Intermediate Perl

Kirrily Robert

Intermediate Perl
by Kirrily Robert

Table of Contents

List of Tables

Chapter 1. Introduction

Welcome to Netizen's *Intermediate Perl* training course. This is a one-day module in which we extend on the material covered in *Introduction to Perl* and explore the topics of references, advanced regular expressions, and interacting with the operating system.

1.1. Course outline

- Revise introduction to Perl material
- File I/O
  - Line input and globbing operators
  - Opening files and directories
  - Opening pipes
  - Finding information about files
  - Recursing down directories
  - File locking
  - Handling binary data
- Advanced regular expressions
  - Review of basic regexps
  - Multiline strings
  - Backreferences
- More functions

    - The `grep()` function
    - The `map()` function
    - `printf()` and `sprintf()`
    - `pack()` and `unpack()`
    - List manipulation with `splice()`

- System interaction
    - `system()` and `exec()`
    - Backticks
    - Interacting with the file system
    - Dealing with users, groups and permissions
    - Interacting with processes
    - Security considerations

- References and complex data structures
    - Creating and dereferencing
    - Complex data structures
    - Anonymous data structures

1.2. Assumed knowledge

This training module assumes the following prior knowledge and skills:

- Basic Perl fluency, including a familiarity with Perl variable types, functions and operators, conditional constructs, and basic regular expressions
- Some Unix experience, including logging in, moving around directories, and editing files

1.3. Module objectives

- Be able to open files and directories to read and write data, using various techniques
- Perform tests on files and directories
- Open pipes to read or write data through another program
- Use regular expressions to handle multiline data
- Use backreferences to create complex regular expressions
- Use and understand more complex Perl functions such as `grep()` and `map()`
- Use Perl functions to call system commands
- Use Perl to interact with the file system, users, and processes
- Understand the security implications of running system commands from Perl, and how to increase security
- Understand and use Perl references to create complex data structures and anonymous data structures

1.4. Platform and version details

This module is taught using Unix or a Unix-like operating system. Most of what is learnt will work equally well on Windows NT or other operating systems; your instructor will inform you throughout the course of any areas which differ.

All Netizen's Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs signficantly from previous versions of Perl, so you will need a Perl 5 interpreter to use what you have learnt. However, older Perl programs should work fine under Perl 5.

1.5. The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what ap-
pears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *`like this`*

Notes and tips appear offset from the text like this.

Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is

not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

1.6. Other materials

In addition to these notes, you should have a copy of the required text book for this course: *Programming Perl* (2nd edition), more commonly referred to as "the Camel book". The Camel book will be used throughout the day, and will be a valuable reference to take home and keep next to your computer.

You will also have received a floppy disk containing these notes in HTML form (with working links to external resources etc) and all the example scripts and data used in this course.

1.7. Logging into your account

Your username and password will have been given to you with these course notes.

1. Open the telnet program
2. Connect to the training server using the hostname or IP number given by your instructor
3. Login using the username and password you were given

You will find yourself at a Unix shell prompt. Hopefully (if you met the pre-requisites of this course) you will now be able to see that your account has a subdirectory called `exercises/` which are the example scripts and exercises given in these course notes. If you’re not quite up to speed with Unix, there’s a cheat-sheet in Appendix A of these notes.

Chapter 2. File I/O

2.1. In this chapter...

In this section, we learn how to open and interact with files and directories in various ways.

2.2. Assumed knowledge

You should already have encountered the `open()` function and the <> line input operator in a previous Perl training session or in your previous Perl experience.

2.3. Angle brackets - the line input and globbing operators

You will have encountered the line input operator <> before, in situations such as these:

```
# reading lines from STDIN
while (<>) {
        ...
        ...
}

# reading a single line of user input from STDIN
my $input = <STDIN>
```

The line input operator is discussed in-depth on page 53 of the Camel. Read it now.

- In scalar context, the line input operator yields the next line of the file referenced by the filehandle given.
- In list context, the line input operator yields all remaining lines of the file referenced by the filehandle.
- The default filehandle is `STDIN`, or any files listed on the command line of the Perl script (eg **myscript.pl file1 file2 file3**).

The *globbing* operator is nearly, but not quite, identical to the line input operator. It looks the same, and it acts partly in a similar way, but it really is a separate operator.

The filename globbing operator is documented on page 55 of the Camel.

If the angle brackets have anything in them other than a filehandle or nothing, it will work as a globbing operator and whatever is between the angle brackets will be treated as a filename wildcard. For instance:

```
my @files = <*.txt>
```

The filename glob `*.txt` is matched against files in the current directory, then either they are returned as a list (in list context, as above) or one scalar at a time (in scalar context).

If you get a list of files this way, you can then open them in turn and read from them.

```
while (<*.txt>) {
        open (FILEHANDLE, $_) || die ("Can't open $_: $!");
        ...
        ...
        close FILEHANDLE;
}
```

The `glob()` function behaves in a very similar manner to the angle bracket globbing operator.

```
my @files = glob("*.txt")
```

```
foreach (glob "*.txt") {
        ...
}
```

The `glob()` is considered much cleaner and better to use than the angle-brackets globbing operator.

2.3.1. Exercises

1. Use the line input operator to accept input from the user then print it out
2. Modify your previous script to use a `while` loop to get user input repeatedly, until they type "Q" (or "q" - check out the `lc()` and `uc()` functions in chapter 3 of your Camel book) (Answer: `exercises/answers/userinput.pl`)
3. Use the file globbing function or operator to find all Perl scripts in your home directory and print out their names (assuming they are named in the form `*.pl`) (Answer: `exercises/answers/findscripts.pl`)

2.3.1.1. Advanced exercises

1. Use the above example of globbing to print out all the Perl scripts one after the other. You will need to use the `open()` function to read from each file in turn. (Answer: `exercises/answers/printscripts.pl`)

2.4. open() and friends - the gory details

2.4.1. Opening a file for reading, writing or appending

The `open()` function is used to open a file for reading or writing (or both, or as a pipe - more on that later).

