Learning to Program in Perl

by Graham J Ellis

computer training

Well House Consultants LTD

PERL
JAVA
PHP
TCL/TK
MYSQL

Languages of the Web

Learning to Program in Perl version 1.7

Written by Graham Ellis *graham@wellho.net*
Design by Lisa Ellis

Well House Consultants, Ltd.
404, The Spa, Melksham, Wiltshire SN12 6QL England

+44 (0) 1225 708 225 (phone)
+44 (0) 1225 707 126 (fax)

Find us on the World Wide Web at: *http://www.wellho.net*
Or contact us at: *enquiries@wellho.net*

Printing History

May 1999	1.0	First Edition
February 2000	1.1	Minor additions
June 2000	1.2	Compliation of modules
October 2000	1.3	Name change, revisions
April 2002	1.4	Added modules
September 2002	1.5	Added modules
January 2003	1.6	Updated modules
February 2003	1.7	Updated modules

This manual was printed on 21 May 2003.

Table of Contents

1 Introduction

1.1 What is Perl?

Perl is a computer language

You write a series of instructions and the computer then performs them. Unless you state otherwise, statements are performed in order. You can, though, have conditional code, loops and calls to blocks of code elsewhere just as in other languages.

Indeed, Perl has much more in its language than most other programming languages. It's both eclectic (many ways of doing things from many different sources) and wide ranging in itself. And there are a lot of other resources available to let it go even further.

What does Perl cost?

Perl itself costs nothing. It's distributed under an artistic license[1] which gives you the right to copy and use it for free under most conditions, and even to modify it in many circumstances!

What is Perl used for?

Deep breath. Anything. Everything.

Seriously, though ...

- Data manipulation
- Installation scripts
- System management
- Daemons
- Network services
- World Wide Web interaction
- Database interfacing

What computer do I need to run Perl?

Perl is a very efficient language. You don't need anything too powerful and it runs cross-platform. Common and supported ports include:

implementation has been verified for this course

- Windows 95, 98, 2000, ME, NT and XP
- Solaris 2.x, Solaris 7, Solaris 8 and Solaris 9
- MacOS (from System 7, including OSX)
- Linux (All flavours including Mandrake, SuSE, Redhat, and Caldera)
- AIX3, 4
- SunOS 4.1.x
- Free/Open/Net BSD
- Irix 4, 5, 6
- Ultrix 4
- HPUX 9, 10

1 the full text is on our server

- Digital UNIX / DEC OSF/1 1, 2, 3, 4
- Ms-Dos
- Windows 3.1
- Amiga
- AS400
- VMS
- Tandem Guardian
- MVS
- Lynxos
- Novell Netware
- NextStep
- OS2
- Acorn RiscOS
- Siemens Sinix
- SCO Unixware

And it runs virtually the same way on all platforms!

Is Perl loaded onto my computer?

If Perl is loaded, it's very likely you can just type in `perl` at a command prompt.

It's also possible that it is loaded, but under a different name, or not in your "path", i.e. where your computer looks for executable programs. This is system-dependent; it varies not only from one manufacturer's computer to the next, but can also be varied by how the system administrator has configured things.

If you're not sure if Perl is available on your computer, ask your system or network administrator. Even though Perl hasn't been explicitly installed, it's often there; it's included in Linux distribution, it comes with many commercial packages as an installation language, etc.

When you just type in `perl`, the cursor will hang.

What's gone wrong?

Nothing!

Just typing in `perl` starts the Perl language; you may now type in a program at the command line. When you're finished typing, enter an end-of-file and the program is interpreted and run.

Let's try that:

```
    graham@otter:~> perl
    print 5 + 5;
    print " something\n";
^D  10 something
    graham@otter:~>
```

or, at a different type of prompt:

```
    C:\> perl
    print 5 + 5;
    print " something\n";
^Z  10 something
    C:\>
```

On systems running Linux and Solaris, enter `[Control]D` for end of file. On Win32 systems (Windows 98, Windows 2000, Windows NT, Windows XP, etc.), use `[Control]Z` instead.

1.2 Perl Platforms

Perl runs on a wide variety of different computers and operating systems, and the programs are remarkably portable. For example, the author of these notes uses a Perl program to collect his email when he's away from the office. The same program runs without alteration on Windows, Unix and Linux systems, and it's the type of program that, in other languages, would be hard to transfer around.

Perl originated as a freely distributed (open source) program from a Unix environment, in 1988. In the early days, if you wanted Perl you downloaded it through a modem, in source code (it's written in C), compiled it, and installed it. You can still download it from the Internet in source form and do that if you wish, but you may already have it anyway. Various parties have come to realise what an advantage it is to supply Perl as a part of their product.

Perl on Unix

Perl can be downloaded for all common and modern versions of Unix and Unix derivatives, as well as many older and more obscure versions. The installation procedures will compile Perl for you, and will tune it for your particular operating system.

Users of Sun's Solaris 8 and 9 operating systems will find that Perl is now shipped as a standard part of the operating system.

Perl on Linux

Perl is a necessary part of the Linux operating system. There’s no need to question whether you have it or not if you're running Linux, it will be there!

Perl on Windows

If you're using Windows NT, you'll find Perl is shipped by Microsoft on the optional software CD.

For other Windows operating systems, a pre-compiled, easy-to-install version can be downloaded from the ActiveState web site at no charge. ActiveState is sponsored by Microsoft.

Perl on the Macintosh

If you're running an operating system up to and including OS9, you'll find links from the various Perl web sites to "MacPerl" which you can download and install.

On OSX (also known as OS10), Perl is supplied as standard with its Unix-based operating system.

1.3 Perl Versions

The current stable version of Perl is Perl 5.8.0 (July, 2002). The previous version was 5.6.1.

Prior to version 5.6.0, a different numbering system was used. If you have versions 5.004 or 5.005, they're quite recent. We suggest that you don't upgrade unless you need to use facilities that have been added in the very latest versions.

Versions with an odd number in the second position (e.g. 5.9.0) are often available, and with a higher release number too. The odd number signifies a development release, and the majority of our trainees should stick with the latest production version even if it appears to be older.

There may be a 5.10 release of Perl at some stage and there will be a Perl 6. This will be a rewrite; some features that have become time-expired will be removed from the language, and a number of converter tools are also to be provided to allow you to convert Perl 5 code to work in Perl 6. O'Reilly are publishing a book on Perl 6 (Perl 6 Essentials, ISBN 0-596-00499-0) in July, 2003 to give a detailed look ahead to Perl6; your tutor will give you an update during your course on the development, and our "Of Course" magazine, published twice a year, will also give you news if you're on our mailing list.

Older Versions

It's possible that you're still using version 4 of Perl. Perl 4 was a smaller language (without object orientation). It is integrated into a number of commercial products, and it's also faster to start on heavily loaded computers. Perl 4.019 and 4.036 are good, stable languages. Please let your tutor know if you're using one of these and he will tell you which of the facilities on your course are not going to be available to you.

If you're running Perl 5.0 through 5.003, you'll probably want to upgrade to the current release soon. Whilst they're good and stable versions, on balance you would be best to upgrade.

How do I find what version I have?

Windows 98 / Windows 2000 / Windows NT

```
C:\>perl -v
This is perl, version 5.005_02 built for MSWin32-
  x86-object (with 1 registered patch, see perl -
  V for more detail)
Copyright 1987-1998, Larry Wall
Binary build 509 provided by ActiveState Tool
  Corp. http://www.ActiveState.com
Built 13:37:15 Jan  5 2003
Perl may be copied only under the terms of either
  the Artistic License or the GNU General Public
  License, which may be found in the Perl 5.0
  source kit. Complete documentation for Perl,
  including FAQ lists, should be found on this
  system using 'man perl' or 'perldoc perl'. If
  you have access to the Internet, point your
  browser at http://www.perl.com/, the Perl Home
  Page.
C:\>
```

Linux

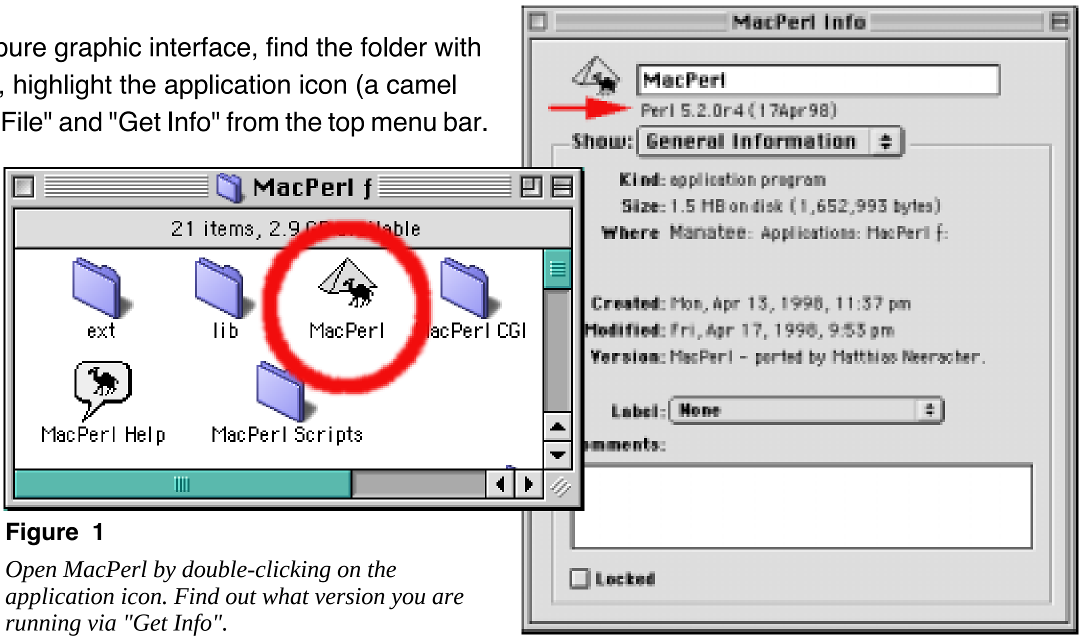
```
$ perl -v
This is perl, v5.6.0 built for i386-linux
Copyright 1987-2000, Larry Wall
Perl may be copied only under the terms of either
  the Artistic License or the GNU General Public
  License, which may be found in the Perl 5.0
  source kit.
Complete documentation for Perl, including FAQ
  lists, should be found on this system using `man
  perl' or `perldoc perl'.  If you have access to
  the Internet, point your browser at http://
  www.perl.com/, the Perl Home Page.
$
```

Solaris

```
seal% perl -v
This is perl, version 5.003 with EMBED
   built under solaris at Jan 30 1997 21:13:45
   + suidperl security patch
Copyright 1987-1996, Larry Wall
Perl may be copied only under the terms of either
  the Artistic License or the GNU General Public
  License, which may be found in the Perl 5.0
  source kit.
seal%
```

<u>Macintosh</u>

On the Mac, with its pure graphic interface, find the folder with the MacPerl application, highlight the application icon (a camel and a pyramid!), select "File" and "Get Info" from the top menu bar.



MacPerl
Perl 5.2.0r4 (17Apr98)
Show: General Information
Kind: application program
Size: 1.5 MB on disk (1,652,993 bytes)
Where Manatee: Applications: MacPerl ƒ:
Created: Mon, Apr 13, 1998, 11:37 pm
Modified: Fri, Apr 17, 1998, 9:53 pm
Version: MacPerl - ported by Matthias Neeracher.
Label: None
mments:
Locked

Figure 1

Open MacPerl by double-clicking on the application icon. Find out what version you are running via "Get Info".

1.4 Examples of Perl in use

Let's log in.

You log in as ________________________________ (please write your account name here[1])

A window will appear with your computer name[2] as the prompt. Move the mouse into this window and type:

```
calc
```

and press the `[return]` or `[enter]` key.

```
seal% calc 6 + 7 + 7 + 7 + 7
calc:  Copyright Well House Consultants 2003
result: 34
seal% calc
calc:  Copyright Well House Consultants 2003
Calculate what: 6 + 4 * 7
result: 34
Calculate what: $/5
result: 6.8
Calculate what:
seal%
```

Figure 2

This example was run on "seal", a Unix workstation running Solaris.

