Intermediate Perl

Kirrily Robert

Paul Fenwick

Jacinta Richardson

Intermediate Perl

by Kirrily Robert, Paul Fenwick, and Jacinta Richardson

Open Publications License 1.0

This document is a revised and edited copy of the training notes originally created by Kirrily Robert and Netizen Pty Ltd. These revisions were made by Paul Fenwick and Jacinta Richardson.
Copies of the original training manuals can be 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 version 4.23 of Perl Training Australia's "Intermediate Perl" 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.

Doctor Damian Conway, who provides many of our advanced courses, is one of the three core Perl 6 language designers, and is one of the leading Perl experts worldwide. Damian was the winner of the 1998, 1999, and 2000 Larry Wall Awards for Best Practical Utility. He is a member of the technical committee for OSCON, a columnist for The Perl Journal, and author of the book "Object Oriented Perl".

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:	03 9354 6001
Fax:	03 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://www.perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058

Chapter 2. Introduction

Welcome to Perl Training Australia's *Intermediate Perl* training course. This is a two-day module in which we extend on the material covered in *Introduction to Perl* and explore the topics of references, file input and output, interacting with the operating system and using modules.

Credits

This course is based upon the Intermediate Perl training module written by Kirrily Robert of Netizen Pty Ltd.

Course outline

Day 1

- Revise assumed knowledge
- References and complex data structures
- Introduction to modules and packages
- Writing packages and modules
- Using Perl objects

Day 2

- Advanced regular expressions
- File I/O
- System interaction
- Bonus material

Assumed knowledge

This training module assumes the following prior knowledge and skills:

- Basic Perl fluency, including a familiarity with Perl variable types, functions and operators, conditional constructs, and basic regular expressions.

Module objectives

- Understand and use Perl references to create complex data structures and anonymous data structures.
- Understand how to use Perl modules and where to find them.
- Understand how to write basic Perl modules and how to use exporter to export symbols for later use.
- Understand how to use Perl objects.
- Understand how to capture data using regular expressions and work with greedines, multi-line data and back references.
- Be able to open files and directories to read and write data, using various techniques.
- Perform tests on files and directories.
- Open pipes to read or write data through another program.
- Use Perl functions to call system commands.
- Use Perl to interact with the file system, users, and processes.
- Understand the security implications of running system commands from Perl, and how to increase security.

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.

At the time of writing, the most recent stable release of Perl is 5.8.6, however older versions of Perl (particularly 5.6.1 and 5.005) are still common. Your instructor will inform you of any features which may not exist in earlier 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 appears *like this*


Notes and tips appear offset from the text like this.


Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.


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


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, unless otherwise stated. References to the 2nd edition will be shown in parentheses.

Logging into your account

However you're doing this course, you will have an account which you'll want to log into at this point. If you're doing a corporate course, this is most likely on a machine that you're familiar with. Otherwise, your instructor will give you details on how to do this.

In any case, you should find yourself at a Unix shell prompt. You should find that your account has an `exercises/` subdirectory, or your instructor will tell you how to set this up. The `exercises/` directory contains example scripts and answers that are referred to throughout these notes. If your Unix stills are rusty, you can find a cheat sheet in Appendix C of these notes.

Chapter 3. References and complex data structures

In this chapter...

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

Assumed knowledge

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

Introduction to references

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

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

The following sources also provide useful and comprehensive information about references:

- Chapter 8 (chapter 4, 2nd Ed) of the Camel book, and in **perldoc perlref**.
- Chapter 1 of Advanced Perl Programming (O'Reilly's Panther book).
- Tom Christiansen's FMTYEWTK (Far More Than You Ever Wanted To Know) tutorials available from the Perl website (http://www.perl.com/).

Uses for references

There are three main uses for Perl references.

Creating complex data structures

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

Passing arrays and hashes to subroutines and functions

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

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

printargs(@a1, @a2);

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

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

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

Object oriented Perl

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

Creating and dereferencing references

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

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

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

Note that all references are scalars, because they contain a single item of information: the memory address of the actual data. This is what a reference looks like if you print it out:

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

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

```
print ref($scalar_ref);          # prints SCALAR
print ref($array_ref);           # prints ARRAY
print ref($hash_ref);            # prints HASH
```

The `ref()` function is documented on page 773 (page 204, 2nd Ed) of the Camel book or in **perldoc -f ref**.

Dereferencing (getting at the actual data that a reference points to) is achieved by prepending the appropriate sigil to the name of the reference. For instance, if we have a hash reference `$hash_reference` we can dereference it by adding a percentage sign: `%$hash_reference`.

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

Here's one way to access array elements or slices, and hash elements:

```
print $$array_ref[0];           # prints the first element of the array
                                # referenced by $array_ref: a
print @$array_ref[1,2];         # prints an array slice: b, c
print $$hash_ref{'sky'};        # prints a hash element's value: blue
```

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

```
print $array_ref->[0];       # prints the first element of the array
                             # referenced by $array_ref: a
print $hash_ref->{'sky'};    # prints a hash element's value: blue
```

The notation here is exactly the same as selecting elements from an array or hash, except that an arrow is inserted between the variable name and the element to fetch. So where `$foo[1]` gets the first (ie, position 2) element from the array `@foo`, `$foo->[1]` gets the first element from the array pointed to by the reference `$foo`.

It's not possible to get an array or hash slice using arrow notation.

Taking an array slice of a single element from an array reference does not result in a warning from Perl, although it's certainly not recommended. Perl does however try to be helpful in this case and returns the scalar referred to by the array slice, rather than the length of the array slice which would be `1`.

```
my $value = @$array_ref[0];     # Oops, this should be $$array_ref[0];
print $value;                   # Prints 'a' as desired but is not obvious
```

Exercises

1. Create an array called `@friends`, and populate it with the name of some of your friends.
2. Create a reference to your array called `$friends_ref`. Using this reference, print the names of three of your friends.

Assigning through references

Assigning values to the underlying array or hash through a reference is much the same as accessing the value:

```
my @trees = qw/lemon orange grapefruit/;
my $treeref = \@trees;

$treeref->[3] = 'mandarin';
print "@trees";                 # prints "lemon orange grapefruit mandarin"

my %fruit = ( kumquat => "sour",
              orange => "sweet",
              lemon => "sour",
              mandarin => "sweet");

my $fruit = \%fruit;            # scalars, arrays and hashes can all
                                # have the same names if it makes sense

$fruit->{grapefruit} = "sour and sweet";
```

Passing multiple arrays/hashes as arguments

When we pass multiple arrays to a subroutine they are flattened out to form one large array.

```
my @colours = qw/red blue white green pink/;
my @chosen = qw/red white green/;

print_unchosen(@chosen, @colours);

sub print_unchosen {
        my (@chosen, @colours) = @_;

        # at this point @chosen contains:
        # (red white green red blue white green pink)
        # and @colours contains () - the empty list.
}
```

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

```
ref_print_unchosen(\@chosen, \@colours);

sub ref_print_unchosen {
        my ($chosen, $colours) = @_;

        print "Chosen list:\n";
        foreach (@$chosen) {
                print "$_\n";
        }
        print "Colour list:\n";
        foreach (@$colours) {
                print "$_\n";
        }
}
```

When we pass references into a subroutine we're allowing that subroutine full access to the structure that the reference refers to. All changes that the subroutine makes to that structure will remain after the subroutine has returned. If you wish to make a copy of the structure that the reference refers to and modify that locally, you can do the following:

```
sub ref_print_unchosen {
        my ($chosen, $colours) = @_;

        my @chosen = @$chosen;          # this @chosen is now a copy
        my @colours = @$colours;        # this @colours is now a copy
}
```

The above paragraph discusses a concept that is often referred to as *call by reference*. Typically when we call Perl subroutines we consider them to be called *by value*. Technically, however, this is incorrect.

In the case where we pass scalars into a subroutine, we usually `shift` them from `@_` or we copy the contents from `@_` into another list. However if we instead modify the contents of `@_` directly we will actually be modifying the contents of the variables given to the subroutine.

We don't recommend this practice, however, as it makes your code much harder for other people to maintain. It's much better to do something like the following:

```
($x, $y) = modify($x, $y);
```

If you do use this functionality, be careful, as it's a fatal error to attempt to modify a read-only value, such as a literal string.

Anonymous data structures

We can use anonymous data structures to create complex data structures without having to declare many temporary variables. Anonymous arrays are created by using square brackets instead of round ones. Anonymous hashes use curly braces instead of round ones.

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

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

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

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

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

# more useful yet, we can put these anon arrays straight into a hash:

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

The same technique can be used to create anonymous hashes:

```
# The old, two-step way:
my %hash = (
        a       =>   1,
        b       =>   2,
        c       =>   3
);
my $hash_ref = \%hash;

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

Data is pulled out of an anonymous data structure using the arrow notation:

```
my $value = $hash_ref->{a};
print $value;              # prints 1
```

Exercise

1. Change your previous program to initialise `$friends_ref` using an anonymous array constructor. You should no longer need your original `@friends` array. Test that your program still works.

Complex data structures

References are most often used to create complex data structures. Since references are scalars, they can be used as values in both hashes and arrays. This makes it possible to create both deep and complex multi-dimensional data structures. These are covered more deeply in Appendix A.

The use of references in data structures allows you to create arrays of arrays, arrays of hashes, hashes of arrays and hashes of hashes. We saw an example of a hash of arrays in the previous section. Here is an example of an array of hashes:

```
my %alice = (
                name => "Alice Jane",
                age => 34,
                employeenumber => 12003,
);
my %bob = (
                name => "Bob Jane",
                age => 32,
                employeenumber => 12345,
);

my @employees = (
        \%alice,
        \%bob,
);

# to print out Alice's employee number:
print $employees[0]->{employeenumber};
```

```
# Or, to use anonymous data structures
my @employees2 = (
        {
                name => "Alice Jane",
                age => 34,
                employeenumber => 12003,
        },
        {
                name => "Bob Jane",
                age => 32,
                employeenumber => 12345,
        },
);

# to print out Bob's age:
print $employees2[1]->{age};
```

Exercises

1. Create data structures as follows:
   a. Create a hash called `%pizza_prices` which contains prices for small, medium and large pizzas.
   b. Create a hash called `%pasta_prices` which contains prices for small, medium and large serves of pasta.
   c. Create a hash called `%milkshake_prices` which contains prices for small, medium and large milkshakes.
   d. Create a hash called `%menu` containing references to the above hashes, so that given a type of food and a size you can find the price of it. Don't forget that your hash must contain both keys (the type of food), and values (a reference to the data structure containing the prices).
2. Write code which accepts food-type and size from the user and returns the price for the food.
3. Convert the above hash to use anonymous data structures (to get a hash of hashes) instead of the original three pizza, pasta and milkshake hashes, and modify your customer code appropriately.
4. Add a new element to your foods hash which contains the prices of salads. Rather than adding this in when you create the hash, instead add it separately.
5. Create a subroutine which can be passed a scalar and a hash reference. Check whether there is an element in the hash which has the scalar as its key. Hint: use `exists` for this.

Answers for the above exercises can be found in `exercises/answers/food.pl`.

Disambiguation and curly braces

Often in our code, we need to treat a reference as its underlying data structure. For a simple reference, this is easy; we prepend the reference with the appropriate sigil and it just works:

```
my $hashref = { a => 1, b => 2, c => 4, d => 8 };

foreach ( keys %$hashref ) {
        ...
}
```

What can cause us problems is when the reference isn't so simple. What should Perl do, in the following case?

