Perl for System Administration

Jacinta Richardson
Paul Fenwick

Perl for System Administration

by Jacinta Richardson and Paul Fenwick

Conventions used throughout this text are based upon the conventions used in the Netizen training manuals by Kirrily Robert, and found at http://sourceforge.net/projects/spork

This training manual is maintained by Perl Training Australia, and can be found at http://www.perltraining.com.au/notes.html.

This is revision 1.2 of Perl Training Australia’s "Perl for System Administrators" training manual.

Table of Contents

List of Tables

Chapter 1. About Perl Training Australia

Training

Perl Training Australia (http://www.perltraining.com.au) offers quality training in all aspects of the Perl programming language. We operate throughout Australia and the Asia-Pacific region. Our trainers are active Perl developers who take a personal interest in Perl's growth and improvement. Our trainers can regularly be found frequenting online communities such as Perl Monks (http://www.perlmonks.org/) and answering questions and providing feedback for Perl users of all experience levels.

Our primary trainer, Paul Fenwick, is a leading Perl expert in Australia and believes in making Perl a fun language to learn and use. Paul Fenwick has been working with Perl for over 10 years, and is an active developer who has written articles for *The Perl Journal* and other publications.

Consulting

In addition to our training courses, Perl Training Australia also offers a variety of consulting services. We cover all stages of the software development life cycle, from requirements analysis to testing and maintenance.

Our expert consultants are both flexible and reliable, and are available to help meet your needs, however large or small. Our expertise ranges beyond that of just Perl, and includes Unix system administration, security auditing, database design, and of course software development.

Contact us

If you have any project development needs or wish to learn to use Perl to take advantage of its quick development time, fast performance and amazing versatility; don't hesitate to contact us.

Table 1-1. Perl Training Australia's contact details

Phone:	+61 3 9354 6001
Fax:	+61 3 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058 AUSTRALIA

Chapter 2. Introduction

Welcome to Perl Training Australia’s *Perl for System Administration*. This is a one-day module in which we will cover system administration users for Perl.

Course outline

- Brief introduction to Perl.
- Filesystem analysis and traversal.
- Mail processing and filtering.
- Privilege and security considerations.
- Logfile processing and monitoring.
- System interaction, wrappers, and process manipulation.
- Interacting with network services.

Assumed knowledge

This training module assumes the following prior knowledge and skills:

- Previous programming experience.
- Thorough understanding of operators and functions, conditional constructs, subroutines and basic regular expressions concepts.

Module objectives

- Some objectives

Platform and version details

Perl is a cross-platform computer language which runs successfully on approximately 30 different operating systems. However, as each operating system is different this does occasionally impact on the code you write. Most of what you will learn will work equally well on all operating systems; your instructor will inform you throughout the course of any areas which differ.

All Perl Training Australia’s Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs significantly 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.

At the time of writing, the most recent stable release of Perl is version 5.8.8, however older versions of Perl 5 are still common. Your instructor will inform you of any features which may not exist in older versions.

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 appears 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 appear *`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.

Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Other materials

In addition to these notes, it is highly recommend that you obtain a copy of Programming Perl (2nd or 3rd edition) by Larry Wall, et al., more commonly referred to as "the Camel book". While these notes have been developed to be useful in their own right, the Camel book covers an extensive range of topics not covered in this course, and discusses the concepts covered in these notes in much more detail. The Camel Book is considered to be the definitive reference book for the Perl programming language.

The page references in these notes refer to the *3rd edition* of the camel book. References to the 2nd edition will be shown in parentheses.

Chapter 3. Why use Perl for System Administration?

For years, Perl has been the scripting language of choice for many system administrators. There are many factors which have influenced this choice. Some of these are:

- Excellent text manipulation capabilities. Perl excels at manipulating log files and other regular data. This makes it easy to automate much of the general house keeping associated with system maintenance. It also makes it easy to extract data and trends from different kinds of application log files.
- CPAN. The Comprehensive Perl Archive Network, gives Perl almost infinite extensibility, full database connectivity and Unicode support. There are literally thousands of third party modules to solve all sorts of different problems. If you have a task to fulfil then chances are reasonable that someone else has already done some of it for you.
- DBI. Perl's Database interface supports a wide range of third party databases. Further it presents a consistent interface for each. Using this module simplifies the management of disparate database platforms.
- Portability. Perl exists on more than 30 different operating systems. This allows well written code to be developed on one platform and deployed across many, simplifying automation tasks.
- Speed. Perl is fast to write and fast to run, making it perfect for small once-off tasks. Yet Perl is also great for large projects with support for full test coverage, documentation and modules.
- Documentation. Perl has extensive documentation freely available. This is one of Perl's biggest assets. Every built in function comes with a full description and many with usage examples. Perl's modules also come with extensive documentation as well as test suites and example code.
- Familiarity. Much of what can be done in bash, sed, awk and C can be transferred almost directly into Perl code. Likewise the format of many functions are equivalent to common Unix commands.
- Low-level access. As well as allowing access to high-level functionality, Perl makes it easy to work directly with hardware, sockets and to fulfil other low-level requirements.
- Freedom. Perl is licenced under both the Artistic license and the GNU Public License and is freely available.

Chapter 4. Perl Basics

In this chapter...

This chapter aims to provide a quick tour of Perl's basics. You can skip much of this material if you already know Perl.

The concepts in this chapter are used extensively throughout the rest of these notes, and this information is intended for quick reference rather than in-depth analysis.

For a greater discussion on these concepts, refer to Perl Training Australia's *Programming Perl* course notes (available online at http://perltraining.com.au/notes.html), or *Programming Perl, 3rd Ed* by Larry Wall et al (commonly referred to as the *Camel Book*).

Important basics

Help

Perl comes with a very detailed help system called `perldoc`. This is installed on most systems, and works similarly to the Unix `man`. Useful pages are listed below.

```
perldoc perldoc                # Instructions on using perldoc

perldoc perltoc                # Perl table of contents

perldoc perl                   # Overview of Perl

perldoc perlfunc               # Full list of Perl functions
perldoc -f <function_name>     # Help with a specific function

perldoc perlop                 # Full list of Perl operators

perldoc perlmodlib             # List of modules installed with Perl
perldoc perllocal              # List of locally installed modules

perldoc <module_name>          # Documentation for specific module
```

Shebang line

All Perl programs should start with a shebang line. On Unix and Unix-like operating systems, this line should specify where to find Perl. For example:

```
#!/usr/bin/perl
```

On Microsoft Win32, and other systems which rely on other data to determine where to find the interpretor this can be shortened to:

```
#!perl
```

It is a good practice, regardless of your operating system, to include the full Unix path, as this makes your programs more portable between systems.

Strictures and warnings

Perl comes with two great programming aids; strictures and warnings. We strongly recommend you turn these on and leave them on for every program you write.

```
#!/usr/bin/perl -w
use strict;
```

Or alternately (versions of Perl 5.6.0 and above):

```
#!/usr/bin/perl
use strict;
use warnings;
```

Strict

Strict ensures that you pre-declare your variables, don't use symbolic references and don't have barewords. Pre-declaring your variables is just a matter of preceding the variable name with a scoping keyword (such as `my`) the first time you use it. It saves you from making accidental spelling mistakes:

```
# without strict;
$num_of_freinds = 5;    # Oops, poor spelling!

print "I have $num_of_friends friends\n";
```

With strict, compilation of your program would die with an error:

```
Global symbol "$num_of_friends" requires explicit package name
```

telling you that Perl has never seen the `$num_of_friends` variable before.

Symbolic references are only really needed for very advanced operations in Perl; for everything else the same job can be done faster and more cleanly using a *hash*. As such, we will not mention symbolic references further in this course, except to say that you don't want to use them by mistake.

Barewords are words in your programs with no identifying characteristics. For each case of a bareword, Perl has to *guess* at run-time whether it's a string, or a call to a subroutine, and this can introduce bugs if Perl guesses differently to what you intended. Since it's trivial to be clear on this distinction, you will never need to use barewords either.

Warnings

Warnings turns on helpful advice to let you know that Perl thinks you've probably done something wrong. These warnings aren't necessarily show-stoppers, but if you're getting them, it's worth spending some time wondering why. A few things that trigger warnings are:

- Trying to read from or write to an unopened filehandle, socket or device.
- Treating a string of non-numeric characters as if it were a number.
- Printing or performing comparisons with undefined values.
- Assigning an odd number of elements to a hash (collection of key-value pairs).

Comments

Comments are wonderful things which help future maintainers, including yourself in 6 months time, decipher your code. These should be liberally spread through your code.

To start a comment just add a `#`. Your comment will then last until the end of line:

```
# This comment takes the whole line

print "Hello World!";  # This comment starts part way through
```

It's a good idea to include a comment at the top of your code saying what it does, and who wrote it. This allows the future maintainer of your code contact you, and tell you how grateful they are that you provided such good comments. It's also recommended you include the date (at least a month and year) when you wrote the code.

Starting your program

Each of your programs should start with:

```
#!/usr/bin/perl -w
# This program....
# Author:  Your Name <you@some.address.somewhere>
# Date:    Month Year
use strict;
```

Variables

There are two rules on user-defined variable names. They are:

- Variable names may only consist of alphabet, numerical and the underscore (_) characters.
- Variable names must start with an alphabet character.

There are variables whose names do not conform to these rules, however they are *Special* variables. We'll cover them later.

Perl has three basic variable types, and each is preceded by a punctuation character known as a *sigil*. The variables and sigils are scalars (`$`), arrays (`@`), and hashes (`%`).

Scalars

Perl's fundamental type is the scalar. A scalar contains a *single* piece of information; such as a number, a character, a string, a filehandle, or a reference (pointer). The sigil for a scalar variable is the dollar (`$`). A mnemonic for this is the `$` looks a bit like an `S` for single or scalar.

```
my $name   = "Perl Training Australia";
my $number = 123;
my $float  = 234.54;
my $char   = "a";
```

Unlike strictly typed programming languages (such as C and Java), Perl does not care what kind of value you're putting in a scalar. If you treat a scalar containing a number as a string, Perl will turn it into a string. If you treat a scalar containing a string as a number, Perl will try to turn it into a number. Adding integers and floating point numbers results in a floating point result. If you want to coerce it back into an integer, that's possible too. If you assign a string to a variable which was previously a filehandle, Perl doesn't mind.

```
my $new_num = $number + $float;   # 357.54
my $silly   = $number + $name;    # 123 (and a warning)

print $silly . $char;             # prints "123a"
```

Further, Perl sets no limit on the length of your strings, or the size of your numbers. However, limits may still exist due to environmental influences such as machine precision and memory availability. There is no need to tell Perl how long your string will be.

Quotes and interpolation

Perl has two sets of quote that are used for delimiting strings. Double quotes (`"`) and single quotes (`'`). In many cases in your program these can be used interchangeably:

```
my $name = 'Perl Training Australia';
my $home = "Melbourne";
```

However there is one difference. Double quotes `interpolate` while single quotes do not. Interpolation allows us to add variables within a set of double quotes and have those variables be replaced with their contents. For example:

```
print "I work at $name";  # prints "I work at Perl Training Australia"

print 'I work at $name';  # prints "I work at $name"
```

Control characters such as `\n` for newline, `\t` for tab and `\b` for bell can also be interpolated within double quotes. These are merely treated as pairs of characters within single quotes.

To *escape* characters within quotes, to remove any special interpolative meanings, use the backslash (`\`) character. To escape a backslash use two: `\\`.

```
print "He said \"Hi Sally"";
print 'It is Tim\'s sandwich';
```

Perl also allows the programmer to pick their own quotes, by using the `q` (single-quotes) and `qq` (double-quotes) operators. The following are equivalent to the two lines above:

```
print qq{He said "Hi Sally"};
print q{It is Tim's sandwich};
```

Arrays

An array is an ordered list of scalars. Arrays can contain any number of scalars (again within memory and other machine constraints), and there are no restrictions on what those scalars may contain. The sigil for an array is an at-sign (`@`). A mnemonic for this is that `@` looks like `a` for array or all.

```
my @numbers = (1, 2, 3, 4, 5);
my @friends = ("Jane", "Bob", "Alice", "Eve");
my @mixed   = (1, "Jane", 4, "Jacob", 7, 12.12);
my @info    = ($name, $home);
```

Array indexes start at 0. So `@numbers` has the indexes 0 through to 4.

Array lookups

To look up a single element in an array we do the following:

```
print $friends[3];      # prints "Eve"
```

notice that we use a `$` sign here rather than an `@` sign. This is because we're getting a *single* thing from the array: a *scalar*.

Changing array elements

To change an element in the array we use the same syntax:

```
$numbers[3] = 20;       # @numbers is now (1, 2, 3, 20, 5)
```

Adding array elements

Adding an element to the array is the same as changing an element, except in this case, the previous value was empty.

```
$mixed[5] = "Ben";      # (1, "Jane", 4, "Jacob", 7, 12.12, "Ben");
```

A *better* way of doing this is to `push` the value on to the end of the array, as this saves us having to know what index value we are up to.

```
push @mixed, "Joe";     # (1, "Jane", 4, "Jacob", 7, 12.12, "Ben", "Joe");
```

Counting backwards

We can also count backwards through our array. `-1` represents the last element, `-2` the second last, `-3` the third last and so on. Thus:

```
print $numbers[-2];     # prints "20"
```

Last index

To find the last index of an array we use a strange looking notation as follows:

```
my @friends = ("Jane", "Bob", "Alice", "Eve");
print $#numbers;                # prints "3" (last index)
```

unfortunately it's easy to swap the `$` and `#`, resulting in:

```
print #$numbers;                # Whoops!
```

which comments out `$numbers` so that print has to look for its arguments on the next line of code. More often than not, we actually want the *length* of the array, rather than the last index.

Array length

There is one inherently *scalar* piece of information for an array, and that is its length. Since Perl does it's best to *do what I mean (dwim)*, treating an array like a scalar will return its length.

```
my $length = @friends;   # length is 4
```

Interpolation

As a convenience, Perl allows us to interpolate arrays into strings in the same way we do scalars:

```
print "The lucky numbers are @numbers";
```

In this case, each element of the array is joined together, separated with single spaces.

Hashes

A hash is an unordered mapping of key-value pairs. Every key and value must be a scalar. Hashes can contain any number of key-value pairs and, like arrays, there are no restrictions on the scalar contents, although the keys are always treated as strings.

To understand this mapping consider a telephone book. In the telephone book we have names (keys) which map to numbers (values). It is easy enough to find a telephone number given a name, but very time-consuming to find a name given a telephone number. Perl's hashes are the same.

Likewise it doesn't make sense for a telephone book to have multiple entries for the exact same name (and address) details. How would you know which number to call? Thus, hash keys must be unique.

The sigil for hashes is the percent (`%`). There's no good mnemonic for this one.

```
my %age_of = (
        Jane    => 23,
        Bob     => 63,
        Alice   => 38,
        Eve     => 47,
);

my %favourite_colour_of = (
        Jane    => "Blue",
        Bob     => "Brown",
        Alice   => "Green",
        Eve     => "Yellow",
);
```

The strange arrow `=>` is called the *fat comma*. It behaves like an ordinary comma except it's bigger (and therefore easy to see) and it automatically quotes the value to its left. Values on the right hand side, still need to be quoted.

Hash lookups

To look up a single element in a hash we do the following:

```
print $age_of{Jane};    # prints "23"
```

Again we use a `$` sign instead of a `%` sign. This is because we're getting a *single* thing from the hash: a *scalar*.

Changing hash values

To change a value in the hash we use the same syntax:

```
$age_of{Jane} = 24;
```

Adding hash pairs

Adding a key-value pair to the hash uses the same as changing a value, if the key was not previously in the hash, it will spring into existence.

```
$age_of{Donald} = 15;   # Donald is now in the hash.
```

Hash size

To find out how many pairs of keys and values we have, we have to use either the `keys` or `values` function. These return all of the keys and values respectively. Taking the result of either function in a scalar context returns us the result we want.

```
my $num_of_pairs = keys(%age_of);
```

Interpolation

There is no one obvious way to display hash data, so hashes do not interpolate in double quoted strings.

Special Variables

Perl has a number of special variables. The three that we will see most often in this course are are `$_`, `@ARGV` and `%ENV`.

$_

`$_` is at the same time the most used and least seen special variable. It is usually pronounced as *dollar underscore* but is sometimes referred to simply as *it*. Many of Perl's built-in functions take `$_` as their default argument. Such as `print`.

```
# prints $_;
print;
```

The usefulness of `$_` will become apparent as we explore many of the common input, output, and string-processing functions of Perl.

@ARGV

`@ARGV` is the array which stores all the command line arguments which the Perl program was called with. These may include filenames, switches, and other input.

%ENV

`%ENV` is a hash of your program's environment. The keys in this hash depend on your operating system. Changing values in this hash changes the environment for your program and any other processes it spawns. However, changes do not affect the parent process; in other words they are lost after your program has finished running.

Conditionals and truth

Perl's conditional structures should look pretty familiar to most programmers. However, before we start this section we should take a brief detour into what Perl views as true and false.

In fact, it's easier to look at what Perl views as false, because this is a very short list. Perl sees the following four things as false:

1. The undefined value.
2. The number zero: `0`.
3. The string of the single digit zero: `"0"` (or `'0'`).
4. The empty string: `""` (or `''`).