The `open()` function is documented on pages 191-195 of the Camel book, and also in **perldoc perlfunc**. Read the documentation for `open()` before going any further.

In a typical situation, we might use `open()` to open and read from a file:

```
open(LOGFILE, "/var/log/httpd/access.log")
```

Note that the < (less than) used to indicate reading is assumed; we could equally well have said `"</var/log/httpd/access.log"`.

You should *always* check for failure of an `open()` statement:

```
open(LOGFILE, "/var/log/httpd/access.log") || die "Can't open
        /var/log/httpd/access.log: $!";
```

`$!` is the special variable which contains the error message produced by the last system interaction. It is documented in chapter 2 of the Camel, on page 134.

Once a file is opened for reading or writing, we can use the filehandle we specified (in this case `LOGFILE`) for a variety of useful purposes:

```
open(LOGFILE, "/var/log/httpd/access.log") || die "Can't open
        /var/log/httpd/access/log: $!";

# use the filehandle in the in the <> line input operator...
while (<LOGFILE>) {
        print if /netizen.com.au/;
}

close LOGFILE;
```

```
# open a new logfile for appending
open(SCRIPTLOG, ">>myscript.log") || die "Can't open myscript.log: $!";

# print() takes an optional filehandle argument -
defaults to STDOUT
print SCRIPTLOG "Opened logfile successfully.\n";

close SCRIPTLOG;
```

Note that you should always close a filehandle when you're finished with it (though admittedly any open filehandles will be automatically closed when your script exits).

You can also use `sysopen()` and friends to open a file in a C-like way. See page 229 of your Camel book for details or **perldoc -f sysopen**.

2.4.1.1. Exercises

1. Write a script which opens a file for reading. Use a `while` loop to print out each line of the file.
2. Use the above script to open a Perl script. Use a regular expression to print out only those lines not beginning with a hash character (i.e. non-comment lines). (Answer: `exercises/answers/delcomments.pl`)
3. Create a new script which opens a file for writing. Write out the numbers 1 to 100 into this file. (Answer: `exercises/answers/100count.pl`)
4. Create a new script which opens a logfile for appending. Create a `while` loop which accepts input from STDIN and appends each line of input to the logfile. (Answer: `exercises/answers/logfile.pl`)
5. Create a script which opens two files, reads input from the first, and writes it out to the second. (Answer: `exercises/answers/readwrite.pl`)

2.4.2. Reading directories

It is also possible to open directories (using `opendir()` and read from them. However, it is not possible to read the contents of files in that directory simply by opening it and looping through it. Opening a directory simply makes the filenames in that directory accessible via functions such as `readdir()`.

`opendir()` is documented on page 195 of the Camel. `readdir()` is on page 202. Don't forget that function help is also available by typing **perldoc -f opendir** or **perldoc -f readdir**

```
opendir(HOMEDIR, $ENV{HOME});

my @files = readdir(HOMEDIR);

closedir HOMEDIR;

foreach (@files) {
        open(THISFILE, "<$_") || die "Can't open file $_: $!");
        ...
        ...
        close THISFILE;
}
```

2.4.2.1. Exercises

1. Use `opendir()` and `readdir()` to obtain a list of files in a directory. What order are they in?
2. Use the `sort()` function to sort the list of files asciibetically (Answer: `exercises/answers/dirlist.pl`)

2.4.3. Opening files for simultaneous read/write

Files can be opened for simultaneous read/write by putting a + in front of the > or < sign. +< is almost always preferable, however, as +> would overwrite the file before you had a chance to read from it.

Read/write access to a file is not as useful as it sounds --- you can't write into the middle of the file using this method, only onto the end. The main use for read/write access is to read the contents of a file and then append lines to the end of it.

A more flexible way to read and write a file is to import the file into an array, manipulate the array, then output each element again.

```
# program to remove duplicate lines
open(INFILE, "file.txt") || die "Can't open file.txt for in-
put: $!";
my @lines = <INFILE>;
close INFILE;

# dup-remover taken from The Perl Cookbook
my @unique = grep { ! $seen{$_} ++ } @lines;

open(OUTFILE, ">file.txt") || die "Can't open file.txt for out-
put: $!";
foreach (@unique) {
        print OUTFILE $_;
}

close OUTFILE;
```

One thing to watch out for here is memory usage. If you have a ten megabyte file, it will use at least that much memory as a Perl data structure.

2.4.3.1. Exercises

1. Open a file, reverse its contents (line by line) and write it back to the same filename (Answer: `exercises/answers/reversefile.pl`)

2.4.4. Opening pipes

If the filename given to `open()` begins with a pipe symbol (`|`), the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `|`, the filename is to be interpreted as a filename which pipes input to us.

This is often used when you want to take input from the system a line at a time. Here's an example which reads from the **rot13** filter (a simple routine which rotates the letters of its input by 13 letters, providing a very simple cipher for encoding the answers to jokes, spoilers to movies, or other low-security information):

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

use strict;

open (ROT13, "rot13 < /etc/motd |") || die "Can't open pipe: $!";

while (<ROT13>) {
        print;
}

close ROT13;
```

Conversely, we can output something through rot13:

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

use strict;
```

```
open (ROT13, "|rot13") || die "Can't open pipe: $!";

print "This is some rot13'd text:\n";
print ROT13 "This is some rot13'd text.\n";

close ROT13;
```

If you reverse the two print lines above, the output will nevertheless be in the same order as before. You'll need to set `$|` to flush the output pipe. It's on page 130 of your Camel, or in **perldoc perlvar**.

