, passing it an integer followed by a reference to a hash. The integer is the number of times that you want the tests to be run. The hash contains details of the code that you want tested. The keys to the hash are unique names for each of the subroutines and the values are references to the functions themselves. `timethese` will run each of your functions the given number of times and will print out the

[3] A rather old 200 MHz P6 with 64 MB of RAM, running Microsoft Windows 98 and ActivePerl build 521.

results. As you can see from the results we get above, our functions fall into three sets. Both concat and interp took about 8 seconds of CPU time to run 1,000,000 times; join was a little longer at 9 seconds; and sprintf came in at 12 seconds of CPU time.

You can then use these figures to help you decide which version of the code to use in your application.

3.5 Command line scripts

Often data munging scripts are written to carry out one-off tasks. Perhaps you have been given a data file which you need to clean up before loading it into a database. While you can, of course, write a complete Perl script to carry out this munging, Perl supplies a set of command line options which make it easy to carry out this kind of task from the command line. This approach can often be more efficient.

The basic option for command line processing is -e. The text following this option is treated as Perl code and is passed through to the Perl interpreter. You can therefore write scripts like:

```
perl -e 'print "Hello world\n"'
```

You can pass as many -e options as you want to Perl and they will be run in the order that they appear on the command line. You can also combine many statements in one -e string by separating them with a semicolon.

If the code that you want to run needs a module that you would usually include with a use statement, you can use the -M option to load the module. For example, this makes it easy to find the version of any module that is installed on your system[4] using code like this:

```
perl -MCGI -e 'print $CGI::VERSION'
```

These single-line scripts can sometimes be useful, but there is a whole set of more powerful options to write file processing scripts. The first of these is -n, which adds a loop around your code which looks like this:

```
LINE:
while (<>) {
  # Your -e code goes here
}
```

This can be used, for example, to write a simple grep-like script such as:

```
perl -ne 'print if /search text/' file.txt
```

[4] Providing that the module uses the standard practice of defining a $VERSION variable.

which will print any lines in `file.txt` that contain the string "search text". Notice the presence of the `LINE` label which allows you to write code using `next LINE`.

If you are transforming data in the file and want to print a result for every line, then you should use the `-p` option which prints the contents of `$_` at the end of each iteration of the `while` loop. The code it generates looks like this:

```
LINE:
while (<>) {
  # Your -e code goes here
} continue {
  print
}
```

As an example of using this option, imagine that you wanted to collapse multiple zeroes in a record to just one. You could write code like this:

```
perl -pe 's/0+/0/g' input.txt > output.txt
```

With the examples we've seen so far, the output from the script is written to `STDOUT` (that is why we redirected `STDOUT` to another file in the last example). There is another option, `-i`, which allows us to process a file in place and optionally create a backup containing the previous version of the file. The `-i` takes a text string which will be the extension added to the backup of the file, so we can rewrite our previous example as:

```
perl -i.bak -pe 's/0+/0/g' input.txt
```

This option will leave the changed data in `input.txt` and the original data in `input.txt.bak`. If you don't give `-i` an extension then no backup is made (so you'd better be pretty confident that you know what you're doing!).

There are a number of other options that can make your life even easier. Using `-a` turns on autosplit, which in turn splits each input row into `@F`. By default, autosplit splits the string on any white space, but you can change the split character using `-F`. Therefore, in order to print out the set of user names from `/etc/passwd` you can use code like this:

```
perl -a -F':' -ne 'print "$F[0]\n"' < /etc/passwd
```

The `–l` option switches on line-end processing. This automatically does a `chomp` on each line when used with `-n` or `-p`. You can also give it an optional octal number which will change the value of the output record separator (`$\`).[5] This value is

[5] Special variables like `$\` are covered in more detail in chapter 6.

appended to the end of each output line. Without the octal number, `$\` is set to the same value as the input record separator (`$/`). The default value for this is a newline. You can change the value of `$/` using the `-0` (that's dash-zero, not dash-oh) option.

What this means is that in order to have newlines automatically removed from your input lines and automatically added back to your output line, just use `–l`. For instance, the previous `/etc/passwd` example could be rewritten as:

```
perl -a -F':' -nle 'print $F[0]' < /etc/passwd
```

For more information about these command line options see the `perlrun` manual page which is installed when you install Perl.

3.6 Further information

More discussion of the Schwartzian transform, the Orcish Manoeuvre, and other Perl tricks can be found in *Effective Perl Programming* by Joseph Hall with Randal Schwartz (Addison-Wesley) and *The Perl Cookbook* by Tom Christiansen and Nathan Torkington (O'Reilly).

The more academic side of sorting in Perl is discussed in *Mastering Algorithms with Perl* by Jon Orwant, Jarkko Hietaniemi, and John Macdonald (O'Reilly).

More information about benchmarking can be found in the documentation for the `Benchmark.pm` module.

Further information about the DBI and DBD modules can be found in *Programming the Perl DBI* by Tim Bunce and Alligator Descartes (O'Reilly) and in the documentation that is installed along with the modules. When you have installed the DBI module you can read the documentation by typing

```
perldoc DBI
```

at your command line. Similarly you can read the documentation for any installed DBD module by typing

```
perldoc DBD::<name>
```

at your command line. You should replace `<name>` with the name of the DBD module that you have installed, for example "Sybase" or "mysql".

3.7 Summary

- Sorting can be very simple in Perl, but for more complex sorts there are a number of methods which can make the sort more efficient.
- Database access in Perl is very easy using the DBI.
- `Data::Dumper` is very useful for seeing what your internal data structures look like.
- Benchmarking is very important, but can be quite tricky to do correctly.
- Command line scripts can be surprisingly powerful.