Everything else is true.

```
my $undefined;          # false
undef;                  # false
"0";                    # false
"";                     # false
0;                      # false
"apple";                # true
'banana';               # true
1;                      # true
-1;                     # true
"00";                   # true
my @array;              # false in scalar context (size 0)
@array = (1,2,3);       # now true in scalar context
```

Comparison operators

Perl has two flavours of comparison operators, strings and numbers.

```
$a < $b                 # Numerical less than
$a > $b                 # Numerical greater than
$a <= $b                # Numerical less than or equal
$a >= $b                # Numerical greater than or equal
$a == $b                # Numerical equality
#a != $b                # Numerical inequality

$a lt $b                # String less than
$a gt $b                # String greater than
$a le $b                # String less than or equal
$a ge $b                # String greater than or equal
$a eq $b                # String equality
$a ne $b                # String inequality
```

It's important to use the correct comparison operator for your intention.

```
"10" lt "9";            # true (1 comes before 9)
"00" == 0;              # true ("00" is 0 when treated as a number)
"3" == "3com";          # true (but generates a warning)
"3" eq "3com";          # false
```

Boolean operators

Perl has two flavours of boolean operators, C-like and English-like. The primary difference between them is one of precedence. English-like operators have almost the lowest precedence possible and are always evaluated last. C-like operators have the same precedence as they do in C. It is always possible to use parentheses to force the order of execution, and it is recommended that you do so if you feel any ambiguity exists.

For more information read `perldoc perlop`.

```
$a && $b                # AND:  True if $a and $b are true
$a and $b               #       As above.

$a || $b                # OR:  True if $a or $b is true (or both)
$a or $b                #      As above.

! $a                    # NOT: True if $a is false
not $a                  #      As above.

$a xor $b               # Exclusive-OR: True if either $a or $b
                        # is true, but not both.
```

if-elsif-else

Like most imperative languages, Perl has a fairly standard if-then-else structure:

```
if( <condition> ) {

}
elsif( <condition> ) {

}
else {

}
```

In Perl's case both the parentheses and the braces are required. The `elsif` and `else` blocks are optional. Multiple `elsif` blocks may appear after the `if` and before any `else`.

unless

Perl also has an `unless` construct. `unless` is the same as *if not*. For example the following two code snippets do the same thing.

```
if( not $I_have_apples ) {      unless( $I_have_apples ) {
        buy_apples();                   buy_apples();
}                               }

make_apple_pie();               make_apple_pie();
```

Trailing conditionals

Perl provides trailing conditional statements.

```
buy_apples() if not $I_have_apples;
```

```
buy_apples() unless $I_have_apples;
```

In this form the parentheses and curly braces are not required. However only a single statement may appear on the left.

Because the conditional appears on the right, trailing conditionals have the potential to reduce readability of your code. If the condition is important, you should always use the full form. Consider the following example:

```
launch_nuclear_missiles() if red_button_pushed();
```

For someone skimming down the left of the code, this can be quite disconcerting.

Looping constructs

Perl has two main looping constructs. `while` and `foreach`.

while

```
while( <condition> ) {

}
```

Just like Perl's `if` statement, the parentheses and braces are required.

`while` is typically used to iterate over input from the user or file and in cases where the number of iterations is either not known beforehand, or not relevant.

The following code echos back data passed in on STDIN:

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

This takes advantage of `$_` in two ways. `while( <STDIN> )` is a short-cut for:

```
while( defined( $_ = <STDIN> ) )
```

In fact, we can further reduce our above example to the following:

```
while( <> ) {

}
```

<> is a highly magical operator. First it checks `@ARGV` to see if there are arguments to use a filename. If there are, it will open each file in order, and iterate through each line. If `@ARGV` is empty, it checks for input on `STDIN`.

foreach

```
# using $_
foreach ( @array ) {

}
```

```
foreach my $value (@array) {

}
```

Again, parentheses and braces are required.

`foreach` is very handy for iterating over arrays and lists. In the first example, `$_` is set to each array element as we walk through. In the second example `$value` is set instead, and `$_` remains untouched.

In `foreach` loops the iterator (`$_` or `$value` in the above examples) *is* the element in the array. Thus the below code squares the values in the array:

```
foreach my $value (@array) {
        $value = $value*$value;
}
```

Subroutines

```
sub name {

}
```

Subroutines are user-written functions. They are compiled at the same time as the rest of your code, but do not get executed (regardless of where they appear in your program) until they are called.

```
# Call the buy_apples subroutine:
buy_apples();

# then later...

# The buy_apples subroutine
sub buy_apples {
        go_shopping();
        select_apples();
        pay();
}
```

Subroutines take one or more scalar arguments (remember that arrays and hashes can be treated as just lists of scalars), and can return one or more scalars. Arguments are stored in the `@_` array.

```
print second_arg( @array );

sub second_arg {
        my ($first, $second) = @_;

        return $second;
}
```

```
print first_last( @array );

sub first_last {
        my $first = shift @_;
        my $last = pop @_;

        return ($first, $last);
}
```

Passing hashes and arrays into subroutines causes them to lose their identity.

```
if( greater_length( @array1, @array2 ) ) {
        # ...
}

sub greater_length {
        my ( @array1, @array2 ) = @_;

        # @array1 now has *all* of the elements
        @ @array2 is *empty*

        return @array1 > @array2;       # Always true!
}
```

To avoid this use references:

```
if( greater_length( \@array1, \@array2 ) ) {
        # ...
}

sub greater_length {
        my ( $array1, $array2 ) = @_;

        my @array1 = @$array1;
        my @array2 = @$array2;

        return @array1 > @array2;
}
```

File I/O

To open files in Perl we usually the `open` function for convenience. We can also use the `sysopen` function if we need precision. The `open` function allows files to be opened in the following modes:

<

Reading. If file doesn't exist an error will occur.

>

Writing. If the file already exists, it will be clobbered, just like the Unix >. If the file doesn't exist, it will be created.

>>

Appending. If the file already exists, data will be added to the end. If the file doesn't exist, it will be created.

|

Pipe. Execute the specified process and either pipe input to it, or take output from it. This will be covered more later.

A plus (+) character can be added to the mode (+<, >+, >>+) in order open the file for both reading and writing. This is very rarely as useful as it might at first sound.

Reading

```
# Open file for reading, die on failure
open(FILE, "<", $filename) or die "Could not open $filename: $!";

# open(FILE, "< $filename") or die "Could not open $filename: $!";

while(<FILE>) {
        # process line
}
```

The three argument version of `open` has the following security advantages over the two argument version, and is recommended.

- The mode must be specified. In the two argument version of `open` it is possible to omit the mode. If however, the filename then contains a mode character (for example `$filename = "> /etc/passwd"`, that will be assumed to be the file mode. This can have undesirable consequences.
- Filenames are taken literally. In the two argument version of `open` whitespace before and after the filenames is ignored. Having Perl treat your filenames literally makes it possible to more easily specify filenames which include unescaped spaces and shell meta-characters.

Traditionally, bareword filehandles in Perl are true globals. If another part of your script, or a module you import, opens a file and uses the same filehandle name as an earlier section of your code, the old file will be closed.

Fortunately in Perl versions 5.6.0 and above, we can use scalar filehandles:

```
open(my $fh, "<" $filename) or die "Could not open $filename: $!";

while(<$fh>) {
        # process line
}
```

These have the advantage that access to the file now has scope. As soon as the filehandle goes out of scope the file will be closed.

Changing the input record separator

By default, files will be read in line by line. To change this we need to change the input record separator `$/`. Changing this also changes what `chomp` removes when called.

```
$/ = undef;     # Read the whole file in at once
$/ = "";        # Read in paragraph by paragraph
$/ = "\n%\n";   # Read in Unix fortunes

open(my $fh, "<", $fortunes) or die $!;

while(<$fh>) {
        chomp;  # remove \n%\n
        # Do something with fortune
}
```

Keep in mind that `$/` is a true global. Changing it in one part of your program changes it for all later parts of your program. If you need to change `$/` within a large program, *localise* your change:

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

        open(my $fh, "<", $fortunes) or die $!;

        while(<$fh>) {
                chomp;  # remove %

                # Do something with fortune
        }
}
```

Using `local` here, tells Perl to ensure that this change only occurs for the duration of the block (the outer curly braces). Once execution leaves the block `$/` will automatically revert to its previous value. Subroutines called from within your block will see the localised value of `%/`.

Writing

```
# Open file for writing, die on failure
open(my $fh, ">", $filename1) or die "Could not open $filename: $!";
open(FILE,  ">>", $filename2) or die "Could not open $filename: $!";

foreach my $number (1 .. 10) {
        print {$fh} $number, "\n";
        print FILE  $number, "\n";
}
```

The example above shows how to print the numbers 1 through to 10 to two different files. In the first, we clobber the file if it already exists, in the second, we *append* to it.

Notice that we do **not** include a comma after the filehandle when we are printing to it. Inserting a comma would tell Perl to print out the filehandle memory location (which wouldn't look very interesting) rather than print to that location.

The curly braces around `$fh` in the first `print` statement are not required, but help make the filehandle stand out and hopefully remove the temptation to add a comma after it.

CPAN

Perl's biggest strength comes from its community. As an extension to that, many Perl programmers write and maintain modules for free use for all as part of the Comprehensive Perl Archive Network (CPAN).

CPAN provides more than 10,000 modules, making it an excellent starting point to help solve your particular problem. However, you should keep in mind that not all CPAN modules are created equal. Some are much better documented and written than others. As with any situation when you're using third party code, you should take the time to determine the suitability of any given module for the task at hand.

Many of the popular CPAN modules are pre-packaged for popular operating systems. In addition, the `CPAN.pm` module that comes with Perl can make the task of finding and installing modules from CPAN much easier.

For modules that aren't packaged for your operating system, you can use the *CPAN shell*. This requires administrator privileges, but on most operating systems can be as simple typing `cpan` at the shell prompt:

```
hostname:/root# cpan

cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support enabled

cpan>
```

Once inside the shell, `help` provides a list of help, and `install` will install a particular module. For example, to install the module `HTML::Template`

```
cpan> install HTML::Template
```

The CPAN shell will locate the module, download it, check its dependencies, and perform any testing required.

For ActiveState Perl installations (which includes most Microsoft Windows machines) the use of PPM (Programmer's Package Manager) is recommended. PPM provides a command line interface for downloading and installing pre-compiled versions of most CPAN modules.

Installing modules using PPM is just as easy as the CPAN shell:

```
C:\> ppm
PPM - Programmer's Package Manager version 3.4.
Copyright (c) 2001 ActiveState Software Inc.  All Rights Reserved.

Entering interactive shell. Using Term::ReadLine::Perl as readline library.

Type 'help' to get started.

ppm>
```

PPM expects double-colons in module names to be replaced with dashes for package names. So to install the `HTML::Template` module we would use:

```
ppm> install HTML-Template
```

If automated installation fails using either system, or we do not have administrator access to the machine, then we can also install a CPAN module manually. CPAN modules come in compressed tarballs (.tar.gz), and should contain a `README` and/or `INSTALL` file that contains instructions for installation. However for almost all modules the proceedure is the same:

```
perl Makefile.PL
make
make test
make install
```

On Windows systems the free `nmake` utility from Microsoft can be used instead of `make` (but needs to be installed separately).

autodie

Many Perl functions return a true value on success and a false value on failure. Assuming success without checking for failure can cause very strange errors. Thus, it is a wise idea to always check your return values.

```
open( my $fh, "<", $filename ) or die "Failed to open: $!";
...
close $fh;      # Oops!  Forgot to check for failure!
```

Unfortunately it’s very easy to forget to add an "or die" to a function call, and making sure you add them all does tend to clutter up your code. A good alternative is to use the `autodie` module. `autodie` replaces functions with equivalents which succeed or die:

```
use autodie qw(open close);

open( my $fh, "<", $filename );
...
close $fh;
```

Now if any calls to `open` or `close` fail, our program will automatically die with an error message. We can use `autodie` with any Perl built-in function except `print`.

Chapter summary

This chapter gave a whirl-wind tour through Perl’s essentials: the variables, conditionals, looping constructs, subroutines and file I/O. We also briefly covered how to install modules via CPAN, and the joys of the `autodie` module.

Chapter 5. Regular expressions

In this chapter...

In this chapter we begin to explore Perl's powerful regular expression capabilities, and use regular expressions to perform matching and substitution operations on text.

Regular expressions are a big reason of why so many people learn Perl. One of Perl's most common uses is string processing and it excels at that because of its built-in support for regular expressions.

Patterns and regular expressions are dealt with in depth in chapter 5 (chapter 2, 2nd Ed) of the Camel book, and further information is available in the online Perl documentation by typing **perldoc perlre**.

What are regular expressions?

The easiest way to explain this is by analogy. You will probably be familiar with the concept of matching filenames under DOS and Unix by using wild cards - `*.txt` or `/usr/local/*` for instance. When matching filenames, an asterisk can be used to match any number of unknown characters, and a question mark matches any single character. There are also less well-known filename matching characters.

Regular expressions are similar in that they use special characters to match text. The differences are that more powerful text-matching is possible, and that the set of special characters is different.

Regular expressions are also known as REs, regexes, and regexps.

Regular expression operators and functions

m/PATTERN/ - the match operator

The most basic regular expression operator is the matching operator, `m/PATTERN/`.

- Works on `$_` by default.
- In scalar context, returns true (`1`) if the match succeeds, or false (the empty string) if the match fails.
- In list context, returns a list of any parts of the pattern which are enclosed in parentheses. If there are no parentheses, the entire pattern is treated as if it were parenthesised.
- The `m` is optional if you use slashes as the pattern delimiters.
- If you use the `m` you can use any delimiter you like instead of the slashes. This is very handy for matching on strings which contain slashes, for instance directory names or URLs.
- Using the `/i` modifier on the end makes it case insensitive.

```
while (<>) {
        print if m/foo/;        # prints if a line contains "foo"
        print if m/foo/i;       # prints if a line contains "foo", "FOO", etc
        print if /foo/i;        # exactly the same; the m is optional
        print if m#foo#i;       # the same again, using different delimiters
        print if /http:\/\//;   # prints if a line contains "http://"
                                # suffers from "leaning-toothpick-syndrome".
        print if m!http://!;    # using ! as an alternative delimiter
        print if m{http://};    # using {} as delimiters
}
```

s/PATTERN/REPLACEMENT/ - the substitution operator

This is the substitution operator, and can be used to find text which matches a pattern and replace it with something else.

- Works on `$_` by default.
- In scalar context, returns the number of matches found and replaced.
- In list context, behaves the same as in scalar context and returns the number of matches found and replaced (a cause of more than one mistake...).
- You can use any delimiter you want, the same as the `m//` operator.
- Using `/g` on the end of it matches globally, otherwise matches (and replaces) only the first instance of the pattern.
- Using the `/i` modifier makes it case insensitive.

```
# fix some misspelled text

while (<>) {
        s/freind/friend/g;    # Correct freind to friend on entire line.
        s/teh/the/g;
        s/jsut/just/g;
        s/pual/Paul/ig;       # Correct (case insensitive) all occurrences
                              # of "pual" (or "Pual" or "PuAl" etc)
        print;
}
```

Exercises

The above example can be found in `exercises/spellcheck.pl`.

1. Run the spelling check script over the `exercises/spellcheck.txt` file.
2. There are a few spelling errors remaining. Change your program to handle them as well. An answer can be found in `exercises/answers/spellcheck.pl`.

Binding operators

If we want to use `m//` or `s///` to operate on something other than `$_` we need to use binding operators to bind the match to another string.

Table 5-1. Binding operators

Operator	Meaning
`=~`	True if the pattern matches
`!~`	True if the pattern doesn’t match

```
print "Please enter your homepage URL: ";
my $url = <STDIN>;

if($url !~ /^http:/) {
        print "Doesn’t look like a http URL.\n";
}

if ($url =~ /geocities/) {
        print "Ahhh, I see you have a geocities homepage!\n";
}

my $string = "The act sat on the mta";
$string =~ s/act/cat/;
$string =~ s/mta/mat/;

print $string;  # prints: "The cat sat on the mat";
```

Easy modifiers

There are several modifiers for regular expressions. We’ve seen two already.

Table 5-2. Regexp modifiers

Modifier	Meaning
`/i`	Make match/substitute match case insensitive
`/g`	Make substitute global (all occurrences are changed)

You can find out about the other modifiers by reading **perldoc perlre**.

Meta characters

The special characters we use in regular expressions are called *meta characters*, because they are characters that describe other characters.

Some easy meta characters

Table 5-3. Regular expression meta characters

Meta character(s)	Matches...
`^`	Start of string
`$`	End of string
`.`	Any single character except `\n`
`\n`	Newline
`\t`	Matches a tab
`\s`	Any whitespace character, such as space, tab, or newline
`\S`	Any non-whitespace character
`\d`	Any digit (0 to 9)
`\D`	Any non-digit
`\w`	Any "word" character - alphanumeric plus underscore (_)
`\W`	Any non-word character
`\b`	A word break - the zero-length point between a word character (as defined above) and a non-word character.
`\B`	A non-word break - anything other than a word break.

Any character that isn't a meta character just matches itself. If you want to match a character that's normally a meta character, you can escape it by preceding it with a backslash.

These and other meta characters are all outlined in chapter 5 (chapter 2, 2nd Ed) of the Camel book and in the `perlre` manpage - type **perldoc perlre** to read it.

It's possible to use the `/m` and `/s` modifiers to change the behaviour of the first three meta characters (`^`, `$`, and `.`) in the table above. These modifiers are covered in more detail later in the course.

Under newer versions of Perl, the definitions of spaces, words, and other characters is locale-dependent. Usually Perl ignores the current locale unless you ask it to do otherwise, so if you don't know what's meant by locale, then don't worry.

Some quick examples:

```
# Perl regular expressions are often found within slashes

/cat/                                   # matches the three characters
                                        # c, a, and t in that order.