```
my @result = @$array[0];
```

Does this mean:

- Find `@array`.
- Look up index `0`: `$array[0]`
- Turn that (`$array[0]`) into an array: `@$array[0]`

or:

- Find the array reference `$array`
- Treat that as an array: `@$array`
- Take an array slice with index `0`: `@$array[0]`

Perl does the latter, however if that is what we wanted then we should have written `$$array[0]`, as that explicitly returns a single (scalar) result.

We can force Perl to evaluate our expression as the first interpretation above by using curly braces. This allows us to clearly write:

```
my @result = @{$array[0]};
```

We can use `${...}`, `@{...}` or `%{...}` syntax to evaluate any expression and dereference the result.

Data::Dumper

Typically, to print out a data structure you have to understand its underlying structure and then write a number of loops to print it out in full. If the structure is relatively simple such as a hash of hashes of values, or even a hash of hash of arrays this isn't too difficult.

However, often data structures are very complex, and negotiating and printing these structures can be a tiresome exercise. It's also an unnecessary one, as all the hard work has already been done for you. To save you from having to write specialised printing code in every program for debugging purposes, there's a special library you may find useful called `Data::Dumper`.

`Data::Dumper` provides a function which takes Perl data structures and turns them into human readable strings representing the data with in them. It can be used just like this:

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

```
my %HoH = (
        Jacinta => {
                        age => 26,
                        favourite_colour => "blue",
                        sport => "swimming",
                        language => "Perl",
                    },


        Paul => {
                        age => 27,
                        favourite_colour => "green",
                        sport => "cycling",
                        language => "Perl",
                },
);

print Dumper \%HoH;
```

This will print out something similar to:

```
$VAR1 = {
          'Paul' => {
                      'language' => 'Perl',
                      'favourite_colour' => 'green',
                      'sport' => 'cycling',
                      'age' => 27
                    },
          'Jacinta' => {
                         'language' => 'Perl',
                         'favourite_colour' => 'blue',
                         'sport' => 'swimming',
                         'age' => 26
                       }
        };
```

Not only is this easy to read, but it's also perfectly valid Perl code. This means you can use `Data::Dumper` to easily give you a structure that you can paste into another program, or which can be 'serialised' to a file and re-created at a later date. `Data::Dumper` has a lot more uses beyond simple debugging.

`Dumper` expects to be given one or more references to data structures to dump. If `Dumper` is provided with a hash or array then every element of the array, or every key and value of the hash, will be considered a separate data structure, and dump separately. The results are not particularly useful:

```
# result of: print Dumper %HoH;
$VAR1 = 'Paul';
$VAR2 = {
          'language' => 'Perl',
          'favourite_colour' => 'green',
          'sport' => 'cycling',
          'age' => 27
        };
$VAR3 = 'Jacinta';
$VAR4 = {
          'language' => 'Perl',
          'favourite_colour' => 'blue',
          'sport' => 'swimming',
          'age' => 26
        };
```


You can read more about `Data::Dumper` on page 882 of the Camel book or in **perldoc Data::Dumper**.

Exercises

1. Use `Data::Dumper` to print out your data structures from the previous exercise.
2. Use **perldoc Data::Dumper** to read about `Data::Dumper`'s many options and configuration variables.

Chapter summary

- References are scalar data consisting of the memory address of a piece of Perl data, and can be used in arrays, hashes, and other places where you would use a normal scalar
- References can be used to create complex data structures, to pass multiple arrays or hashes to subroutines, and in object-oriented Perl.
- References are created by prepending a backslash to a variable name.
- References are dereferenced by replacing the name part of a variable name (eg `foo` in `$foo`) with a reference, for example replace `foo` with `$foo_ref` to get `$$foo_ref`
- References to arrays and hashes can also be dereferenced using the arrow `->` notation.
- References can be passed to subroutines as if they were scalars.
- References can be included in arrays or hashes as if they were scalars.
- Anonymous arrays can be made by using square brackets instead of round; anonymous hashes can be made by using curly brackets instead of round. These can be assigned directly to a reference, without any intermediate step.
- `Data::Dumper` allows complex data structures to be printed out verbatim without requiring full knowledge of the underlying data structure.

Chapter 4. External Files and Packages

In this chapter...

In this chapter we'll discuss how we can split our code into separate files. We'll discover Perl's concept of packages, and how we can use them to make our code more robust and flexible.

Splitting code between files

When writing small, independent programs, the code can usually be contained within a single file. However there are two common occurrences where we would like to have our programs span multiple files. When working on a large project, often with many developers, it can be very convenient to split a program into smaller files, each with a more specialised purpose. Alternatively, we may find ourselves working on many programs that share some common code base. This code can be placed into a separate file which can be shared across programs. This saves us time and effort, and means that bug-fixes and improvements need to be made only in a single location.

Require

Perl implements a number of mechanisms for loading code from external files. The most simplest of these is by using the `require` function:

```
require 'file.pl';
```

Perl is smart enough to make sure that the same file will not be included twice if it's required through the same specified name.

```
# The file is only included once in the following case:
require 'file.pl';
require 'file.pl';
```

Required files *must* end with a true value. This is usually achieved by having the final statement of the file being:

```
1;
```

Conflicts can occur if our included file declares subroutines with the same name as those that appear in our main program. In most circumstances the subroutine from the included file takes precedence, and a warning is given.

We will learn how to avoid these conflicts later in this chapter when we discuss the concept of *packages*.

The use of `require` has been largely deprecated by the introduction of modules and the `use` keyword. If you're writing a code library from scratch we recommend that you create it as a module. However, `require` is often found in legacy code and is a useful thing to understand.

Any code in the file (except for subroutines) will be executed immediately when the file is required. The `require` occurs at run-time, this means that Perl will not throw an error due to a missing file until that statement is reached, and any subroutines inside the file will not be accessible until after the `require`.

Variables declared with `my` are not shared between files, they are only visible inside the block or file where the declaration occurs. To share packages between files we use *package variables* which are covered later in this chapter.

The use of modules (which we will learn about later) allows for external files to be loaded at compile-time, rather than run-time.

Use strict and warnings

Perl pragmas, such as `strict` and `warnings` are lexically scoped. Just like variables declared with `my`, they last until the end of the enclosing block, file or eval.

This means that you can turn strict and warnings on in one file without it influencing other parts of your program. Thus, if you're dealing with legacy code, then your new libraries, modules and classes can be strict and warnings compliant even though the older code is not.

Example

The use of `require` is best shown by example. In the following we specify two files, `Greetings.pl` and `program.pl`. Both are valid Perl programs on their own, although in this case, `Greetings.pl` would just declare a variable and a subroutine, and then exit. As we do not intend to execute `Greetings.pl` on its own, it does not need to be made executable, or include a shebang line.

Our library code, to be included.

```
# Greetings.pl
# Provides the hello() subroutine, allowing for greetings
# in a variety of languages.  English is used as a default
# if no language is provided.

use strict;
use warnings;

my %greeting = (
        en      => "Hello",
        'en-au' => "G'day",
        fr      => "Bonjour",
        jp      => "Konichiwa",
        zh      => "Nihao",
);
```

```
sub hello {
        my $language = shift || "en";

        my $greeting = $greeting{$language}
                or die "Don't know how to greet in $language";

        return $greeting;
}

1;
```

Our program code.

```
# program.pl
# Uses the Greetings.pl file to provide another hello() subroutine
use strict;

# Get the contents from file.pl
require "Greetings.pl";

print "English: ",    hello("en"),   "\n";       # Prints "Hello"
print "Australian: ", hello("en-au"),"\n";       # Prints "G'day"
```

Exercises

1. Create a file called `MyTest.pl` Define at least two functions; `pass` and `fail` which print some amusing output. Make sure that it uses `strict`.
2. Test that your code compiles by running **perl -c MyTest.pl**. (The **-c** tells Perl to check your code).
3. Create a simple Perl script which requires `MyTest.pl` and calls the functions defined within.

Introduction to packages

The primary reason for breaking code into separate files is to improve maintainability. Smaller files are easier to work with, can be shared between multiple programs, and are suitable for dividing between members of large teams. However they also have their problems.

When working with a large project, the chances of naming conflicts increases. Two entirely different files may have two different subroutines with the same name; however it is only the last one loaded that will be used by Perl. Files from different projects may be re-used in new developments, and these may have considerable name clashes. Multiple files can also make it difficult to determine where subroutines are originally declared, which can make debugging difficult.

Perl's *packages* are designed to overcome these problems. Rather than just putting code into separate files, code can be placed into independent packages, each with its own namespace. By ensuring that package names remain unique, we also ensure that all subroutines and variables can remain unique and easily identifiable.

A single file can contain multiple packages, but convention dictates that each file contains a package of the same name. This makes it easy to quickly locate the code in any given package.

Writing a package in Perl is easy. We simply use the `package` keyword to change our current package. Any code executed from that point until the end of the current file or block is done so in the context of the new package.

```
# By declaring that all our code is in the "Greetings" package,
# we can be certain not to step on anyone else's toes, even if
# they have written a hello() subroutine.

package Greetings;

use strict;
use warnings;

my %greeting = (
        en      => "Hello",
        'en-au' => "G'day",
        fr      => "Bonjour",
        jp      => "Konichiwa",
        zh      => "Nihao",
);

sub hello {
my $language = shift || "en";

        my $greeting = $greeting{$language}
                or die "Don't know how to greet in $language";

        return $greeting;
}

1;
```

The package that you're in when the Perl interpreter starts (before you specify any package) is called `main`. Package declarations use the same rules as `my`, that is, it lasts until the end of the enclosing block, file, or eval. Here's an example:

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

sub hello { print "This is hello in the main package\n"; }

{
        package Foo;
        sub hello { print "This is hello in the Foo package\n"; }
}

# Here we're back in the main package again.

hello();        # main's hello
```

Perl convention states that package names (or each part of a package name, if it contains many parts) starts with a capital letter. Packages starting with lower-case are reserved for pragmas (such as `strict`).

The scoping operator

Being able to use packages to improve the maintainability of our code is important, but there's one important thing we have not yet covered. How do we use subroutines, variables, or filehandles from other packages?

Perl provides a *scoping operator* in the form of a pair of adjacent colons. The scoping operator allows us to refer to information inside other packages, and is usually pronounced "double-colon".

```
require "Greetings.pl"

# This calls the hello() subroutine in our main package,
# printing "Greetings Earthling".
print hello(),"\n";

# Greetings in English.
print Greetings::hello("en"),"\n";

# Greetings in Japanese.
print Greetings::hello("jp"),"\n";

sub hello {
        return "Greetings Earthling";
}
```

Calling subroutines like this is a perfectly acceptable alternative to exporting them into your own namespace (which we'll cover later). This makes it very clear where the called subroutine is located, and avoids any possibility of an existing subroutine clashing with that from another package.

Occasionally we may wish to change the value of a variable in another package. It should be very rare that we should need to do this, and it's not recommended you do so unless this is a documented feature of your package. However, in the case where we do need to do this, we use the scoping operator again.

```
use Carp;

# Turning on $Carp::Verbose makes carp() and croak() provide
# stack traces, making them identical to cluck() and confess().
# This is documented in 'perldoc Carp'.

$Carp::Verbose = 1;
```

There's a shorthand for accessing variables and subroutines in the `main` package, which is to use double-colon without a package name. This means that `$::foo` is the same as `$main::foo`.