Calculator

Our little calculator can take sums entered on the command line[3] and print out the results.

If you just typed `calc` and pressed `[return]`, you'll be prompted to enter a sum, and the result will be printed. You can then do a series of calculations ... just keep typing them in. When finished, just press `[return]` on its own. The result on the last calculation can be used in the next one by using the `$` character.

Screen locator and counter

Another Well House Consultants' activity is providing support for a product called "RasterFlex" which allows users to add extra screens to their workstations.

To check the installation has worked, we use a script to search for and report the names and numbers of screens that the operating system can see:

```
seal% s_c3

screen found: VITec,RasterFLEX-HR0 at sbus0: SBus slot 2 0x0 SBus level 5 sparc ipl 7
That is /devices/sbus@1,f8000000/VITec,RasterFLEX-HR@2,0
it is known as    /dev/fb   /dev/rfx0   /dev/fbs/rfx

screen found: cgsix0 at sbus0: SBus slot 3 0x0 SBus level 5 sparc ipl 7
That is /devices/sbus@1,f8000000/cgsix@3,0
it is known as    /dev/fb1   /dev/fbs/cgsix0

system appears to have 2 screens

seal%
```

Figure 3

Running the screen locator

1 Example: `p1`
2 Our computers are usually named after fish, like "cod".
3 i.e. after the word "calc" but before you pressed [return]

A talker

One of our servers is permanently running a Perl script and anyone who wants on our LAN can connect and use it as a general talking board. Here's a sample:

Figure 4

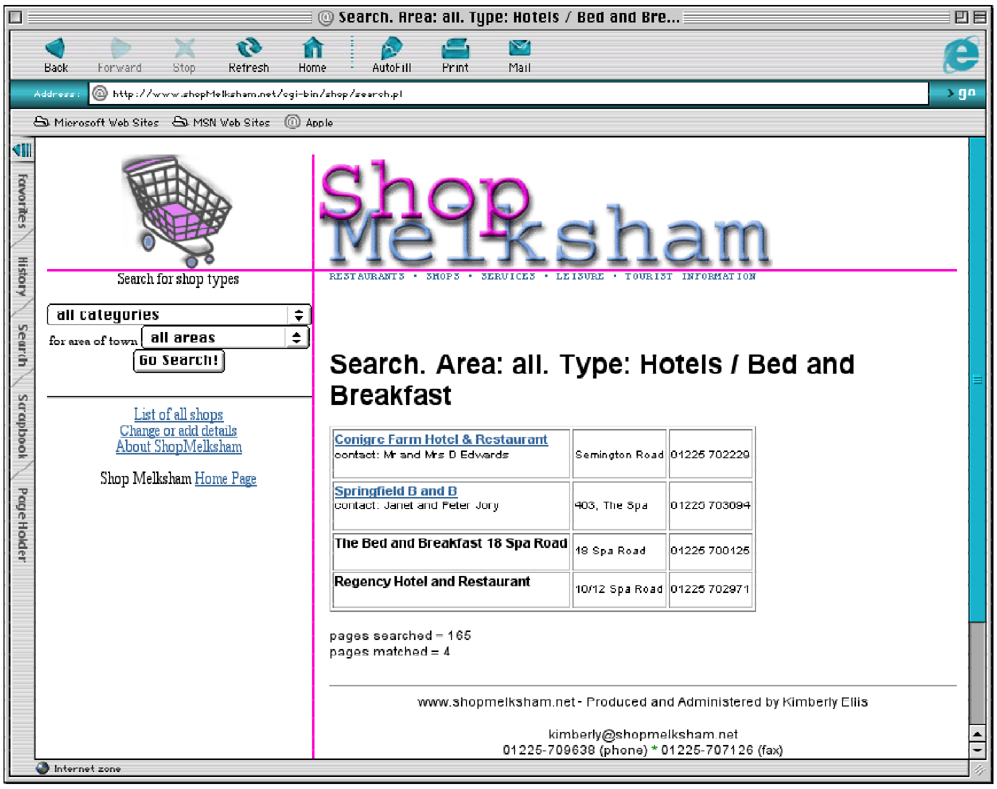
Running the talker

```
flipper% telnet lecht 7777
Trying 192.168.200.130...
Connected to lecht.
Escape character is '^]'.
*** flipper has come online ***
Link Established! .q to quit, .w for who
.w
flipper enquires who:  flipper
*** dolphin has come online ***
dolphin enquires who:  dolphin flipper
Hi there, Dolphin! How's the course going?
flipper: Hi there, Dolphin! How's the course going?
dolphin: .name peacock
Test OK ... Thanx!
flipper: Test OK ... Thanx!
dolphin: you see my "attempts" too...not fair!
.q
Connection closed by foreign host.
flipper%
```

A changing web page

The illustration below shows a page that changes whenever you request a certain category of shops.

We don't have to prepare all the pages ahead of time; we simply divert requests for this page to a Perl script which generates them on the fly, according to whatever the user has requested.



http://www.shopMelksham.net/cgi-bin/shop/search.pl

Search. Area: all. Type: Hotels / Bed and Breakfast

Conigre Farm Hotel & Restaurant contact: Mr and Mrs D Edwards	Semington Road	01225 702229
Springfield B and B contact: Janet and Peter Jory	403, The Spa	01225 703094
The Bed and Breakfast 18 Spa Road	18 Spa Road	01225 700125
Regency Hotel and Restaurant	10/12 Spa Road	01225 702971

pages searched = 165
pages matched = 4

www.shopmelksham.net - Produced and Administered by Kimberly Ellis

kimberly@shopmelksham.net
01225-709638 (phone) * 01225-707126 (fax)

Figure 5

This page changes according to the user's request.

Let's get on to the fundamentals of the language now. We're assuming for the purpose of this course that you've already had some practical programming experience.

2 Hello Perl World

2.1 Let's study a first program

Here's the code that we've edited into a file called "hello".

```
# Copyright - Well House Consultants, 2003
# Perl Basics - first program!
print "Hello; Welcome to this Perl course ";
```

How do we enter our program?

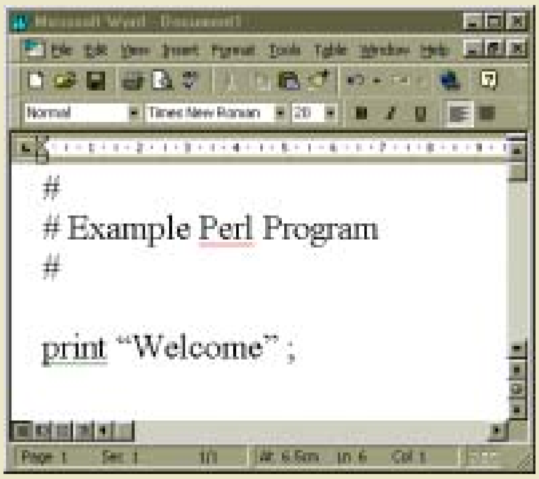
You can use any text editor you like ... a point-and-click editor such as Notepad on a PC, SimpleText on a Macintosh, or Textedit on a Unix box, through to a more sophisticated editor such as vi or emacs. Provided you produce a text file, it doesn't matter!

We don't want this to become a course in operating systems and editors, so we've wrapped a point-and-click editor under the name "edit" on your workstation. It will be at the command line, in a pull-down menu, or both!

Users familiar with vi may use that editor on our Unix and Linux boxes.

When you've finished typing in a program, save it away in a file. Why? Because when you run a program, the file is read and the instructions in that file are performed. If you forgot to save the file, you'll be running the previous version!

At the moment, you can choose to give the file any name you like as long as it's acceptable to the computer. Later you'll learn that there sometimes are rules to file names you must stick to in certain circumstances.



```
#
# Example Perl Program
#

print "Welcome" ;
```

If you're familiar with word processors such as Microsoft Word and wish to use them ... usually you can, provided that you save just the text without all the formatting information. In Word, do that by saving as "MS-DOS Text with Line Breaks". If you use "Text Only" you'll have a problem with smart quotes and other similar facilities!

You may get a warning message that you'll be losing formatting information when you save in this way. Yes, you WILL loose that information ... but then, perl doesn't want it!

How do we run our program?

By typing in the word `perl` followed by the name of the file into which we've saved the program.[1]

Let's run the example of the "hello" program we listed out above. Firstly on "coypu" (our Microsoft Windows machine):

```
C:\perlcourse>perl hello
Hello; Welcome to this Perl course
C:\perlcourse>
```

and then on "seal" (our Unix box):

```
seal% perl hello
Hello; Welcome to this Perl course seal%
```

Good. It works on both of them. But, notice the two different prompts and where the new lines end up.

What were the components of that program file we typed in?

[1] You'll learn tomorrow how to run your program without having to type the word "perl" every time!

- **Executable Statements**

At least one executable statement (something that the computer will execute when the program is run).

How this statement is written must be correct in format.[1] If you get it wrong, either the program will do the wrong thing, or it won't run at all -- it will just display an error message.

In the case of our first program, there's just one executable statement:

```
print "Hello; Welcome to this Perl course ";
```

and it's made up of three parts.

The first two will vary from one statement to the next. Here we have `print` telling Perl to

1) print the following part(s) and
2) " ..... " -- a literal string of text to be printed.

All statements should end with a ";" character so that Perl knows where one statement ends and the next begins.

- **Comments**

If this is your first time programming, you'll find it very useful being able to add some information that Perl does not read. A comment allows you to document what the command is doing so that the program will be easy to understand if you or someone else has to come back to it.

If you've programmed before, you already know it's worth spending a little time ensuring your programs are well commented unless they're going to be used just once or twice and then thrown away.

In our Perl example, a comment is started by a # character.

<u>Example</u>:

```
print "Greetings";
# Say hi to the user!
```

You can also add comments onto the end of a line by adding a space character followed by a # and then your comment.

<u>Example</u>:

```
print "Greetings"; # Say hi to the user!
```

Although the syntax of Perl demands that you put spaces in certain places, and that you don't put them in certain others,[2] you can often add spaces where you like.

And that can be any number of spaces, or any other white space characters such as tabs, new lines and line feeds.

You can put several statements on one line, or split a statement over several, or inset a series of statements from the left margin so that the human reader can see that they go together.

Perl doesn't care about comments, but you and subsequent human readers do!

[1] Its "syntax"

[2] Don't start splitting up the word "print" for example.

How do I do more than one thing in a program?

Simply place several statements in your program file. Perl will run the statements in turn, and in that same order.

```
# hello_again - second program!
# Good idea to say what the program does,
# who wrote it, which version it is ...
print# message to user
"Hello; Welcome to this Perl course. ";
print# another message to user
"We'll do an exercise soon. ";
# We'll make it more interesting soon!
```

```
graham@otter:profile/book> perl hello_again
Hello; Welcome to this Perl course. We'll do
  an exercise soon.
graham@otter:profile/book>
```

Figure 6

Running "hello_again" using Linux this time.

What if I make a mistake?

There are two types of mistakes you can make as you enter your program, syntax errors and errors of meaning.

- **Syntax errors**

  In such a case, Perl won't understand your program.
  Let's make an intentional error:

```
# wrong - program with a mistake!
print   # message to user
    "Hello; Welcome to this Perl course. "
print   # another message to user
        "We'll do an exercise soon. ";
```

The error is tiny (can you spot it?), but when I try and run it:

Figure 7

Running "wrong" to illustrate an error message.

```
graham@otter:profile/book> perl wrong
syntax error at wrong line 5, next token???
Execution of wrong aborted due to compilation errors
graham@otter:profile/book>
```

The code doesn't run.
Solution:
- Pull the file into an editor
- Find the error
- Correct the error, save the code, run it again!