/^cat/                                  # matches c, a, t at start of line

/\scat\s/                               # matches c, a, t with spaces on
                                        # either side

/\bcat\b/                               # Same as above, but won’t
                                        # include the spaces in the text
                                        # it matches.  Also matches if
                                        # cat is at the very start or
                                        # very end of a string.

# we can interpolate variables just like in strings:

my $animal = "dog"                      # we set up a scalar variable
/$animal/                               # matches d, o, g
/$animal$/                              # matches d, o, g at end of line

/\$\d\.\d\d/                            # matches a dollar sign, then a
                                        # digit, then a dot, then
                                        # another digit, then another
                                        # digit, eg $9.99
                                        # Careful! Also matches $9.9999
```

Quantifiers

What if, in our last example, we’d wanted to say "Match a dollar, then any number of digits, then a dot, then only two more digits"? What we need are quantifiers.

Table 5-4. Regular expression quantifiers

Quantifier	Meaning
`?`	0 or 1
`*`	0 or more
`+`	1 or more
`{n}`	match exactly n times
`{n,}`	match n or more times
`{n,m}`	match between n and m times

Here are some examples to show you how they all work:

```
/Mr\.? Fenwick/;            # Matches "Mr. Fenwick" or "Mr Fenwick"
/camel.*perl/;              # Matches "camel" before "perl" in the
                            # same line.
/\w+/;                      # One or more word characters.
/x{1,10}/;                  # 1-10 occurrences of the letter "x".
```

Exercises

For these exercises you may find using the following structure useful:

```
while(<>) {
        chomp;

        print "$_ matches!\n" if (/PATTERN/);   # put your regexp here
}
```

This will allow you to specify test files on the command line to check against, or to provide input via STDIN. Hit **CTRL-D** to finish entering input via STDIN. (Use the key combination **CTRL-Z** on Windows).

You can find the above snippet in: `exercises/regexploop.pl`.

1. Earlier we mentioned writing a regular expression for matching a price. Write one which matches a dollar sign, any number of digits, a dot and then exactly two more digits.

   Make sure you're happy with its performance with test cases like the following: `12.34`, `$111.223`, `$.24`.
2. Write a regular expression to match the word "colour" with either British or American spellings (Americans spell it "color")?
3. How can we match any four-letter word?

See `exercises/answers/regexp.pl` for answers.

Grouping techniques

Let's say we want to match any lower case character. `\w` matches both upper case and lower case so it won't do what we need. What we need here is the ability to match any characters in a *group*.

Character classes

A character class can be used to find a single character that matches any one of a given set of characters.

Let's say you're looking for occurrences of the word "grey" in text, then remember that the American spelling is "gray". The way we can do this is by using character classes. Character classes are specified using square brackets, thus: `/gr[ea]y/`

We can also use character sequences by saying things like `[A-Z]` or `[0-9]`. The sequences `\d` and `\w` can easily be expressed as character classes: `[0-9]` and `[a-zA-Z0-9_]` respectively.

Inside a character class some characters take on special meanings. For example, if the first character is a caret, then the list is negated. That means that `[^0-9]` is the same as `\D` --- that is, it matches any non-digit character.

Here are some of the special rules that apply inside character classes.

- `^` at the start of a character class negates the character class, rather than specifying the start of a line.
- `-` specifies a range of characters. If you wish to match a literal -, it must be either the first or the last character in the class.
- `$ . () {} * +` and other meta characters taken literally.

Exercises

Your instructor will help you do the following exercises as a group.

1. How would we find any word starting with a letter in the first half of the alphabet, or with X, Y, or Z?
2. What regular expression could be used for any word that starts with letters *other* than those listed in the previous example.
3. There's almost certainly a problem with the regular expression we've just created - can you see what it might be?

Alternation

The problem with character classes is that they only match one character. What if we wanted to match any of a set of longer strings, like a set of words?

The way we do this is to use the pipe symbol `|` for alternation:

```
/rabbit|chicken|dog/                # matches any of our pets
```

The pipe symbol (also called *vertical bar*) is often found on the same key as `\`.

However this will match a number of things we might not intend it to match. For example:

- rabbiting
- chickenhawk
- hotdog

We need to specify that we want to only match the word if it's on a line by itself.

Now we come up against another problem. If we write something like:

```
/^rabbit|chicken|dog$/
```

to match any of our pets on a line by itself, it won't work quite as we expect. What this actually says is match a string that:

- starts with the string "rabbit" or
- has the string "chicken" in it or

- ends with the string "dog"

This will still match the three incorrect words above, which is not what we intended. To fix this, we enclose our alternation in round brackets:

```
/^(rabbit|chicken|dog)$/
```

Finally, we will now only match any of our pets on a line, by itself.

Alternation can be used for many things including selecting headers from emails for printing out:

```
# a simple matching program to get some email headers and print them out

while (<>) {
        print if /^(From|Subject|Date):\s/;
}
```

The above email example can be found in `exercises/mailhdr.pl`.

The concept of atoms

Round brackets bring us neatly into the concept of atoms. The word "atom" derives from the Greek *atomos* meaning "indivisible" (little did they know!). We use it to mean "something that is a chunk of regular expression in its own right".

Atoms can be arbitrarily created by simply wrapping things in round brackets --- handy for indicating grouping, using quantifiers for the whole group at once, and for indicating which bit(s) of a matching function should be the returned value.

In the example used earlier, there were three atoms:

1. start of line
2. rabbit or chicken or dog
3. end of line

How many atoms were there in our dollar prices example earlier?

Atomic groupings can have quantifiers attached to them. For instance:

```
# match four words (without punctuation)
/(\b\w+\s*){4}/;

# match three or more words starting with "a" in a row
# eg "all angry animals"
/(\ba\w*\s*){3,}/;

# match a consonant followed by a vowel twice in a row
# eg "tutu" or "tofu"
/\b([^\W\d_aeiou][aeiou]){2}\b/;
```

Exercises

1. Determine whether your name appears in a string (an answer's in `exercises/answers/namere.pl`).
2. What pattern could be used to match a blank line? (Answer: `exercises/answers/blanklinere.pl`)
3. Remove footnote references (like [1]) from some text (see `exercises/footnote.txt` for some sample text, and `exercises/answers/footnote.pl` for an answer). (Hint: have a look at the footnote text to determine the forms footnotes can take).
4. Write a script to search a file for any of the names "Yasser Arafat", "Boris Yeltsin" or "Paul Keating". Print out any lines which contain these names. You can find a file including these names and others in `exercises/famous_people.txt`. (Answer: `exercises/answers/namesre.pl`)
5. What pattern could be used to match any of: Elvis Presley, Elvis Aron Presley, Elvis A. Presley, Elvis Aaron Presley. You can find a test file in `exercises/elvis.txt`. (Answer: `exercises/answers/elvisre.pl`)
6. What pattern could be used to match an IP address such as `192.168.53.124`, where each part of the address is a number from 0 to 255? (Answer: `exercises/answers/ipre.pl`)

Chapter summary

- Regular expressions are used to perform matches and substitutions on strings.
- Regular expressions can include meta-characters (characters with a special meaning, which describe sets of other characters) and quantifiers.
- Character classes can be used to specify any single instance of a set of characters.
- Alternation may be used to specify any of a set of sub-expressions.
- The matching operator is `m/PATTERN/` and acts on `$_` by default.
- The substitution operator is `s/PATTERN/REPLACEMENT/` and acts on `$_` by default.
- Matches and substitutions can be performed on strings other than `$_` by using the `=~` (and `!~`) binding operator.

Chapter 6. Advanced regular expressions

In this chapter...

This chapter builds on the basic regular expressions taught earlier in the 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.

Assumed knowledge

You should already be familiar with the following topics:

- Regular expression meta characters
- Quantifiers
- Character classes and alternation
- The `m//` matching function
- The `s///` substitution function
- Matching strings other than `$_` with the `=~` matching operator

Patterns and regular expressions are dealt with in depth in chapter 5 (chapter 2, 2nd Ed) of the Camel book, and further information is available in the online Perl documentation by typing **perldoc perlre**.

Capturing matched strings to scalars

Perl provides an easy way to extract matched sections of a regular expression for later use. Any part of a regular expression that is enclosed in parentheses is captured and stored into special variables. The substring that matches first set of parentheses will be stored in `$1`, and the substring that matches the second set of parentheses will be stored in `$2` and so on. There is no limit on the number of parentheses and associated numbered variables that you can use.

```
/(\w)(\w)/;     # matches 2 word characters and stores them in $1, $2
/(\w+)/;        # matches one or more word characters and stores them in $1
```

Parentheses are numbered from left to right by the *opening* parenthesis. The following example should help make this clear:

```
$_ = "fish";
/((\w)(\w))/;   # captures as follows:
                # $1 = "fi", $2 = "f", $3 = "i"

$_ = "1234567890";
/(\d)+/;        # matches each digit and then stores the last digit
                # matched into $1
/(\d+)/;        # captures all of 1234567890
```

Evaluating a regular expression in list context is another way to capture information, with parenthesised sub-expressions being returned as a list. We can use this instead of numbered variables if we like:

```
$_ = "Our server is training.perltraining.com.au.";
my ($full, $host, $domain) = /(([\w-]+)\.([\w.-]+))/;
print "$1\n";                   # prints "training.perltraining.com.au."
print "$full\n";                # prints "training.perltraining.com.au."
print "$2 : $3\n";              # prints "training : perltraining.com.au."
print "$host : $domain\n"       # prints "training : perltraining.com.au."
```

A regular expression that fails to match the given string does not always reset `$1`, `$2` etc. Therefore, if we do not explicitly check that our regular expression worked, we can end up using data from a previous match. This can mean that the following code may cause unexpected surprises:

```
while(<>) {
        # check that we have something that looks like a date in
        # YYYY-MM-DD format.

        if(/(\d{4})-(\d{2})-(\d{2})/) {
                print STDERR "valid date\n";
        }
        next unless $1;

        if($1 >= $recent_year) {
                print RECENT_DATA $_;
        }
        else {
                print OLD_DATA $_;
        }
}
```

If this code encounters a line which doesn't appear to be a valid date, the line may be printed to the same file as the last valid line, rather than being discarded. This could result in lines with dates similar to "1901-3-23" being printed to `RECENT_DATA`, or lines with dates like "2003-1-1" being printed to `OLD_DATA`.

Extended regular expressions

Regular expressions can be difficult to follow at times, especially if they're long or complex. Luckily, Perl gives us a way to split a regular expression across multiple lines, and to embed comments into our regular expression. These are known as *extended regular expressions*.

To create an extended regular expression, we use the special `/x` switch. This has the following effects on the match part of an expression:

- Spaces (including tabs and newlines) in the regular expression are ignored.
- Anything after an un-escaped hash (`#`) is ignored, up until the end of line.

Extended regular expressions do not alter the format of the second part in a substition. This must still be written exactly as you wish it to appear.

If you need to include a literal space or hash in an extended expression you can do so by preceeding it with a backslash.

By using extended regular expressions, we can change this:

```
# Parse a line from 'ls -l'
m{^([\w-]+)\s+(\d+)\s+(\w+)\s+(\w+)\s+(\d+)\s+(\w+\s+\d+\s+[\d:]+)\s+(.*)$};
```

into this:

```
# Parse a line from 'ls -l'

m{
    ^                           # Start of line.
    ([\w-]+)\s+                 # $1 - File permissions.
    (\d+)\s+                    # $2 - Hard links.
    (\w+)\s+                    # $3 - User
    (\w+)\s+                    # $4 - Group
    (\d+)\s+                    # $5 - File size
    (\w+\s+\d+\s+[\d:]+)\s+     # $6 - Date and time.
    (.*)                        # $7 - Filename.
    $                           # End of line.
}x;
```

As you can see, extended regular expressions can make your code much easier to read, understand, and maintain.

Exercise

Web server access logs typically contain long lines of information, only some of which is of interest at any given time. In the `exercises/access-pta.log` file you'll see an example taken from Perl Training Australia's webserver.

1. Write a regular expression which captures the request origin, the access date and requested page. Print this out for each access in the file. A starting program can be found in `exercises/log-process.pl`.

You can find an answer to this exercise in `exercises/answers/log-process.pl`.

Advanced exercise

1. Split tab-separated data into an array then print out each element using a `foreach` loop (an answer's in `exercises/answers/tab-sep.pl`, an example file is in `exercises/tab-sep.txt`).

Greediness

Regular expressions are, by default, "greedy". This means that any regular expression, for instance `.*`, will try to match the biggest thing it possibly can. Greediness is sometimes referred to as "maximal matching".

Greediness is also left to right. Each section in the regular expression will be as greedy as it can while still allowing the whole regular expression to match if possible. For example,

```
$_ = "The cat sat on the mat";

/(c.*t)(.*)(m.*t)/;

print $1;               # prints "cat sat on t"
print $2;               # prints "he "
print $3;               # prints "mat";
```

It is possible in this example for another set of matches to occur. The first expression `c.*t` could have matched `cat` leaving `sat on the` to be matched by the second expression `.*`. However, to do that, we need to stop `c.*t` from being so greedy.

To make a regular expression quantifier not greedy, follow it with a question mark. For example `.*?`. This is sometimes referred to as "minimal matching".

```
$_ = "The fox is in the box.";

/(f.*x)/;             # greedy               -- $1 = "fox is in the box"
/(f.*?x)/;            # not greedy           -- $1 = "fox"

$_ = "abracadabra";

/(a.*a)/              # greedy               -- $1 = "abracadabra"
/(a.*?a)/             # not greedy           -- $1 = "abra"

/(a.*?a)(.*a)/        # first is not greedy  -- $1 = "abra"
                      # second is greedy     -- $2 = "cadabra"

/(a.*a)(.*?a)/        # first is greedy      -- $1 = "abracada"
                      # second is not greedy -- $2 = "bra"

/(a.*?a)(.*?a)/       # first is not greedy  -- $1 = "abra"
                      # second is not greedy -- $2 = "ca"
```

Exercise

1. Write a regular expression that matches the first and last words on a line, and print these out.

More meta characters

Here are some more advanced meta characters, which build on the ones covered earlier.

Table 6-1. More meta characters

Meta character	Meaning
`\c`*x*	Control character, i.e. **CTRL-*x***
`\0`*nn*	Octal character represented by *nn*
`\x`*nn*	Hexadecimal character represented by *nn*
`\l`	Lowercase next character
`\u`	Uppercase next character
`\L`	Lowercase until `\E`
`\U`	Uppercase until `\E`

Meta character	Meaning
`\Q`	Quote (disable) meta characters until `\E`
`\E`	End of lowercase/uppercase/quote
`\A`	Beginning of string, regardless of whether /m is used.
`\Z`	End of string (or before newline at end), regardless of whether /m is used.
`\z`	Absolute end of string, regardless of whether /m is used.

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

/C++/         # wrong! regexp engine complains about the plus signs
/C\+\+/       # this works
/\QC++\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
```

Read about all of these and more in **perldoc perlre**.

Working with multi-line 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:

```
All language designers are arrogant.  Goes with the territory... :-)
             -- Larry Wall in <1991Jul13.010945.19157@netlabs.com>
%
Although the Perl Slogan is There’s More Than One Way to Do It, I hesitate
to make 10 ways to do something.  :-)
             -- Larry Wall in <9695@jpl-devvax.JPL.NASA.GOV>
%
And don’t tell me there isn’t one bit of difference between null and space,
because that’s exactly how much difference there is.  :-)
             -- Larry Wall in <10209@jpl-devvax.JPL.NASA.GOV>
%
"And I don’t like doing silly things (except on purpose)."
             -- Larry Wall in <1992Jul3.191825.14435@netlabs.com>
%
:        And it goes against the grain of building small tools.
Innocent, Your Honor.  Perl users build small tools all day long.
             -- Larry Wall in <1992Aug26.184221.29627@netlabs.com>
%
/* And you’ll never guess what the dog had */
/*   in its mouth... */
             -- Larry Wall in stab.c from the perl source code
%
Be consistent.
             -- Larry Wall in the perl man page
```

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 `$/`. This is called the input record separator.

```
$/ = "\n%\n";
while (<>) {
        # $_ now contains one RECORD per loop iteration
}
```

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

Changing `$/` doesn't just change how readline (`<>`) works. It also affects the `chomp` function, which always removes the value of `$/` from the end of its argument. The reason we normally think of `chomp` removing newlines is that `$/` is set to newline by default.

It's usually a very good idea to use `local` when changing special variables. For example, we could write:

```
{
        local $/ = "\n%\n";
        $_ = <>;        # first fortune cookie is in $_ now
}
```

to grab the first fortune cookie. By enclosing the code in a block and using local, we restrict the change of `$/` to that block. After the block `$/` is whatever it was before the block (without us having to save it and remember to change it back). This localisation occurs regardless of how you exit the block, and so is particularly useful if you need to alter a special variable for a complex section of code.

Variables changed with `local` are also changed for any functions or subroutines you might call while the `local` is in effect. Unless it was your intention to change a special variable for one or more of the subroutines you call, you should end your block before calling them.