When referring to a variable in another package, the sigil (punctuation denoting the variable type) always goes *before* the package name. Hence to get to the scalar `$bar` in the package `Foo`, we would write `$Foo::bar` and not `Foo::$bar`.

It is not possible to access lexically scoped variables (those created with `my`) in this way. Lexically scoped variables can *only* be accessed from their enclosing block.

Exercises

1. Print out the version of the `Cwd` module installed on your training server. The version number is in `$Cwd::VERSION`.
2. Look at the documentation for the `Carp` module using the **perldoc Carp** command. This is one of Perl's most frequently used modules.

Package variables and our

It is not possible to access lexically scoped variables (those created with `my`) outside of their enclosing block. This means that we need another way to create variables to make them globally accessible. These global variables are called *package variables*, and as their name suggests they live inside their current package. The preferred way to create package variables, under Perl 5.6.0 and above, is to declare them with the `our` statement. Of course, there are alternatives you can use with older version of Perl, which we also show here:

```
package Carp;

our $VERSION = '1.01';          # Preferred for Perl 5.6.0 and above

# use vars qw/$VERSION/;        # Preferred for older versions
# $VERSION = '1.01';

# $Carp::VERSION = '1.01';      # Acceptable but requires that we then
                                # always use this full name under strict
```

In all of the cases above, both our package and external code can access the variable using `$Carp::VERSION`.

Exercises

1. Change your `MyTest.pl` file to include a package name `MyTest`
2. Update your program to call the MyTest functions using the scoping operator.
3. Create a package variable `$pass_mark` using `our` inside `MyTest.pl` which defines an appropriate pass mark.
4. In your Perl script, create a loop which tests 10 random numbers for pass or fail with reference to the `$pass_mark` package variable. Print the appropriate `pass` or `fail` message.

Answers for the above exercises can be found in `exercises/answers/MyTest.pl` and `exercises/answers/packages.pl`.

Chapter summary

- A package is a separate namespace within Perl code.

- A file can have more than one package defined within it.
- The default package is `main`.
- We can get to subroutines and variables within packages by using the double-colon as a scoping operator for example `Foo::bar()` calls the `bar()` subroutine from the `Foo`
- To write a package, just write `package` *`package_name`* where you want the package to start.
- Package declarations last until the end of the enclosing block, file or eval (or until the next package statement).
- Package variables can be declared with the `our` keyword. This allows them to be accessed from inside other packages.
- The `require` keyword can be used to import the contents of other files for use in a program.
- Files which are included using `require` must end with a true value.

Chapter 5. Modules

In this chapter...

In this chapter we'll discuss modules from a user's standpoint. We'll find out what a module is, how they are named, and how to use them in our work.

In the remainder of the chapter, we will investigate how to write our own modules.

Module uses

Perl modules can do just about anything. In general, however, there are three main uses for modules:

- Changing how the rest of your program is interpreted. For example, to enforce good coding practices (`use strict`) or to allow you to write in other languages, such as Latin (`use Lingua::Romana::Perligata`), or to provide new language features (`use Switch`).
- To provide extra functions to do your work (`use Carp` or `use CGI qw/:standard/`).
- To make available new classes (`use HTML::Template` or `use Finance::Quote`) for object oriented programming.

Sometimes the boundaries are a little blurred. For example, the `CGI` module provides both a class and the option of extra subroutines, depending upon how you load it.

What is a module?

A module is a separate file containing Perl source code, which is loaded and executed at compile time. This means that when you write:

```
use CGI;
```

Perl looks for a file called `CGI.pm` (.pm for *Perl Module*), and upon finding it, loads it in and executes the code inside it, before looking at the rest of your program.

Sometimes you need to tell Perl where to look for your Perl modules, especially if some of them are installed in a non-standard place. Like many things in Perl, There's More Than One Way To Do It. Check out **perldoc -q library** for some of the ways to tell Perl where your modules are installed.

Sometimes you might choose to pass extra information to the module when you load it. Often this is to request the module create new subroutines in your namespace.

```
use CGI qw(:standard);
use File::Copy qw(copy);
```

Note the use of `qw()`, this is a list of words (in our case, just a single word). It's possible to pass many options to a module when you load it. In the case above, we're asking the `CGI` module for the `:standard` bundle of functions, and the `File::Copy` module for just the `copy` subroutine.

Each module has a different set of options (if any) that it will accept. You need to check the documentation of the module you're dealing with to which (if any) are applicable to your needs.

To find out what options exist on any given module read its documentation: **perldoc** ***module_name***.

Exercise

1. Using `File::Copy` make a copy of one of your files. If you're eager, ask the user which file to copy and what to name the copy.

Where does Perl look for modules?

Perl searches through a list of directories that are determined when the Perl interpretor is compiled. You can see this list (and all the other options Perl was compiled with), by using **perl -V**.

The list of directories which Perl searches for modules is stored in the special variable `@INC`. It's possible to change `@INC` so that Perl will search in other directories as well. This is important if you have installed your own private copy of some modules.

Of course, being Perl, there's more than one way to change `@INC`. Here are some of the ways to add to the list of directories inside `@INC`:

- Call Perl with the `-I` command-line switch with the location of the extra directory to search. For example:

  ```
  perl -I/path/to/libs
  ```

  This can be done either in the shebang line, or on the command-line.
- Use the `lib` pragma in your script to inform Perl of extra directories. For example:

  ```
  use lib "/path/to/libs";
  ```
- Setting the `PERL5LIB` environment variable with a colon-separated list of directories to search. Note that if your script is running with taint checks this environment variable is ignored.

Since `use` statements occur before regular Perl code is executed, modifying `@INC` directly usually does not have the desired effect.

Finding installed modules

Perl comes with many modules in its standard distribution. You can get a list of all of them by doing a **perldoc perlmodlib**. The Camel book describes the standard modules in chapters 31 and 32 (chapter 7, 2nd Ed).

Besides from the modules in the standard distribution, you can also see any other modules that were installed on your system by using **perldoc perllocal**. Generally this file only lists other modules that were installed by hand, or using one of the CPAN installers (more on this later).

> Modules installed through your operating system's packaging system may not appear in **perldoc perllocal**.

You can get more information on any module that you have installed by using **perldoc** ***module_name***. For example, **perldoc English** will give you information about the `English` module.

Most importantly, there's a great resource for finding modules called the *Comprehensive Perl Archive Network*, or *CPAN* for short. The CPAN website (http://www.cpan.org/) provides many ways of finding the modules you're after and browsing their documentation on-line. It's highly recommended that you become familiar with CPAN's search features, as many common problems have been solved and placed in CPAN modules.

Exercise

1. Open a web browser to CPAN's search site (http://search.cpan.org) and spend a few minutes browsing the categories provided.
2. Perform a search on CPAN for a problem domain of your choice. If you can't think of one, search on `CGI`, `XML` or `SOAP`.

Using CPAN modules

At the time of writing, CPAN provides more than 5,500 separate and freely available modules. This makes CPAN an excellent starting point when you wish to find modules 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. Some (such as the `CGI` or `DBI`) modules have become de-facto standards, whereas others may not be used by anyone except the module's author.

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. However, in almost all circumstances it's better to use or extend a suitable module from CPAN rather than trying to re-invent the wheel.

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

Most CPAN modules come with `README` and/or `INSTALL` files which tell you how to install the modules. However in almost every case, the process is the same:

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

If you install your module in a different directory than your other Perl modules you may have to use the `lib` pragma, mentioned in the previous section, to tell Perl where to find your files. Once a module is installed, you can use it just like any other Perl module.

The double-colon

Sometimes you'll see modules with double-colons in their names, like `Finance::Quote`, `Quantum::Superposition`, or `CGI::Fast`. The double-colon is a way of grouping similar modules together, in much the way that we use directories to group together similar files. You can think of everything before the double-colon as the category that the module fits into.

In fact, the file analogy is so true-to-life that when Perl searches for a module, it converts all double-colons to your directory separator and then looks for that when trying to find the appropriate file to load. So `Finance::Quote` looks for a file named `Quote.pm` in a directory called `Finance`. That two modules are in the same category doesn't necessarily mean that they're related in any way. For example, `Finance::Quote` and `Finance::QuoteHist` have very similar names, and their maintainers even enjoy very similar hobbies, they certainly have similar uses, but neither package shares any code in common with the other.

It's perfectly legal to have many double-colon separators in module names, so `Chicken::Bantam::SoftFeather::Pekin` is a perfectly valid module name.

Writing modules

Modules contain regular Perl code, and for most modules the vast majority of that code is in subroutines. Sometimes there are a few statements which initialise variables and other things before any of those subroutines are called, and those get executed immediately. The subroutines get compiled and tucked away for later use.

Besides from the code that's loaded and executed, two more special things happen. Firstly, if the last statement in the module did not evaluate to true, the Perl compiler throws an exception (usually halting your program before it even starts). This is so that a module could indicate that something went wrong, although in reality this feature is almost never used. Virtually any Perl module you care to look at will end with the statement `1;` to indicate successful loading.

The other thing that happens when a module is `use`d is that its `import` subroutine (if one exists) gets called with any directives that were specified on the `use` line. This is useful if you want to export functions or variables to the program that's using your module for functional programming but is almost never used (and very often discouraged) for object oriented programming.

As you've no doubt guessed by now, modules and packages often go hand-in-hand. We know how to use a module, but what are the rules on writing one? Well, the big one is this:

> A module is a file that contains a package of the same name.

That's it. So if you have a package called `Tree::Fruit::Citrus::Lime`, the file would be called `Tree/Fruit/Citrus/Lime.pm`, and you would use it with `use Tree::Fruit::Citrus::Lime;`.

A module can contain multiple packages if you desire. So even though the module is called `Chess::Piece`, it might also contain packages for `Chess::Piece::Knight` and `Chess::Piece::Bishop`. It's usually preferable for each package to have its own module, otherwise it can be confusing to your users how they can load a particular package.

When writing modules, it's important to make sure that they are well-named, and even more importantly that they won't clash with any current or future modules, particularly those available via CPAN. If you are writing a module for internal use only, you can start its name with `Local::` which is reserved for the purpose of avoiding module name clashes.


You can read more about writing modules in **perldoc perlmodlib**, and a little on pages 554-556 of the Camel book.

Use versus require

Perl offers several different ways to include code from one file into another. `use` is built on top of `require` and has the following differences:

- Files which are `used` are loaded and executed at compile-time, not run-time. This means that all subroutines, variables, and other structures will exist before your main code executes. It also means that you will immediately know about any files that Perl could not load.
- `use` allows for the import of variables and subroutines from the used package into the current one. This can make programming easier and more concise.
- Files called with `use` can take arguments. These arguments can be used to request special features that may be provided by some modules.

Both methods:

- Check for redundant loading, and will skip already loaded files.
- Raise an exception on failure to find, compile or execute the file.
- Translate `::` into your systems directory separator (covered more shortly).

Where possible `use` and Perl *modules* are preferred over `require`.

Warnings and strict

When your module is used by a script, whether or not it runs with warnings depends upon whether the calling script is running with warnings turned on. You can (and should) invoke the `use warnings` pragma to turn on warnings for your module without changing warnings for the calling script.

Your modules should always use strict.

```
use strict;
use warnings;
```

Exercise

This exercise will have you adapt your `MyTest.pl` code to become a module. There's a list at the end of this exercise of things to watch out for.