2.4.4.1. Exercises

1. Modify the second example above (provided for you as `exercises/rot13.pl` in your exercises directory to accept user input and print out the **rot13**'d version.
2. Change your script to accept input from a file using `open()` (Answer: `exercises/answers/rot13.pl`)
3. Change your script to pipe its input through the **strings** command, so that if you get a file that's not a text file, it will only look at the parts of the file which are strings. (Answer: `exercises/answers/strings.pl`)

2.5. Finding information about files

We can find out various information about files by using file test operators and functions such as `stat()`

Table 2-1. File test operators

Operator	Meaning
`-e`	File exists.
`-r`	File is readable
`-w`	File is writable
`-x`	File is executable
`-o`	File is owned by you
`-z`	File has zero size.
`-s`	File has nonzero size (returns size).
`-f`	File is a plain file.
`-d`	File is a directory.
`-l`	File is a symbolic link.
`-p`	File is a named pipe (FIFO), or Filehandle is a pipe.
`-S`	File is a socket.
`-b`	File is a block special file.
`-c`	File is a character special file.
`-t`	Filehandle is opened to a tty.
`-u`	File has setuid bit set.
`-g`	File has setgid bit set.
`-k`	File has sticky bit set.
`-T`	File is a text file.
`-B`	File is a binary file (opposite of -T).
`-M`	Age of file in days when script started.
`-A`	Same for access time.
`-C`	Same for inode change time.

The file test operators are documented fully in **perldoc perlfunc**.

Here's how the file test operators are usually used:

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

use strict;

unless (-e "config.txt") {
        die "Config file doesn't exist";
}

# or equivalently...
die "Config file doesn't exist" unless -e config.txt;
```

The `stat()` function returns similar information for a single file, in list form. `lstat()` can also be used for finding information about a file which is pointed to by a symbolic link.

2.5.1. Exercises

1. Write a script which asks a user for a file to open, takes their input from STDIN, checks that the file exists, then prints out the contents of that file. (Answer: `exercises/answers/fileexists.pl`)
2. Write a script to find zero-byte files in a directory. (Answer: `exercises/answers/zerobyte.pl`)
3. Write a script to find the largest file in a directory: `exercises/answers/largestfile.pl`)

2.6. Recursing down directories

The built-in functions described above do not enable you to easily recurse through subdirectories. Luckily, the **File::Find** module is part of the standard library distributed with Perl 5.

The **File::Find** module is documented in chapter 7 of the Camel, on page 439, or in **perldoc File::Find**.

File::Find emulates Unix's **find** command. It takes as its arguments a block to execute for each file found, and a list of directories to search.

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

use strict;
use File::Find;

print "Enter the directory to search: ";
chomp(my $dir = <STDIN>);

find (\&wanted, $dir);

sub wanted {
        print "$_\n";
}
```

For each file found, certain variables are set. `$File::Find::dir` is set to the current directory name, `$File::Find::name` contains the full name of the file, i.e. `$File::Find::dir/$_`.

2.6.1. Exercises

1. Modify the simple script above (in your scripts directory as `exercises/find.pl`) to only print out the names of plain text files only (hint: use file test operators)
2. Now use it to print out the contents of each text file. You'll probably want to pipe your output through **more** so that you can see it all. (Answer: `exercises/answers/find.pl`)

2.7. File locking

File locking can be achieved using the `flock()` function. This can be used to guard against race conditions or other problems which occur when two (or more) users open the same file in read/write mode.

`flock()` is documented on page 166 of the Camel book, or use **perldoc -f flock** to read the online documentation.

2.8. Handling binary data

If you are opening a file which contains binary data, you probably don't want to read it in a line at a time using `while (<>) { }`, as there's no guarantee that there will be any line breaks in the data.

Instead, we use `read()` to read a certain number of bytes from a file handle.

`read()` is documented on page 202 of the Camel book, or by using **perldoc -f read**.

`read()` takes the following arguments:

- The filehandle to read from
- The scalar to put the binary data into
- The number of bytes to read
- The byte offset to start from (defaults to 0)

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

use strict;

my $image = "picture.gif";

open (IMAGE, $image) or die "Can't open image file: $!";
open (OUT, ">backup/$image") or die "Can't open backup file: $!";
```

```
my $buffer;

binmode IMAGE;