It is a compile-time error to try and declare a special variable using `my`.

Special variables are covered in Chapter 28 of the Camel book, (pages 127 onwards, 2nd Ed). 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 884 (page 403, 2nd Ed) of the Camel book or in **perldoc English**. You can find out about all of Perl's special variables' English names by reading **perldoc perlvar**.

Exercise

1. In your directory is a file called `exercises/perl.txt` which is a set of Perl-related fortunes, formatted as in the above example. This file contains a great many quotes, including the ones in the example above and many many more. Use multi-line regular expressions to find only those quotes which are from the `perl man page`. (Answer: `exercises/answers/fortunes.pl`)

Regexp modifiers for multi-line data

Perl has two modifiers for multi-line data. `/s` and `/m`. These can be used to treat the string you're matching against as either a single line or as multiple lines. Their presence changes the behaviour of caret (`^`), dollar (`$`) and dot (`.`).

By default caret matches the start of the string. Dollar matches the end of the string (regardless of newlines). Dot matches anything but a newline character.

With the `/s` modifier, caret and dollar behave the same as in the default case, but dot will match the newline character.

With the `/m` modifier, caret matches the start of any line within the string, dollar matches the end of any line within the string. Dot does not match the newline character.

```
my $string = "This is some text
and some more text
spanning several lines";

if ($string =~ /^and some/m) {                  # this will match because
        print "Matched in multi-line mode\n";   # ^ matches the start of any
}                                               # line in the string

if ($string =~ /^and some/) {                   # this won't match
        print "Matched in single line mode\n";  # because ^ only matches
}                                               # the start of the string.

if($string =~ /^This is some/) {                # this will match
        print "Matched in single line mode\n";  # (and would have without
}                                               # the /s, or with /m)

if($string =~ /(some.*text)/s) {   # Prints "some text\nand some more text"
        print "$1\n";              # Note that . is matching \n here
}


if($string =~ /(some.*text)/)   {   # Prints "some text"
        print "$1\n";               # Note that . does not match \n
}
```

The differences between default, single line, and multi-line mode are set out very succinctly by Jeffrey Friedl in Mastering Regular Expressions (see the Further Reading 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 multi-line mode" describes one in which each of `^`, `$` and `.` all do what many programmers expect them to do. That is `.` will match newlines as well as all other characters, and `^` and `$` each work on start and end of lines, rather than the start and end of the string.

Table 6-2. Effects of single and multi-line 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
clean multi-line	both `/m` and `/s`	start of line	end of line	Yes

Modifiers may be clumped at the end of a regular expression. To perform a search using “clean multi-line” irrespective of case your expression might look like this

```
/^the start.*end$/msi
```

and if we had the following strings

```
$string1 = "the start of the day
is the end of the night";

$string2 = "10 athletes waited,
the starting point was ready
how it would end
was anyone’s guess";

$string3 = uc($string2); # same as string 2 but all in uppercase
```

we’d expect the match to succeed with both `$string2` and `$string3` but not with `$string1`.

Back references

Special variables

There are several special variables related to regular expressions. The parenthesised names beside them are their long names if you use the English module.

- `$&` is the matched text (MATCH)
- `$‘` (dollar backtick) is the unmatched text to the left of the matched text (PREMATCH)
- `$’` (dollar forwardtick) is the unmatched text to the right of the matched text (POSTMATCH)
- `$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 the same way that other scalar variables can be used.

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

```
# or alternately...
m/^\d+/;
print $&;

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

You can also use `$1` and other special variables in substitutions:

```
$string = "It was a dark and stormy night.";
$string =~ s/(dark|wet|cold)/very $1/;
```

When Perl sees you using PREMATCH (`$‘`), MATCH (`$&`), or POSTMATCH (`$’`), it assumes that you may want to use them again. This means that it has to prepare these variables after every successful pattern match. This can slow a program down because these variables are "prepared" by copying the string you matched against to an internal location.

If the use of those variables make your life much easier, then go ahead and use them. However, if using `$1`, `$2` etc can be used for your task instead, your program will be faster and leaner by using them.

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 set of parentheses are non-capturing
m/(?:Dr|Prof) (\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:

```
print if m{(t\w+) $1};
```

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

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

```
# print if found repeated words starting with ’t’: ie "this this"
# (note, this contains a subtle bug which you’ll find in the exercise)
print if m{(t\w+) \1};
```

Exercises

1. Write a script which swaps the first and the last words on each line.
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`)

Advanced exercises

1. Make your swapping-words program work with lines that start and end with punctuation characters. (Answer: `exercises/answers/firstlast.pl`)
2. Modify your repeated word script to work across line boundaries (Answer: `exercises/answers/multiline_double.pl`)
3. What about case sensitivity with repeated words?

Chapter summary

- Input data can be split into multi-line strings using the special variable `$/` , also known as `$INPUT_RECORD_SEPARATOR`.
- The `/s` and `/m` modifiers can be used to treat multi-line 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 meta characters such as `\1`, `\2` etc may be used *within* the regular expression itself, to refer to text previously matched.

Chapter 7. System interaction, wrappers, and process manipulation

In this chapter...

Perl is a popular tool for system administration as it makes it extremely easy to call existing shell scripts and tools to do your work.

In this chapter we will examine a number of ways that we can call external programs, and how we can control their input and output.

Platform independence

A number of the methods we'll cover below sacrifice portability for utility. This is because a large number of the system commands you may wish to call from your programs are different between operating systems. To counter this, there are a wide number of Perl functions and modules which allow you to interact with the system in an operating system independent function. We recommend that you use these where possible.

Exit values

Experienced shell programmers are familiar with the idea of an *exit value* or *exit status*. When a command terminates, it can return an integer value to its parent, indicating success, failure, or other states. Traditionally, a value of zero means success, and anything else indicates failure. The reasoning behind this is that there is often only one way to succeed, but many ways to fail.

Later in this text we'll discuss how to capture the exit value of other commands. However if you want your Perl programs to interact nicely with your shell scripts, then you'll almost certainly want to use Perl's `exit` function to indicate success or value:

```
exit(0);        # Exit with a value of '0'.
exit;           # The default exit value is '0'.

exit(1);        # Exit with a value of '1'
```

`exit` causes our program to halt immediately and exit with the specified value. The `exit` function shouldn't be used if there's a chance that something else in your program may wish to catch and interpret the error, for that the use of `die` is recommended instead.

Invoking shell commands using system

You can learn more about the `system` command by executing `perldoc -f system`

If you're used to using the shell to execute commands or run other scripts, then you're almost certainly eager to do the same thing in Perl. Doing so couldn't be easier, we just use the `system` command:

```
system("echo Hello World");     # Use the shell to print a greeting
```

Perl always uses the *standard shell* on your operating system, regardless of what your own preferences may be. That means that Perl will invoke `/bin/sh -c` on Unix systems, `command.com` on Windows 95 lineage systems, and `cmd.exe /x/d/c` on Windows NT lineage systems.

On Windows (only) the `PERL5SHELL` environment variable can be set to determine which shell is used.

Commands entered into system work the same as if you had entered them on the command line:

```
# Search for errors in syslog
system("tail /var/log/syslog | grep -i ERROR");

# Use notepad to edit a file
system("notepad example.txt");
```

The `system` command will execute the command (or commands) specified, and wait for them to finish before returning execution to Perl. The commands will share their standard input, standard output, and standard error with Perl.

Multiple argument system

Where possible it is generally better to use the multiple-argument version of `system`. This version assumes its first argument is the system command and that all others are arguments to that command. These arguments are treated literally (not passed via the shell) and are therefore less open to security issues.

When supplied with multiple arguments, `system` will completely bypass the shell. This is faster, and can avoid unintentional interpretation of shell meta-characters:

```
# Run 'cat' on a file named '*.txt'.  By avoiding the shell there
# is no interpretation of shell meta-characters

system('cat', '*.txt');

# Run 'cat' on all files ending in '.txt', but avoiding the shell.
# This uses Perl's built-in glob() function:

system('cat', glob('*.txt') );

# Run 'cat' on a list of files, each name will be interpreted
# literally.

system('cat', @filenames);
```

Problems with system

Of course, there are problems that you can encounter when using `system`. To begin with, your command might fail, either by not starting at all, or by returning some sort of error status in its exit value.

After executing a `system` command, Perl sets a few special variables. The `$?` variable packs up the exit value of the process, as well as information on whether it was killed by a signal, and if it dumped core.

There are a few special values for . If it’s equal to `-1`, then your process never even started, and the reason for this will be in the special variable `$!`. If it’s equal to zero, then your process ran to completion and exited with a zero exit status, which usually means it thought it was successful.

If `$?` is anything else, you have to do use a number of bit-masking and bit-shifting operations to extract the required values:

```
system("some_command");

if ($? == -1) {
        print "Couldn’t run some_command - $!\n";
} elsif ($? == 0) {
        print "some_command ran successfully\n";
} else {
        print "Exit value is ",    $? >> 8,  "\n";
        print "Signal number is ", $? & 127, "\n";
        print "Dumped core\n"   if $? & 128;
}
```

Perl also has a few macros that can make dealing with system easier. These are both easier to understand than the bit-masking operations, and more portable.

```
use POSIX qw(WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG);

system("some_command");

if (WIFEXITED($?)) {
        print "Command terminated normally with exit value ",
              WEXITSTATUS($?),"\n";
} elsif (WIFSIGNALED($?)) {
        print "Command killed by signal ",WTERMSIG($?),"\n";
} else {
        print "Command did not run, or terminated abnormally.\n";
}
```

Of course, having to do all that error checking every time you call to the shell gets very bothersome. Luckily, there’s an easier way.

IPC::System::Simple and autodie

Both the `IPC::System::Simple` module (available from the CPAN) and `autodie` can take the hard work out of checking the return value from system commands:

```
use IPC::System::Simple qw(system);

system("some_command");
```

With `IPC::System::Simple` enabled, the `system` function will execute the command provided and check the result. If the command fails to start, dies from a signal, dumps core, or returns a non-zero exit status, then `IPC::System::Simple` will throw an exception with detailed diagnostics. Unless you take steps to prevent it, a failure from this command will cause your program to die with an error. If you want to capture the error, you can do so:

```
# The 'eval' block allows us to capture errors, which
# are then placed in $@.  If any of the commands below
# fail, the 'eval' is exited immediately.  This means if
# we fail to backup the files, we won't delete them.

eval {
        system('backup_files');
        system('delete_files');
};

if ($@) {
        warn "Error in running commands: $@\n";
}
```

The `autodie` pragma uses `IPC::System::Simple` underneath to provide the same changes to `system` but with lexical scope (until the end of the current block, file, or eval). The same code as above could be written as:

```
eval {
        use autodie qw(system);

        system('backup_files');
        system('delete_files');
};

if ($@) {
        warn "Error in running commands: $@\n";
}
```

When using either module, it's possible to specify a range of acceptable return values as a first argument.

```
use IPC::System::Simple qw(system);

# Run a command, insisting it return 0, 1 or 2:
system( [0,1,2], "some_command");

# Run a command and capture its exit value:
my $exit_value = system( [0,1,2], 'some_command');

# Specify return values using the range operator:
my $exit_value = system( [0..2], 'some_command');
```

Just like regular `system`, the `run` command uses the standard shell when running a single command, or invokes the command directly when called in a multiple argument fashion:

```
# Run 'cat *.txt' via the shell.
system('cat *.txt');

# Run 'cat' on the file called '*.txt', bypassing the shell.
system('cat','*.txt');

# Run 'cat' on all files matching '*.txt', bypassing the
# shell.
system('cat',glob('*.txt'));
```

The `IPC::System::Simple` module also provides a `systemx()` command for running commands, but which *never* invokes the shell, even when called with a single argument.

You can read more about `IPC::System::Simple` at http://search.cpan.org/perldoc?IPC::System::Simple and `autodie` at http://search.cpan.org/perldoc?autodie.

Capturing a program's output

`system` is great for calling processes which either don't generate output, or which send their output to files. But what if you want to run a command that normally prints to STDOUT? Running it with `system` will work, but if you want to capture that output you'll have to redirect it to a file, and then open that file.... It's a lot of unnecessary hard work. Fortunately Perl gives us a few other methods of grabbing an external program's output.

backticks/qx

Just like backticks in `bash` or `sh`, backticks in Perl can be used to execute an external process and capture its output:

```
my $result = ‘finger pjf‘;

my $result2 = qx{finger $name};
```

`qx{}` is an alternative to using backticks. It has the same effect, but is easier to identify when using fonts which represent forward and backticks similarly.

In a scalar context (as above) the whole return result will be returned as a string with embedded newlines. In a list context you will receive a list with one line of output per element.

```
my $directory = qx{dir};        # ’dir’ in a single string.

my @dir_lines = qw{dir};        # One line per element.
```

Backticks *always* invoke the shell, so be careful of unwanted shell meta-characters.

Piped open

Just as we can use `open` for opening files for reading and writing, we can also use `open` for opening processes. After all, there is much similarity between printing to a filehandle, and sending data to a process, or reading from a filehandle and reading data from a process.

```
open (my $ssh, "ssh $host cat $file |") or die "Can’t open pipe: $!";

while(<$ssh>) {
        # We can process the file in any way we like here.
        # In this particular case, we’ll simply print it to
        # our STDOUT.

        print;
}
```

```
close $ssh or die "Failed to close: $! $?";
```

In the above example, our filehandle `$ssh` provides us input from the process.

When opening a process for writing, we need to set up a handler to catch any SIGPIPEs. These might be generated if we try to write to a pipe which has closed; for example if we opened a process that doesn't exist. We do this by adding subroutine reference to the special `%SIG` hash.

```
# Set up a handler in case our pipe breaks, the process doesn't
# exist, or other error occurs.

local $SIG{PIPE} = sub { die "Pipe broke." };

# Open process to pipe to
open(my $out, "| $process1")  or die $!;

print {$out} "Some text";

close $out or die "Failed to close: $! $?";
```

It is important to be aware that the command provided may go via the shell. Thus it is essential to be certain that any variables or data do not contain any unexpected shell meta-characters.

This construct cannot be used for both piping into and out of a process. For tips on how to achieve that read **perldoc perlipc** and `IO::Pipe`.

Multi-arg open

To avoid passing the process command via the shell, it is possible to use a multiple argument version of `open` just like we can with `system` and `exec`. Thus the above examples would become:

```
open (my $ssh, "-|", "ssh", $host, "cat", $file)

# and

open(my $out, "|-", $process1)  or die $!;
```

exec

To pass execution over to an external program after manipulating the environment we can use `exec`. `exec` works very similarly to `system` with one key difference: code occurring in the file after the call to `exec` will only be executed if the call fails.

`exec` is very useful if you're writing a *wrapper* program, something which performs a series of tasks before executing some larger process. For example, you may wish to ensure that certain environment variables are set before calling a given program. This also allows you to have the exact same program and wrapper on a number of machines but each using appropriate environment variables.

```
use Config::General;
my %config = ParseConfig("config.txt");

# Set up environment variables for Oracle
$ENV{TNS_ADMIN}       = $config{tns_admin};
$ENV{ORACLE_HOME}     = $config{oracle_home};
$ENV{LD_LIBRARY_PATH} = $config{ld_path};
```

```
# Run program which assumes environment is done
exec('my_oracle_application');
```

Just as with `system`, `exec` has both a single argument and a multiple argument version. When you do not intend shell meta-characters to be interpreted, the multiple-argument version is recommended for both speed and safety.

Example - Tape backups

Being able to call out to the shell and make use of other programs as components in our program, gives Perl a lot of power. In the below example we write a basic (but effective) program that uses the system's `dump` command to make backups to tape. If the file `/usr/local/etc/fulldump` is found then a full dump is performed and the tape is ejected. This provides a simple mechanism so that other processes (such as a script running on a web server) can influence how our backup is performed.

The code below is optimised to be run from a scheduler such as `cron` that will forward any script output to an administrator. It forwards the output of the `dump` command to STDOUT, and so ensures that full dump reports are sent by mail each evening.

```
#!/usr/bin/perl -wT
use strict;

# Clean our path
$ENV{PATH} = "/usr/local/sbin:/usr/sbin:/usr/bin";
$ENV{RSH} = "ssh";

# These are the list of file systems we want to dump.
# We can include extra options here; in our case
# we specify the '-L' switch to add a tape label.

my @filesystems = (
        '-L boot /boot',
        '-L database /mnt/database',
        '-L home /mnt/home',
);

# If this files exists, we want a full dump.
use constant FULLDUMPFILE  => "/usr/local/etc/fulldump";

# Which program should be use for tape control?
use constant MT            => '/bin/mt';

# Where is our dump command?
use constant DUMP          => '/sbin/dump';

# Default dump level.  -1 is incremental.
my $DEFAULT_LEVEL          = "-1";

# If my full-dump file exists, then do a full dump instead.
if (-e FULLDUMPFILE) {
        $DEFAULT_LEVEL = "-0";
}

# @ARGV is our list of command line arguments.  If we
# don't get a dump level on the command line, we'll
# use the default.

my $level = shift(@ARGV) || $DEFAULT_LEVEL;
```

```
# We expect our dump level to always be a minus, followed
# by a single digit.  This is a simple check to ensure that
# it’s not anything else.