- **Errors of meaning**

What happens when Perl can understand your program, but you've not asked it to do the right thing? Here's an example:

```
# poor - program with a mistake!
print# message to user
   Hello  ;
print# another message to user
   "We'll do an exercise soon. ";
```

When it runs, we get:

```
graham@otter:profile/book> perl poor
We'll do an exercise soon.
graham@otter:profile/book>
```

Figure 8
Running "poor" to illustrate an error message.

It HAS run. And at first glance you might say "fine". But it isn't. The word "Hello" didn't print out. Why?

YOU MUST CHECK THAT YOUR PROGRAM DOES EXACTLY WHAT YOU WANT.

Some languages will spot a lot of errors, but Perl has so many facilities that making a small change often causes not an ILLEGAL syntax, but means something different but still LEGAL. And even if you do something silly, Perl assumes you know what you're doing.

You can specify options on the Perl command line to tell the language to behave in different ways. If you run it with `-w` you'll be able to get Perl to give you warnings about legal things you're doing that are silly / dangerous / a bit odd. Let's try that on our latest example:

```
graham@otter:profile/book> perl -w poor
Name "main::Hello" used only once: possible typo at poor line 3.
Filehandle main::Hello never opened at poor line 2.
We'll do an exercise soon.
graham@otter:profile/book>
```

Figure 9
Running "poor" with a -w option to find where the error occurred.

Well, that certainly shouted about "Hello" didn't it?

As you learn more about Perl during this week, you'll come to understand the other terms such as `main::` and "file handle" that have cropped up. [After all, you did just used a facility you didn't even know existed!]

What if I want to print on several lines?

You could place a new line character into a double-quoted string, but it would mean the source code doesn't look good.

Perl provides a special way of writing a new line:

```
\n
```

Since the backslash itself is a special character, the way to write a "\" character within Perl is to use two backslashes together (`\\`).

2.2 Summary

- Use your favourite editor to write a Perl program to a plain text file.
- Run that program by typing in `perl <program name>` (where "program name" is the name you've given the file).
- If the program's syntax is correct, the instructions it contains are performed; otherwise it gives an error message or performs the function incorrectly.
- `perl -w` can be used to give warning messages if you're doing something odd.

Within the Perl program, you have:

executable statements	ending with	`;`
comments	staring with	`#`

The executable statement we used was the word `print` followed by a constant string of text we wanted to print, written in double quotes. We can use most characters directly, but there are some specials:

`\n`	new line
`\\`	backslash
`\"`	double quote character
`\$`	dollar character

You can place white space almost anywhere you like in your program to aid readability.

Exercise

Change to the trainee area of your account[1] before you do this exercise.

- Set up a Perl script yourself (call it "mine") to print out your name.
- Run it; check it worked correctly.

Our example answer is mine

This is what your results could look like:

Sample

```
graham@otter:profile/answers> perl mine
Hello Pooh Bear. There's honey in the larder!
graham@otter:profile/answers>
```

[1] Just type in `trainee` at the command line.

3 Variables and Operations

3.1 Reading from the user

You will naturally want to do more than just print text. After all, a plain word processor can do this. You've come on this course to learn how to write a program whose action varies depending on what is entered, such as reading in information from a person running your program.

Let's go through this process step by step:

1) Prompt the user so that he knows he has to enter something.
   `print "please enter your name: ";` [1]
2) Read from the keyboard.[2]
   By putting `<` and `>` around it, we say "read from"
3) Store away the result of this operation, using what we call an " assignment". On the right-hand side of an equals sign (=), we write the read operation from stage 2.
   On the left-hand side ...
4) The assignment needs to know where we want to store the information. We don't have to give some highly technical computer memory location code. Instead, we choose a name ourselves that's relevant to the application. We must, though, put a `$` character in front of it so that Perl knows it's the name for a memory slot, the contents of which can vary.

```
   $users_name = <STDIN> ;
```

To show this has worked, we'll now print out the contents of the variable. It's a `print` statement like we used before, but rather than a constant string of text, we'll print the variable's contents:

```
print $users_name;
```

Here's the whole program:

```
# greeting - read the user name and
# echo it back in a greeting!
# input
print "please enter your name: ";
$users_name = <STDIN> ;
# output
print "This is a program to greet ";
print $users_name;
print "on this course\n";
```

```
graham@otter:profile/book> perl greeting
please enter your name: Pooh
This is a program to greet Pooh
on this course
graham@otter:profile/book>
```

Figure 10
Running Perl program "greeting".

[1] Note: Don't add a new line, but do leave a space after the last word.

[2] The keyboard is known as STDIN. It's something called a "file handle" that we'll explore later.

3.2 More about variables

We've read into a variable and printed out its contents.

What are the rules for choosing a name for a variable?

- Variables of this type have names that start with `$`.
  That's to distinguish them from other parts of the Perl language.
- The `$` must be followed by a letter (upper or lower case).
- Then any combination of letters, digits, underscores.
- Names are case significant.

Some suggestions:

- be descriptive
- make it not too long, but not too short
- avoid too many similar names for variables

Whilst you can use almost any variable name you wish, you'll learn as you go on that some names have special significance and so ... a further suggestion:

- avoid names that have other uses such as `$a`, `$b`, `$MATCH`, `$RS` and `$ARGV.`

  Apart from `$a` and `$b`, you should be safe if you use any lower case letters!

How much information can a variable contain?

You can put as much text as you like into a variable -- up to the memory capacity of your computer.[1]

Do I have to tell Perl about a variable before I use it?

No, just use the name and Perl will create it as the program runs, making it large enough to take the text that was entered.

Can I reuse a variable?

Yes, if you put something into a variable which already exists and has text in it, the old contents will be lost and the new contents stored. Perl even shortens or lengthens the variable if you've assigned less or more text this time.

1 And sometimes beyond its capacity if it uses "swap space" on disk when memory gets full.

Can I copy a variable?

You can copy the *contents* of a variable. Study this example and see if you can work out what's happening.

```
# couple - reads two names and echos them
# input
print "please enter first name: ";
$users_name = <STDIN> ;
$first_name = $users_name;
print "please enter second name: ";
$users_name = <STDIN> ;
$second_name = $users_name;
# output
print "This is a program to greet ";
print $first_name; print " and ";
print $second_name;
print " on this course\n";
```

In normal practice, we would have read straight into `$first_name` and `$second_name`, but then we wanted to illustrate the points we've just been talking about. Let's see that in action:

Figure 11
Running Perl program "couple".

```
graham@otter:profile/book> perl couple
please enter first name: Christopher
please enter second name: Pooh
This is a program to greet Christopher
and Pooh
on this course
graham@otter:profile/book> perl couple
please enter first name: Pooh
please enter second name: Christopher
This is a program to greet Pooh
and Christopher
on this course
graham@otter:profile/book>
```

That's good, but I think you might have hoped for a one-line greeting such as:

Figure 12
Running Perl program "couple" as though it output a continuous line.

```
graham@otter:profile/book> perl couple
please enter first name: Pooh
please enter second name: Christopher
This is a program to greet Pooh and Christopher on this course
graham@otter:profile/book>
```

What happened?

The new line problem

Why are all the extra lines on my output?

What did your user enter when prompted for the first name?

```
P-o-o-h-[enter]
```

So what was saved into the variable $users_name?

```
P-o-o-h-[enter]
```

So what was copied into the variable $first_name?

```
P-o-o-h-[enter]
```

So what was printed when that variable's contents were printed?

```
P-o-o-h-[enter]
```

EXACTLY!

How can I get rid of that new line character?

You can do things to change the contents of variables; much of programming is about doing that, and one of the functions that's built into Perl is `chop`.

```
chop $first_name;
```

means:

- Take the contents of the variable `$first_name`
- Remove the last character
- Store the result back into `$first_name` (overwriting the old value)

So ... this program:

```
# one_line - two names are echoed back on one line
# input
print "please enter first name: ";
$users_name = <STDIN> ;
$first_name = $users_name;
print "please enter second name: ";
$users_name = <STDIN> ;
$second_name = $users_name;
# manipulation
chop $first_name;
chop $second_name;
# output
print "This is a program to greet ";
print $first_name;
print " and ";
print $second_name;
print " on this course\n";
```

is run as:

```
graham@otter:profile/book> perl one_line
please enter first name: Christopher
please enter second name: Pooh
This is a program to greet Christopher and Pooh on this course
graham@otter:profile/book>
```

Figure 13

Running Perl program "one_line"

3.3 How do I do calculations?

You've seen a number of operations already ...

`print`	write a variable or constant out
`chop`	remove last character from a variable
`< ... >`	read in from a file handle
=	save what's on the right into a variable named on the left

Arithmetic operations

You can also perform arithmetic operations on the contents of variables:

```
$difference = $age1 - $age2 ;
```

Let's see it in action:

```
graham@otter:profile/book> perl ages
please enter first age: 46
please enter second age: 43
The second person is 3 years younger than the first
graham@otter:profile/book>
```

Figure 14

Running Perl program "ages"

```
# ages - compare two ages
# input
print "please enter first age: ";
$age1 = <STDIN> ;
print "please enter second age: ";
$age2 = <STDIN> ;
# get difference
$difference = $age1 - $age2 ;
# output
print "The second person is ";
print $difference;
print " years younger than the first\n";
```

You'll notice that we didn't have to do anything special to convert the strings of text that were entered into numbers; no need to `chop` or anything.

If we do a subtraction, Perl knows it's an arithmetic operation and extracts the numbers from the strings of text that were typed in. That also means it gets rid of the new line characters as part of the subtraction so that after the subtraction line, the variables contain:

`$age14-6-[return]`(held as a string)
`$age24-3-[return]`(held as a string)
`$difference3` (held as a number)

Other calculations include:

+	addition
*	multiplication (not "x" for multiply!)
/	division

Several operations at the same time?

Yes, you can. You can write a complicated expression involving as many operators, constants and variables as you like!

Perl has a complex set of rules to tell it in what order all the operators should be carried out.

The operators you've met so far are executed in the following order:

`< ... >`	order irrelevant
`*` and `/`	left to right
`+` and `-`	left to right
`=`	right to left

Let's work out the average age of two people:

```
# average - average 2 ages
# input
print "please enter first age: ";
$age1 = <STDIN> ;
print "please enter second age: ";
$age2 = <STDIN> ;
# get average
$average = $age1 / 2 + $age2 / 2 ;
# output
print "The average age is ";
print $average; print "\n";
```

```
graham@otter:profile/book> perl average
please enter first age: 43
please enter second age: 46
The average age is 44.5
graham@otter:profile/book>
```

Figure 15
Running Perl program "average"

Can I change the order things are done within a statement?

Yes, you can. You can use round brackets `( )` to change the order of evaluation. Instructions within inner brackets are performed first.

Just as you can change the order in which operations, such as `+` and `-` are performed, you can also change the order in which functions such as `chop` and `print` are performed.

While we're busy reducing the number of statements (but also making them more complex), it would be a good time to reduce the number of `print` statements. You are allowed to give a comma-separated list of things to print.

Let's see these facilities in use:

```
# ave2 - average 2 ages
# input
print "please enter first age: ";
$age1 = <STDIN> ;
print "please enter second age: ";
# get average
print "The average age is ",
  $average = ($age1 + ($age2 = <STDIN>))
  / 2 ,     "\n";
# following line not good - too confusing!
print "Ages ",$age1+0," and ",$age2;
# but an excellent class discussion
```

```
graham@otter:profile/book> perl av2
please enter first age: 46
please enter second age: 43
The average age is 44.5
Ages 46 and 43
graham@otter:profile/book>
```

Figure 16
Running Perl program "ave2"

Remember, comment it well. Better clear than concise!

3.4 Summary

Reading from the user

- use file handle STDIN
- surround it by the read from operator parts `<STDIN>`
- assign what you read (using `=`)
- into a variable

Variables

- names start with `$` character
- then a letter
- then letters, numbers, underscores

Variable names are case significant. Variables are created and reused on the fly. Their length is adjusted automatically so they can hold as much text as you like.[1] When you enter a string, it has a `[return]` on the end. Use the chop function if you need to get rid of this.