while (read IMAGE, $buffer, 1024) {
        print OUT $buffer;
}

close IMAGE;
close OUT;
```

If you are using Windows, DOS, or some other types of systems, you may need to use `binmode()` to make sure that certain linefeed characters aren't translated when Perl reads a file in binary mode. While this is not needed on Unix systems, it's a good idea to use it anyway to enhance portability.

2.9. Chapter summary

- Angle brackets <> can be used for simple line input. In scalar context, they return the next line; in list context, all remaining lines; the default filehandle is `STDIN` or any files mentioned in the command line (ie `@ARGV`).
- Angle brackets can also be used as a globbing operator if anything other than a filehandle name appears between the angle brackets. In scalar context, returns the next file matching the glob pattern; in list context, returns all remaining matching files.

- The `open()` and `close()` functions can be used to open and close files. Files can be opened for reading, writing, appending, read/write, or as pipes.
- The `opendir()`, `readdir()` and `closedir()` functions can be used to open, read from, and close directories.
- The **File::Find** module can be used to recurse down through directories.
- File test operators or `stat()` can be used to find information about files
- File locking can be achieved using `flock()`
- Binary data can be read using the `read()` function. The `binmode()` function should be used to ensure platform independence when reading binary data.

Chapter 3. Advanced regular expressions

3.1. In this section...

This section builds on the basic regular expressions taught in Netizen's *Introduction to Perl* course. We will learn how to handle data which consists of multiple lines of text, including how to input data as multiple lines and different ways of performing matches against that data.

3.2. Assumed knowledge

You should already be familiar with the following topics:

- Regular expression metacharacters
- Quantifiers
- "Greediness" in regular expressions, aka maximal and minimal matching
- Character classes and alternation
- The `m//` matching function
- The `s///` substitution function
- Matching strings other than `$_` with the `=~` matching operator
- Assigning matched strings to lvalues

Patterns and regular expressions are dealt with in depth in chapter 2 of the Camel book, and further information is available in the online Perl documentation by typing **perldoc perlre**.

3.2.1. Review exercises

The following exercises are intended to refresh your memory of basic regular expressions:

1. Write a script to search a file for any of the names "Yasser Arafat", "Boris Yeltsin" or "Monica Lewinsky". Print out any lines which contain these names. (Answer: `exercises/answers/namesre.pl`)
2. What pattern could be used to match any of: Elvis Presley, Elvis Aron Presley, Elvis A. Presley, Elvis Aaron Presley. (Answer: `exercises/answers/elvisre.pl`)
3. What pattern could be used to match a blank line? (Answer: `exercises/answers/blanklinere.pl`)
4. What pattern could be used to match an IP address such as `203.20.104.241`, where each part of the address is a number from 0 to 255? (Answer: `exercises/answers/ipre.pl`)

3.3. More metacharacters

Here are some more advanced metacharacters, which build on the ones already covered in the Introduction to Perl module:

Table 3-1. More metacharacters

Metacharacter	Meaning
`\B`	Match anything other than a word boundary
`\c`*X*	Control character, i.e. **CTRL-*X***
`\0`*nn*	Octal character represented by *nn*

Metacharacter	Meaning
\x*nn*	Hexadecimal character represented by *nn*
\l	Lowercase next character
\u	Uppercase next character
\L	Lowercase until \E
\U	Uppercase until \E
\Q	quote (disable) metacharacters until \E
\E	End of lowercase/uppercase

```
# search for the C++ computer language:

/C++/        # wrong! regexp engine com-
plains about the plus signs
/C\+\+/      # this works
/C\Q++\E/    # this works too

# search for "bell" control characters, eg CTRL-G

/\cG/        # this is one way
/\007/       # this is another -- CTRL-G is octal 07
/\x07/       # here it is as a hex code
```

3.4. Working with multiline strings

Often, you will want to read a file several lines at a time. Consider, for example, a typical Unix fortune cookie file, which is used to generate quotes for the **fortune** command:

```
%
Let’s call it an accidental feature.
        -- Larry Wall
%
```

```
Linux: the choice of a GNU generation
%
When you say "I wrote a program that crashed Windows", peo-
ple just stare at
you blankly and say "Hey, I got those with the sys-
tem, *for free*".
        -- Linus Torvalds
%
I don't know why, but first C pro-
grams tend to look a lot worse than
first programs in any other language (maybe except for for-
tran, but then
I suspect all fortran programs look like 'firsts')
        -- Olaf Kirch
%
All language designers are arrogant. Goes with the territory...
        -- Larry Wall
%
We all know Linux is great... it does infi-
nite loops in 5 seconds.
        -- Linus Torvalds
%
Some people have told me they don't think a fat penguin re-
ally embodies the
grace of Linux, which just tells me they have never seen a an-
gry penguin
charging at them in ex-
cess of 100mph.  They'd be a lot more careful
about what they say if they had.
        -- Linus Torvalds, announcing Linux v2.0
%
```

The fortune cookies are separated by a line which contains nothing but a percent sign.

To read this file one item at a time, we would need to set the delimiter to something other than the usual `\n` - in this case, we'd need to set it to something like `\n%\n`.

To do this in Perl, we use the special variable `$/`.

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

Conveniently enough, setting `$/` to `""` will cause input to occur in "paragraph mode", in which two or more consecutive newlines will be treated as the delimiter. Undefining `$/` will cause the entire file to be slurped in.

```
undef $/;
$_ = <FH>; # whole file now here
```

Special variables are covered in Chapter 2 of the Camel book, from page 127 onwards. We're going to be looking at more special variables soon, so mark the page now. The information can also be found in **perldoc perlvar**.

Since `$/` isn't the easiest name to remember, we can use a longer name by using the **English** module:

```
use English;

$INPUT_RECORD_SEPARATOR = "\n%\n";      # long name for $/
$RS = "\n%\n";                          # same thing, awk-like
```

The **English** module is documented on page 403 of the Camel or in **perldoc English**.

3.4.1. Exercises

1. In your directory is a file called `exercises/linux.txt` which is a set of Linux-related fortunes, formatted as in the above example. Use multiline regular expressions to find only those quotes which were uttered by Larry Wall. (Answer: `exercises/answers/larry.pl`)

3.4.2. Regexp modifiers for multiline data

The `/s` and `/m` modifiers can be used to treat the string you're matching against as

either a single or multiple lines. In single line mode, ^ will match only at the start of the entire string, and $ will match only at the end of the entire string. In multiline mode, they will match at embedded newlines as well.