($level) = $level =~ /^-(\d)$/;
defined($level) or die "No dump level available\n";

# Dump each file system
foreach my $filesystem (@filesystems) {

        system("$DUMP -$level $options 2>&1");
        if ($?) {       # Croak if there were problems.
                die "\nErrors encountered!  Entire dump halted.\n";
        }
        sleep 1;
}



# If we had a full dump, clean up and eject the tape.
# Otherwise we leave the tape in the drive.
if ($level eq "-0") {
        system(MT, "offline");
        unlink(FULLDUMPFILE);
        print "Full dump successful.  Tape ejected\n";
}
```

Sending signals

Sometimes we want to send a signal to another process, usually because we want it to terminate. We can do this using Perl’s `kill` function:

```
my $success = kill $signal, $process_id;
```

If the signal is zero then it simply checks that the given process is alive, returning a true value if it is, and a false value if not.

On Unix systems `kill` sends the specified signal to the process in question. You can use either the signal name (without the leading ’SIG’) or its number. Specifying a negative process_id sends the signal to all processes within that group:

```
# Both of these statements send a SIGHUP to the given
# process.

kill ’HUP’, $process_id;
kill 1, $process_id;

# Sends a SIGHUP to the given process and all other
# members of its process-group (usually its children).

kill ’HUP’, -$process_id;
```

To get a list of signals available on a Unix system, use the shell command `kill -l`.

On a Windows system `kill` will terminate the given process, causing it to exit with a status identified by the first argument:

```
# Windows-only, cause $process_id to exit with a value
# of ’42’
```

```
kill 42, $process_id;
```

Sending a value of zero to a process simply returns whether or not it’s still alive, just like in Unix.

Chapter summary

This chapter covered how to call external programs and send data to them, or receive data from them. It also covered sending signals to other processes. For more information on this material read chapter 16 of the Perl Cookbook.

Chapter 8. The command line

In this chapter...

This chapter explores some of Perl’s command line options. To find out more about these read `perldoc perlrun`.

Once off scripts

Occasionally we find a task that only ever needs to be done once. Perhaps we need to change a file so that all strings `A002` become `B005`, or we want to find out how many times a particular IP address accesses the web-server today. In these cases, rather than use a throw-away script, we may be able to write our script directly onto the command line.

Keep in mind as you do this though, that sometimes throw-away scripts turn into programs that become essential to the business. If you think you’re ever likely to run this same program again, or if it is non-trivial, write it into a program, comment it, use strict and warnings, as well as the appropriate modules and keep it. You’ll be glad you did.

Using the execute switch (-e) to convert from epoch-time

Let’s say that you’ve got a timestamp in *seconds from the epoch*; the number of seconds since midnight, 1st January, 1970 GMT. This time format is used by a number of applications, and has the advantage of being an absolute measurement of time that is independent of timezone or daylight savings. It’s also completely useless to most humans. By default, the *squid* proxy server records times in seconds from the epoch.

We can use Perl to convert epoch-time to local time very easily, and we can do so on the command-line using Perl’s *execute switch*, `-e`:

```
perl -e ’print scalar(localtime(1150946643)).qq{\n}’;
```

Under Perl 5.10, we can use the capital `-E` switch to execute code, but turning on all the new 5.10 features first:

```
perl -E ’say scalar localtime(1150946643)’
```

When using the `-e` and `-E` switches, you need to be very careful of interactions with the shell. Most Unix shells pass single-quoted strings to the application without alteration. DOS and Windows shells, on the other hand, use double quotes for this purpose:

```
# Unix, single-quotes
perl -e’print scalar(localtime(1150946643)).qq{\n};’

# Windows, double-quotes
perl -e"print scalar(localtime(1150946643)).qq{\n};"
```

In these notes we'll be using single-quotes when working on the command-line. If you're working on a Windows system, then you'll need to change these to double-quotes before trying any examples.

The `qq{\n}` represents a newline character, which you may more commonly see written as `"\n"`. We use `scalar` to force `localtime` into a *scalar context*. Without this, Perl would instead return us a long list consisting of the year, month, time, hour, minute, second and so forth. Not exactly what we're after.

When writing a script on the command line, it's *always* recommended that you use `q{}` for single quotes, and `qq{}` for double quotes. This avoids any unwanted interaction with the shell, and can also make your code visually easier to read.

To perform multiple operations, just use semi-colons between your statements, in the same way that you do in a program:

```
perl -e 'foreach(<*.txt>) { s/.txt$//; rename(qq{$_.txt},qq{$_-2006.txt}) }'
```

This moves all files with a `.txt` extension to instead end with `-2006.txt`.

Script-less programming

You may have a snippet of Perl that you wish to execute, perhaps from an e-mail or web page, but which you don't want to save as a permanent program. In that case you can invoke Perl and give it a script on STDIN:

```
% perl
foreach(<*.txt>) {
        s/.txt$//;
        rename("$_.txt","$_-2006.txt");
}
```

This will tell you of syntax errors immediately, but script execution will not start until you send Perl an *end-of-file* character, or more commonly known as `EOF`. On Unix systems this is done by hitting *CTRL-D* at the start of a line, and under Windows is done by hitting *CTRL-Z* at the start of a line.

If your program accepts input from STDIN, you will need to provide its input after you've sent the `EOF` character and then send `EOF` again. In this case, you're almost certainly better off writing your code into a file.

Printing switch (-p)

Using `-p` tells Perl to act as a stream editor. It will read input from STDIN, or from files mentioned on the command line, and place each line of input into `$_`. The body of your program is then executed, and the contents of `$_` are printed. It's most commonly used with Perl's substitution operator `s///`, which is covered in the regular expressions chapters of this course.

The following command line snippet can be used to correct a common spelling mistake in one of our documents:

```
perl -pe 's/freind/friend/g' essay.txt > spellchecked-essay.txt
```

It's the same as writing:

```
while(<>) {
        s/freind/friend/g;
        print;
}
```

As a more advanced example, the following snippet can be used to convert *seconds from the epoch* time-stamps into human readable dates for *squid* logfiles:

```
perl -pe’s/^([\d.]+)/localtime($1)/e’ access.log
```

It works by finding a number at the start of each line (the timestamp), and replacing it with the result of calling `localtime` on that timestamp.

Non-printing switch (-n)

```
perl -ne ’print if /perltraining\.com\.au/’
```

Using `-n` makes Perl act almost the same as `-p`. However, the `print` line is excluded. This allows us to write code like the above which only prints when we want it to. It is equivalent to:

```
while(<>) {
        print if /perltraining\.com\.au/;
}
```

Module switch (-M)

Perl has a great number of useful modules, and we may wish to use these on command-line programs. We can load them quickly and easily using the `-M` switch. The following example prints what Perl can find in our environment using `Data::Dumper`:

```
perl -MData::Dumper -e ’print Dumper(\%ENV);’
```

Multiple modules can be used by including multiple `-M` flags.

If you need to provide options to the module, you can do so as follows:

```
perl -Mautodie=open,close -e ’open(my $file, q{> /tmp/foo});
print {$file} qq{12345\n};’
```

The above program will die with an error if the `open` fails, even though we are not explicitly catching this error. This is because of our use of the `autodie` module. It is equivalent to:

```
use autodie qw(open close);
open(my $file, q{> /tmp/foo});
print {$file} qq{12345\n};
```

In-place switch (-i)

```
perl -i -pe ’s/freind/friend/’ file
perl -i.old -pe ’s/freind/friend/’ file
```

Using `-i` on its own allows you to edit the file in place, overwriting the original version. This can be dangerous, as a bug in your program can result in data-loss, and if your program terminates unexpectedly your file can be left in an inconsistent state.

A better solution is to provide an argument to the switch: `-i.old`. This creates a backup copy of the original file `file.old` and then overwrites the original.

This is equivalent to:

```
mv file file.old
perl -pe 's/freind/friend/' file.old > file
```

If your operating system or file-system does not allow an opened file to be removed, then you *must* specify a backup extension when using `-i`. In particular, Windows systems always require an extension.

If the backup file contains an asterisk, then it is replaced with the current filename. This allows you to add a *prefix* instead of a suffix if needed. For example:

```
perl -i'badly_spelled_*' -e's/freind/friend/' file
```

would create a backup called `badly_spelled_file`. You can get fancy and place the asterisk in the middle of the backup name, or even have multiple asterisks if you prefer.

Autosplit switch (-a)

`-a` is Perl's autosplit switch. When using autosplit (with `-n` or `-p`), Perl automatically does a split on whitespace and assigns the result to the `@F` variable.

Let's say that we want to parse the output of `ls -l` from a Unix system. It consists of a series of lines in the following format:

```
-rw-r--r--  1 pjf pjf   10201 Jul 17 13:52 command.pod
-rw-r--r--  1 pjf pjf   17739 Jul 17 15:51 command.sgml
-rw-r--r--  1 pjf pjf 1320760 Jul 18 14:57 sysadmin.ps
-rw-r--r--  1 pjf pjf    2010 Jul 14 17:31 sysadmin.sgml
```

If we want to print all lines which have a file-size greater than 1MB we could use:

```
ls -l  | perl -ane 'print if $F[4] > 1_000_000;'
```

Note that Perl always counts fields starting from zero. The above code run over our sample input would display the single line :

```
-rw-r--r--  1 pjf pjf 1320760 Jul 18 14:57 sysadmin.ps
```

The above Perl code is equivalent to:

```
while (<>) {
        our @F = split(" ", $_, 0);

        print if $F[4] > 1_000_000;
}
```

Note that the `0` as a final argument to `split` means that empty fields are simply discarded; the effect of this is that any sequence of space characters is considered a seperator. You can also use the `-F` switch can be used to specify an alternative pattern on which to split.

Parsing the results of `ls -l` to get file information is *not* a recommended way to gain information about files. It's both slow and prone to error. A better way is to use Perl's in-built `stat` function, or the file test operators which are covered in the directories chapter of this course.

You could use an example similar to the above if you did not have direct access to the filesystem, such as the output of `ls -l` stored in a file.

Other switches

Perl has many other switches. Below are some common ones.

Check switch (-c)

```
perl -c program.pl
```

`-c` causes Perl to check the program for syntactic errors and to exit without executing the main body of code. Code in `BEGIN` and `CHECK` blocks, as well as `use` lines will still be executed.

Warnings switch (-w)

```
perl -w program.pl
```

The `-w` switch runs your program with warnings turned on. Running with warnings helps catch common mistakes, and is highly recommended.

Debugging switch (-d)

```
perl -d program.pl
```

Runs the program under the Perl debugger.

You can learn more about the Perl debugger by using `perldoc perldebug`

Include switch (-I)

```
perl -I/home/pjf/perl/lib/ program.pl
```

Specifies which additional directories should be searched when looking for modules. This modifies Perl's special `@INC` variable.

Taint switch (-T)

```
perl -T program.pl
```

Turns on taint mode. Any input from outside the program must be cleaned before being used to cause effects outside the program. For example data received from a user must be cleaned before being passed as an argument to a system call.

We'll cover taint mode in more detail later in the course.

To learn more about Perl's taint mode, read Perl Training Australia's *Perl Security* course manuals available at http://perltraining.com.au/courses/perlsec.html and Perl's security documentation at `perldoc perlsec`.

Chapter summary

Perl's command line interface makes it a great filter when passing the output of one program to another with a little editing on the way. It also makes it easy for us to perform basic tasks without having to write a program for it.

Chapter 9. Filesystem analysis and traversal

In this chapter...

Many system administrators are familiar with shell-based tools when it comes to filesystem manipulation, and Perl makes it very easy to integrate with existing shell commands. Unfortunately calling out to the shell is comparatively slow, difficult to debug, and can be operating-system dependent. Luckily Perl comes with built-in functions for filesystem manipulation, which are fast, cross-platform, and provide better diagnostics. We'll be covering them in this chapter.

This chapter covers how to perform common filesystem operations in Perl. To find out more about these functions read `perldoc -f function` or where the function is provided by a module: `perldoc Modulename`.

More information about writing cross-platform code can be found in `perldoc perlport`.

Directory separators

Different operating systems have different directory separators. Unix systems use forward-slash (`/`), DOS and Windows uses backslash (`\`), and MacOS 9 systems use a colon (`:`).

Perl interprets a forward-slash as a directory separator on both *Unix and Windows* systems, and we'll be using forward-slash as the directory separator throughout these notes. Using a forward-slash also avoids any problems where Perl may interpret a backslash as a *meta-character*, such as using "\n" for a newline.

For code that is truly independent of filesystem considerations, we'll examine the `File::Spec` module later in this chapter.

Working with files

Copying, moving and renaming files

One of the most common filesystem operations is that of copying or moving files. Perl comes with the `File::Copy` module that provides a portable, cross-platform way to copy and move files.

```
use File::Copy;

# Copy one filename to another.
copy($existing, $new) or die "Failed to copy: $!";

# Copy the contents of a file to STDOUT.
copy($existing, \*STDOUT) or die "Failed to copy: $!";

# Move (rename) a file.
move($old_location, $new_location) or die "Failed to move: $!";
```

If you're copying from one filename to another, then under VMS, OS/2, Win32, and MacOS Classic `File::Copy` will attempt an attribute-preserving system copy.

Perl also has an in-built `rename` function, which is a thin wrapper around any system call provided by the operating system:

```
rename($old_name, $new_name) or die "Failed to rename: $!";
```

Be aware that behaviour of this function varies significantly depending on the system implementation. For example, it may not work across file system boundaries. In many cases `File::Copy`'s `move` function provides a more portable and reliable alternative.

For more information on copying files, see `perldoc File::Copy`

Deleting files

Perl has an in-built function called `unlink` for deleting files.

```
unlink $file or die "Failed to remove $file: $!";
```

`unlink` can be passed multiple files, and returns the number of files successfully deleted. It's recommended that you delete files one at a time, so if a failure does occur you know *which* file failed to be deleted:

```
foreach my $filename (@list_of_files) {
        unlink($filename) or warn "Could not remove $filename - $!";
}
```

`unlink` will not delete directories, see `rmdir` later in these notes.

Some filesystems, particularly under VMS, keep multiple versions of files. Thus a portable method to make sure all copies of a file are removed is to use:

```
1 while unlink "file";
```

Finding information about files

To find out information about files we can use the file-test operators. These are similar to the ones used by the `bash` shell, and a full list can be found in **perldoc -f -x**.

```
if( -r $file ) {
        print "$file is readable.\n";
}

if( -e $file ) {
        print "$file exists.\n";
}
```

Perl also has a `stat` function that returns a large amount of information on a file at once.

You should be mindful that while the file-test operators will provide you with information about each file at the current time, this may change as your program is running. It would be foolish to assume the size of a file is constant if you know it to be a logfile that is being actively written.

Open the file only if...

Let's say that you wish to write a new file, but your program should never overwrite an existing one. You could write code that looks like this:

```
# DANGER!  This code contains a race condition, and
# should not be used.

if (not -e $filename) {
        open(my $fh, ">", $filename) or die "Can't open $file - $!"
}
```

However that code contains a problem. In between testing to see if our file exists, and opening the file, another process may create a file with that name. Perhaps it's because we're on a busy system, or our program is running multiple times, or because someone is intentionally trying to trick our system into doing something it should not. In any case we run the risk of clobbering an existing file. On a filesystem that allows symbolic links, we may even clobber an existing file in an entirely different location.

A much better way of opening files when we need careful control is to use Perl's `sysopen` function:

```
use Fcntl;

# Open a NEW file for writing.  This fails if the
# file already exists, or is a symlink.
sysopen(my $fh, $filename, O_WRONLY|O_CREAT|O_EXCL)
        or die "Failed to open $outfile: $!";
```

The reasons for using `sysopen` are twofold. Firstly, it's faster, we're performing one operation instead of two. The second, and more important reason, is that it's much more secure. The `O_CREAT|O_EXCL` flag combination tells Perl that it *must* create a new file, it can't open an existing file for writing, nor may it chase a symlink. This means we don't run the risk of accidently clobbering an existing file, even on a very active system.

You can learn more about race conditions and `sysopen` in Perl Training Australia's *Perl Security* course materials at http://perltraining.com.au/courses/perlsec.html .

Temporary files

Opening a temporary file is a very common operation. In line with Perl's design of making "simple jobs easy, hard jobs possible", opening a temporary file securely in Perl is a very easy task.

In many situations, there's no need to have a temporary file with an actual *name*. If a file is temporary, and is only to be manipulated by the current process and its children, then it's possible to use that file without referring to the file system at all.

The lack of name has numerous advantages. The file is automatically cleaned up when the last filehandle to it is closed. It's also possible to keep very tight controls on what can access that file, as it's not accessible via the regular file system.

Creating an anonymous file in Perl version 5.8.0 and beyond is a very simple operation using `open`:

```
my $fh;
open($fh,"+>",undef) or die "Could not open temp file - $!";
```

Using an undefined filename indicates to Perl that an anonymous temporary file is desired. This can be written to and read from just like a normal file, however you will need to use the `seek()` function to read the contents of the file once you've written to it.

You can also use the `File::Temp` module under any version of Perl to safely create temporary files:

```
use File::Temp qw(tempfile);

my $fh = tempfile() or die "Could not open temp file - $!";

print {$fh} "This is written to my tempfile\n";
```

The `File::Temp` module provides an excellent cross-platform interface for working with temporary files, and contains a number of additional safety checks to ensure that files are created in a secure fashion. The `File::Temp` module also provides ways of securely creating temporary directories, and safely deleting temporary files.

File locking

Perl comes with a portable locking mechanism called *flock*, which is short for file-lock. This allows us to apply *advisory* locks to any filehandle.