1. Create a directory named `lib`.
2. Move your `MyTest.pl` file into your `lib` directory and rename it to `MyTest.pm`.
3. Make sure `MyTest.pm` uses `strict` and `warnings`.
4. Test that your module has no syntax errors by running **perl -c MyTest.pm**.
5. Change your Perl script from before to `use` your module. Check that everything still works as you expect.

6. Add a print statement to your module (outside any subroutines). This should be printed when the module is loaded. Check that this is so.

Answers can be found in `exercises/answers/lib/MyTest.pm` and `exercises/answers/modules.pl`

Things to remember...

The above exercises can be completed without reference to the following list. However, if you're having problems, you may find your answer herein.

- A module is a file that contains a package of the same name.
- Perl modules must return a true value to indicate successful loading. (Put `1;` at the end of your module).
- To use a module stored in a different directory, add this directory to the `@INC` array. (Put `use lib 'path/to/modules/'` before the other `use` lines.
- To call a subroutine which is inside a module, you can access it via the double-colon. Eg: `MyModule::test();`

Exporting and importing subroutines

Writing your own `import` function for each and every module would be a tiresome and error-prone process. However, Perl comes with a module called `Exporter`, which provides a highly flexible interface with optimisations for the common case.

`Exporter` works by checking inside your module for three special data structures, which describe both what you wish to export, and how you wish to export them. These structures are:

- `@EXPORT` symbols to be exported into the user's name space by default
- `@EXPORT_OK` symbols which the user can request to be exported
- `%EXPORT_TAGS` that allows for bundles of symbols to be exported when the user requests a special export tag.

@ISA

To take advantage of `Exporter`'s `import` function we need to let Perl know that our package has a special relationship with the `Exporter` package. We do this by telling Perl that we *inherit* from `Exporter`. Our package and the rest of our program does not need to be written in an object oriented style for this to work.

Now when Perl goes looking for the `import` function it will first look in our package. If it can't be found there, Perl will look for a special array called `@ISA`. The contents of the `@ISA` array is interpreted as a list of parent classes, and each of these will be searched for the missing method.

To specify that this package is a sub-class of the Exporter module we include the following lines:

```
use Exporter;
our @ISA = qw(Exporter);
```

use base

An alternative to adding parent modules to `@ISA` yourself is to use the `base` pragma. This allows you to declare a derived class based upon the listed parent classes. Thus the two lines above becomes:

```
use base qw(Exporter);
```

The `base` pragma takes care of ensuring that the `Exporter` module is loaded.

The `base` pragma is available for all versions of Perl above 5.6.0.

An example

Here's an example of just using `@EXPORT` and `@EXPORT_OK`. Our hypothetical module, `People::Manage` is used for managing interpersonal relations.

```
package People::Manage;
use base qw(Exporter);
use vars qw(@EXPORT @EXPORT_OK);

@EXPORT     = qw(invite $name @friends %addresses);        # invite is a subroutine
@EXPORT_OK  = qw(&taunt $spouse @enemies %postcodes);  # so is taunt
```

The ampersand in front of subroutines is optional.

Exporting by default

Exporting your symbols by default, by populating the `@EXPORT` array, means that anyone using your module will receive these symbols without having to ask for them. This is generally considered to be bad style, and is sometimes referred to as 'polluting' the caller's namespace.

The reason this is considered to be bad style is that there is nothing in the `use` line to indicate that anything is being exported. A programmer who is not familiar with the module may inadvertently define their own subroutines or variables which clash with those that are exported. Likewise, a reviewer examining the code will not easily be able to determine from which module a given subroutine may have been exported, especially if many modules are used.

Using the `@EXPORT` array is highly discouraged.

Using `@EXPORT_OK` allows the user to choose which symbols they wish to bring into their name space. All other symbols can be accessed by using their full name, such as `People::Manage::invite()`, when required.

An example

Our module:

```
###### People/Manage.pm ###########
package People::Manage;         # create a package of the same name
use strict;
use warnings;
use base qw(Exporter);

# List out the things we wish to export
our @EXPORT_OK  = qw(invite $name @friends %addressbook
                     taunt $spouse @enemies @children $pet);

# Only package variables can be exported, as such all of these
# variables need to be declared with 'our' not 'my'.

our $name   = "Fred";
our $spouse = "Wilma";

our @children = qw(Pebbles);
our @friends  = qw(Barney Betty);
our $pet      = "Dino";
my $address = "301 CobbleStone Way, Bedrock";

our %addressbook = (
        Barney         => "303 Cobblestone Way, Bedrock",
        Betty          => "303 Cobblestone Way, Bedrock",
        "Barney's Mom" => "142 Boulder Ave, Granitetown",
);


sub invite {
        my ($friend, $date) = @_;
        return "Dear $friend,\n $spouse and I would love you to come to".
               "dinner at our place ($address) on $date.\n\n".
               "Yours sincerely, $name\n";
}

sub taunt {
        my ($enemy) = @_;
        return "Dear $enemy, my pet $pet has more brains than you.\n";
}

1;                              # module MUST end with something true
```

Our program:

```
###### dinner.pl ###########
#!/usr/bin/perl -w
use strict;
use People::Manage qw(invite %addressbook);     # only using a few things

# Invite some people over for dinner.

foreach my $person (keys %addressbook) {
        print invite($person,"next Tuesday");
}
```

Importing symbols

Once your module is written and it exports a few symbols, it's time to use it. This is done with the `use` command that we've seen with `strict` and other modules. We can load our module in three ways:

- `use People::Manage;` which imports all of the symbols stored in `@People::Manage::EXPORT`.
- `use People::Manage ();` which imports *none* of the symbols in either `@People::Manage::EXPORT` or `@People::Manage::EXPORT_OK`.
- `use People::Manage qw($name $spouse invite);` which imports all the listed symbols. If a symbol is mentioned which is not in either `@People::Manage::EXPORT` or `@People::Manage::EXPORT_OK` then a fatal error will occur.

Exercises

These exercises build on the previous exercises.

1. Change your MyTest.pm module to export the `pass` and `fail` symbols and import those into your script. Change your script to call `pass` and `fail` instead of their fully qualified names.
2. Change your module to export the `$pass_mark` variable and use that instead of its fully qualified name.

Exporting tags

If you wish to export groups of symbols that are related to each other, there is an `%EXPORT_TAGS` hash which provides this functionality. This can be used in the follow manner:

```
%EXPORT_TAGS = (family => [ qw/$name $spouse @children $pet/ ],
                social => [ qw/%address invite taunt @friends/ ],
);
```

Names which appear in `%EXPORT_TAGS` must also appear in `@EXPORT` or `@EXPORT_OK`. Tags themselves cannot be used in either export array.

Importing symbols through tags

Symbols grouped in tags can be imported normally, by specifying each symbol, or by using the tag provided. This is done by prepending the tag name with a colon:

```
use People::Manage qw/:family/;            # Family related information.
# use People::Manage qw/:social/;          # Social-related symbols.
# use People::Manage qw/:family :social/;  # Both
```

Exercise

1. In your `MyTest` module, create a tag which contains both subroutines and use that instead of specifying them both during the import.

Chapter summary

- A module is a separate file containing Perl source code.
- We can use modules by writing `use module_name;` before we want to start using it.
- Perl looks for modules in a list of directories that are determined when the Perl interpretor is compiled.
- Module names may contain double-colons (`::`) in their names such as `Finance::Quote`, these tell where Perl to look for a module (in this case in the `Finance/` directory.
- Modules can be used for class definitions or as libraries for common code.
- A module can contain multiple packages, but this is often a bad idea.
- It's often a good idea to put your own modules into the `Local` namespace.

Chapter 6. Using Perl objects

In this chapter...

While discussion of Object Oriented programming is beyond the scope of this course, a great many modules you may encounter while programming provide an object oriented interface. This chapter will teach you what you need to know to use these modules.


Perl Training Australia runs a two day course on Object Oriented Programming in Perl, for more information visit our website (http://www.perltraining.com.au/) or talk to your instructor during the break.

Objects in brief

An *object* is a collection of data (attributes) and subroutines (methods) that have been bundled into a single item. Objects often represent real-world concepts, or logical constructs. For example, an *invoice* object may have attributes representing the date posted, date date, amount payable, GST, vendor, and so on. The invoice may have various methods that allow for payment to be made, and possibly a payment to allow the invoice to be disputed.

An object in Perl is a reference to a specially prepared data structure. This structure may be a hash, an array, a scalar or something more complex. However, as the user of an object, we don't need to know (and should not care) what sort of structure is actually being used. What matters are the *methods* on the object, and how we can use them.

A Perl object is a *special* kind of reference because it also knows what class it belongs to. In other words, an object knows what kind of object it is.

Object orientation allows us to create *multiple* objects from the same class which can each store different information and behave differently according to that information. This makes it very easy for the users of those objects, as it makes the information easy to track and manipulate.

Using an object

To use a Perl module which provides an object oriented interface we `use` it without specifically importing any methods. For our examples we will use the `DBI` module, which allows us to interact with a number of databases, and is one of the most commonly used modules in Perl.

```
#!/usr/bin/perl -w
use strict;
use DBI;               # We can now create DBI objects.
```


To learn more about `DBI` read **perldoc DBI** and the `DBI` homepage (http://dbi.perl.org/).

> Perl Training Australia also runs a Database Programming with Perl course which you may find of interest. For more information visit our website (http://www.perltraining.com.au/) or talk to your instructor during the break.

Instantiating an object

To create a new object we call the constructor method on the *name* of the class. In many cases this method is called `new`, however with `DBI` it is called `connect`; as we get our database handle by *connecting* to a database.

```
use DBI;

# Create a DBI object (database connection handle)
my $dbh = DBI->connect($data_source, $username, $password);
```

By convention, our connected database object is called `$dbh`, for "database handle".

We can create a number of database handles (objects), with each connecting to different databases or with different usernames and passwords. We could also create a number of database handles connecting to the same database. This could potentially be useful if we wished to execute multiple SQL commands simulatenously, particularly if we're dealing with a clustered database system.

```
use DBI;

my $oracle_dbh   = DBI->connect($oracle_dsn, $oracleuser, $oraclepasswd);
my $postgres_dbh = DBI->connect($postgres_dsn, $postgresuser, $postgrespasswd);
my $mysql_dbh1   = DBI->connect($mysql_dsn, $mysqluser1, $mysqlpasswd1);
my $mysql_dbh2   = DBI->connect($mysql_dsn, $mysqluser2, $mysqlpasswd2);
```

Each of these objects represent a different database connection and we can call the other `DBI` methods on these objects from now on. Each object will remember which database it refers to without further work on behalf of the programmer.

Calling methods on an object

As we covered earlier, we can get at the contents of a normal reference by using the arrow operator:

```
$array_ref->[$index];          # Access array element via array reference
$hash_ref->{$key};             # Access array element via hash reference
```

It should come as no big surprise that Perl object methods (or functions, if you'd prefer) can be accessed the same way:

```
$object->method();             # Call method() on $object
```

In a specific case, we can call a method on one of our `DBI` objects as follows:

```
use DBI;

my $dbh = DBI->connect($data_source, $username, $password);

$dbh->do("UPDATE friends SET phone = '12345678' WHERE name = 'Jack'");
```

Destroying an object

When you no longer need an object you can let it go out of scope, just as when you no longer need any other Perl data structure. In some cases the documentation may recommend calling certain clean up functions. In the case of `DBI` it is considered polite to disconnect from the database.

```
$dbh->disconnect();
```

Chapter summary

- Perl objects are special references to Perl data structures which know which class they belong to.
- Object orientation allows us to create multiple objects from the same class to store different information.
- To use a Perl class we just `use` the module.
- To create an object we call the constructor method on the class.
- Many objects of the same class can be created.
- To call a method on an object we use the arrow operator.
- Objects are destroyed when they go out of scope.