```
my $string = qq(
This is some text
and some more text
spanning several lines
);

if ($string =~ /^and some/m) {                  # this will match
        print "Matched in multiline mode\n";
}

if ($string =~ /^and some/s) {                  # this won't match
        print "Matched in single line mode\n";
}
```

In single line mode, the dot metacharacter will match \n. In multiline mode, it won't.

The differences between default, single line, and multiline mode are set out very succinctly by Jeffrey Friedl in Mastering Regular Expressions (see the Bibliography at the back of these notes for details). The following table is paraphrased from the one on page 236 of that book.

His term "clean multiline mode" refers to a mode which is similar to multi-line, but which does not strip the newline character from the end of each line.

Table 3-2. Effects of single and multiline options

Mode	Specified with	^ matches...	$ matches...	Dot matches newline
default	neither /s nor /m	start of string	end of string	No
single-line	/s	start of string	end of string	Yes
multi-line	/m	start of line	end of line	No

Mode	Specified with	`^` matches...	`$` matches...	Dot matches newline
clean multi-line	both `/m` and `/s`	start of line	end of line	Yes

3.5. Backreferences

3.5.1. Special variables

There are several special variables related to regular expressions.

- `$&` is the matched text
- `$‘` is the unmatched text to the left of the matched text
- `$’` is the unmatched text to the right of the matched text
- `$1`, `$2`, `$3`, etc. The text matched by the 1st, 2nd, 3rd, etc sets of parentheses.

All these variables are modified when a match occurs, and can be used in any way that other scalar variables can be used.

```
# this...
my ($match) = m/^(\d+)/;
print $match;

# is equivalent to this:
m/^\d+/;
print $&;

# match the first three words...
m/^(\w+) (\w+) (\w+)/;
print "$1 $2 $3\n";
```

You can also use `$&` and other special variables in substitutions:

```
$string = "It was a dark and stormy night.";
$string =~ s/dark|wet|cold/very $&/;
```

If you want to use parentheses simply for grouping, and don't want them to set a `$1` style variable, you can use a special kind of *non-capturing* parentheses, which look like `(?: ... )`

```
# this only sets $1 - the first two sets of parentheses are non-
capturing
m/^(?:\w+) (?:\w+) (\w+)/;
```

The special variables `$1` and so on can be used in substitutions to include matched text in the replacement expression:

```
# swap first and second words
s/^(\w+) (\w+)/$2 $1/;
```

However, this is no use in a simple match pattern, because `$1` and friends aren't set until after the match is complete. Something like:

```
my $word = "this";
print if m/($word) $1/;
```

... will *not* match "this this". Rather, it will match "this" followed by whatever `$1` was set to by an earlier match.

In order to match "this this" we need to use the special regular expression metacharacters `\1`, `\2`, etc. These metacharacters refer to parenthesized parts of a match pattern, just as `$1` does, but *within the same match* rather than referring back to the previous match.

```
my $word = "this";
print if m/($word) \1/;
```

3.5.2. Exercises

1. Write a script which swaps the first and the last words on each line (Answer: `exercises/answers/firstlast.pl`)
2. Write a script which looks for doubled terms such as "bang bang" or "quack quack" and prints out all occurrences. This script could be used for finding typographic errors in text. (Answer: `exercises/answers/double.pl`)

3.5.3. Advanced

1. Modify the above script to work across line boundaries (Answer: `exercises/answers/multiline_double.pl`)
2. What about case sensitivity?

3.6. Section summary

- Input data can be split into multiline strings using the special variable `$/`, also known as `$INPUT_RECORD_SEPARATOR`.
- The `/s` and `/m` modifiers can be used to treat multiline data as if it were a single line or multiple lines, respectively. This affects the matching of `^` and `$`, as well as whether or not `.` will match a newline.
- The special variables `$&`, `$‘` and `$’` are always set when a successful match occurs
- `$1`, `$2`, `$3` etc are set after a successful match to the text matched by the first, second, third, etc sets of parentheses in the regular expression. These should only be

used *outside* the regular expression itself, as they will not be set until the match has been successful.

- Special non-capturing parentheses `(?:...)` can be used for grouping when you don't wish to set one of the numbered special variables.
- Special metacharacters such as `\1`, `\2` etc may be used *within* the regular expression itself, to refer to text previously matched.

Chapter 4. More functions

4.1. In this chapter...

In this chapter, we discuss some more advanced Perl functions.

4.2. The grep() function

The `grep()` function is used to search a list for elements which match a certain regexp pattern. It takes two arguments - a pattern and a list - and returns a list of the elements which match the pattern.

The `grep()` function is on page 178 of your Camel book.

```
# trivially check for valid email addresses
my @valid_email_addresses = grep /\@/, @email_addresses;
```

The `grep()` function temporarily assigns each element of the list to `$_` then performs matches on it.

There are many more complicated uses for the grep function. For instance, instead of a pattern you can supply an entire block which is to be used to process the elements of the list.

```
my @long_words = grep { (length($_) > 8); } @words;
```

`grep()` doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression. Have a look at the documentation for this function to see how this is described.

4.2.1. Exercises

1. Use `grep()` to return a list of elements which contain numbers (Answer: `exercises/answers/grepnumber.pl`)
2. Use `grep()` to return a list of elements which are
   a. keys to a hash (Answer: `exercises/answers/grepkeys.pl`)
   b. readable files (Answer: `exercises/answers/grepfiles.pl`)

4.3. The map() function

The `map()` function can be used to perform an action on each member of a list and return the results as a list.

```
my @lowercase = map lc, @words;
my @doubled = map { $_ * 2 } @numbers;
```

`map()` is often a quicker way to achieve what would otherwise be done by iterating through the list with `foreach`.