```
use Fcntl qw(:flock);

flock($fh, LOCK_EX) or die "Cannot get an exclusive lock: $!;

# or

flock($fh, LOCK_SH) or die "Cannot get a shared lock: $!;

# use our locked file
# closing releases the lock

close $fh;
```

Perl's flock mechanism can be used to lock any filehandle, including sockets and streams like `STDIN`. If the lock fails, or your operating system does not support locking on the requested filehandle, flock will return false.

Locks in Perl are *advisory*, meaning that other processes can ignore them if they wish. In fact, most operating systems only have advisory locking of files, or only support mandatory locking in very special cases. There are good reasons for this; on a Unix system a mandatory lock on the `/etc/passwd` file by a hung or malicious program could potentially prevent access to the entire system.

By default, flock will wait indefinitely until a lock is obtained, however we can request a lock be made in a non-blocking fashion by using the special constant `LOCK_NB`:

```
use Fcntl qw(:flock);

if( flock(FILE, LOCK_EX|LOCK_NB) ) {
        # we got the lock
        # do something with it
}
```

While Perl allows us to unlock files by using the `LOCK_UN` constant, its use is often a mistake. Normally when we're finished with a file it is best to close it, as this automatically releases the lock, and avoids any possibility of us accidently reading or writing to an unlocked file. Under older versions of perl unlocking a file did not always flush any output buffers, and this could result in subtle errors as data would often be written to the (open but now unlocked) file on program exit.

Locking your process

It's common to see external lock files being used to ensure that only a single instance of a program is running on a machine. This has the additional overhead of creating and tidying up the lock file. Luckily for us, this is rarely needed in Perl.

We can take advantage of the fact that our program's source code will be stored in a file, and that file must be accessible to the Perl interpreter in order for it to run. Rather than locking an external file, we can simply lock our own source code, the filename of which can be found in the special variable `$0`.

```
use Fcntl qw(:flock);

open(SELF,"<",$0) or die "Cannot open $0 - $!";

flock(SELF, LOCK_EX|LOCK_NB) or die "Already running.";
```

If this causes any problems, Perl programs also allow data to be stored at the end of their source code, in a special `__DATA__` section. If this exists, the data is accessible through a special filehandle called DATA. We can use this as an alternative method to lock our own program.

```
use Fcntl qw(:flock);

flock(DATA, LOCK_EX|LOCK_NB) or die "Already running.";

# ...

__DATA__
Don't remove this data section!
```

This is a less optimal solution as the `DATA` section must be at the end of your code, and is therefore a long way away from your locking code. If the `__DATA__` section does not exist, `flock` will fail with our message `Already running` rather than a warning that `DATA` doesn't exist.

File Permissions

Available file permissions are not consistent across operating systems. In Unix-based operating systems, file permissions are represented as octal numbers. 1 stands for execute, 2 for write, 4 for read. These values are added to indicate multiple permissions with the common values being 5 - read and execute, 6 - read and write, 7 - read, write and execute.

These permissions are then applied to cover "owner", "group" and "other" permissions. Thus a file with permissions of `0750` means that the owner can read, write and execute it, people in the same group as the owner can read and execute it, but everyone else has no permission to do anything.

This permission model is also used for Unix directories. To add something to a directory you need to be able to write to it, to see a listing you need to be able to read it, and to enter it at all you need to be able to execute it.

Many of Perl’s file permissions functions assume this model. The various Unix/POSIX compatibility layers attempt to map these to meaningful values for other operating systems, but sometimes there is no good mapping. Read **perldoc perlport** for information on your operating system.

When specifying permissions in Perl, it is important to do so in *octal*. Perl considers a number to be an octal number if it starts with a zero, such as `0644` or `0755`. Forgetting the leading zero will have Perl interpret the number as *decimal*, and you will end up with very different permissions than what you expect.

Changing permissions

`chmod` changes the permissions on a list of files. Be aware that Unix-like permissions do not make sense on all operating systems.

```
chmod 0775, $file_a or die "Failed to change permissions: $!";

# or a list:
chmod 0775, $file_b, $file_c;
```

Default permissions (umask)

The *umask* represents permission bits that are *never* set when creating a file. Perl’s `umask` function can be used to both get and set the umask used by the current process.

```
my $current = umask();

umask 0022;
```

The *umask* is applied to all files that are created. For example, the following code will create a new file with permissions 0755:

```
use Fcntl;

umask 0022;

sysopen(FILE, "runme", O_WRONLY|O_CREAT|O_EXCL, 0777);
```

If no umask is set in the file, then the process owner’s umask will be used. You should *always* have a good reason when setting the *umask* in your program, as this takes away the user’s choice in setting their own.

Changing ownership

```
my ($login,$pass,$uid,$gid) = getpwnam($user)
        or die "$user not in passwd file";

chown $uid, $gid, $file;
```

The above snippet looks up a given username, to get their UID and GID from the password file. This is then used to change the ownership and group ownership of a file to that user.

`chown` is not implemented on a number of operating systems, and even when it is you can rarely change the owner of a file unless you're the superuser. Use of this function will reduce the portability of your program. For more information read `perldoc perlport` and `perldoc -f chown`

Links

For filesystems that support links, Perl has three functions for link manipulation.

To create a symbolic link in Perl, use the `symlink` function:

```
symlink $old_file, $new_file or die "Failed to create symlink: $!";

# To check that your system allows symlinks:
$symlinks_ok =  eval { symlink("",""); 1 };
```

To create a hard link, use the `link` function:

```
link $old_file, $new_file or die "Failed to create link: $!";
```

To read the destination name of a symbolic link, use the `readlink` function:

```
my $linked_to = readlink $link;
```

Working with directories

Reading directories

There are two ways to read the contents of a directory in Perl. `opendir` and its associate `readdir` give you very fast access to all files including dot files. Files are returned in "file-system order" which may not be sorted and only filenames (and not paths) are returned.

```
opendir( HOMEDIR, $ENV{HOME} )  or die "Failed to read $ENV{HOME}: $!";
my @files = readdir(HOMEDIR);
closedir(HOMEDIR);

# Newer versions of Perl (5.6.1 and beyond) support opening
# directory handles into scalars.

opendir( my $home, $ENV{HOME} ) or die "Failed to read $ENV{HOME}: $!";
my @files = readdir($home);
closedir($home);

# In either case, once we have our filenames, we can then process
# them.  He we walk through each one and print the filename:

foreach my $file ( @files ) { print "$file\n"; }
```

Alternately, we can use `glob`.

```
my @files = glob("*.txt");      # files ending with .txt

# or less commonly:
my @files = <*.txt>;
```

Glob is slower, returns the files in ascii-betical order, with full path names and does not include dot files (such as `.forward`). On the other hand, readdir returns file names in file system order (which may not be sorted).

Sub-directories are considered to be files.

Returning normal files

Often when we process a directory we want to skip over sub-directories, we can do this with the file operators from above.

```
opendir( my $home, $ENV{HOME} ) or die "Failed to read $ENV{HOME}: $!";

foreach my $file ( readdir($home) ) {
        next unless -f $file;

        # process file
}
```

Creating and removing directories

```
mkdir $new_dir or die "Failed to make $new_dir $!";
mkdir $new_dir, $mask or die "Failed to make $new_dir: $!";

rmdir $new_dir or die "Failed to remove $new_dir: $!";
```

For `mkdir`, if the mask is omitted it defaults to `0777`, with modifications from `umask` if applicable. `rmdir` will fail if the directory is not empty.

To create or remove a directory tree we can instead use `File::Path`.

```
use File::Path;

mkpath( ’shop/inventory/shelf’ );

mkpath( ’shop/inventory/shelf’, 0, $mode );

rmtree( ’shop/inventory/shelf’ );
```

`mkpath` returns a list of all directories created upon success and throws an exception on failure. `rmtree` behaves like the Unix `rm -r` command; deleting both files and directories in the tree. Upon success it returns the number of files deleted. Symlinks are not followed.

For more information about `mkpath` and `rmtree` read `perldoc File::Path`.

Directory paths

Different operating systems have different directory separators. This can make writing portable code much harder. Fortunately `File::Spec` can be used to work with directories in an operating system independent manner.

```
use File::Spec;

my $dir = File::Spec->catfile( ’shop’, ’inventory’, ’shelf’, ’price.txt’ );
print $dir;
```

```
# Alternately split the path into parts.
my ($volume,$directories,$file) = File::Spec->splitpath( $dir );
```

The print will generate:

shop/inventory/shelf/price.txt

  on Unix and Unix-like operating systems.

shop\inventory\shelf\price.txt

  on Win32 operating systems.

shop:inventory:shelf:price.txt

  on Mac OS 9.

Directory representations

Just as different operating systems have different separators, they also have different representations for other common directories. `File::Spec` makes many of these more manageable:

```
use File::Spec;

my $current_dir = File::Spec->curdir();   # '.' on both Unix and Win32
my $updir       = File::Spec->updir();    # '..'   ""        ""
my $root_dir    = File::Spec->rootdir();  # '/' Unix, '\' Win32

my $null_device = File::Spec->devnull();  # /dev/null on Unix
                                          # nul       on Win32

my $tempdir     = File::Spec->tmpdir();   # /tmp      on both
```

Preventing path traversal attacks

A common issue with accepting file names from untrusted users is avoiding path traversal attacks. For example consider the following:

```
$filename = "../../../../etc/passwd";    # assume came from user

# write to the file specified by the user
open(FILE, ">", $filename) or die "Failed to open file $filename: $!";
```

Oops! We might just have clobbered `/etc/passwd`! Fortunately we can use `File::Spec` to spot attempts to climb up the directory structure in an operating system independent manner:

```
use File::Spec;

$filename = "../../../../etc/passwd";    # assume came from user

# If we have an absolute path, then complain.
if( File::Spec->file_name_is_absolute( $filename )  ) {
        die "Absolute path not allowed";
}



# If our path contains any "parent directory" elements,
# then complain.
```

```
my $updir = File::Spec->updir();
if ( grep {$_ eq $updir} File::Spec->splitdir( $filename ) ) {
        die "Parent directories not allowed in pathnames."
}

# write to the file specified by the user
open(FILE, ">", $filename) or die "Failed to open file $filename: $!";
```

Changing directories

```
use File::Spec;
chdir( File::Spec->updir() ) or die "Failed to change up a dir: $!";
```

Changes your program's current working directory, if possible. This changes the working directory for the rest of your program and for all processes your program may spawn. Be aware that this will have no effect on your current working directory once your program terminates.

Current working directory, absolute path for files

```
use Cwd;
my $pwd = getcwd();

use Cwd qw/abs_path/;
my $pwd = abs_path($file);
```

`getcwd` returns the current working directory for your program when called.

`abs_path` returns the absolute path of the given file.

File::Find

It is possible to use Perl's `opendir` and `readdir` functions to recurse through directories; but it's not easy or elegant. Fortunately there's a module called `File::Find` which replaces the need. This emulates Unix's `find` command but is portable across operating systems. `File::Find` comes standard with typical Perl installs.

```
use File::Find;
my $YEAR = 365;      # Days in year (good enough for this)
my $SIZE = 100_000;  # 100k bytes

# For each directory passed in on the command line
foreach my $dir (@ARGV) {
        find ( \&find_old_music, $dir );
}

# All music which hasn't been accessed for a year, 100k+ in size
sub find_old_music {
        if( /(\.(mp3|ogg)$/i and -A > $YEAR and -s > $SIZE) {
                print "$File::Find::name\n";
        }
}
```

Our `\&find_old_music` argument in our call to `find` is a subroutine reference. This subroutine will be called for each file `File::Find` finds (including directories and other special files). When the `find_old_music` subroutine gets called it has three variables set up:

$_

Set to the name of the current file.

$File::Find::dir

Set to the current directory.

$File::Find::name

Full name of the file. Equivalent to `$file::Find::dir/$_`.

`File::Find` automatically changes your current working directory to the same as the file you are currently examining.

File::Find::Rule

Some people find the call-back interface to `File::Find` difficult to understand. Further, storing both your rules and your actions in the call-back subroutine hides a lot of detail from someone glancing over your code. As a result, an alternative exists called `File::Find::Rule`.

```
use File::Find::Rule;
my $YEAR_AGO = time() - 365 * 24 * 60 * 60;     # Year ago in secs
my $SIZE = 100_000;                             # 100k bytes

my @old_music = File::Find::Rule->file()
                                ->name ( ’*.mp3’, ’*.ogg’)
                                ->atime( "< $YEAR_AGO" )
                                ->size ( "> $SIZE" )
                                ->in   ( @ARGV );

# Do something with @old_music files
```

`atime` actually returns the file access time in seconds since the 1st January 1970. Thus `->atime( "< $YEAR_AGO" )` says that it was last accessed at a point that was earlier in time than a year ago was.

Chapter summary

This chapter covered portable methods to work with files and directories with some attention paid to portability issues. For more information about these subjects please read Chapter 7 of the Perl Cookbook.

Chapter 10. Mail processing and filtering

In this chapter...

Email is an excellent method to send non-urgent information to any number of recipients. This chapter deals with two common problems: how to send email from programs, to let us know how things went, and how to deal with the already incredible amount of mail we currently receive.

Sending mail

A very easy module for sending email is `Mail::Send`. By default it will search for your mail executable and use the first it finds. You can change this behaviour by explicitly setting which mailer you wish to use in the call to `open`. `Mail::Send` is part of `MailTools`.

```
use Mail::Send;
my $msg = new Mail::Send;
my $time = localtime();

$msg->to( ’user1@example.com’, ’user2@example.com’);
$msg->cc( ’user3@example.com’);
$msg->bcc(’user4@example.com’);
$msg->subject("Webserver is down! ($time)");

my $fh = $msg->open;            # use the default mailer on the system

print {$fh} "Web server response for page: $page was: $response."

$fh->close;         # complete the message and send it
```

With attachments

`Mail::Send` doesn’t handle attachments. For simple work with attachments, you may want to look at `MIME::Lite`.

```
use MIME::Lite;

# Create a new multi-part message:
$msg = MIME::Lite->new(
         From    => ’user1@example.com’,
         To      => ’user2@example.com’,
         Cc      => ’user3@example.com, user4@example.com’,
         Type    => ’multipart/mixed’
         Subject => "Web server is down! ($time)",
);

# Attachments
# Text part
$msg->attach(
         Type     => ’TEXT’,
         Data     => "Web server response for page: $page ".
                     "was: $response." .
                     "See the attached image for recent load.",
);
```

```
# Attach Image.
$msg->attach(
         Type        => 'image/gif',
         Path        => '/var/www/data/load.gif',
         Filename    => 'load.gif',
         Disposition => 'attachment'
);

$msg->send;
```

Filtering mail

There's a good chance you receive lots of e-mail. If you're a system administrator with machines that send you status reports, or the designated contact person for a project or business, then there's a chance that you'll receive a truly amazing amount of e-mail.

Managing all that e-mail can be hard. There are lots of solutions that can do basic operations, like sorting into folders, but sometimes you'll want to perform more powerful operations. Maybe you need to send an SMS when an important e-mail arrives. Maybe you need to send different vacation messages to your work colleagues than to your friends. Maybe you want to strip incoming files and place them somewhere on the filesystem. Whatever you want, you may find that existing tools don't quite do the job.

Luckily for us, it's quite easy to allow Perl to control the delivery of e-mail.

Mail::Audit

Simon Cozen's `Mail::Audit` module has a simple-to-use interface, understands a great many mailbox formats, and possesses a surprising array of plug-ins.

`Mail::Audit` is most commonly used as a mail-filter, with incoming mail being delivered to a program you've written instead of to your regular mailbox. With many common Unix mailers you can do that by putting the following in your `~/.forward` file:

```
|~/bin/my-mail-filter
```

Although if you're using `qmail`, you'll want to edit your `.qmail` file instead to add:

```
preline ~/bin/my-mail-filter
```

Setting a program as your local delivery agent depends upon the mail transport agent installed on your system. It's also *strongly* recommended that you test your program carefully before enabling it. Losing mail will ruin your day.

Using `Mail::Audit` is easy. We start by loading the module, and creating a new `Mail::Audit` object. This automatically reads our mail (from `STDIN` by default), and parses it:

```
#!/usr/bin/perl -w
use strict;

use Mail::Audit;

my $mail = Mail::Audit->new(emergency=>"~/emergency_mbox");
```

You'll note that we've specified an *emergency* mailbox. Should anything go horribly wrong, `Mail::Audit` will write the message here. If this isn't set then `Mail::Audit` will try to hand the mail back to your mail transport agent if things go wrong.

Once we've got a `Mail::Audit` object, delivering our mail is easy:

```
# Mail containing ’root’ in the from line goes into a
# maildir folder.  Note the trailing slash.

$mail->accept("~/Maildir/.root/") if $mail->from    =~ /root/i;

# Mail with ’joke’ in the subject gets delivered to a ’jokes’
# mbox file.  Note the is NO trailing slash.

$mail->accept("~/Mail/jokes")     if $mail->subject =~ /joke/i;

# Everything else goes to our default mailbox:
# /var/spool/mail/username

$mail->accept();
```

`Mail::Audit` understands both *mbox* and *Maildir* mailboxes, and will try to auto-detect the format if the file or directory exists on disk already. If auto-detection fails, then it will default to *Maildir* if the filename ends in a slash, and *mbox* otherwise. It is *strongly* recommended that you always include the trailing slash for *Maildir* delivery, even if you think the directory already exists.

In these notes we will assume that you are using *Maildir* directories, as they have rapidly grown in popularity. Our examples can be easily modified to work with *mbox* files just by omitting the trailing slash in folder names.

Accepting and filtering mail

Calling `accept` on a mail normally terminates your program. If you want to accept mail to multiple locations at once, you can do so by passing all those locations as arguments to `accept`.

The following example automatically saves all incoming mail into Maildirs based upon the sender, as well as to a central inbox.