Chapter 7. Advanced regular expressions

In this chapter...

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

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. Hence code similar to the following 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";
        }

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

In this code, should we have a line which doesn't match the regular expression for a valid date, this line will be printed to whichever file the previous valid line was printed to. This may 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 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 a 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'
/^([\w-]+)\s+(\d+)\s+(\w+)\s+(\w+)\s+(\d+)\s+(\w+\s+\d+\s+[\d:]+)\s+(.*)$/;
```

into this:

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

/
    ^                           # 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

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

```
while(<>) {
        chomp;

        my ($origin, date, page) = (/PATTERN/); # put your regexp here

        ...
}
```

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

   Write a regular expression which captures the request origin, the access date and requested page. Print this out for each access in the file.

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 in the Introduction to Perl course.

Table 7-1. 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

Meta character	Meaning
\u	Uppercase next character
\L	Lowercase until \E
\U	Uppercase until \E
\Q	Quote (disable) meta characters until \E
\E	End of lowercase/uppercase/quote

```
# 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:

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

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

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

To do this in Perl, we use the special variable `$/`. This is called the input record separator.

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

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

```
undef $/;
$_ = <> # 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/linux.txt` which is a set of Linux-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 were uttered by Larry Wall. You might also want to refresh your memory of `chomp()` at this point. (Answer: `exercises/answers/larry.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
        print "Matched in multi-line mode\n";   # because ^ matches the
}                                               # start of any line
                                                # in the string

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

if($string =~ /^This is some/s) {                # 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)/m)  {   # 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 7-2. 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 `$&` and other special variables in substitutions:

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

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 two sets of parentheses are non-capturing
m/^(?:\w+) (?:\w+) (\w+)/;
```

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

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

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

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

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

In order to match "this this" we need to use the special regular expression 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.

```
my $word = "this";
print if m/($word) \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 8. File I/O

In this chapter...

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

Assumed knowledge

You should already have seen the <> line input operator in a previous Perl training session or in your previous Perl experience.

Angle brackets - the line input and globbing operators

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

```
# reading lines from STDIN (or from files on the command line)
while (<>) {
        # Process the line of input in $_
}

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

# reading all lines from STDIN into an array
my @input = <STDIN>;
```


The line input operator is discussed in-depth on page 81 (page 53, 2nd Ed) of the Camel book. You can read about the closely-related `readline` function using **perldoc -f readline**.

- In scalar context, the line input operator yields the next line of the file referenced by the filehandle given.
- In list context, the line input operator yields all remaining lines of the file referenced by the filehandle. (Be careful when using this as you may use up all your memory if the file is large).
- The default filehandle is `STDIN`, or any files listed on the command line of the Perl script (eg **myscript.pl file1 file2 file3**).

The *globbing* operator looks the same as the line input operator, but is really quite different.


The filename globbing operator is documented on page 83 (page 55, 2nd Ed) of the Camel book. You can also read about it with **perldoc perlop**.

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

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

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

Perl's globs operate the same way as they do in the UNIX C-shell. Don't worry if you don't know C-shell, the basic pattern matching operators (such as `*` and `?`) have the same behaviour as just about any other shell that you may have used.

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

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

        # Read from the file

        close FILEHANDLE;
}
```

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

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

foreach (glob("*.txt")) {
        # Process the filename in $_
}
```

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

Like all functions, you can read more about `glob` using `perldoc -f glob`.

Exercises

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

Advanced exercises

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

open() and friends - the gory details

Opening a file for reading, writing or appending

The `open()` function is documented on pages 747-755 (pages 191-195, 2nd Ed) of the Camel book, and also in **perldoc -f open**.

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

In brief, Perl uses the same characters as shell does for file operations. That is:

- < says to open the file for reading
- > says to open the file for writing
- >> says to open the file for appending.

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

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

Note that the < (less than) character used to indicate reading is assumed; we could equally well have said

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

Although reading mode is assumed when opening files, it is still always a good idea to explicitly open your files for reading by using the < character. Doing so protects you from the case where your filename has odd characters in it, such as <, > and | which all mean special things to `open`.

Watch out when using > to open files for writing. Like shell, using > will *clobber* any contents of your file. This is because > truncates the file when it is opened. So even if you don't write anything to the file, the original contents will be lost upon opening.

Using > or >> will cause the files to spring into existence if they do not already exist, so you don't have to worry about how to create them before writing.

If you need more control over how you open your files, check out the `sysopen` function by using **perldoc -f sysopen**. Using `sysopen` is especially important if you're running with elevated privileges, as it can help protect against dangerous race conditions. You can read more about that on pages 571-573 in the Camel book (3rd Ed only).

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

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

`die` is a Perl function which takes an error message and terminates the program displaying that message to the user. In this example, the die statement (which is always true) is executed only if the open statement does not return true, that is, if there was an error in opening the file.

Attempting to read from an unopened file may cause unexpected results.

`$!` is the special variable which contains the error message produced by the last system interaction. It is documented in on page 669 (page 134, 2nd Ed) of the Camel book and also in `perldoc perlvar`.

Perl tries to be helpful when dying on errors and will append the appropriate filename and line number of your script to the end of the die message, with a newline. If you don't want this behaviour, end the die message with a newline (`\n`) character. For example:

```
# The following provides an error with file and line-number:
open(LOGFILE, "< $file") or die "Cannot open $file: $!";

# Here the file and line-number are omitted.
open(LOGFILE, "< $file") or die "Cannot open $file: $!\n";
```

Make sure you don't do this by accident, and miss out on this important information.

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

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

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

close LOGFILE;

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

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

close SCRIPTLOG;
```

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

Under Perl version 5.6.0 and above, you can provide a scalar as the first argument to the `open` function. This means that your filehandles can have scope, and makes it easier to pass them to subroutines and put into structures such as hashes and arrays.

```
my $fh;
open($fh,"< /path/to/file") or die "...";
```

In versions before 5.6.0 you can do the same thing by using the `FileHandle` module, but you need to declare your intentions first:

```
use FileHandle;

my $fh = FileHandle->new;     # $fh is now a FileHandle object.
open ($fh,"< /path/to/file") or die "...";
```

You use scalar filehandles the same way as you use regular ones:

```
while(<$fh>) {
        # do something with each line of the file
}
```

Using the `FileHandle` module also works in Perl 5.6.0 and above, so if compatibility with older versions of Perl is important to you, you should use the `FileHandle` module for scalar filehandles.

For more information see `perldoc FileHandle` and pages 895-898 (page 442-444, 2nd Ed) in the Camel book.

Be careful when trying to open a file whose name contains characters that might have special meaning to `open()`, in particular those that end with `|` (pipe), or begin with `>` or `<`, as these may result in `open()` not doing what you expect. Leading and trailing spaces are also ignored.

Under Perl 5.6.0 and above, a three-argument version of `open()` exists. This version of `open()` treats the filename literally, including special characters and spaces. You use it like this:

```
my $filename = "filename ending with spaces   ";

open(FILE, "<", $filename)
        or die "Failed to open file: $filename for reading: $!";
while(<FILE>) {
        # Process the line of input in $_
}
```

The three argument version of `open` is much safer than the two-argument version, especially if you're dealing with untrusted user input, as no special interpretation is done on the filename. It's described with the rest of the `open` documentation.

For a safe file open for those who can't upgrade to Perl 5.6, have a look at `sysopen`. Information about sysopen can be found in **perldoc -f sysopen** and pages 808-810 (pages 194, 2nd Ed) of the Camel book.

Exercises

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

Reading directories

In addition to being able to open files, it's also possible to open directories using the `opendir()` function. Once a directory is open, you can read filehandles from it using the `readdir()` function.

To read the contents of files in the directory, you still need to open each one using the `open()` function.

`opendir()` is documented on page 755 (page 195, 2nd Ed) of the Camel book. `readdir()` is on page 770 (page 202, 2nd Ed). Don't forget that function help is also available by typing **perldoc -f opendir** or **perldoc -f readdir**

```
# $ENV{HOME} stores the home directory on Unix platforms, use
# $ENV{HOMEPATH} for MS Windows
opendir(HOMEDIR, $ENV{HOME}) or die "Can't read dir $ENV{HOME}: $!";

my @files = readdir(HOMEDIR);

closedir HOMEDIR;

foreach (@files) {
        # Skip over directories and non-plain files (eg devices)
        next unless -f "$ENV{HOME}/$_";

        open(THISFILE, "< $ENV{HOME}/$_")
                or die "Can't open file $ENV{HOME}/$_: $!";

        # Read from the file...

        close THISFILE;
}
```

The HOMEDIR in the previous example is a *directory handle* not a filehandle, even though they look the same. Attempting to use a directory handle as a filehandle (or the opposite) will result in an error.

Under Perl 5.6.1 and above you can provide a scalar as the argument to `readdir`. This allows you to have scalar directory handles which have scope and makes it easier for you to pass them to subroutines or include them in hashes and arrays.

```
my $homedir;
opendir($homedir, $ENV{HOME}) or die "Can't read dir $ENV{HOME}: $!";
my @files = readdir($homedir);
```

glob and readdir

There are some major differences between `glob()` and `readdir()`. `glob()` is not as fast but gives you flexibility over which filenames you get back: `glob("*.c")` for example, returns only files with the ".c" extension. `glob()` also gives you back filenames in asciibetical order, whereas `readdir` gives you back the files in whatever order they're stored in the internal representation of your system.

`glob("some/path/*")` will return filenames with path intact whereas readdir will return only the filenames of the files in the directory.

The last difference between these is their behaviour with "." files. For example ".bashrc". `glob("*")` will not return these files (although `glob(".*")` will), whereas `readdir()` will always return "." files.

Table 8-1. Differences between glob and readdir

glob	readdir
Slower	Faster
Allows you to filter filenames	Gives you all filenames
Returns files in asciibetical order	Returns files in file-system order
Returns filename with path intact	Returns filename only
Does not return dot files when called as `glob("*")`	Returns all filenames

rewinddir

We can rewind the current position of the directory handle back to the beginning by using the `rewinddir` function.

```
opendir(HOMEDIR, $ENV{HOME}) or die "Can't read dir $ENV{HOME}: $!";

foreach my $file (readdir(HOMEDIR)) {
        next unless -f "$ENV{HOME}/$_";
        open(THISFILE, "< $ENV{HOME}/$_")
                or die "Can't open file $ENV{HOME}/$_: $!");
        # Read from the file...
}
```

```
rewinddir HOMEDIR;
# Now we can read through our directory contents again...
# if we wish to.
```

`rewinddir` does not refresh the directory listing when it rewinds. To see whether the directory listing has changed since your program started you'll have to close the directory and reopen it.

You can find more about `rewinddir` by reading **perldoc -f rewinddir** or page 777 (page 208, 2nd Ed) in the Camel book.

Changing directories

The function `chdir` allows you to change your program's working directory. All relative file access from that point on will use the new directory. Note that this does *not* change the working directory of the calling process.

```
        # Archive log files
my $tar = "/bin/tar";
my $date = "01-01-00";
my $directory = "/var/log/apache/";

unless( chdir($directory) ) {
        die "Failed to chdir to $directory: $!";
}

# Get all files in this directory
my @files = glob("*.log*");
system("$tar -czf weblogs.$date.tgz @files") if @files;

# We learn how to check that system worked later in the course.
```

You can find more about `chdir` by reading **perldoc -f chdir** or page 688 (page 148, 2nd Ed) in the Camel book.