```
foreach (@words) {
        push (@lowercase, lc($_);
}
```

Like `grep()`, it doesn't require a comma between its arguments if you are using a block as the first argument, but does require one if you're just using an expression.

4.3.1. Exercises

1. Create an array of numbers. Use `map()` to find the square of each number. Print out the results.

4.4. Chapter summary

- The `grep()` function can be used to find items in a list which match a certain regular expression
- The `map()` function can be used to perform an operation on each member of a list.

Chapter 5. System interaction

5.1. In this section...

In this section, we look at different ways to interact with the operating system. In particular, we examine the `system()` function, and the backtick command execution operator. We also look at security and platform-independence issues related to the use of these commands in Perl.

5.2. system() and exec()

The `system()` and `exec()` functions both execute system commands.

`system()` forks, executes the commands given in its arguments, waits for them to return, then allows your Perl script to continue. `exec()` does not fork, and exits when it's done. `system()` is by far the more commonly used.

```
% perl -we 'system("/bin/true");  print "Foo\n";'
Foo

% perl -we 'exec("/bin/true");  print "Foo\n";'
Statement unlikely to be reached at -e line 1.
(Maybe you meant system() when you said exec()?)
```

If the system command fails, the error message will be available via the special variable `$!`.

```
% perl -e 'system("cat non-existant-file") || die "$!";'
cat: non-existant-file: No such file or directory
```

5.2.1. Exercises

1. Write a script to ask the user for a username on the system, then perform the **finger** command to see information about that user. (Answer: `exercises/answers/finger.pl`)

5.3. Using backticks

Single quotes can be used to specify a literal string which can be printed, assigned to a variable, et cetera. Double quotes perform interpolation of variables and certain escape sequences such as `\n` to create a string which can also be printed, assigned, etc.

A new set of quotes, called *backticks*, can be used to interpolate variables then run the resultant string as a shell command. The output of that command can then be printed, assigned, and so forth.

Backticks are the backwards-apostrophe character (`‘` which appears below the tilde (~), next to the number `1` on most keyboards.

Just as the `q()` and `qq()` functions can be used to emulate single and double quotes and save you from having to escape quotemarks that appear within a string, the equivalent function `qx()` can be used to emulate backticks.

Backticks and the `qx()` function are discussed in the Camel on pages 52 and 41 respectively or in **perldoc perlop**.

5.3.1. Exercises

1. Modify your earlier finger program to use backticks instead of `system()` (Answer: `exercises/answers/backtickfinger.pl`)
2. Change it to use `qx()` instead (Answer: `exercises/answers/qxfinger.pl`)

3. The Unix command **whoami** gives your username. Since most shells support backticks, you can type **finger ‘whoami‘** to finger yourself. Use shell backticks inside your `qx()` statement to do this from within your Perl program. (Answer: `exercises/answers/qxfinger2.pl`)

5.4. Platform dependency issues

Note that the examples given above will not work consistently on all operating systems. In particular, the use of `system()` calls or backticks with Unix-specific commands will not work under Windows NT, MacOS, etc. Slightly less obviously, the use of backticks on NT can sometimes fail when the output of a command is sent explicitly to the screen rather than being returned by the backtick operation.

5.5. Security considerations

Many of the examples given above can result in major security risks if the commands executed are based on user input. Consider the example of a simple finger program which asked the user who they wanted to finger:

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

use strict;

print "Who do you want to finger? ";
my $username = <STDIN>;
print ‘finger $username‘;
```

Imagine if the user’s input had been `skud; cat /etc/passwd`, or worse yet, `skud; rm -rf /`. The system would perform both commands as though they had been entered into the shell one after the other.

Luckily, Perl’s `-T` flag can be used to check for unsafe user inputs.

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

Documentation for this can be found by reading the **perldoc perlsec**, or on page 356 of the Camel.

`-T` stands for "taint checking". Data input by the user is considered "tainted" and until it has been modified by the script, may not be used to perform shell commands or system interactions of any kind. This includes system interactions such as `open()`, `chmod()`, and any other built-in Perl function which interacts with the operating system.

The only thing that will clear tainting is referencing substrings from a regexp match. The `perlsec` online documentation contains a simple example of how to do this. Read it now, and use it to complete the following exercises.

Note that you'll also have to explicitly set `$ENV{'PATH'}` to something safe (like `/bin`) as well.

5.5.1. Exercises

1. Modify the finger program above to perform taint checking (Answer: `exercises/answers/taintfinger.pl`)
2. Take one of your scripts using `open()` or `opendir()` and modify it to accept a filename as user input. Turn taint checking on. What sort of regular expression could you use to check for valid filenames? (Answer: `exercises/answers/taintfile.pl`)

5.6. Section summary

- The `system()` function can be used to perform system commands. `$!` is set if any error occurs.

- The backtick operator can be used to perform a system command and return the output. The `qx()` quoting function/operator works similarly to backticks.
- The above methods may not result in platform independent code.
- Data input by users or from elsewhere on the system can cause security problems. Perl's `-T` flag can be used to check for such "tainted" data
- Tainted data can only be untainted by referencing a substring from a pattern match.

Chapter 6. References and complex data structures

6.1. In this section...

In this section, we look at Perl's powerful reference syntax and how it can be used to implement complex data structures such as multi-dimensional lists, hashes of hashes, and more.

6.2. Assumed knowledge

For this section, it is assumed that you have a good understanding of Perl's data types: scalars, arrays, and hashes. Prior experience with languages which use pointers or references is helpful, but not required.

6.3. Introduction to references

Perl's basic data type is the *scalar*. Arrays and hashes are made up of scalars, in one- or two-dimensional lists. It is not possible for an array or hash to be a member of another array or hash under normal circumstances.

However, there is one thing about an array or hash which is scalar in nature -- its memory address. This memory address can be used as an item in an array or list, and the data extracted by looking at what's stored at that address. This is what a reference is.