```
#!/usr/bin/perl -w
use strict;

use Mail::Audit;
use Mail::Address;
use constant INBOX => "~/Maildir/";

my $mail = Mail::Audit->new(emergency=>"~/emergency_mbox");

my $from_header = $mail->from;
my @senders     = Mail::Address->parse($from_header);

# This following line walks through all the senders mentioned
# in the From header (almost always just one), extracts the
# username (p.fenwick@perltraining.com.au would be just
# ’p.fenwick’.

my @usernames   = map { $_->user  } @senders;
```

```
# We now adjust our senders to replace dots (which have
# special meanings in Maildirs) with underscores (which do
# not).

foreach (@usernames) {
        s{\.}{_}g;
}

# Finally, we map those usernames into directories.
# Our p.fenwick example would become ~/Maildir/.users.p_fenwick/

my @user_archives = map { INBOX. ".users.$_/" } @usernames;

# If we’ve failed to extract any e-mail addresses from our From
# header, then @senders will be empty, and we’ll end up with an
# empty @user_archives.  In that case we’ll only be delivering
# to the main mailbox.

$mail->accept(INBOX, @user_archives);
```

One of the most commonly used features of `Mail::Audit` is the ability to separate incoming mail into folders, particularly for mailing lists. We could do on a list-by-list basis:

```
my $from = $mail->from;

if ($from =~ /melbourne-pm\@pm\.org/) {
        $mail->accept(INBOX.".lists.perl.melbourne-pm/");
} elsif ($from =~ /jobs\@perl\.org/) {
        $mail->accept(INBOX.".lists.perl.jobs/");
} elsif ($from =~ /debian-security-announce/) {
        $mail->accept(INBOX.".lists.security/");
}

$mail->accept(INBOX);
```

If you’re on a lot of mailing lists then you may find it more convenient for Perl to automatically detect and sort your mailing lists for you:

```
use Mail::Audit;
use Mail::ListDetector;
use constant INBOX => "~/Maildir/";

my $mail = Mail::Audit->new(emergency=>"~/emergency_mbox");

# Let’s see if we’re dealing with a post to a mailing list...

my $list = Mail::ListDetector->new($mail);

if ($list) {
        # It is a post to a list!  Find its name...
        my $list_name = $list->listname;

        # Replace dots with underscores ...
        $list_name =~ s{\.}{_}g;

        # And accept it to ~/Maildir/.lists.$list_name/
        $mail->accept(INBOX.".lists.$list_name/");
}

# If it’s not a list, then just throw it in the regular Mailbox.
$mail->accept(INBOX);
```

Of course, we may want to do perform actions based upon the mailing list name, rather than blindly save it to a folder. In any case, the `Mail::ListDetector` module can do all the hard work of identifying the list for us.

Chapter summary

In this chapter we have only really scratched the surface of using Perl for mail filtering. A wide variety of modules exist for creating, editing, searching, filtering, and processing email. The popular *spamassassin* system also exists as a Perl module.

More information and modules for Mail handling can be found on the Comprehensive Perl Archive Network (CPAN), at http://search.cpan.org/search?q=mail .

Chapter 11. Security considerations

In this chapter...

Perl is a very powerful language which attempts to make almost everything possible. This, of course means that it makes it very easy to write large security holes into your code. Fortunately, a little bit of knowledge can make this much less likely.

In this chapter we cover potential security pitfalls and how to avoid most of them. We also touch on privileges under Perl.

This is not a complete coverage of Perl security. For more comprehensive coverage of programming securely in Perl refer to Perl Training Australia’s *Perl Security* course notes (available online at at http://perltraining.com.au/notes.html).

Potential security pitfalls

Most of us wouldn’t give shell access on a secure machine to any random person who asked. Neither would we install code from an unknown party just on their request. Yet it’s surprising how often security is overlooked when writing code. Any time that a program accepts input from an unknown party and does not verify that input before using it to affect your system, it is inviting a security violation.

Cleaning up after security violations can be a tremendous job. It makes sense, therefore, to try to avoid them. Being aware of the issues is the first step; knowing how to avoid most of them is the second.

The biggest security pitfall in most programs (regardless of language) is best summed up as *unintended consequences*. Consider the following Perl code:

```
#!/usr/bin/perl -w
# DON’T USE THIS CODE
use strict;
use CGI;

my $filename = CGI->param(’file’);

open(FILE, "/home/test/$filename")
      or die "Failed to open /home/test/$filename for reading: $!";

# print out contents of requested file
print <FILE>;
```

In this code we have used the two-argument version of `open`. Further, we haven’t specified a mode for opening the file. Under normal circumstances, Perl will assume we meant to open this file for reading. To many beginners, this code looks innocent. Yet imagine that we pass in the value:

```
../../etc/passwd
```

Oops. We just printed out the contents of `/etc/passwd`! Now imagine that we pass in the value:

```
../../bin/rm -rf /home/test/ |
```

This tells Perl to execute the command on the left and pipe the output to the given filehandle. Printing out the contents of `/etc/passwd` is bad, but executing arbitrary commands is a disaster.

This isn't rocket science. An average attacker can exploit this mistake to see the contents of files they shouldn't, overwrite existing files and run system commands. Writing code like the above is like giving shell access to anyone who asks. And yet it's such a common mistake.

Coding for security

Perl's `open` function isn't the only place where you can go wrong. Any function or operator that passes input via the shell requires careful attention, as it may contain *shell meta-characters*. Assuming you can't just avoid all such functions and operators, the only way to ensure your code is safe is to *never trust input from the user*.

Fortunately this isn't too hard, and can be done without too much effort. If we know what characters a field is allowed to have, we can use a regular expression to make sure that only these characters are used:

```
#!/usr/bin/perl -w
use strict;
use CGI;

my $filename = CGI->param('file');

unless ($filename =~ /^([\w.-]+)$/) {
        die "Filename is not valid!\n";
}

# Filename is okay (only contains A-Z, a-z, 0-9,  _, . and -)

open(FILE, "<", "/home/test/$filename")
      or die "Failed to open /home/test/$filename for reading: $!";

# print out contents of requested file
print <FILE>;
```

It is always better to specify what is allowed, rather than what is not allowed. This is because it's much easier to modify your expression to allow a few extra characters if necessary, whereas it is almost impossible to be sure that you've listed *all* the potentially bad characters.

However, even if we're careful, we can still make mistakes. Wouldn't it be nice if Perl could provide some extra level of security to ensure that we don't use untrusted input by accident? It can, by using *taint mode*.

Taint checking

It's always important that we validate our input, and this is particularly true if we're working in a security sensitive context. Unfortunately it's easy to forget our validation steps, even if you are programming defensively.

To help prevent this; Perl has a *Taint mode*. Taint mode enforces the following rule:

> You may not use data derived from outside your program to affect something else outside your program -- at least, not by accident.

Taint mode achieves its aim by marking all data that comes from external sources as *tainted*. This data will then be considered unsuitable for certain operations:

- Executing system commands
- Modifying files
- Modifying directories
- Modifying processes
- Invoking any shell
- Performing a match in a regular expression using the `(?{ ... })` construct
- Executing code using string eval

Attempting to use tainted data for any of these operations results in an exception:

> Insecure dependency in open while running with -T switch at insecure.pl line 7.

Tainted data is communicable. Thus the result of any expression containing tainted data is also considered tainted.

Turning on taint

Taint mode automatically enabled when Perl detects that it's running with differing real and effective user or group ids -- which most commonly occurs when the program is running setid.

Taint mode can also be explicitly turned on by using the `-T` switch on the shebang line or command line.

```
#!/usr/bin/perl -wT       # Taint mode is enabled
```

It's highly recommended that taint mode be enabled for any program that's running on behalf of someone else, such as a CGI script or a daemon that accepts connections from the outside world. Once taint checks are enabled, they cannot be turned off.

Using taint checks is often a good idea even when we're not in a security-sensitive context. This is because it strongly encourages the good programming (and security) practice of checking incoming data before using it.

Untainting your data

The only way to clear the taint flag on your data is to use a capturing regular expression on it.

```
($clean_filename) = ($filename =~ /^([\w.-]+)$/);

if (not defined $clean_filename) {
        die "Filename is not valid!\n";
}

# Filename is okay (only contains A-Z, a-z, _, . and -)
```

The contents of the special variables `$1`, `$2`, (and so on) are also considered clean, but it's *strongly* recommended that you use the list-capturing syntax shown above. `$1`, `$2` can be set to indeterminate-yet-clean values if your regular expression fails, whereas a list-capturing syntax guarantees `$clean_filename` will be undefined on failure.

Passing your data through a regular expression does not mean that it's safe to use. However it should force you to think about it first. There's nothing to stop you from bulk-untainting data with an expression like `/(.*)/s`, but doing so is extremely trusting of your data, and certainly not recommended.

Dangerous environment variables

In addition to data our program receives while running, we also have to be aware of environment variables that can be set. Taint mode requires that each of these be either empty or untainted before they may be used.

- `PATH` - the directories searched when finding external executables.
- `IFS` - Internal Field Separator; the characters used for word splitting after expansion.
- `CDPATH` - a set of paths first searched by `cd` when changing directory with a relative path.
- `ENV` - the location of a file containing commands to execute upon shell invocation.
- `BASH_ENV` - similar to `ENV` but only comes into effect when bash is started non-interactively (eg. to run a shell script).
- `PERL5SHELL` (Windows only) - The shell that Perl will use to invoke when calling system commands. This is only checked for taintedness in Perl 5.8.9 and above.

Not all of these are used by all shells, but Perl will err on the side of caution and check them all regardless. If any are set, and we attempt to perform an operation which makes use of them, Perl will throw an exception:

```
Insecure $ENV{ENV} while running with -T switch at insecure.pl line 4.
```

The best way to avoid encountering these errors is to set these values yourself. For the most part this means the start of your script will look similar to:

```
#!/usr/bin/perl -wT
use strict;

delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
$ENV{PATH} = "/usr/bin/:/usr/local/bin";
```

At the very least you should make sure that any script running in taint mode sets its own `$ENV{PATH}`.

PERL5LIB, PERLLIB, PERL5OPT

The `PERL5LIB` and `PERLLIB` environment variables can be set to tell the perl interpreter where to look for Perl modules (before it looks in the standard library and current directory). These can be used instead of including `use lib "path/to/modules"` in your code.

The `PERL5OPT` environment variable can be set to tell the perl interpreter which command-line options to run with. These consist of `-[DIMUdmtw]` switches.

These environment variables are silently ignored by Perl when taint checking is in effect.

Set-user-id Perl programs

`suidperl`, which allows Perl programs to run with elevated privileges has regularly been the cause of security problems for Perl. In August 2000 a root shell exploit was discovered. This was consequently fixed, however further security vulnerabilities are always possible.

`suidperl` is neither built or nor installed by default, and may be removed from later version of Perl. It is recommended that you use dedicated, single-purpose tools such as `sudo` instead of `suidperl` where possible.

You can learn more about running setuid and setgid programs safely in Perl Training Australia's security notes that can be found at http://perltraining.com.au/courses/perlsec.html .

Chapter summary

This chapter covered using Perl's *taint mode* to help us ensure that we always validate input from external sources. Taint mode does not trust any information from external sources and thus insists that environment variables are cleaned before they are used.

Chapter 12. Logfile processing and monitoring

In this chapter...

This chapter covers some of Perl's modules which make working with log files easier.

Tailing files

Perl is often used to process log files, sometimes even while those log files are being written. `File::Tail` makes this task easy.

```
use File::Tail;

my $file = File::Tail->new("/var/log/apache/access.log");

while ( defined( my $line = $file->read() )) {
        # do something with the line
}
```

`File::Tail` does its best to ensure that it does not "busy-wait" on a file that has little traffic. Further, if the file does not change for some time, `File::Tail` will check to make sure that it's still there and hasn't been *rolled-over* to a new file. If this has occurred it will re-open the original file name for you.

Optional arguments

`File::Tail` can be given a number of arguments upon creation to change how it performs. Some of these are listed below:

name

  The name of the file to open.

interval

  The initial time to wait between checks to see if new data has been written to the file. The default value is 10 seconds.

maxinterval

  The maximum number of seconds that will be spend sleeping between checks to the file for new input. Each time `File::Tail` reads new data it counts the number of new lines and divides that by the time it just waited. This is used as the average time before new data is used as the interval to wait, so long as this interval is not greater than `maxinterval`. By default this is 60 (as in `File::Tail` will never wait for more than 60 seconds to check the file).

adjustafter

  The resistance to increasing the wait interval upwards. The default is 10, so `File::Tail` will wait for the current interval 10 times before adjusting the interval upwards.

resetafter

> The number of seconds after the last change that `File::Tail` should wait before checking to see if the file has been closed and reopened. The default is `adjustafter*maxinterval`.

We use these arguments as follows:

```
use File::Tail;

my $file = File::Tail->new(
        name        => "/var/log/apache/access.log",
        maxinterval => 60,
        adjustafter => 10,
);

while ( defined( my $line = $file->read() )) {
        # print the line out
        print $line;
}
```

In most cases, the defaults should work fine, so you should only adjust them if `File::Tail` is not responsive enough, or is causing undue load on your system.

File::Tail::App

`File::Tail` has one major limitation, if your program halts for some reason there is no good way to resume reading from where you got up to. If this is a requirement of your project you may want to look at `File::Tail::App`.

```
use Unix::PID '/var/run/logfile_app.pid';
use File::Tail::App qw(tail_app);

tail_app({
        new => [
                name        => '/var/log/apache/access.log',
                interval    => 1,
        ],
        lastrun_file => 'logfile_app.lastrun',
        do_md5_check => 1,
        line_handler => \&process_line,
});

sub process_line {
        my ($line) = @_;
        # do something with the line
}
```

`Unix::PID` records our process' PID in the given file, or exits with an error if the file already contains the PID of a running process. This ensures our process isn't running twice, and makes it easier to locate our long-running process if we need to stop or restart it.

`lastrun_file` is a scratch-pad to which our process can record details of where its up to. This means that if the process is terminated unexpectedly, it will be able to seek to the correct place in the log file when it next runs. `File::Tail::App` checks to see if the file has changed drastically since the lastrun information written - such as being truncated - and starts at the beginning if so.

`do_md5_check` records a MD5 sum on a small part of data at the beginning of the file. If this value changes between invocations of your program, then file processing will start at the beginning of the file regardless of the value in `lastrun_file`.

`line_handler` is given a reference to the subroutine we wish to use to handle each line, in this case `process_line`. If this callback is not specified, then `File::Tail::App` simply prints each line.

Exercises

Your instructor will tell you which file to use as your input for these exercises.

1. Use `File::Tail` to print out each line in the given file as it is generated. You may find it useful to set `interval => 1` for more responsive results.
2. Run your program. Notes will be printed to the file before and after it has been rotated. Make sure that `File::Tail` is correctly handling rotated files. An answer can be found in `exercises/answers/file_tail.pl`.
3. Use `File::Tail::App` to print out each line in the given file as it is generated. You can skip the `use Unix::PID` line from the example; if you do use it, make sure you try to write to a file in your own directory, and not in `/var/run`.

   Once you're happy that your program is working, stop it from running. Run it a second time and check that it starts from where it left off.

Interesting data

A common task for a Perl program is to watch a logfile for interesting lines of data. These may be warnings or errors, or just things entirely out of the ordinary. You could be tempted to write a program and specify what the interesting lines look like, and this works very well if you're looking for accesses to a particular file, or connections from a particular machine.

However in the more general case of show me *all* the "interesting" data that's written to a file, specifying regexps for that "interesting" data becomes more difficult. Let's take the example of a program that watches the Unix `syslog` file. If a line is written to `syslog` that you've never ever seen before, it's probably very interesting and unusual, but if you've never ever seen it before, your regexp probably won't catch it.

A much better technique is to specify lines which are *boring*. For example, the DNS daemon on many systems will report about problems with other people's servers. While this may be useful to determine why a particular name is not resolving (or resolving strangely), it's not something we can usually control or care about. We may ignore such lines with:

```
# Regular expressions of boring data
my @boring = (
’named\[[0-9]+\]: bad referral’,
’named\[[0-9]+\]: ns_resp: query\(.*\) All possible A RR lame’,
’named\[[0-9]+\]: ns_resp: query\(.*\) No possible A RRs’,
’named\[[0-9]+\]: ns_forw: query\(.*\) All possible A RR’s lame’,
’named\[[0-9]+\]: sysquery: query\(.*\) All possible A RR’s lame’,
’named\[[0-9]+\]: .* NS points to CNAME’,
’named\[[0-9]+\]: unrelated additional info .* type A from’,
);

# Build one big regular expression to match all above
my $boring_re = "(?:". join(")|(?:", @boring). ")";
$boring_re = qr/$boring_re/o;
```

If we use `logcheck` or a similar program which already has regular expressions to cover all the boring cases we can just walk through those rather than including them into our file:

```
my @boring;

# Get regular expressions from logcheck
foreach my $file ( glob("/etc/logcheck/ignore.d.paranoid/*" ) {

        # Skip files we can’t read
        open(RE, "<", $file) or next;

        push @boring, <RE>;
}

# Build one big regular expression to match all above
chomp @boring;
my $boring_re = "(?:". join(")|(?:", grep({$_}, @boring)). ")";
$boring_re = qr/$boring_re/;
```

Once we have a regular expression which can help us filter out the boring messages, we can then do something useful with the rest:

```
use File::Tail;

my $file=File::Tail->new("/var/log/syslog");

while ( defined( my $line = $file->read() )) {

        # Skip if the line looks boring
        next if $line =~ /$boring_re/o;

        # Do something useful here.
        print $line;
}
```

This being Perl, we can do more useful things with the interesting lines than just print them out. We could e-mail them to an administrator (encrypted first, if we prefer), announce them on an IRC channel, or send them via instant message to whoever is responsible for monitoring our machine that day.

Parsing Apache Logfiles

Once we can tail a file, we may find it useful to parse the contents. There are a great many modules on CPAN that allow us to parse logs of certain formats. A common example is looking through Apache log files, for which we can use `Parse::AccessLogEntry`.