When you perform an arithmetic calculation on a string, Perl automatically ignores the `[return]` for the purpose of the calculation. Calculations may be performed using operators such as `+` `-` `*` and `/`. You can alter the order in which they're performed by using round brackets.

You can specify a whole list of things to be printed in a single `print` operation, such as constants, contents of variables and even expressions to be worked out within the line.

1 A "string"

Exercise

Write a program to ask the user his/her name, and then print out a personalised welcome message.

Our example answer is amulree

Sample

```
seal% amulree
Please enter your name Graham Ellis
This is to welcome Graham Ellis
seal%
```

For Advanced Students

Write a program to ask for a temperature in fahrenheit, and print it out converted to centigrade.

To convert F -> C

Take away 32

Divide by 9

Multiply by 5

Test it. Comment it as well.

Our example answer is dunkeld

Sample

```
seal% dunkeld
Please enter a temperature in degrees fahrenheit 212
212 degrees F converts to 100 degrees C
seal% dunkeld
Please enter a temperature in degrees fahrenheit 32
32 degrees F converts to 0 degrees C
seal% dunkeld
Please enter a temperature in degrees fahrenheit -40
-40 degrees F converts to -40 degrees C
seal% dunkeld
Please enter a temperature in degrees fahrenheit 100
100 degrees F converts to 37.7777777777778 degrees C
seal%
```

4 Perl Fundamentals

4.1 First Perl program

When you're writing your program, you use an editor to enter the program (the "source") and save the text into a plain ASCII text file. Use any editor you like, provided it can write a plain text file.

The system you're using for this course will have a variety of editors available depending on what platform you're using. You'll have some of vi, emacs, pico, ex, edit, notepad, simpletext and textedit available.

If you do have an editor you're familiar with, that's great. If not, we have set up all the systems on this course so that you can type the filename and a point-and-click style editor will appear. Once you've entered your program, remember to save the file! Here's our "first program"

```
#
# Copyright Well House Consultants, 2003
# "language" - showing some Perl language constructs
#

print ("Welcome to Perl Programming\n");

$temperature = 212;   $factor = 5.0 / 9.0;
$becomes = ($temperature - 32) * $factor;

$units_1 = "Farenheight";
$units_2 = "Centigrade";

=head1 Documentation for "langauge"

"language" is a sample program that converts
a temperature in Farenheight into
Centrigrade

=cut

print $temperature," degrees ",$units_1,  # from
      " becomes ",
      $becomes," degrees ",$units_2,       # to
      "\n";

__END__

Since the Perl interpreter stops at the __END__
line, I can supply further comments and documentation
at that point, and it's efficient at run time.
```

You can run that as we did *(right)* by typing in `perl language.`

Let’s look at the component parts.

```
seal% perl language
Welcome to Perl Programming
212 degrees Fahrenheit becomes 100 degrees Centigrade
seal%
```

Figure 17

Running Perl program "language".

4.2 Comments and documentation

Firstly, every program that you write should include documentation. Are you going to remember what some obscure piece of code did when you come back to it six months later, and are your users going to be able to work out how to use your program from reading it?

You'll want to include:

• comments	for yourself and other programmers
• documentation	for your users

Both comments and built-in documentation are ignored by the Perl language interpreter itself, and you are encouraged to be generous in using them.

Comments

Every programming language supports comments, and every good programmer comments his code well. Although there are no specific programs that make use of comments, when a maintainance programmer comes back to the code at a later date his job will be eased by their generous provision. It's said that the majority of the cost of code is in the maintenance rather than the development, and that only one in five pieces of code are maintained by their original author throughout their useful life.

There are four ways that you might like to consider to comment your programs:

1. If you start a line in Perl with a `#` character, that line will be treated as a comment.
2. Under most circumstances, a `#` in the middle of a line also signifies the start of a comment that runs to the end of a line.
3. White space may be placed between any language elements in Perl, and you can use as much or as little white space as you like, thus setting out your code to be more readable.
4. If you include a line that reads
   `__END__`
   in your file, then the Perl interpreter will stop parsing the file at that point, which means that anything thereafter will be treated as comments or documentation. Since documentation requires lines that start with a `=` character, it follows that you can place more or less whatever you wish after the `__END__` and have it taken as a comment.

Documentation

Also known as "documenation comments" in some languages.

Your users will all want documentation but may have different views as to what format they would like; some would like HTML, others Postscript, and yet others plain text files. In Perl, all can be provided from a single set of embedded documentation comments written in a notation known as POD ("Plain Old Documentation").

A POD directive starts with the `=` character on column1, followed by a keyword (such as `head`, `over`, `back` or `for`). There may be other text on that directive line, then there's a number of lines of text making up the body of the documentation, followed by an `=cut` line.

In our first example program there was one POD block:

```
=head1 Documentation for "langauge"

"language" is a sample program that converts
a temperature in Farenheight into
Centrigrade

=cut
```

which calls up a major heading, and then provides a chunk of description.

When run as a Perl program, POD blocks are ignored, but extra programs are supplied with the Perl distribution which include POD interpreters. Let's get plain text out of our program using `pod2text`:

```
$ pod2text language
Documentation for "langauge"
    "language" is a sample program that converts
    temperature in Farenheight into Centrigrade.
$
```

Without comments and PODs, the Perl program will still work but will be impractical to read:

```
print
("Welcome to Perl Programming\n");
$temperature=212;$factor=5.0/9.0;
$becomes=($temperature-32)*
$factor;$units_1=
"Fahrenheit";$units_2=
"Centigrade";print
$temperature,"degrees ",$units_1,
" becomes ",$becomes," degrees "
,$units_2,"\n";
```

Executable statements

The active statements in our Perl program (after we've discounted the comments) are each ended with a `;` character, and each is executed in turn.

It's the `;` character which tells the Perl language compiler[1] where one ends and the next starts. Rather like a full stop (.) in English.

It is not the end-of-line character that separates statements as it does in shell programming languages, Fortran and some others. Leave the `;` off and you'll probably get an error reported!

```
# language -
#
# Copyright Well House Consultants, 2003
# "language" - showing some Perl language constructs
#

print ("Welcome to Perl Programming\n");

$temperature = 212;    $factor = 5.0 / 9.0;
$becomes = ($temperature - 32) * $factor;

$units_1 = "Farenheight";
$units_2 = "Centigrade";

=head1 Documentation for "langauge"

"language" is a sample program that converts
a temperature in Farenheight into
Centrigrade

=cut

print $temperature," degrees ",$units_1,  # from
      " becomes ",
      $becomes," degrees ",$units_2,       # to
      "\n";

__END__

Since the Perl interpreter stops at the __END__
line, I can supply further comments and documentation
at that point, and it's efficient at run time.
```

Figure 18

Running Perl program "language" -- comments have been greyed.

Here are our executable statements in order:

```
print("Welcome to Perl Programming\n");
$temperature=212;
$factor=5.0/9.0;
$becomes=($temperature-32)*$factor;
$units_1="Fahrenheit";
$units_2="Centigrade";
print$temperature," degrees ",$units_1,
" becomes ",$becomes," degrees ",$units_2,"\n";
```

and it still runs the same!

1 more about the compiler later!

Print statement

To start, we'll print information out from our Perl program using a `print` function. A `print` function comprises the word "print" followed by a comma-separated list of things to be printed. And it ends, of course, with the mandatory semicolon.

In our example, the output from our `print` statements came to the screen, but Perl really sends output to something called "Standard Out" or `STDOUT`. What happens to information sent to `STDOUT` is controlled by the program which called up Perl -- at the moment, your command-line interpreter. What are the alternatives? Lots, including:

- The screen
- A file
- Another command
- A web visitor's browser

It's not Perl that controls this, nor the programmer. It's the user and the environment in which he runs your Perl program!

Our comma-separated list can contain a variety of things to print out and we'll introduce many more as we go through. In this example, we used

- Constant (unchanging) text strings written in double quotes (may include special codes like `\n` for a new line)
- Variable (changing) text strings
- Variable (changing) numbers

Variables and assignments

We need to be able to store information within our program and to have it be in common with almost every other language, we use the concept of a named variable. It's much easier to refer to something by name than by an obscure location number. Using a name also means that our program's memory can be re-arranged and we won't have to change the code!

Perl allocates memory for variables dynamically. Let's look again at the first two executable statements of our program:

```
print("Welcome to Perl Programming\n");
$temperature=212;
```

- While the first statement is being executed, the variable `$temperature` does not exist. There is no memory allocated for it and its name is not held in any "symbol table".
- The second statement is an assignment. As it's executed, Perl works out the result of whatever is to the right of the `=` sign and saves it into a variable named on the left.

When Perl comes to save the value 212, the variable `$temperature` does not exist, and so Perl:

- Looks at the type and size of the information being assigned
- Allocates as much space as necessary
- Creates the name in its "symbol table" pointing to that space

In order for the Perl language to understand that part of a statement is a reference to a variable, the variable name must conform to certain rules:

- 1st character must be `$`
- 2nd character must be a letter -- upper or lower case
- Subsequent characters must be letters, digits or underscores
- Variable names can be as short as `$j,` or as long as you wish
- Variable names are case sensitive -- `$well` is not the same as `$Well`

Notice differences here to other languages:

IN OTHER LANGUAGES, you often have to declare variables before you use them.In Perl, you can just use them!

IN OTHER LANGUAGES, you often have to state what type of information will be held in a variable. In Perl, you can just assign the information!

IN OTHER LANGUAGES, you often have to tell the system how much memory to allocate for each variable (especially with a character string), but in Perl the memory allocation is automatic.

IN OTHER LANGUAGES, you can start a variable name with a letter (or you only have to state `$` when you're using rather than setting the variable). In Perl, you must always use the `$` or it means something else!

Because Perl has dynamic memory allocation, you'll discover later that you can get rid of a variable and release the memory it used by writing something like

```
undef $temperature;
```

but you won't do that very often. It's pointless to release individual variables unless you're concerned about memory use. And, in any case, the memory will be released as the Perl program exits!

Calculations

The next statements in our first sample program are:

```
    $factor=5.0/9.0;
    $becomes=($temperature-32)*$factor;
```

In each case, Perl calculates the expression on the right, then assigns the result to the variable named on the left.

The expression being calculated will be comprised of variables (the current value of which will be used) and / or constants, which will be linked together using a number of operators. Being Perl, of course, there's a huge number of operators. Here are some, with examples:

`$g + $h`	add values of `$g` and `$h`
`$g - $h`	subtract value of `$h` from value of `$g`
`$g * $h`	multiply values of `$g` and `$h`
`$g / $h`	divide value of `$g` from value of `$h`
`$g % $h`	divide value of `$g` from value of `$h` -- result is the remainder
`$g ** $h`	raise value of `$g` to the power of value of `$h`
`- $g`	value of `$g`, negated
`+ $g`	value of `$g`
`abs $g`	the absolute value of `$g`
`atan2 $g,$h`	arc tangent of value of `$g` divided by value of `$h`
`cos $g`	the cosine of the value of `$g` (taken to be radians)
`exp $g`	e (2.71828182845905) to the power of the value of `$g`
`int $g`	the integer part of the value of `$g`
`log $g`	the natural logarithm of the value of `$g`
`rand $g`	a random number between 0 and the value of `$g`
`sin $g`	the sine of the value of `$g` (taken to be radians)
`sqrt $g`	the square root of the value of `$g`

In which order are these operations performed?

`**`	first - from right to left, then
unary + and -	then
`*` `/` and `%`	left to right, then
+ and -	left to right, then
functions	right to left

which is the same as in other languages.

Also, as in other languages, you can use round brackets to change the order of operations. Our second example calculation did just that as we wanted the subtraction done before the multiplication. Many programmers always put brackets around values to the right of functions for clarity; write `sin($g)` rather than `sin $g`. We have done the same with the first `print` of our sample program.

4.3 Summary

Type your Perl program into a plain text file, using any suitable editor. A program consists of comments and executable statements. Comments are:

- lines starting with # characters
- text after # if it's within a line
- blocks from `=for` to `=cut`
- white space

The assignment statement says "calculate what's on the right of an = sign and save it in the variable named on the left". You can choose any variable name provided that it starts with a `$` and a letter, and only contains letters, digits, and _ characters.