References are covered in detail in chapter 4 of the Camel book, and in **perldoc perlref**. Chapter 1 of Advanced Perl Programming (O'Reilly's Panther book) contains a very good explanation of references. Lastly, Tom Christiansen's FMTEYEWTK (Far

More Than You Ever Wanted To Know) tutorials contain information about references. They're available from the Perl website (http://www.perl.com/)

6.4. Uses for references

There are three main uses for Perl references.

6.4.1. Creating complex data structures

Perl references can be used to create complex data structures, for instance hashes of arrays, arrays of hashes, hashes of hashes, and more.

6.4.2. Passing arrays and hashes to subroutines and functions

Since all arguments to subroutines are flattened to a list of scalars, it is not possible to use two arrays as arguments and have them retain their individual identities.

```
my @a1 = qw(a b c);
my @a2 = qw(d e f);

printargs(@a1, @a2);

sub printargs {
        print "@_\n";
}
```

The above example will print out `a b c d e f`.

References can be used in these circumstances to keep arrays and hashes passed as arguments separate.

6.4.3. Object oriented Perl

References are used extensively in object oriented Perl. In fact, Perl objects *are* references to data structures.

6.5. Creating and dereferencing references

To create a reference to a scalar, array or hash, we prefix its name with a backslash:

```
my $scalar = "This is a scalar";
my @array  = qw(a b c);
my %hash = (
        'sky'           =>      'blue',
        'apple'         =>      'red',
        'grass'         =>      'green'
);

my $scalar_ref = \$scalar;
my $array_ref  = \@array;
my $hash_ref   = \%hash;
```

Note that all references are scalars, because they contain a single item of information: the memory address of the actual data.

This is what a reference looks like if you print it out:

```
% perl -e 'my $foo_ref = \$foo; print "$foo_ref\n";'
SCALAR(0x80c697c)
% perl -e 'my $bar_ref = \@bar; print "$bar_ref\n";'
ARRAY(0x80c6988)
% perl -e 'my $baz_ref = \%baz; print "$baz_ref\n";'
HASH(0x80c6988)
```

You can find out whether a scalar is a reference or not by using the `ref()` function, which returns a string indicating the type of reference, or undef if a scalar is not a reference..

The `ref()` function is documented on page 204 of the Camel book or in **perldoc -f ref**.

Dereferencing (getting at the actual data that a reference points to) is achieved by prepending the appropriate variable-type punctuation to the name of the reference. For instance, if we have a hash reference `$hash_reference` we can dereference it by looking for `%$hash_reference`

```
my $new_scalar = $$scalar_ref;
my @new_array  = @$array_ref;
my %new_hash   = %$hash_ref;
```

In other words, wherever you would normally put a variable name (like `new_scalar`) you can put a reference variable (like `$scalar_ref`).

Here's how you access array elements or slices, and hash elements:

```
print $$array_ref[0];           # prints the first ele-
ment of the array
                                # referenced by $array_ref
print @$array_ref[0..2];        # prints an array slice
print $$hash_ref{'sky'};        # prints a hash element's value
```

The other way to access the value that a reference points to is to use the "arrow" notation. This notation is usually considered to be better Perl style than the one shown above, which can have precedence problems and is less visually clean.

```
print $array_ref->[0];
print $hash_ref->{'sky'};
```

The above arrow notation is best explained in the book Advanced Perl Programming. Your instructor should have a copy if you are interested.

6.6. Passing multiple arrays/hashes as arguments

If we were attempt to pass two arrays together to a subroutine, they would be flattened out to form one large array.

```
my @fruits  = qw(apple orange pear banana);
my @rodents = qw(mouse rat hamster gerbil rabbit);
my @books   = qw(camel llama panther sheep);

mylist(@fruit, @rodents);

# print out all the fruits and then all the rodents
sub mylist {
        my @list = @_;
        foreach (@list) {
                print "$_\n";
        }
}
```

If we want to keep them separate, we need to pass the arrays by references:

```
myreflist(\@fruit, \@rodents);

sub myreflist {
        my ($firstref, $secondref) = @_;
        print "First list:\n";
        foreach (@$firstref) {
                print "$_\n";
        }
        print "Second list:\n";
        foreach (@$secondref) {
                print "$_\n";
        }
}
```

6.7. Complex data structures

References are most often used to create complex data structures. Since hashes and arrays only accept scalars as elements, references (which are inherently scalars) can be used to create arrays of arrays or hashes, and hashes of arrays or hashes.

```
my %categories = (
        'fruits'        =>      \@fruits,
        'rodents'       =>      \@rodents,
        'books'         =>      \@books,
);

# to print out "gerbil"...
print $categories{'rodents'}->[3];
```

6.8. Anonymous data structures

We can use anonymous data structures to create complex data structures, to avoid having to declare many temporary variables. Anonymous arrays are created by using square brackets instead of round ones. Anonymous hashes use curly brackets instead of round ones.

```
# the old two-step way:
my @array = qw(a b c d);
my $array_ref = \@array;

# if we get rid of $array_ref, @array will still hang round us-
ing up
# memory.  Here's how we do it without the intermediate step, by
# creating an anonymous array:

my $array_ref = ['a', 'b', 'c', 'd'];

# look, we can still use qw() too...
```

```
my $array_ref = [qw(a b c d)];

# more useful yet, we can put these anon ar-
rays straight into a hash:

my %transport = (
        'cars'          =>   [qw(toyota ford holden porsche)],
        'planes'        =>   [qw(boeing harrier)],
        'boats'         =>   [qw(clipper skiff dinghy)],
);
```

The same technique can be used to create anonymous hashes:

```
# The old, two-step way:

my $hash = (
a => 1,
b => 2,
c => 3
);
my $hash_ref = \$hash;

# the quicker way, with an anonymous hash:
my $hash_ref = {
a => 1,
b => 2,
c => 3
};
```