```
use File::Tail::App;
use Parse::AccessLogEntry;

my $parser = Parse::AccessLogEntry->new();
tail_app({
        new => [
                name        => ’/var/log/apache/access.log’,
                interval    => 5,
        ],
        lastrun_file => ’logfile_app.lastrun’,
        do_md5_check => 1,
        line_handler => \&process_line,
});
```

```
sub process_line {
        my ($line) = @_;

        my $contents = $parser->parse($line);

        print "Host: $contents->{host} ";
        print "Date: $contents->{date} ";
        print "File: $contents->{file} ";
}
```

Generating reports from logfiles with Logfile

The `Logfile` module can be used to generate simple reports for a variety of different web log types. For example we can find out the top 5 most popular web pages on our site for the time period our log covers with:

```
use Logfile::Apache;

my $logfile = new Logfile::Apache(
        File => '/var/log/apache/access-pta.log',
        Group => [qw(File)],
);

$logfile->report(
        Group => "File",
        Sort  => "Records",
        Top   => 5,
);
```

which returns:

```
File                                Records
===========================================
/tips/index                    1659 17.70%
/pta                            851  9.08%
/favicon                        617  6.58%
/images/logo                    486  5.18%
/images/vcss                    484  5.16%
```

As we have two index pages in our `/tips/` directory: `index.html` and `index.atom` these have been aggregated into the one record. We can also see what files were most popular by bytes downloaded as well as their overall popularity:

```
use Logfile::Apache;

my $logfile = new Logfile::Apache(
        File => '/var/log/apache/access-pta.log',
        Group => [qw(File Bytes)],
);

$logfile->report(
        Group => "File",
        List  => [qw(Bytes Records)],
        Sort  => "Bytes",
        Top   => 5,
);
```

which returns:

```
File                                 Bytes          Records
============================================================
/notes/progperl            72057469 26.35%        91  0.97%
/talks/optimisation        28316164 10.35%        15  0.16%
/notes/perloo              23956582  8.76%        62  0.66%
/notes/perldbi             22112175  8.09%        45  0.48%
/notes/sysadmin            21887840  8.00%        51  0.54%
```

The first and third to fifth of these are our course notes (PDF) which can be downloaded from our site.

Logging with Perl

When writing programs, it can often be useful to send status updates to a log of some form. One should always consider the method of program invocation in such a case. For example, if your program is going to be run as a daemon process, then it makes sense for all information to go into a log file. On the other hand, if your program is going to be called as an application by a user, then it's important to share that important information with the user (potentially as well as logging it) so that the user has all the information they need about the program state.

Really simple logging

The easiest way to write a log in Perl is to append to a file:

```
use IO::Handle;
use autodie;

# Open file for appending, turn off buffering
open(my $log_fh, ">>", "my_app.log");
$log->autoflush(1);

...

print {$log_fh} "Interesting event happened here";
```

This is fine in most cases, but it will make your life *very* difficult if you want to change how logging is done in the future, such as logging to a database rather than a file.

A much better method is to write a subroutine for logging. This is not only easier to call, but gives you the flexibiliy of updating a single routine to change how all your logging is done. The following code provides a minmal example, which includes date-stamps at the start of each line.

```
    use IO::Handle;
    use constant LOGFILE => '/var/log/my_app.log';

    {
my $log_fh;     # This variable is persistent, but only
                # accessible to the subroutine below.

sub mylog {
    use autodie;

    if (not $log_fh) {
        open ($log_fh, '>>', LOGFILE);
        $log_fh->autoflush(1);
    }
    my $date = localtime();

    print {$log_fh} "[$date] @_\n";

}

    }
```

In Perl 5.10, the `state` and `say` keywords are available, which makes this even easier:

```
    use IO::Handle;
    use constant LOGFILE => '/var/log/my_app.log';

    sub mylog {
use autodie;
use feature qw(say state);

state $log_fh;  # State variables are persistent

if (not $log_fh) {
    open ($log_fh, '>>', LOGFILE);
    $log_fh->autoflush(1);
}

my $date = localtime();

say {$log_fh} "[$date] @_";

    }
```

On systems with native appending support (which includes most Unix systems writing to local disks), these simple logging routines can be used without any sort of locking, provided that writes remain small.

For more complex systems, it's recommended to use a pre-built logging system, such as `Log4perl` (described below).

Log4perl

There are a whole range of more advanced logging options. One very popular heavy-weight option is `Log::Log4perl` which implements Java's `log4j` interface. This allows you to change the logging behaviour of your application without restarting your code. `Log4perl` supports graduated logging (error, warning, info, etc), and logging to differnet destinations, including files, sockets, e-mail, RRDtool, and user-defined interfaces.

You can learn more about `Log::Log4perl` at http://search.cpan.org/perldoc?Log::Log4perl and also in the perl.com article at `http://www.perl.com/pub/a/2002/09/11/log4perl.html`.

Logging to Syslog

If you are writing a system daemon, or similar tool, you may wish to add your log messages to your system's log. On Unix systems this is the syslog file often found in `/var/log/syslog`. On Windows systems this is the event log. Perl comes with a standard module to allow you to write messages to the Unix Syslog: `Sys::Syslog`.

```
    use Sys::Syslog;
```

```
openlog("perl/messenger", "perror, pid", LOG_USER);
syslog(LOG_INFO, "connect: Connection closed unexpectedly");
```

This module is designed to provide a very similar interface as the C libraries. The first argument to `openlog` is the program identifier, the second is a set of options for how to handle calls to `syslog`. The final argument is the facility. `LOG_USER` in this case is for any generic user-level messages. If this were a mail, cron or ntp error though, there are other facility options for those.

`syslog` takes a priority for the message, and the message to print. This module can also do a whole lot more. Read `perldoc Sys::Syslog` for more.

Chapter summary

- `File::Tail` and `File::Tail::App` provide a way to process changing files.
- Judicious use of regular expressions can allow us to avoid dealing with boring data.
- We can use `Parse::AccessLogEntry` to process Apache log files.
- `Logfile` allows us to get basic reports on log files.
- A simple file-appender in a subroutine can provide a quick-and-simple logging solution.
- The `Log::Log4perl` module can provide very flexible and extensible logging.
- The `Sys::Syslog` module can be used to write to syslog.

Chapter 13. Interacting with network services

In this chapter...

As well as handling the sending of email, Perl is a great tool for working with network services. Be these instant messaging services such as IRC and AIM, using voice synthesis engines such as festival, scraping web pages or talking to LDAP services, Perl can do it. Perl can also do much, much more. This chapter covers some of these ideas.

For a detailed discussion on network programming with Perl, consult `perldoc perlipc` .

Sending data to IRC

Whether you're dealing with interesting lines from log files, tracking changes on a wiki, or monitoring a repository of source code, IRC bots are a popular choice for reporting information. The prevalence of instant messaging and the number of clients which now handle IRC makes an excellent way to distribute information between a large number of users.

Perl's `Net::IRC` module can be used to connect to IRC, send and receive messages, and perform other tasks. Here's a simple example:

```
#!/usr/bin/perl -w
use strict;

use Net::IRC;
use constant CHANNEL    => '#Syslog';

# Setup connection
my $irc = Net::IRC->new;
my $connection = $irc->newconn(
        Nick    => "ReportBot",
        Server  => "irc.example.com",
        Ircname => "IRC Reporting Bot",
) or die "Can't connect";

# Connect and report on status
$connection->join(CHANNEL) or die "Can't join";
$connection->privmsg(CHANNEL,"Tailing syslog messages");

# At this stage use $connection to report as required.
# For example combined with syslog processing from Logfiles chapter

while ( defined( my $line = $file->read() )) {

        # The following line clears any pending messages for
        # the bot; in our case they're just ignored.
        $connection->do_one_loop();

        next if $line =~ /$boring_re/o;

        $connection->privmsg(CHANNEL, $line);
}
```

Event driven services

The above code includes the strange line:

```
$connection->do_one_loop();
```

This line tells `Net::IRC` to process any waiting messages and events, and is essential to avoid us queueing up data on our IRC connection that never gets handled. It's essential for programs such as ours where `Net::IRC` isn't the main loop, but `File::Tail` is.

In our simple example we take the default action on all events (which is usually to ignore them), but we could have code run when particular actions are noticed (such as a user entering the channel, or a particular message being sent). We demonstrate some examples of call-backs in our discussion on AIM/ICQ below.

Sending an AOL instant message

IRC messages are great if the channel is quiet. However if the channel gets busy important messages could be missed. An alternative is to use something like AOL's Instant Messaging service (AIM) or ICQ. Both of these use the OSCAR protocol, and we can use the `Net::OSCAR` module to interface with this.

```
use strict;
use Net::OSCAR qw(:standard);
use File::Tail;

use constant USERNAME => "example";             # Bot username
use constant PASSWORD => "secret";
use constant SYSADMIN => "my_aim_username";     # Human username

my $file = File::Tail->new("/var/log/example.log");

my $oscar     = Net::OSCAR->new();
my $logged_in = 0;

# Set some call-backs to make our lives easier
$oscar->set_callback_signon_done( sub { $logged_in = 1 } );

$oscar->signon(
        screenname => USERNAME,
        password   => PASSWORD,
) or die "Failed to connect";

# A timeout of -1 means "wait forever" until events occur.
# This means we'll do the minimum amount of processing to
# login.
$oscar->timeout(-1);

# Wait until we're logged in.
while(not $logged_in) {
        $oscar->do_one_loop();
}

# Now reset our timeout to 0.01 seconds, so we don't
# wait too long while reading our file.
$oscar->timeout(0.01);

# Now that we're connected, we'll just copy lines
# from our logfile to our remote user as we see them.
```

```
while ( defined( my $line = $file->read() )) {
        $oscar->send_im(SYSADMIN, $line);
        $oscar->do_one_loop();
}
```

Call-backs

In the above code we register a call-back with our `$oscar` object. The `set_callback_signon_done` method takes a reference to a subroutine as its argument. In our example we've supplied an anonymous subroutine that sets a flag, but we can also create a reference to a subroutine:

```
 $oscar->set_callback_signon_done ( \&signon_done );

 # then later ...

 sub signon_done {
        print "Logged in!\n";
 }
```

The `Net::OSCAR` module allows for a wide variety of callbacks to be set, on anything from buddies logging in and out, to messages and chat-invites being received.

Sending data to a speech engine

With the amount of visual data we have to deal with from day to day, sometimes it helps to use a different channel to deal with really important information. Other times, it's just easier to sit back and listen to a report, than to read it yourself. In any case you can use the `Speech::Synthesis` module to help fulfil your aims.

This module provides access to a number of engines: SAPI4, SAPI5 and MSAgent (all Win32 only), MacSpeech (OS X only) and Festival.

```
use Speech::Synthesis;
my $engine = 'Festival';

my $ss = Speech::Synthesis->new(
        engine   => "Festival",
        language => "en_AU",
        voice    => "rab_diphone",
);

$ss->speak("All your base are belong to us.");
```

Web browsing and scraping

Perl is all about making our lives easier, and a lot of this is about doing our work for us. Well, wouldn't it be great if there was a Perl module to do our web browsing? It turns out that there is. `WWW::Mechanize`.

`WWW::Mechanize` (or Mech, as it is commonly known) allows you to automate interaction with websites. It supports fetching pages, following links, submitting forms, and much more.

The following example goes to http://search.cpan.org/ and performs a module search. It then locates all the module links on the first page, and displays their names and URLs.

```
#!/usr/bin/perl -w
use strict;
use WWW::Mechanize;

# Get our argument from the command line, or use
# 'Acme' as a default

my $query = $ARGV[0] || 'Acme';

# Create our Mechanize agent.

my $mech = WWW::Mechanize->new();

# Get our page

$mech->get('http://search.cpan.org/');

# Find our query form (named f), fill it in, and submit

$mech->form_name('f');
$mech->field('query', $query);
$mech->submit;

my @links = $mech->links;

# All our modules end in a ".pm" or ".pod" extension.

my @module_links = grep { $_->url =~ /\.(pm|pod)$/ } @links;

# Walk though each of our links and print the text and url.

foreach my $link (@module_links) {
        my $text = $link->text;
        my $url  = $link->url;
        print "$text\n\t$url\n\n";
}
```

When run with a command-line argument of `Quantum` the following results are produced (truncated for space):

```
Quantum::Random
        /author/FOX/Quantum-Random-0.01/lib/Quantum/Random.pod

Acme::MetaSyntactic::quantum
        /author/BOOK/Acme-MetaSyntactic-0.83/lib/Acme/MetaSyntactic/quantum.pm

Quantum::Entanglement
        /author/AJGOUGH/Quantum-Entanglement-0.32/Entanglement.pm

Quantum::Superpositions
        /author/LEMBARK/Quantum-Superpositions-2.02/lib/Quantum/Superpositions.pm
```

The `WWW::Mechanize` class provides a very rich interface, allowing one to set the useragent string, handle cookies, and fill in forms.

You can learn more about `WWW::Mechanize` on http://search.cpan.org/ and searching for `WWW::Mechanize`.

Working with LDAP

LDAP (Light-weight Directory Access Protocol) is the de-facto internet directory standard. It allows users to locate organisations, individuals and other resources (such as files and devices) from an internet or intranet directory server. It is supported by many companies including Sun, Microsoft, IBM and Novell.

Perl's `Net::LDAP` module allows you to access an existing LDAP server through Perl. It can be used to search directories as well as add, delete and modify entries. This section assumes some knowledge of the LDAP protocol.

Connecting

Using `Net::LDAP` to connect to our LDAP server is just a matter of creating our object and binding.

```
use Net::LDAP;

my $ldap = Net::LDAP->new(
        'ldap.perltraining.com',
        onerror => 'die',
);

$ldap->bind(
        'cn=root, o=Perl Training Australia, c=AU',
        password => $password,
) ;
```

Searching

To search for an entry we just create our search pattern and search. It's always a good idea to check whether our search was successful, as otherwise it may appear that our search term is not available when instead there was an error.

```
# Perform search
my $results = $ldap->search(
        filter => "(&(sn=Fenwick) (o=Perl Training Australia))",
);

# Handle errors
if ($results->code) {
        die $results->error;
}

# Dump the contents of each entry returned
foreach my $result ($results->entries) {
        $result->dump;
}

# End session.
$ldap->unbind;
```

Adding

To add an entry we can add in all the details in one go, or add in the mere basics and then modify the object.

```
my $result = $ldap->add(
        'cn=Paul Fenwick, o=Perl Training Australia, c=AU',
        attr => [
                'cn'   => ['Paul Fenwick', 'Paul'],
                'sn'   => 'Fenwick',
                'mail' => 'contact@perltraining.com.au',
                'objectclass' =>  [
                        'person',
                        'trainer',
                        'author',
                ],
        ],
);

$ldap->unbind;
```

Modifying

Modifying entries is as easy and searching for the entry we want to change, and making those changes.

```
# First find the entry (gives us the DN)
my $results = $ldap->search(
        filter => "(&(cn=Paul Fenwick) (o=Perl Training Australia))",
        sizelimit => 1,
);

# Handle errors
if ($results->code) {
        die "Failed to add entry: ", $results->error;
}

# If no error, then we should only have one result
# Ask for the first entry.
my $entry= $results=>entry(0);

$ldap->modify(
        $entry,
        changes => [
                add     => [ objectclass => 'director' ],
                replace => [ mail        => 'pjf@perltraining.com.au' ],
                delete  => [ objectclass => 'author' ],
        ]
);

$ldap->unbind;
```

Chapter summary

This chapter has covered connecting to an IRC server, sending AIM messages, sending information through a voice synthesiser, searching CPAN for modules, and working with LDAP. Perl is capable of many more network services, and there are a great many modules available to help you achieve your goals.

Chapter 14. Further Resources

Online Resources

- PerlNet - The Australian Perl Portal - http://perl.net.au/
- The Perl Directory - http://perl.org/
- Comprehensive Perl Archive Network - http://search.cpan.org/
- Perl Mongers user groups - http://pm.org/
- PerlMonks - http://perlmonks.org/
- O'Reilly's Perl.com - http://perl.com/

Books

Perl Best Practices, Damian Conway, O'Reilly and Associates

Programming Perl, Larry Wall et al, O'Reilly and Associates

Perl for System Administration, David N. Blank-Edelman, O'Reilly and Associates

The Perl Cookbook, Tom Christiansen and Nathan Torkington, O'Reilly and Associates

Index