There are many operators available for calculation, including `+ - *` and `/` and the order in which they're performed, can be changed using round brackets.

To print results, use the word `print` followed by a comma-separated list of variables, constants and expressions.

Statements should always be separated by a semicolon.

Exercise

Write a program taking two amounts and adding them together, then converting them to a different currency. The first lines of your program will be:

```
$price1 = 16.50;
$price2 = 9.99;
$exch = 0.6325;
```

The prices are in pounds, and `$exch` is the conversion rate[1] from euros to pounds.

Calculate and print the total bill in euros.

Our example answer is to_euro

Sample

```
$ perl to_euro
converting to Euros
Total is 41.8814229249012 euros
```

[1] Due to the fact that the exchange rate between the pound and the euro changes more frequently than the Perl language, the conversion rate used in this example may be inaccurate at the time the course is run.

4.4 Reading data

Thus far, your program has always converted the same temperature, so it has always printed the same results. You need to be able to read data from the user at run time. That reading may be from a file or from the keyboard or from any number of other sources (including from another program running on the same or a different machine).

File handles

You don't specify directly where you're going to read from. Rather, you use a special type of variable called a "file handle" from which you'll read.

When you come to read from a file later, you'll learn how to associate a file handle with an actual file, but Perl already associates a file handle called `STDIN` (yes, written in capitals with no dollar!) with your main input stream, as defined by the program that called Perl. So it could be that `STDIN` reads from:

- The keyboard
- A file
- Another command
- A web visitor's browser

Read from operator

What are you going to do with the file handle? You're going to read from it. There's a special operator to do that

```
<...>
```

where ... is where you name the file handle. Thus:

```
$temperature = <STDIN>;
```

is "read from `STDIN`, up to and including a new line character, and save the result in `$temperature`."

Let's see the whole program:

```
seal% perl read_write
Please enter a temperature (deg F): 212
212
 degrees Fahrenheit becomes 100 degrees Centigrade
seal% perl read_write
Please enter a temperature (deg F): 98.4
98.4
 degrees Fahrenheit becomes 36.8888888888889 degrees Centigrade
seal%
```

Figure 19
Running Perl program "read_write".

```
# read_write - read, calculate, print results

print ("Please enter a temperature (deg F): ");

$temperature = <STDIN>;   $factor = 5.0 / 9.0;
$becomes = ($temperature - 32) * $factor;

$units_1 = "Fahrenheit";
$units_2 = "Centigrade";

print $temperature," degrees ",$units_1,  # from
      " becomes ",
      $becomes," degrees ",$units_2,       # to
      "\n";
```

Does that look good?

Not really; the converted figures are too accurate (we'll come back to that later) and why is there a new line after the number you entered?

Strings v numbers

The prompt asked for a temperature to be entered. You happen to know that temperatures are numbers and so (it appears!) did your user. But what was actually entered was a string of characters. So the variable `$temperature` contained

2 - 1 - 2 - [return]

and that's exactly what was printed back out later!

What is `2-1-2-[return]` minus 32?

"Silly question," you say. "Can't perform maths on a `[return]`." And Perl knows this. If you perform arithmetic on a string, Perl converts the string to a number as best as it can (and it won't complain). It then performs the maths on that resulting number. So:

```
$becomes = ($temperature - 32) * $factor;
```

starts off with `$temperature` containing a string (and we don't assign anything back to that variable, so it remains a string) and ends with `$becomes` being defined and containing a number.

No new line was printed after `100` on our output then. `$becomes` contained a number.

How would I print out the temperature without the `[return]`?

I could do it by converting it into a number. They look very odd, but any one of these would work:

as the temperature is read in:

```
$temperature = <STDIN> + 0;
$temperature = 1 * <STDIN>;
```

a little later:

```
$temperature = $temperature / 1;
$temperature = $temperature - 0;
```

or even in the `print` statement:

```
print $temperature+0," de...
```

String operators

It's probably better to remove the `[return]` character than manipulate the data as we did above. There are a lot of string functions and operators that will work on strings just like `+` and `cos` worked on numbers.

If I write

```
chop ($temperature);
```

the last character (whatever it is) is removed from the string held in the variable `$temperature`, and the string without that character is saved back into the same variable.

I can even write

```
chop ($temperature = <STDIN>);
```

and that will do what I want. But if I write

```
$temperature = chop (<STDIN>);
```

the variable will end up containing just the `[return]` character. `chop` alters two things -- the variable named as its parameter (it removes the last character from that) AND a return value. It passes back the character actually removed.

Therefore, I could not have corrected my program earlier on by writing

```
print chop($temperature)," de ...
```

`chop` always removes the last character from a string. If you don't know whether you have a new-line character there to be removed, use `chomp`[1] instead. `chomp` only removes the last character if it's a new-line character.

[1] `chomp` is Perl 5 only

Here's the program corrected using best practice:

```
seal% perl rw
Please enter a temperature (deg F): 212
212 degrees Fahrenheit becomes 100 degrees Centigrade
seal% perl rw
Please enter a temperature (deg F): -40
-40 degrees Fahrenheit becomes -40 degrees Centigrade
seal%
```

Figure 20
Running Perl program "rw".

```
# rw - read, calculate, print results (2)

print ("Please enter a temperature (deg F): ");

chop ($temperature = <STDIN>);
$factor = 5.0 / 9.0;
$becomes = ($temperature - 32) * $factor;

$units_1 = "Fahrenheit";
$units_2 = "Centigrade";

print $temperature," degrees ",$units_1,  # from
      " becomes ",
      $becomes," degrees ",$units_2,       # to
      "\n";
```

4.5 Summary

`<....>` is the "read from" operator.

To read from the keyboard, you read from `STDIN`, so `<STDIN>`. When you read, you actually read a string including a new-line character which you can remove using `chop`, `chomp`, or an arithmetic operation.

Exercise

Modify the first three lines of the program you wrote earlier to ask for (and read in) the two prices and the exchange rate. Ensure that the program reports back the exchange rate used.

Our example answer is to_euro2

```
$ perl to_euro2
Converting pounds to Euros
Please enter first amount: 16.99
Please enter second amount: 9.50
Please enter exchange rate for Euros to Pounds: .6325
Total is 41.8814229249012 at an exchange rate of .6325 pounds to 1 euro
$
```

5 More about the Perl Environment

5.1 Integrating your program with your computer

How have you been running your programs?

By typing in the word `perl` followed by the program name. And that's been fine, but in time you'll get tired of having to type `perl` .

And perhaps there will be other programs written in other languages on your computer? Do you want to have to type in the name of the language each time? No, you don't. Apart from the hassle involved, your users aren't going to be able to remember which program was in Perl, which was in C, which was a Korn shell script, and so on.

Although the examples we've looked at so far have been system-independent, this section is not. That's nothing to do with Perl ... it's the effect of the individual operating system.

5.2 Unix and Linux systems

Let's take the example program from the last section and try running it straight from the command line:

```
seal% rw
rw: Command not found
seal%
```

Even though we're currently in the directory that contains the file!

Executable path

Operating systems don't look everywhere for an executable program when you type a name in, only in certain directories. And very often that doesn't even include the current directory!

You could run the command using `./rw`, which states explicitly that it'll be in the current directory. Or you could alter your "path".

Depending on which particular command-line handler (shell program) you're running, the command to alter the path will differ. Here are some examples. If you're doubtful about which is correct for you, or if you want to learn more about Unix or Linux (yes, we do courses on both!), please ask your tutor.

Korn shell

```
    export PATH=$PATH:.
```

C shell and TC shell

```
    set path = ($path .)
```

Bourne shell and Bourne again shell (BASH)

```
    PATH=$PATH:.
    export PATH
```

These changes will only take effect in the current shell. If you log out and log in again, the changes will be lost. And if you move to another window, you'll probably have to set them there too.

To be made permanent, the changes can be edited into Unix or Linux's "dot files":[1]

`.cshrc`	C shell
`.profile`	Bourne shell
`.profile.ksh`	Korn shell
`.tcshrc`	TC shell
`.bashrc`	BASH

Let's see if that helped:

```
seal% set path = ($path .)
seal% rw
rw: Permission denied
seal%
```

We appear to have fixed one problem, just to encounter another.

File Permissions

You can read and write the file "rw", but you don't have permission to execute it. You may find it frustrating but it's to avoid you and your users just typing in a file name and trying to run data. After all, you'll probably have a lot more data files than programs!

Let's set the execute "bit" for the file owner:

```
seal% ls -l rw
-rw-r--r--   1 graham  wellho 414 Jan 18 17:30 rw
seal% chmod u+x rw
seal% ls -l rw
-rwxr--r--   1 graham  wellho 414 Jan 18 17:30 rw
seal%
```

Alternative commands:

`chmod u+x rw`	set program executable for owner only
`chmod ug+x rw`	set program executable for owner and group
`chmod a+x rw`	set program executable for everyone
`chmod 755 rw`	bad practice, but yes, it works!

and we should now be able to run the program:

```
seal% rw
Badly placed ()'s
seal%
```

Not our day, is it?

Telling the calling shell this is Perl

If we don't type in the word `perl` before the program name, how is the calling shell going to know to run the file using that language? It won't!

All shells look at the very first line of the text file they're going to execute and use that line to work out the language.

- If the first line does not start with a `#`, the file's a bourne script.

[1] Locators may vary depending on system configurations.

- If the line starts with `#!`, the rest of the line is taken as the command to run (with a full path).
- If the line starts with a `#` but there's no `!`, the C shell is used.

So our example used the C shell and that strange message was the result of the C shell trying to understand the Perl language.

In order to fix the problem, we need to change the first line of the file.

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

is the most common setting and we can then run the program directly:

```
seal% rw
Please enter a temperature (deg F): 176
176 degrees Fahrenheit becomes 80 degrees
  Centigrade
seal%
```

Do be aware that although `/usr/local/bin/perl` is the most common place to find Perl, it may be elsewhere instead. Other first lines that we have seen include:

```
        #!/bin/perl
        #!/usr/bin/perl
        #!/usr/local/bin/perl5.003
```

and this may even have to be changed between two computers of the same manufacture and running the same operating system!

The good news is that most web servers also understand this line and it works for your server-side scripting too. All you have to do is get your ISP to tell you where he's placed Perl!

It's also good news that you can continue to run your Perl program by typing

```
          perl rw
```

After all, that strange first line starts with a `#` character, so is taken by Perl as a comment.

5.3 Windows 98, 2000 and Windows NT systems

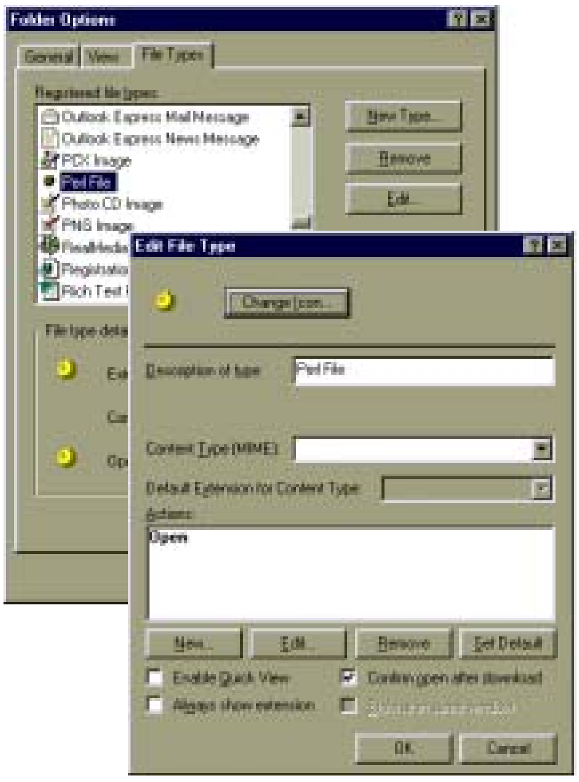
Associating your file with Perl

In order for Windows to recognise the file as a Perl program, you must use a file name ending in ".pl".