6.9. Exercises

1. Create a complex data structure as follows:

a. Create a hash called `%pizza_prices` which contains prices for small, medium and large pizzas.

b. Create a hash called `%pasta_prices` which contains prices for small, medium and large serves of pasta.

c. Create a hash called `%milkshake_prices` which contains prices for small, medium and large milkshakes.

d. Create a hash containing references to the above hashes, so that given a type of food and a size you can find the price of it.

e. Convert the above hash to use anonymous data structures instead of the original three pizza, pasta and milkshake hashes

f. Add a new element to your hash which contains the prices of salads

(Answer: `exercises/answers/food.pl`)

2. Create a subroutine which can be passed a scalar and a hash reference. Check whether there is an element in the hash which has the scalar as its key. Hint: use `exists` for this. (Answer: `exercises/answers/exists.pl`)

6.10. Section summary

- References are scalar data consisting of the memory address of a piece of Perl data, and can be used in arrays, hashes, etc wherever you would use a normal scalar
- References can be used to create complex data structures, to pass multiple arrays or hashes to subroutines, and in object-oriented Perl.
- References are created by prepending a backslash to a variable name.
- References are dereferenced by replacing the name part of a variable name (eg `foo` in `$foo`) with a reference, for example replace `foo` with `$foo_ref` to get `$$foo_ref`

- References to arrays and hashes can also be dereferenced using the arrow `->` notation.
- References can be passed to subroutines as if they were scalars.
- References can be included in arrays or hashes as if they were scalars.
- Anonymous arrays can be made by using square brackets instead of round; anonymous hashes can be made by using curly brackets instead of round. These can be assigned directly to a reference, without any intermediate step.

Chapter 7. Conclusion

7.1. What you've learnt

Now you've completed Netizen's Intermediate Perl module, you should be confident in your knowledge of the following fields:

- File I/O, including opening files and directories, opening pipes, finding information about files, recursing down directories, file locking, and handling binary data
- How to use advanced regular expression techniques such as multiline matching and backreferences
- The use of various Perl functions
- System interaction, including: system calls, the backtick operator, interacting with the file system, dealing with users and groups, dealing with processes, network communications, and security considerations
- Advanced Perl data structures and references

7.2. Where to now?

To further extend your knowledge of Perl, you may like to:

- Borrow or purchase the books listed in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"
- Install Perl on your home or work computer
- Practice using Perl from day to day
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)

- Extend your knowledge with further Netizen courses such as:
  - CGI Programming in Perl
  - Web enabled databases with Perl and DBI

  Information about these courses can be found on Netizen's website (http://netizen.com.au/services/training/). A diagram of Netizen's courses and the careers they can lead to is included with these training materials.

7.3. Further reading

7.3.1. Books

- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O'Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 1997. ISBN 1-56592-257-3.
- Joseph N. Hall and Randal L. Schwartz*Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.

7.3.2. Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)

- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup

Appendix A. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

Table A-1. Simple Unix commands

Action	Command
Change to home directory	**cd**
Change to *directory*	**cd *directory***
Change to directory above current directory	**cd ..**
Show current directory	**pwd**
Directory listing	**ls**
Wide directory listing, showing hidden files	**ls -al**
Showing file permissions	**ls -al**
Making a file executable	**chmod +x *filename***
Printing a long file a screenful at a time	**more *filename* or less *filename***
Getting help for *command*	**man *command***

Appendix B. Editor cheat sheet

This summary is laid out as follows:

Table B-1. Layout of editor cheat sheets

Running	Recommended command line for starting it.
Using	Really basic howto. This is not even an attempt at a detailed howto.
Exiting	How to quit.
Gotchas	Oddities to watch for.

B.1. vi

B.1.1. Running

```
% vi filename
```

B.1.2. Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.

- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

B.1.3. Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

B.1.4. Gotchas

vi has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

B.1.5. Help

`:help` **ENTER** might work. If not, then see the manpage.

B.2. pico

B.2.1. Running

```
% pico -w filename
```

B.2.2. Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has `^X` commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL**-**W** to search for text.
- **CTRL**-**O**to save.

B.2.3. Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL**-**X** from the main menu.

B.2.4. Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar. \\ \hline

B.2.5. Help

CTRL-G from the main menu, or just read the menu bar.

B.3. joe

B.3.1. Running

```
% joe filename
```

B.3.2. Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

B.3.3. Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

B.3.4. Gotchas

Nothing in particular.

B.3.5. Help

CTRL-K then **H**.

B.4. jed

B.4.1. Running

```
% jed
```

B.4.2. Using

- Defaults to the emacs emulation mode.
- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-X** then **S** to save.

B.4.3. Exiting

CTRL-X then **CTRL-C** to exit.

B.4.4. Gotchas

Nothing in particular.

B.4.5. Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

Appendix C. ASCII Pronunciation Guide

Table C-1. ASCII Pronunciation Guide

Character	Pronunciation
!	bang, exlamation
*	star, asterisk
$	dollar
@	at
%	percent
&	ampersand
"	double-quote
'	single-quote, tick
( )	open/close bracket, parentheses
<	less than
>	greater than
-	dash, hyphen
.	dot
,	comma
/	slash, forward-slash
\	backslash, slosh
:	colon
;	semi-colon
=	equals
?	question-mark
^	caret (pron. carrot)
_	underscore

Character	Pronunciation
[ ]	open/close square bracket
{ }	open/close curly brackets, open/close brace
|	pipe, or vertical bar
~	tilde (pron. “til-duh”, wiggle, squiggle)
‘	backtick