Exercises

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

Changing file contents

When manipulating files, we may wish to change their contents. A flexible way of reading and writing a file is to import the file into an array, manipulate the array, then output each element again.

It is important to ensure that should anything go wrong we don't lose our original data. As a result, it's considered *best-practice* to write our data out to a temporary file and them move that over the input file after everything has been successful.

```
# a program that reads in a file and writes the lines in sorted order.
use File::Copy;

open(INFILE, "< file.txt") or die "Can't open file.txt for input: $!";
my @lines = <INFILE>;   # Slurps all the lines into @lines.
close INFILE;

@lines = sort @lines;

# open temporary file to save our sorted data into
open(OUTFILE, ">file.txt.tmp") or die "Can't open file.txt for output: $!";
foreach my $line (@lines) {
        print OUTFILE $line;
}
close OUTFILE;

# at this point we know that we were successful, so write over the original
# file
move("file.txt.tmp", "file.txt");
```

In fact, since the print function can take a list of values to print, instead of the `foreach` loop above, we could have written:

```
print OUTFILE @lines;
```

The `File::Copy` module makes copying and moving files very easy. To find out more about `File::Copy` read **perldoc File::Copy**.

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

Note that this includes doing things such as:

```
foreach (<FILE>) {
        # Process the line of input in $_
}
```

as this will read the whole file into the list given to `foreach` and then walk over it line by line.

Of course, if you don't need to manipulate all of the lines together (eg sorting) you ought to forgo the reading things into an array and just loop over each line. Continue to ensure, however, that your original data cannot be lost if the program terminates unexpectedly.

```
# removes duplicated lines
open(INFILE, "<file.txt") or die "Can't open file.txt for input: $!";
open(OUTFILE, ">outfile.txt") or die "Can't open outfile.txt for output: $!";
```

```
my $lastline;
while(<INFILE>) {
        print OUTFILE $_ unless ($_ eq $lastline);
        $lastline = $_;
}

close INFILE;
close OUTFILE;
```

Exercises

1. Open a file, reverse its contents (line by line) and write it back to the same filename. For example, "this is a line" would be written as "enil a si siht" (Answer: `exercises/answers/reversefile.pl`)

Opening files for simultaneous read/write

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

Read/write access to a file is not as useful as it sounds --- except under special circumstances (notably when dealing with fixed-length records) you can't usefully write into the middle of the file using this method, only onto the end. The main use for read/write access is to read the contents of a file and then append lines to the end of it.

+< puts you at the start of the file. Note that it won't create a new file if the file you're dealing with does not exist (you'll just get an error that the file doesn't exist). If you start writing before you've reached the end of the file, you will overwrite characters in that file (from that point). Even if you're dealing with fixed-length records and think you know what you're doing, this is often still a bad idea. See below.

+>> initially puts you at the end of the file. It will create a new file if necessary and will not clobber an old one. It allows you to read at any point in the file, but all writes will always go to the end.

Since both print and readline (<>) are buffered, you should *not* use them for editing a file in-place. If you must do so, use the lower level functions such as `sysseek()`, `syswrite()` and `sysread()`. Perl also has a `-i` switch, for more useful in-place modification of files. These concepts are not covered in this course.

For more information about `open` including simultaneous read/write, see **perldoc perlopentut**. Also read pages 747-755 (pages 191-195, 2nd Ed) of the Camel book.

For information about the `-i` option to Perl read **perldoc perlrun** and pages 495-497 (page 332, 2nd Ed) of the Camel book.

For information about Perl's buffering and where it can cause problems read Mark Jason Dominus's excellent article on *Suffering from Buffering* available from his Perl FAQs (http://perl.plover.com/FAQs/Buffering.html).

```
# Example: Reading a file and adding to the end.
# Program that checks to see if $username appears in the file
# adds $username to the end, if not.

my $username = <STDIN>;
chomp $username;

open(FILE, "+< file.txt") or die "Failed to open file.txt for read/write: $!";

my $found;
while(<FILE>) {
        chomp;
                # case insensitive matching
        $found = 1 && last if(lc($_) eq lc($username));
}

# We'll be at the end of our file if $found isn't set
unless($found) {
        print FILE "$username\n";
}
close FILE;
```

Opening pipes

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

We can use pipes to read information from any process we can execute on our system. Once the command is open, we can read from the resulting filehandle in the same way we would read from any other file. In the example below, we use secure shell (**ssh**) to read a file on a remote machine.

```
#!/usr/bin/perl -w
# This program allows us to read a file from another machine
# using secure shell.  This is most useful if we can login without
# a password (eg, established keys).
use strict;

# Process our command line arguments, and complain if we don't
# have both a host and filename.
my ($host, $file) = @ARGV;
unless ($host and $file) {
        die "Usage: $0 host filename\n";
}

open (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;
}
```

Here's an example which writes to the **sort** command, which is a standard utility on both Windows and Unix systems. Even though Perl has its own **sort** function, the external command is very good at dealing with large amounts of data in a memory-efficient manner.

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

# Open our external sort command.
open (SORT, "|sort") or die "Can't open pipe: $!";

# Our friends will be printed in sorted order.
foreach my $friend (qw/Jacinta Damian Kirrily Paul/) {
        print SORT "$friend\n";
}
close SORT;
```


If you're interested in reading more about inter-process communication, including pipes, signals, sockets and the like, check out **perldoc perlipc**.

Exercises

1. Modify the second example above (provided for you as `exercises/sort_starter.pl` in your exercises directory) to accept user input and print out the **sort**ed version.
2. Change your script to accept input from a file using `open()` (Answer: `exercises/answers/sort.pl`)
3. If you are using a unix system: change your script to pipe its input through the **strings** command and then **sort**. Now if you specify a file that is not a text file, it will only sort and display printable strings. Try running this over `/usr/bin/perl`. (Answer: `exercises/answers/strings.pl`)

Finding information about files

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

Table 8-2. File test operators

Operator	Meaning
`-e`	File exists.
`-r`	File is readable
`-w`	File is writable
`-x`	File is executable
`-o`	File is owned by you
`-z`	File has zero size.
`-s`	File has nonzero size (returns size).
`-f`	File is a normal file.
`-d`	File is a directory.
`-l`	File is a symbolic link.

Operator	Meaning
`-p`	File is a named pipe (FIFO), or Filehandle is a pipe.
`-S`	File is a socket.
`-b`	File is a block special file.
`-c`	File is a character special file.
`-t`	Filehandle is opened to a tty.
`-u`	File has setuid bit set.
`-g`	File has setgid bit set.
`-k`	File has sticky bit set.
`-T`	File is a text file.
`-B`	File is a binary file (opposite of -T).
`-M`	Age of file in days when script started.
`-A`	Same for access time.
`-C`	Same for inode change time.


The file test operators are documented fully in **perldoc -f -x**.

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

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

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

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

The `stat()` function returns similar information for a single file, in list form. `lstat()` can also be used for finding information about a file which is pointed to by a symbolic link. If you've used these functions in C or other languages, then you'll probably find them somewhat familiar in Perl. Check out **perldoc -f stat** to see the format this data is returned in and how to make use of it.


Occasionally it is desirable to perform several tests on the same file at the same time. Perhaps you'd like to check that a file is both readable and writable. It is possible to perform your test like this:

```
if (-r $file && -w $file) {
        # ...
}
```

but that involves two separate tests which both take time. The file might also change between the tests (which is why file tests are almost always a bad idea in security situations).

Perl caches the result of file tests in a special filehandle called _ (underscore). Performing tests on this filehandle can often avoid subsequent system calls, resulting in a slight performance gain.

```
if(-r $file && -w _) {
        # ...
}
```

There are some caveats on when the _ filehandle can be used with certain operators such as `-l` and `-t`. To find out more about these and to learn more about file test operators read **perldoc -f -x**.

The file test operators expect the file you're testing to be in the current working directory. If this is not the case, make sure you prepend a path to the file before doing your test.

Exercises

1. Use the file test operators to print out only files from a directory which are "normal" files, i.e. not directories, symbolic links or other oddities. (Answer: `exercises/answers/normaldirlist.pl`)
2. Write a script to find zero-byte files in a directory. (Answer: `exercises/answers/zerobyte.pl`)
3. Write a script to find the largest file in a directory: `exercises/answers/largestfile.pl`)
4. Write a script which asks a user for a file to open, takes their input from STDIN, checks that the file exists, then prints out the contents of that file. (Answer: `exercises/answers/fileexists.pl`)

Recursing down directories

The **File::Find** module is documented on pages 889-890 (page 439, 2nd Ed) or more fully in **perldoc File::Find**.

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

`File::Find` emulates Unix's **find** command. It takes as its arguments a subroutine to execute for each file found, and a list of directories to search. Note that to pass a reference to a subroutine we prefix the name of the subroutine with `\&`. In our example below, this is `\&wanted`.

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

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

# find takes a subroutine reference and the directory to start working from.
find (\&wanted, $dir);
```

```
sub wanted {
        if(/\.pl$/) {                           # See if it's a .pl file
                print "$File::Find::name\n";    # Print the current file name.
        }
}
```

For each file found, certain variables are set.

- `$_` is set to the name of the current file.
- `$File::Find::dir` is set to the directory that contains the file.
- `$File::Find::name` contains the full name of the file, i.e. `$File::Find::dir/$_`.

`File::Find` automatically changes your current working directory to the same as the file you are currently examining. There's rarely a need for `$File::Find::dir`. If all you want to do is process the file regardless of its location on the file system you can simply open the file using the name in `$_`. This behaviour can be turned off, see **perldoc File::Find** for further information.

Exercises

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

File locking

File locking can be achieved using the `flock()` function. This can be used to guard against race conditions or other problems which occur when two (or more) processes want to access the same file at the same time.

`flock()` is documented on page 714 (page 166, 2nd Ed) of the Camel book, or use **perldoc -f flock** to read the online documentation.

`flock` is Perl's portable file-locking mechanism, and works on most operating systems (and produces a fatal error on those which it does not). The locks set by `flock` are advisory only, which means that a process that chooses not to use `flock` can (and will) ignore any locks in place. `flock` can only lock entire files, not individual records. Depending upon your setup, `flock` may or may not work over NFS.

```
# using flock
use Fcntl ':flock';             # import LOCK_* constants

flock(FILEHANDLE, LOCK_EX);     # exclusive (write) access
flock(FILEHANDLE, LOCK_SH);     # shared (read-only) access
flock(FILEHANDLE, LOCK_UN);     # unlock
```

As `flock` only works on *filehandles*, instead of filenames, you have to open the file first *before* you try to lock it. It's important to make sure that you open the file for writing, if you intend to write to it, and that you don't clobber the contents of the file when doing so. This is a good use of `+<`. Closing a locked file releases any locks the process holds upon it. This is good because it means that if your process exits unexpectedly all locks it held are released and other processes may then go forward with their locks.