This association of the .pl extension with the Perl language should have been automatically made at the time that Perl was installed on the computer, but if you do need to change or edit the setting, select down menus as follows:

- My Computer
- View
- Folder Options
- File Types
- Perl [Or if not there, "new"]

Here is what you see:



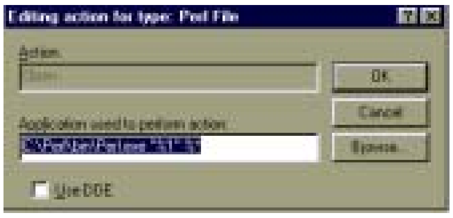
Description of type: Perl File

Actions: Open

Figure 21
Setting file types on the Windows platform.

If you double click on Perl, another menu appears.

If you double click on open, further changes can be made.


Editing action for type: Perl File
Action:
Application used to perform action:
OK
Cancel
Browse...
Use DDE

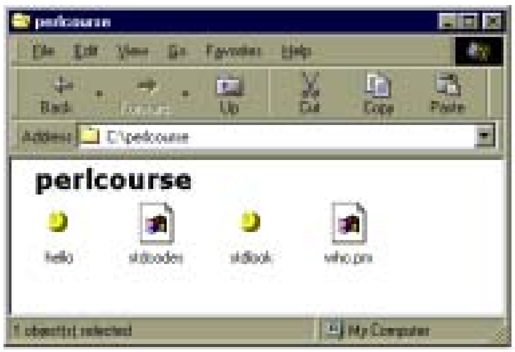
Action: Open

Application used to perform action: C:\Perl\bin\Perl.exe "%1" %*

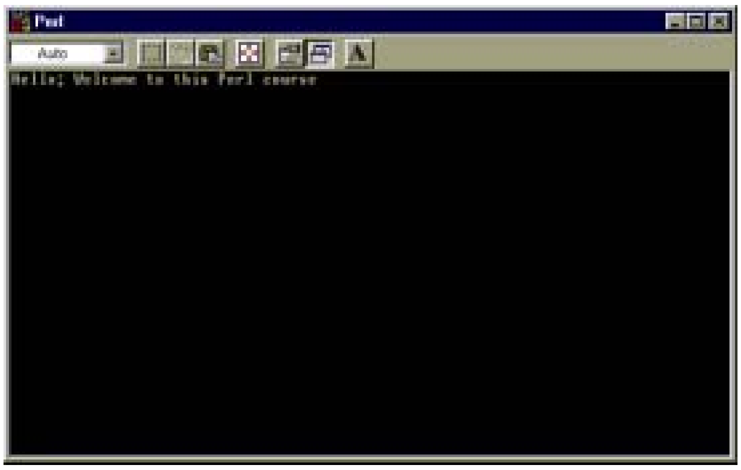
There is no harm in having a `#!/usr/local/bin/perl` line in your Perl program, but it will be ignored.

Running your program

Once you have associated your file with Perl using the extension, any menus that windows offers you will include the icon selected for the Perl program, but won't mention the extension.



Clicking on an icon to run a Perl program will bring up a fresh window and run the program with input and output in that window



```
Hello; Welcome to this Perl course
```

When the Perl program finishes running, the window closes automatically.

If you're looking to display final results for the user to read upon completion of your Perl program, this habit of the window disappearing is frustrating to say the least, and as a short term fix (we'll see better later!) you might like to add the following to the end of your Perl program:

```
<STDIN>;
```

Literally, "read a line from the keyboard ... and throw away what's been typed in!"

5.4 MS-DOS users

You can also run Perl programs in a DOS window. When you installed Perl, the path was probably set in your `\autoexec.bat` file but you may have declined to have that change made. You can set the path manually:

```
SET PATH=%PATH%;C:\PERL\BIN
```

or whatever. This will allow you to run a program simply by typing

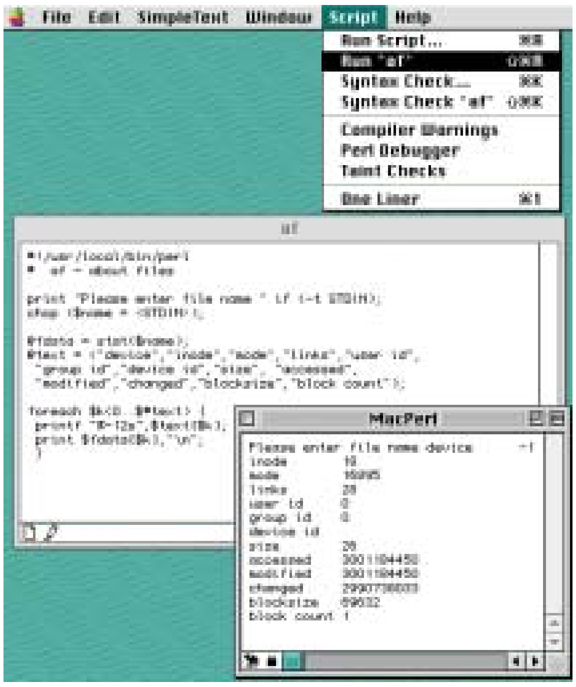
```
perl hello.pl
```

but it is not possible under MS-DOS to have the operating system itself work out that it needs to use Perl to translate and run the file.

Your program will run under MS-DOS using the current window as `STDIN` and `STDOUT`, and upon completion the DOS prompt will re-appear. There is no need for the extra

```
<STDIN>;
```

Note that the MS-DOS prompt starts with a new line, so a Perl program that runs correctly on a Unix system will leave a blank line prior to the prompt on MS-DOS.



```
#!/usr/local/bin/perl
#  af - about files

print "Please enter file name " if (-t STDIN);
chop ($name = <STDIN>);

@fdata = stat($name);
@text = ("device","inode","mode","links","user id",
 "group id","device id","size", "accessed",
 "modified","changed","blocksize","block count");

foreach $k(0..$#text) {
 printf "%-12s",$text[$k];
 print $fdata[$k],"\n";
 }
```

```
Please enter file name device      -1
inode        19
mode         16895
links        28
user id      0
group id     0
device id
size         28
accessed     3001104450
modified     3001104450
changed      2990738033
blocksize    59632
block count  1
```

Figure 22

Running MacPerl on the Macintosh.

5.5 Macintosh

As you might expect, running MacPerl is made easy for the user. Not much more than just knowing Perl will have you using this application "straight out of the box". SimpleText is used in harmony with MacPerl; therefore, all the plusses that you’ve grown to count on are all still there ... the ability to change font and style for ease and readability (without it affecting the final result, keeping it just as cross-platform). Try running a Perl program in speech mode for a laugh (we used "Victoria").

After you’ve double-clicked on the application icon, open up your program (or write a new one and save it) and then run it. With available options, you are able to debug it, ask for compiler warnings and perform other checks.

5.6 The compiler and the interpreter

Historically, there have been two ways that computers have interpreted and run programs.

Compiled languages

In languages such as C and Fortran, the source code is interpreted by a program called a "compiler" which reduces the English-like language written by the programmer to machine low-level instructions. Such low-level instructions are conventionally saved in files called "object files".

Programs in compiled languages are often long and call for many standard facilities shared between different applications. Therefore, after a whole lot of object files have been gathered, they're linked together using another program called a "loader" to form a single executable program.

Such compiled programs should be very fast to run, but they're expensive and slow to write. The executable file is dependent on the type of machine it is on, and there's quite a long procedure to follow if you want to make a change.

Interpreted languages

Basic, batch files, shell scripts and the like are often interpreted as they are run.

The concept is much easier and making alterations is much quicker (since all you do is change the source code and run again), but execution can be s-l-o-w. You now have a program that has to interpret a statement 100 times if it's in a loop that's run 100 times.

The middle way -- Perl

The best of both worlds -- used by Perl (Java, UCSD's Pascal and a number of other languages use similar basic technology).

The Perl code is written into a source file and a COMPILER is run to interpret each statement just once. But the output is not an object file as in a compiled language; instead, it's what's called "byte code" which can be efficiently run by ...

A souped-up INTERPRETER. The Perl interpreter takes the byte code output by the compiler and runs it.

With Perl, both compiler and interpreter phases are run every time your user runs the program. Up to and including Perl 5.004, there was no practical way for the ordinary program to use one independently of the other.

Whilst this system isn't as fast as a truly compiled language, it is much faster to run than an interpreted language. Furthermore, it's portable and simple to alter the programs.

5.7 Some questions on compilers and interpreters

Can I run just the compiler to check if I've got the syntax (language) of my program correct?

Yes, you can run your program using the `-c` option to Perl:

```
seal% perl -c rw
rw syntax OK
seal%
```

What if I make a mistake in my program?

Very often, you'll get an error from the compiler. It will print out an error message and won't let you go on to the interpreted stage at all.

Look carefully at the error message and try to work out the problem. During this course, call the tutor if you need help. Here are some things to look for:

- Have you omitted a `;` or a comma or a `"` character?
- Have you got a comma or a `"` in the wrong order?
- Have you opened but not closed a bracket?
- Have you used a capital letter by mistake? Print in not print!

Still none the wiser? Or perhaps your program runs, but gives totally unexpected results?

Run your program using the `-w` (warnings) option. If Perl feels that something, whilst it's valid, is a bit silly, it will print out a warning message. You can get warning messages from both the compiler and from the interpreter. You might not yet understand every word they display but you'll be prodded in the right direction!

Let's put an error into our "rw" program:

```
# rw2 - read, calculate, print results (2)

print ("Please enter a temperature (deg F): "
       );
chop ($temperature = <STDIN>);
$factor = 5.0 / 9.0;
$becomes = ($temperature - 32) * $factor;

$units_1 = "Fahrenheit";
$units_2 = "Centigrade";

print $temperature " degrees ",$units_1,  # from
      " becomes ",
      $becomes," degrees ",$units_2,       # to
      "\n";
```

```
seal% perl rw2
Please enter a temperature (deg F): 212
seal%
```

Figure 23

Perl program "rw2" **above** *doesn't run properly. Can you find the error? One way to get a "tip" on what's wrong is to run the program with the -w option, as we've done below.*

The error is tiny, but when we run the program it's clearly not good!

```
seal% perl -w rw2
Please enter a temperature (deg F): 212
Filehandle main::212 never opened at
  rw2 line 16, <STDIN> chunk 1.
seal%
```

Perhaps `-w` will help ...

Ah! line 16 ... `$temperature` has the value "212" ... can you spot the error now? [We haven't made it easy]

5.8 Debugging tools

If you're still trying to track down your error -- you have a program that compiles correctly, but doesn't work -- Perl is supplied with a "debugger" that lets you step through the code, examine individual variables, and the rest.

Sometimes, the debugger is overkill and a few extra `print` statements placed in your code will perform miracles!

And -- a halfway house -- the structure of Perl allows you yourself to see what variable names you have defined at any point, and what values they contain. Is this starting to sound a bit complicated? It needn't, for one of the other great aspects of Perl is that there are lots of others out there who are more knowledgable than you or I, and they've probably written the complicated bits and made them publicly available.

To conclude this section, we've modified our "rw" program (now called "rw3") to include a reference to a piece of code obtained from elsewhere:

```
# rw3 - read, calculate, print results
  (2)

use symbols;

print
("Please enter a temperature (deg F): ");

chop ($temperature = <STDIN>);
symbols("after read");
$factor = 5.0 / 9.0;
$becomes = ($temperature - 32) * $factor;

$units_1 = "Fahrenheit";
$units_2 = "Centigrade";

print $temperature," degrees ",$units_1,  # from
      " becomes ",
      $becomes," degrees ",$units_2,       # to
      "\n";

symbols("at end");
```

```
seal% perl rw3
Please enter a temperature (deg F): 212
Symbol list after read:
         temperature 212
212 degrees Fahrenheit becomes 100 degrees
  Centigrade
Symbol list at end:
             becomes 100
              factor 0.555555555555556
         temperature 212
             units_1 Fahrenheit
             units_2 Centigrade
```

Figure 24
Running Perl program "rw3". We're not getting the error messages now.

All we've done is to add that `require` statement to pull in another file of Perl, and these two statements:

```
          symbols("after read");
          symbols("at end");
```

which call code in that file to analyse and list out your variables.

Although examples up to this point (and on the earlier Perl Basics course) were run by typing `perl` followed by the program name, all examples for the rest of this course will have the "magic" first line set and will be marked as executable.

5.9 Summary

On Unix and Linux systems, to make your program executable from the command line, you must:

- include the directory it's in on your path
  (`set path` or similar; make permanent in dot file)
- set the file to be executable
  (`chmod +x` or similar)
- tell the calling shell that this is a Perl file
  (first line `#!/usr/local/bin/perl` or similar)

On Windows systems, the Perl program should be automatically associated with files ending in `.pl;` simply name your Perl programs using that convention and they'll click and run. You may need to extend your program to give the user the opportunity to read the output, though!

If you make an error in your program, Perl may give an error message; if it does, correct the error and try again.

More insidious errors can also occur. Programs can execute but do the wrong thing. You must check carefully. If you can't spot what's wrong with a program, using `-w` on the command line, or adding in extra print statements, may help.

Exercise

Take your exercises from the previous chapter and set them up so that they can execute directly and well. Check they still work directly in a new window, and when you log out and back in again.

Our example answer is to_euro3

Sample

```
graham@otter:~/profile/answers_pp>
  to_euro3
Please enter first price = 19.50
and second price = 9.20
and exchange rate = 0.7
converting to Euros
Total is 41 euros
At a rate of 0.7 euros to 1 pound
graham@otter:~/profile/answers_pp>
```

We provide a sample program called "two" which doesn't work correctly.
Can you make it function as shown in the samples?

Our example answer is two

Sample

```
graham@otter:~/profile/answers_pp> two
please enter your name: Graham
please enter your date of birth: 16/07/54
your name is Graham
16/07/54 is your date of birth
graham@otter:~/profile/answers_pp>
```

6 Conditionals and Loops

Like any other language, Perl has conditionals and loops.

6.1 The "if" statement

The first conditional we'll look at is the if statement. There are two in the following example:

two if statements within one program

```
#!/usr/local/bin/perl
# condition - if statements

print "Enter a number = ";
$number = <STDIN>;

if ($number > 10) {
        print "That number is over 10\n";
        print "Big!\n";
        }

if ($number%5 == 0) {
        print "it's a multiple of 5\n";
    } else {
        print "it's not a multiple of 5\n";
        }

print "program completed\n";
```

```
seal% condition
Enter a number = 4
it's not a multiple of 5
program completed
seal% condition
Enter a number = 15
That number is over 10
Big!
it's a multiple of 5
program completed
seal%
```

Figure 25
Running Perl program "condition".

Structure

An `if` statement comprises:

- The word `if`
- a condition (in brackets)
- a block of code (in braces)

In operation, the statement performs the instructions in the block of code if (and only if) the condition is true.

Whether or not the block of code is performed, Perl continues executing the program immediately beyond the block once the actions of the `if` statement have been completed.

Optionally, the `if` statement may be followed further by:

- the word `else`
- another block of code (also in braces).

Where this extra code is present, the block after the `else` will be executed if (and only if) the condition is false.

The `if` statement now runs either the first block or the second block before continuing on to the following Perl code. It never runs both, and it never runs neither!

Conditions

Perl has a "do I / don't I?" decision to make at the `if` statement based on whether the condition in the brackets gives a result which is true or false.

A number of operators are provided to help you; the ones I'll introduce first return:

- null (""), for false
  (the value 0 is also considered to be false)
- 1 for true
  (any value which is not zero or "" is considered to be true)

and the operators are:

>	greater than	(e.g. `$g > $h`)
==	numerically equal	(e.g. `$g == $h`)
>=	greater than or equal	
!=	not equal	
<	less than	
<=	less than or equal	

Blocks

How does Perl know how many statements are conditional upon the `if` statement? Perhaps you want just one statement to be conditional, or perhaps the whole of the rest of your program.

You tell Perl how many statements to make conditional by indicating the "scope" of the condition using curly braces, rather like wrapping in a sheet of paper the statements that are conditional.

There are many other times in Perl that you'll want to hold a whole group of statements together in this way; it's referred to as a "block" and you'll always use `{` and `}` to surround it.

The blocks in our first example are quite simple, but in more complex use, they can contain almost any code you wish, including another block. This is known as "nesting blocks". When you nest blocks of code, you must ensure that each inner block is contained entirely within its outer block, and that you have the same number of beginning and end curly braces.[1]

There are many other ways of writing conditional statements in Perl, but with this particular use of the `if` statement, you must always put your conditional code into a block. Unlike other languages, you may not leave out the `{` and `}` characters if the block is to consist of just one statement.

6.2 The "while" statement

The `if` statement allowed us for the first time to select code to execute conditionally, but not to keep repeating the code.

The `while` statement allows us to define a block which is going to be executed a number of times (or perhaps not executed at all).

The syntax is similar to the `if` statement:

1 The vi editor has a convenient feature for double-checking your program. The `%` command jumps your cursor from one bracket or brace to its pair -- useful if you're trying to sort out your nesting.

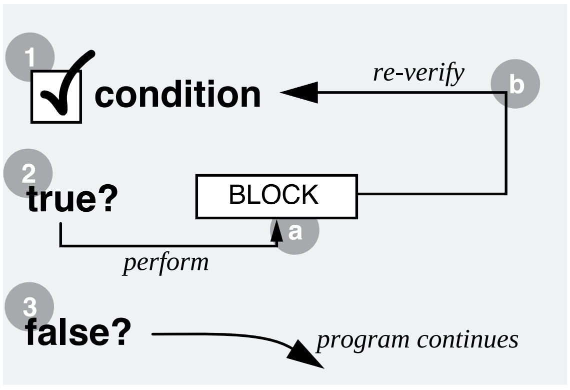


Figure 26

Illustrating how a `while` *loop works.*

```
seal% looper
Enter a number = 5
1 times 5 is 5
2 times 5 is 10
3 times 5 is 15
4 times 5 is 20
5 times 5 is 25
6 times 5 is 30
7 times 5 is 35
8 times 5 is 40
9 times 5 is 45
10 times 5 is 50
11 times 5 is 55
12 times 5 is 60
program completed
seal%
```

Figure 27

Running Perl program "looper".

- The word `while`
- a condition (in brackets)[1]
- a block of code (in braces)[2]

In operation, the statement performs the instructions in the block of code while the condition remains true.

1. The condition is checked
2. If the condition was true
   a) the block is performed
   b) then Perl jumps back to step 1
3. If the condition is false, the loop is completed

In order to prevent the loop running on forever, one of the actions within the block (2a in the above list) should be crafted to change the condition to false when you no longer wish to cycle.

Whether or not the block of code is performed, Perl continues executing the program immediately beyond the block once the actions of the `while` loop have been completed.

```
#!/usr/local/bin/perl
# looper - while loop

print "Enter a number = ";
chop ($number = <STDIN>);
$now = 1;

while ($now <= 12) {
        print $now," times ",$number,
                " is ",$now*$number,"\n";
        $now = $now + 1;
        }

print "program completed\n";
```

[1] brackets ( )
[2] braces { }

6.3 Shorthand operators

Here's a program that calculates course dinner requirements:

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

#  dinners - count course meals!

print "Course name: ";
chop ($course = <STDIN>) ;

while ($course) {
   $error = 0;
   $dinners = 0;
   print "Days= ";
   chop ($daze = <STDIN>);

   if ($daze < 1) {
      print "Too short\n";
      $error = 1;
      }

   if ($daze > 5) {
      print "Too long\n";
      $error = 1;
      }

   if ($error) {
      print
       "Must specify between 1 and 5 days\n";
   } else {
      $day = 1;
      while ($day <= $daze) {
       print "Students for day ",$day,"= ";
       chop ($dc = <STDIN>) ;
       $dinners = $dinners + $dc + 1;
       # adds tutor
       $day = $day + 1;
       }
      print "dinners in ",$day-1,
       " days was ",$dinners,"\n";
      print "That's an average of ",
       $dinners/$daze," per day\n";
      $cc = $cc + 1;
      }
   $alldinners = $alldinners + $dinners;
   print "name of next course: ";
   chop ($course = <STDIN>);
   }
print "Grand total of ",$alldinners," dinners\n";
print "Spread over ",$cc," courses\n";
```

```
seal% dinners
Course name: Perl Basics
Days= 1
Students for day 1= 4
dinners in 1 days was 5
That's an average of 5 per day
name of next course: Perl Programming
Days= 4
Students for day 1= 6
Students for day 2= 6
Students for day 3= 6
Students for day 4= 7
dinners in 4 days was 29
That's an average of 7.25 per day
name of next course:
Grand total of 34 dinners
Spread over 2 courses
seal%
```

Figure 28
Running Perl program "dinners".

It illustrates the `if` and `while` that we just studied, and it works well. But some parts of the code can be improved.

Add and assign

It's very common to write statements such as

```
$alldinners = $alldinners + $d...
```

In Perl we can shorten it to

```
    $alldinners += $d...
```

(other operators are available: `-= *= /= %= **=` and so on)

Increment

It`s also very common to write

```
    $cc = $cc + 1;
```

but that can become as short as

```
    $cc++;
```

or

```
    ++$cc;
```

(we'll see the difference in just a minute)

Multiple assign

There might be times we wish to save the same value or expression in several variables. Instead of

```
    $error = 0;
    $dinners = 0;
```

we can write

```
    $error = $dinners = 0;
```

String expansion

We can start using some clever features of strings, too. Instead of:

```
    print "Students for day ",$day,"= ";
```

we can write simply

```
    print "Students for day $day= ";
```

Assignments within other statements

We can even embed some assignments within others

```
    while ($day <= $daze) {
```

(and later)

```
    $day = $day + 1;
```

becomes

```
    while (++$day <= $daze) {
```

Let's use of all those in our "dinners" program:

```
#!/usr/local/bin/perl
#  din - count course meals!

print "Course name: ";
chop ($course = <STDIN>) ;

while ($course) {
 $error = $dinners = 0;
 print "Days= ";
 chop ($daze = <STDIN>);

 if ($daze < 1) {
 print "Too short\n";
 $error++;
 }
 if ($daze > 5) {
 print "Too long\n";
 $error++;
 }
 if ($error) {
 print "Must specify between 1 and 5 days\n";
 } else {
 $day = 0;
 while (++$day <= $daze) {
   print "Students for day $day= ";
   $dinners += <STDIN> + 1;
   }
 print "dinners in ",$day-1,
   " days was $dinners\n";
 print "That's an average of ",$dinners/
   $daze," per day\n";
 $cc++;
 }
 $alldinners += $dinners;
 print "name of next course: ";
 chop ($course = <STDIN>);
 }
print "Grand total of ", $alldinners," dinners\n";
print "Spread over ",$cc," courses\n";
```

```
seal% din
Course name: Perl Basics
Days= 1
Students for day 1= 4
dinners in 1 days was 5
That's an average of 5 per day
name of next course: Perl Programming
Days= 4
Students for day 1= 6
Students for day 2= 6
Students for day 3= 6
Students for day 4= 7
dinners in 4 days was 29
That's an average of 7.25 per day
name of next course:
Grand total of 34 dinners
Spread over 2 courses
seal%
```

Figure 29

Running Perl program "din", exactly the same results as "dinner".

We've made eight code-reducing changes (can you spot them?), but operationally, it's identical.

More on the increment operator

What's the difference between `$c++` and `++$c` ?
Nothing ... if that's the whole statement.
But look at these two:

```
        $g = $h++;
```

and

```
        $g = ++$h;
```

That roughly translates to:

"If the `++` appears after a variable name, then the variable is incremented after it's been used for any other purpose in the context in which it has appeared".

So: `$g = $h++;`
- take `$h`
- save [the original value] into `$g`
- THEN increment

`$g = ++$h;`
- take `$h`
- increment it
- THEN save [the new value] into `$g`

There is also a `--` operator that can also be specified before or after the variable to which applies.

6.4 Ways of writing numbers

You've seen constant numbers written as 32 and 32.0 ... how else can I write them?