In the following example, we're locking a file before re-writing it. The exclusive lock stops any other process from holding a lock on the file while we perform our operations.

```
use Fcntl ':flock';             # import LOCK_* constants

# Open file for read and write
open FILE, "+< $file" or die "Failed to open $file: $!";

        # Lock the file for writing (exclusive lock) make sure it worked.
flock(FILE, LOCK_EX) or die "Failed to gain lock on $file: $!";

# At this point we have exclusive access to the file.
# Wipe previous process's details out
truncate(FILE, 0) or die "Failed to truncate $file: $!";

# Write to the file, or perform other operations as needed here...
print FILE $data;

close FILE;             # Closing the file releases the lock as well.
```

`flock` will wait indefinitely until the lock is granted, however it can return early if interrupted by a signal or other event. It's important to ensure that flock returns *true* to be sure that you have the lock you requested. It is possible to make `flock` *non-blocking* as follows:

```
use Fcntl ':flock';             # import LOCK_* constants

flock(FILEHANDLE, LOCK_EX | LOCK_NB);     # non-blocking exclusive lock
flock(FILEHANDLE, LOCK_SH | LOCK_NB);     # non-blocking shared lock
```

All attempts to get a *non-blocking* lock return immediately with either *true* for success (the lock was obtained) or *false* for failure (the lock was not obtained).

For an excellent introduction on using `flock`, the slides from Mark Jason Dominus' *File Locking Tricks and Traps* make excellent reading. They can be found at http://perl.plover.com/yak/flock/.

Handling binary data

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

Instead, we can use `read()` to read a certain number of bytes from a file handle. However, before we do that, we should call the `binmode()` function on the filehandle, so that Perl knows that we'll be dealing with a binary file. This means Perl won't try to do any transformations of input based upon the operating system or locale where your program is running.

`binmode()` must be called on the filehandle before any corresponding file I/O. It's best to call it immediately after you open the file.

You can learn more about `read()` by reading page 768 (page 202, 2nd Ed) of the Camel book or **perldoc -f read**.

You can learn more about `binmode()` by reading page 685 (page 147, 2nd Ed) of the Camel book, or **perldoc -f binmode**.

`read()` takes the following arguments:

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

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

my $image = "picture.gif";

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

binmode IMAGE;
binmode OUT;

my $buffer;

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

close IMAGE;
close OUT;
```

Of course, in that particular instance we might have preferred to just copy the file.

Chapter summary

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

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

Chapter 9. System interaction

In this chapter...

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

system() and exec()

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

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

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

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

If the command specified by `system()` could not be run the error message will be available via the special variable `$!`. This value is not set if the command can be run but fails during runtime. The return status of the command can be found in the special variable `$?`, which is also the return value of `system()`. This value is a 16-bit status word which needs to be unpacked to be useful, as demonstrated in the example below.

```
system("/bin/true");
if ($?) {                    # A non-zero exit code means failure
                             # -1 means the program didn't even start!
        if($? == -1) {
                print "Failed to run program: $!\n";
        } else {
                print "The exit value was: " .  ($? >> 8) . "\n";
                print "The signal number that terminated the program was: "
                        . ($? & 127) . "\n";
                print "The program dumped core.\n" if $? & 128;
        }
}
```

Just like `open` the traditional form of calling `system` and `exec` have security issues due to shell expansion. For example consider the following code:

```
my $filename;
print "Please give me a file you want to see: ";
$filename = <>;   # lets pretend: $filename="fred; rm -rf /home/pjf;"
chomp($filename);
system("cat $filename");
        # or
exec("cat $filename");
```

In this case, due to shell expansion, the shell will receive the commands:

```
cat fred
rm -rf /home/pjf
```

and if our program had sufficient permissions to delete pjf's home directory, it would.

As a result, there is another, safer, form of `system` and `exec` that bypasses the shell. If you give `system` or `exec` a list it assumes that the first element is the command to execute, and every other element is an argument to that command. These arguments are not passed to the shell, and so shell expansion will not occur. So:

```
system("cat", "*.txt");
       # or
exec("cat", "*.txt");
```

will give the `"*.txt"` filename to `cat` rather than all files with a `.txt` extension. This is essential in cases like the above where the command may be passed in from a user. In this case, if the file "*.txt" does not exist then we'll receive an non-zero return code. Exec fails and returns false only if the command (in this case **cat**) does not exist. If the file does not exist, the user will receive **cat**'s error message.

*nix Exercise

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

MS Windows Exercise

1. Write a script to ask the user for a filename on the system. Open the nominated file in Notepad using **system**. (Answer: `exercises/answers/notepad.pl`)

Using backticks

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

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

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

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


In this course we tend to use `qx()` because it's much harder to confuse `qx()` with plain old single quotes. Using `qx()` also avoids the problem that in some font sets both single quotes and backticks look exactly the same.

Backticks are different to the `system()` command, in that they capture the output of the command they execute, as opposed to passing it through to the user.

When called in a scalar context backticks return the output of the command they execute as a string with possibly embedded newlines. When called in a list context, the output is returned as a list with each separate line of output being a new list element.

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

# Backticks capture the output of the process they run.  Here,
# we capture the output of the echo command.

my $greeting = qx(echo Hello World);

# $greeting now contains the string "Hello World\n"

# System runs a command without capturing the output, instead it's
# passed straight through.  The following line uses the echo command
# to print a greeting.

system("echo Hello World");
```

The return status of commands called by using backticks can be determined by examining `$?` in the same way as the `system()` example above.


Backticks and the `qx()` function are discussed in the Camel book on page 80 (pages 52 and 41, 2nd Ed) or in **perldoc perlop**.

*nix Exercises

1. Modify your earlier finger program to use backticks instead of `system()` (Answer: `exercises/answers/backtickfinger.pl`)
2. Change it to use `qx()` instead (Answer: `exercises/answers/qxfinger.pl`)
3. The Unix command **whoami** gives your username. Since most shells support backticks, you can type **finger ‘whoami‘** to finger yourself. Use shell backticks inside your `qx()` statement to do this from within your Perl program. (Answer: `exercises/answers/qxfinger2.pl`)

MS Windows Exercises

1. Modify your earlier program to take a directory path from the user. Use backticks to execute the **DIR** command on that path and list out the files in that directory. (Answer: `exercises/answers/backtickdir.pl`)
2. Change it to use `qx()` instead.

3. Time permitting: reverse sort the directory listing contents.

Platform dependency issues

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

Security considerations

This section is not intended as a comprehensive guide to Perl security, rather it is here to show some of the in-built security features that Perl has available. Even `perldoc perlsec` does not give you the whole picture, it just gives you some hints.

The ability to write secure programs is one that is learnt over many years of experience. It's always a good idea to have someone well rehearsed in security and your programming environment to audit your code in case you have missed anything. As well as training, Perl Training Australia also offers security and privacy auditing services.

Perl Training Australia offers a course in Perl Security that covers many common attacks and mistakes, and how they can be prevented in Perl.

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

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

use strict;

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

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

A further, not so obvious problem, can be seen when we ask "Which `finger` program are we calling?". If our program caller has changed our `$ENV{PATH}` then it is very possible that it's not the usual system `finger` found in `/usr/bin/`. It could instead be a malicious `finger` program designed to exploit our program's privileges.

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

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

Documentation for taint checking can be found by reading the **perldoc perlsec**, or on pages 557-568 (page 356, 2nd Ed) of the Camel book.

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

In versions of Perl prior to 5.8.0 files opened for both reading and writing using "`+<`" were not checked for tainted filenames.

Taint checking will not occur on filenames where the file is only being opened for reading. This is due to historical reasons. Good programming practice would have you untaint these filenames anyway.

The only thing that will clear tainting is referencing substrings from a regexp match. Here's an example.

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

$ENV{PATH} = "/bin:/usr/bin"; # Taint requires we set our path.

print "Who do you want to finger?\n";
my $username = <STDIN>;
chomp($username);

# Check $username to make sure it's clean, then finger.

if ( $username =~ /^(\w{1,8})$/ ) {

       # $1 is the contents of the first set of
       # parentheses in the regexp.

       print qx(finger $1);

} else {
       print "That was not a valid username!\n";
}
```

Make sure you remember to check that the regular expression to untaint your variable succeeded. In the case above we only have one regular expression, so `$1` will either be set by the match or will be undefined. Nevertheless we still explicitly tested the match for success. This means that our code won't break if we add any regular expressions before the code used above.

You can also untaint data by capturing the match in a list context:

```
# Check $username to make sure it's clean
my ($safeuser) = ($username =~ /^(\w{1,8})$/);

# safeuser is now either undefined if the match failed or
# the value of $1 if the match succeeded.
if ($safeuser) {
       print qx(finger $safeuser);
} else {
       print "That was not a valid username!\n";
}
```

Note that you'll have to explicitly set the environment's `PATH` variable (found in `$ENV{PATH}`) to something safe (like `/usr/bin`) as well. This variable affects where the shell looks for other executable programs. `finger` is found in `/usr/bin` on our system.

We have to set a safe value for `$ENV{PATH}` because this value can be changed by the user in their environment before running the Perl script. If the user sets their `PATH` to `/home/pjf/bin` then we'd run the `/home/pjf/bin/finger` command rather than the `/usr/bin/finger` command.

For safety's sake, taint checking in Perl always assumes that the `PATH` environment variable has been tampered with by the user.

If you've been calling your Perl program from the command line with **perl *`program.pl`*** you'll be told that you're turning taint checking on too late, even if you've put it in your shebang line.

This is because Perl wants to know that you want to use taint checking as soon as possible. The way to fix this is to include the `-T` option in your call, so: **perl -T *`program.pl`***.

SETUID scripts automatically run with taint checking turned on for your own protection.

Under Perl 5.8.0 and above, there is also the `-t` switch, which causes tainted operations to generate warnings instead of errors. This is no substitute for real taint checking, but can be useful if you're trying to lock down legacy code and see which areas require attention.

Safe.pm

For greater security when using unknown (and possibly hostile) code, or for writing code which adheres to strict standards about what it's allowed to do, there is the *Safe* module. This module allows the creation of compartments in which Perl code can be evaluated. These compartments allow you to define explicitly what the code run within them may and may not do. For example, you may deny access to the file system so that the code may not read or write to files. Or you may only permit the code to use certain operators such that it may add and subtract but not divide, for example. Attempts by the code to perform forbidden tasks result in a compilation error at compile time and a fatal error at run time.

Note that it is always a good idea to audit code that you receive from a third party before executing it on your machine.


Learning how to use the *Safe* module is a course in itself. For more information on this module read **perldoc Safe** and pages 576-581 (489-493 2nd Ed) of the Camel book.

Exercise

1. Ask the user for a filename and if that file does not already exist in your directory, open the file and write a short message to it. Turn on taint checking and try running your script. What sort of regular expression could you use to check for valid filenames? (Answer: `exercises/answers/taintfile.pl`)

Chapter summary

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

Chapter 10. Conclusion

What you've learnt

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

- Advanced Perl data structures and references.
- Finding, using and writing function Perl modules and how to use modules which have an object oriented interface.
- How to use regular expressions to capture data, how greediness works and how to work with multi-line strings.
- File I/O, including opening files and directories, opening pipes, finding information about files, recursing down directories, file locking, and handling binary data.
- System interaction, including: system calls, the backtick operator, interacting with the file system, dealing with users and groups, dealing with processes, network communications, and security considerations.

Where to now?

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

- Work through the material included in the appendices of this book.
- Visit the websites in our "Further Reading" section (below).
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme".
- Install Perl on your home or work computer.
- Practice using Perl from day to day.
- Join a Perl user group such as Perl Mongers (http://www.pm.org/).
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/).
- Extend your knowledge with further Perl Training Australia courses such as:
  - CGI Programming with Perl
  - Database Programming with Perl
  - Perl Security
  - Object Oriented Perl

  Information about these courses can be found on Perl Training Australia's website (http://www.perltraining.com.au/).

Further reading

Books

- Larry Wall, Tom Christiansen and Jon Orwant, *Programming Perl* (3rd Ed), O'Reilly and Associates, 2000. ISBN 0-596-00027-8
- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O'Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 1997. ISBN 1-56592-257-3.
- Joseph N. Hall and Randal L. Schwartz *Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.

Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup
- Comprehensive Perl Archive Network (http://www.cpan.org)

Appendix A. Complex data structures

References are most often used to create complex data structures. Since references are scalars, they can be used as values in both hashes and arrays. This makes it possible to create both deep and complex multi-dimensional data structures. We'll cover some of these in further detail in this chapter.


Complex data structures are covered in detail in chapter 9 (chapter 4, 2nd Ed) of the Camel book.

Arrays of arrays

The simplest kind of nested data structure is the two-dimensional array or matrix. It's easy to understand, use and expand.

Creating and accessing a two-dimensional array

To create a two dimensional array, use anonymous array references:

```
my @AoA = (
        [qw(apple orange pear banana)],
        [qw(mouse rat hamster gerbil rabbit)],
        [qw(camel llama panther sheep)],
);

print $AoA[1]->[3];       # prints "gerbil"
```


The arrow is optional between brackets or braces so the above access could equally well have been written:

```
print $AoA[1][3];
```

Adding to your two-dimensional array

There are several ways you can add things to your two-dimensional array. These also apply to three and four and five and n-dimensional arrays. You can push an anonymous array into your array:

```
push @AoA, [qw/lions tigers bears/];
```

or assign it manually:

```
$AoA[5] = [qw/fish chips vinegar salt pepper-pop/];
```

You can also add items into your arrays manually:

```
$AoA[0][5] = "mango";
```

or by pushing:

```
push @{$AoA[0]}, "grapefruit";
```

You're probably wondering about why we needed the curly braces in our last example. This is because we want to tell Perl that we're looking at the element `$AoA[0]` and asking it to deference that into an array. When we write `@$AoA[0]` Perl interprets that as `@{$AoA}[0]` which assumes that `$AoA` is a reference to an array we're trying to take an array slice on it. It's usually a good idea to use curly braces around the element you're dereferencing to save everyone from this confusion.

Printing out your two-dimensional array

Printing out a single element from your two-dimensional array is easy:

```
print $AoA[1][2];           # prints "hamster"
```

however, if you wish to print our your data structure, you can't just do this:

```
print @AoA;
```

as what you'll get is something like this:

```
ARRAY(0x80f606c)ARRAY(0x810019c)ARRAY(0x81001f0)
```

which are stringified references. Instead you'll have to create a loop to print out your array:

```
foreach my $list (@AoA) {
        print "@$list";
}
```

Hashes of arrays

Arrays of arrays have their uses, but require you to remember the row number for each separate list. Hashes of arrays allow you to associate information with each list so that you can look up each array from a key.

Creating and accessing a hash of arrays

To create a hash of arrays create a hash whose keys are anonymous arrays:

```
my %HoA = (
        fruits =>  [qw(apple orange pear banana)],
        rodents => [qw(mouse rat hamster gerbil rabbit)],
        books =>   [qw(camel llama panther sheep)],
);

print $HoA{rodents}[3];         # prints "gerbil"
```

Adding to your hash of arrays

Adding things to your hash of arrays is easy. To add a new row, just assign an anonymous array to your hash:

```
$HoA{oh_my} =  [qw/lions tigers bears/];
```

To add a single element to an array, either add it in place or push it on the end:

```
$HoA{fruits}[4] = "grapefruit";
push @{$HoA{fruits}}, "mango";
```

Once again you'll notice that we needed an extra set of curly braces to make it clear to Perl that we wanted `$HoA{fruits}` dereferenced to an array.

Printing out your hash of arrays

Printing out a single element from your hash of arrays is easy:

```
print $HoA{fruits}[2];          # prints "pear"
```

Printing out all the element once again requires a loop:

```
foreach my $key (keys %HoA) {
        print "$key => @{$HoA{$key}}\n";
}
```

Arrays of hashes

Arrays of hashes are particularly common when you have number of ordered records that you wish to process sequentially, and each record consists of key-value pairs.

Creating and accessing an array of hashes

To create an array of hashes create an array whose values are anonymous hashes:

```
my @AoH = (
        {
                name => "John",
                age => 31,
        },
        {
                name => "Mary",
                age => 23,
        },
        {
                name => "Paul",
                age => 27,
        },
);

print $AoH[2]{name};            # prints "Paul"
```

Adding to your array of hashes

To add a new hash to your array, add it manually or push it on the end. To add an element to every hash use a loop:

```
$AoH[3] = {                             # adding a new hash manually
                name => "Jacinta",
                age => 26,
          };

push @AoH, {                            # pushing a new hash on to the end
                name => "Judy",
                age => 47
           };

$AoH[0]{favourite_colour} = "blue";     # adding an element to one hash

foreach my $hashref (@AoH) {            # adding an element to every hash
        $hash->{language} = "Perl";
}
```

Printing out your array of hashes

To print a array of hashes we need two loops. One to loop over every element of the array and a second to loop over the keys in the hash:

```
foreach my $hashref (@AoH) {
        foreach $key (keys %$hashref) {
                print "$key => $hashref->{$key}\n";
        }
}
```

Hashes of hashes

Hashes of hashes are an extremely common sight in Perl programs. Hashes of hashes allow you to have a number of records indexed by name, and for each record to contain sub-records. As hash lookups are very fast, accessing data from the structures is also very fast.

Creating and accessing a hash of hashes

To create a hash of hashes, assign anonymous hashes as your hash values:

```
my %HoH = (
        Jacinta => {
                        age => 26,
                        favourite_colour => "blue",
                        sport => "swimming",
                        language => "Perl",
                   },
        Paul => {
                        age => 27,
                        favourite_colour => "green",
                        sport => "cycling",
                        language => "Perl",
                },
```

```
        Ralph => {
                        age => 7,
                        favourite_colour=> "yellow",
                        sport => "little athletics",
                        language => "English"
                 },
);

print $HoH{Ralph}{sport};       # prints "little athletics"
```

Adding to your hash of hashes

```
$HoH{Ralph}{favourite_food} = "Tomato sauce";   # adding to Ralph's hash

$HoH{Tina} = {                                  # adding a new person hash
                age => 19,
                favourite_colour => "black",
                sport => "tai chi",
};
```

Printing out your hash of hashes

Once again, to print out a hash of hashes we'll need two loops, one for each key of the primary hash and the second for each key of the inner hash.

```
foreach my $person ( keys %HoH ) {
        print "We know this about $person:\n";
        foreach $key ( keys %{ $HoH{$person} } ) {
                print "${person}'s $key is $HoH{$person}{$key}\n";
        }
        print "\n";
}
```

More complex structures

Armed with an understanding of the nested data structures we've just covered you should be able to create the best data structure for what you need. Perhaps you need a hash of hashes but where some of your values are arrays. This should pose no problems. Perhaps you want an array of hashes of arrays? This too should be easy.

Appendix B. More functions

The grep() function

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


The `grep()` function is on page 730 (page 178, 2nd Ed) of your Camel book.

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

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

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

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

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

Exercises

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

The map() function

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

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

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

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

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

Exercises

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

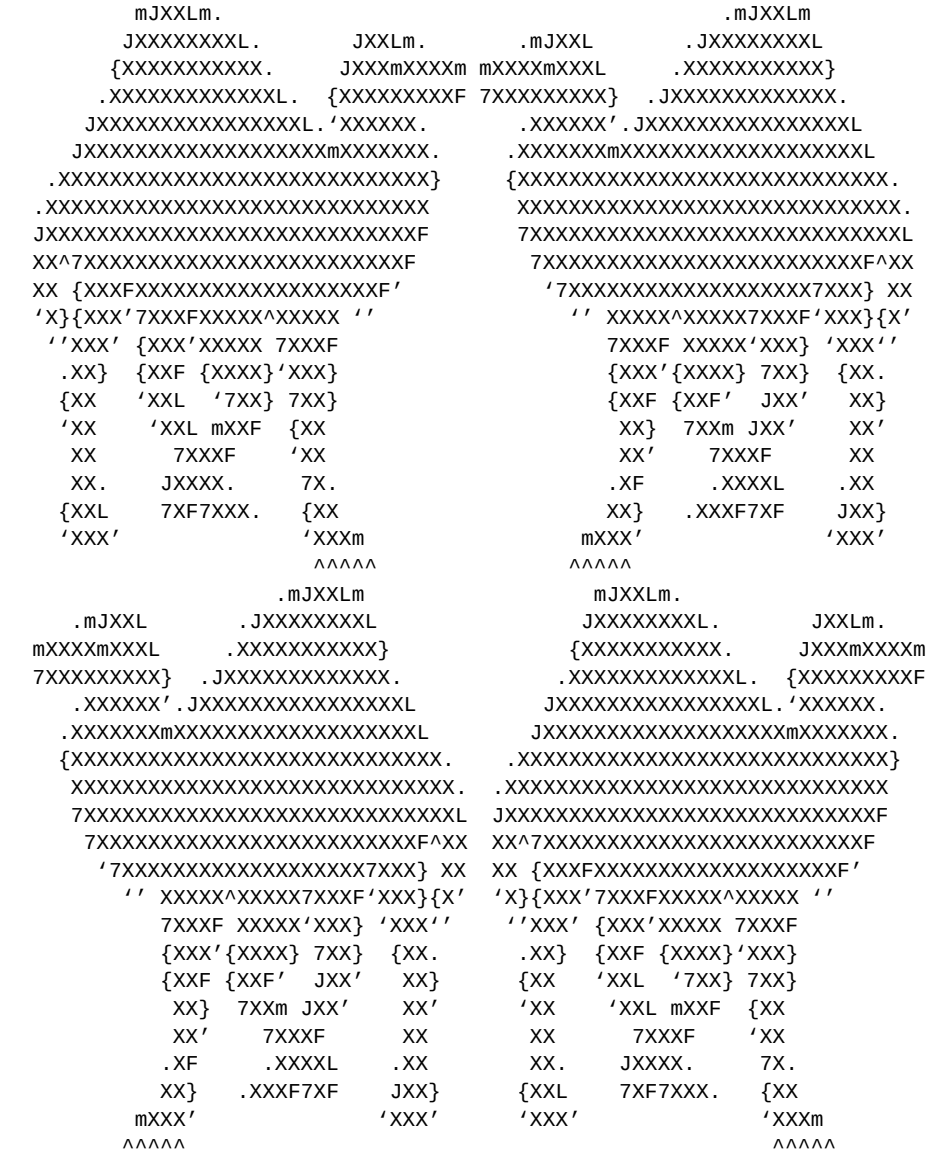
Appendix C. Unix cheat sheet

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

Table C-1. Simple Unix commands

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

Colophon



The use of a camel image in association with Perl is a trademark of O’Reilly & Associates, Inc. Used with permission.

The `camel code` that makes up the cover art was written by Stephen B. Jenkins (aka Erudil). When executed, it generates the images of four smaller camels as shown above. A discussion of the camel code in its native habitat can be found on PerlMonks (http://www.perlmonks.org/index.pl?node=camel+code). More information about Stephen B. Jenkins and his work can be found on his website (http://www.Erudil.com).