```
#!/usr/local/bin/perl
# numbers - how to write constants
$a = 32;    $b = 0x32;
$c = 032;   $d = 3.2e1;
$e = 32.0;  $f = "32";
$g = "32.0";
print 2*$a,"\n";    print 2*$b,"\n";
print 2*$c,"\n";    print 2*$d,"\n";
print 2*$e,"\n";    print 2*$f,"\n";
print 2*$g,"\n";
```

```
seal% numbers
64
100
52
64
64
64
64
seal%
```

Figure 30
Running Perl program "numbers".

Some surprises ...

`032` means 32 octal (base 8) = 3 x 8 + 2 = 26
`0x32` means 32 hexadecimal (base 16) = 3 x 16 + 2 = 50

6.5 Summary

An `if` statement can be used to check a condition, and perform a block of statements[1] if the condition is true.

Optionally, an `else` block can be supplied as an alternative to be performed if the condition is false.

A `while` statement can be used to keep repeating a block of code while a condition remains true.

In Perl, 0 and "" are false and all other values are true. Operators such as `==` and `<` allow you to compare numbers and return you true or false answers.

Perl provides many short forms; you can write:

`$a += 4`	instead of	`$a = $a + 4`
`$a++`	instead of	`$a = $a + 1`
`$a = $b = 0`	instead of	`$a=0; $b=0;`
and `print "$a\n"`	instead of	`print $a,"\n"`

Be careful with `++`; you can write it before or after the variable name and that tells Perl whether to increment before or after it's used for any other operation in the same context.

[1] surrounded by `{ }`

Exercise

Write a program to read in and sum a series of prices. When the user enters a zero, print out the total and number of items, and leave the program.

Our example answer is summer

Sample

```
graham@otter:~/profile/answers_pp> summer
Enter first value 12.50
next value: 3.50
next value: 4.89
next value: 0
3 items, totalling 20.89
graham@otter:~/profile/answers_pp>
```

7 Analysing a Programming Task

7.1 A small job

How do we convert a small job that needs to be done into a Perl program?

We need to:

- learn about the task to be done is some detail
- work out how we're going to achieve the task
- write and test the program

Learning about the job

Although we're talking about "jobs" and "tasks" we're really writing something to take incoming data, handle and manipulate that incoming data and generate results – more data. So the first thing to learn about is the data, input and output.

Let's take an example: We want to write a program to help people understand sound levels in decibels. If they enter a number of decibels, to tell them what it compares to.

This is a typical woolly specification that you might be given, and you need to work it out much more precisely, probably by questioning the originator of the specification.

We'll come up with:

- User Input
- A single number, in decibels
- Output to user
- A line of information that says: "*xxxx* decibels is a bit noisier than *yyyyyyyyyyyyyyyyyyyyyyyyy*"

But there's also other data involved here. What else would you need to know if you were doing this task by hand? You would need a whole list of decibel values and what they equate to. You'll need to question the specification author as to where this data is coming from, and then make a decision as to whether it should form another input (from a file, or even from a web service).

There are other things you need to learn about the job you're doing. In particular just how the inputs get converted into outputs.

Working it out

Once you feel that you understand the job you'll be doing – inputs, outputs and how outputs are worked out from inputs – you may be well advised to draw a flowchart. Although there are a number of schemes for formally flowcharting tasks, you may prefer just to sketch it out on a sheet of paper.

A first flowchart might be as simple as seen in Figure 31:

If that doesn't show you how to convert the job into a program, then you may take one or more of the boxes on your flowchart and expand that box into another flowchart. (see Figure 32)

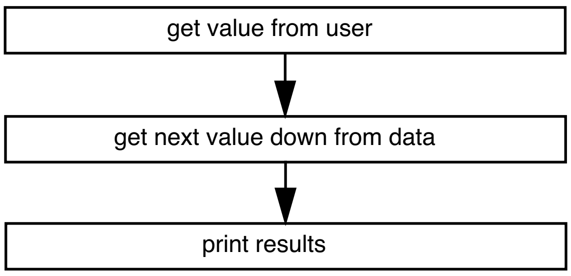


Figure 31

A simple flowchart

Writing

You can go on splitting up flowchart boxes into smaller boxes as you wish, but eventually you'll get to the stage that you can see how each box or group of boxes can be translated into Perl. Some practice is required at this; you'll probably want to bear in mind:

a) That an arrow going back up is probably going to be represented in the code by a loop, and other arrows going back up within it will be `redo` or `next` statements.

b) Decision boxes (shown as diamonds on traditional flowcharts) will often be `if` statements or similar, though they may also translate as the loop condition.

c) Instructional boxes such as "get value from data" will involve using some variables to generate other variables – conditionals and loops will not be involved if you've broken down your flowchart far enough.

Here's an example application, taking our flowcharts and converting them into Perl:

```
#!/usr/bin/perl

# Program to report on decibels - noise_1

#   Ask user to enter a value in decibels
#=========================================

print "Please enter a value in decibels: ";
chop ($yousaid = <STDIN>);

# get next value down from the data
#==================================

# while data is available ... read it
while ($preset = <DATA>) {

# extract value and text
($val,$text) = split(/\s+/,$preset,2);

# is it larger than the user's value?
next if ($val > $yousaid);

# have we already got a higher value?
next if ($val < $best_so_far);

# save as best!
$best_so_far = $val;
$best_text = $text;
```

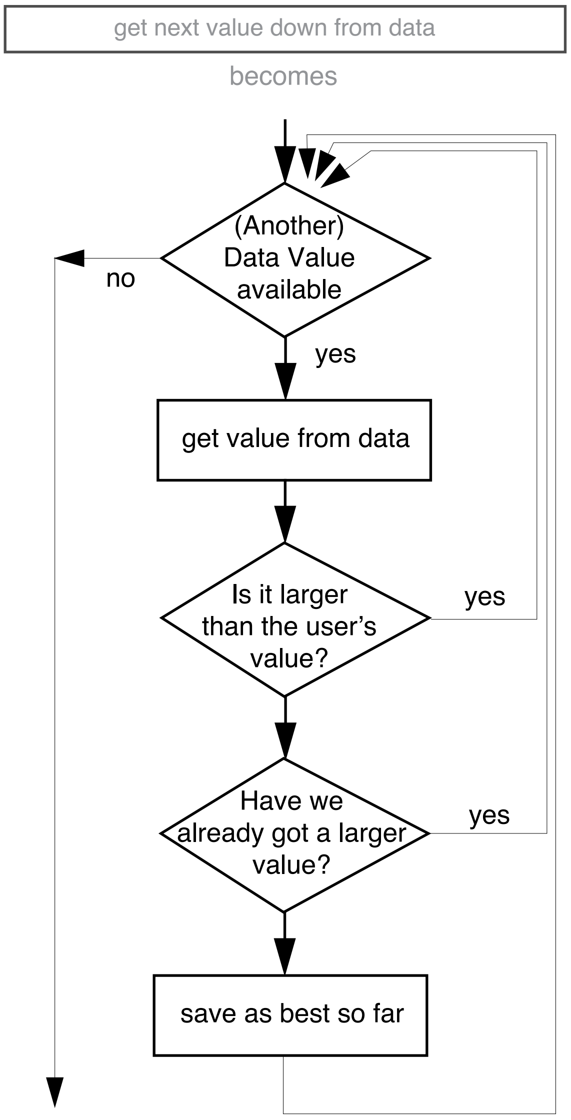

get next value down from data
becomes
(Another) Data Value available
no
yes
get value from data
Is it larger than the user's value?
yes
Have we already got a larger value?
yes
save as best so far

Figure 32

Expanding the simple flowchart

```
# end of loop
}

# print out results
#==================

print "$yousaid decibels is a bit noisier than $best_text";
__END__

5 the threshold of hearing
15 a Broadcasting Studio
25 a Bedroom at Night
35 a Library
45 a Family Living Room
55 a Typical Office
65 Conversational Speech
75 Average Roadside Traffic
85 Inside a Bus
95 Inside a Tube Train
105 a Pop Group at 20 metres
115 a Loud Car Horn at 1 metre
125 a Pneumatic Drill
135 the threshold of Pain
```

We've commented (and underlined the comment) each of the boxes on our original flow chart (Figure 31). We've also commented each of the lines in the middle section (the more detailed flow chart of Figure 32) to show you our translation process.

For a real-life application, it's highly unlikely that we would add so many comments, as the skilled programmer is able to read the detail of well written code line-by-line. Comments would remain for each of the major blocks, however, as would comments that describe any specially "clever" (i.e. obscure) bits of code.

The program runs as follows:

```
[graham@otter pex]$ ./noise_1
Please enter a value in decibels: 20
20 decibels is a bit noisier than a Broadcasting
  Studio

[graham@otter pex]$ ./noise_1
Please enter a value in decibels: 77
77 decibels is a bit noisier than Average Roadside
  Traffic

[graham@otter pex]$
```

Testing

Try to write your program section-by-section, and test each of the sections as you do so to see how you're getting on. Perl does have a debugger that lets you step through the code, but you'll often find that a `print` statement is just as fast and useful.

It's vital to test thoroughly. Put in some data from which the results will be known, and check that you get the right results. If your code includes loops and conditionals, you'll want to run a number of tests in such a way that each loop and each particular condition is tried out in all its various configurations.

Error Handling

In practice, our testing above was not thorough; we made the poor assumption that the user knew enough about decibels to put in a sensible answer. But look at this:

```
[graham@otter pex]$ ./noise_1
Please enter a value in decibels: 2
2 decibels is a bit noisier than [graham@otter pex]$

[graham@otter pex]$ ./noise_1
Please enter a value in decibels: Forty
Forty decibels is a bit noisier than [graham@otter pex]$

[graham@otter pex]$ ./noise_1
Please enter a value in decibels: 23456
23456 decibels is a bit noisier than the threshold of Pain

[graham@otter pex]$
```

None of these is a correct answer. Let's look at each:

```
Please enter a value in decibels: 2
2 decibels is a bit noisier than [graham@otter pex]$
```

In this case, the number entered was less than the quietest (smallest) value in our preset examples. No match was found, and an unsuitable output was printed.

```
Please enter a value in decibels: Forty
Forty decibels is a bit noisier than [graham@otter pex]$
```

A number wasn't entered at all – just text. The Perl program assumed that the letters (as they didn't contain digits) meant zero, and we got the same type of error as the first example.

```
Please enter a value in decibels: 23456
23456 decibels is a bit noisier than the threshold of Pain
[graham@otter pex]$
```

Perhaps this doesn't look like an error, but it is. The threshold of pain is around 135 decibels, so to say that 23456 decibels is "a bit nosier" is crazy.

You'll need to add error handling to your code. Sometimes a lot of error handling. You can tackle error handling on the basis of looking for all possible errors, or taking the opposite approach and checking that the inputs were valid. The latter is perhaps the best and most secure approach in Perl, although it's the less common approach in many more traditional programming languages.

Let's check our input value, with examples using both schemes:

```
#!/usr/bin/perl

# Program to report on decibels - noise_2
# Ask user to enter a value in decibels

#========================================
print "Please enter a value in decibels: ";
chop ($yousaid = <STDIN>);
if ($yousaid =~ /\D/)
{
die ("Only digits allowed\n");
}
if ($yousaid < 5) {
die ("Must specify a number 5 or greater\n");
}
if ($yousaid > 150) {
die ("Must specify a number 150 or less\n");
}

# get next value down from the data
#==================================
```

(Rest of application unchanged)

Here's the outputs from testing. Note how we started off by testing that the program still works for valid inputs, and then we tried out each of the error conditions in turn.

```
[graham@otter pex]$ ./noise_2
Please enter a value in decibels: 73
73 decibels is a bit noisier than Conversational
  Speech

[graham@otter pex]$ ./noise_2
Please enter a value in decibels: forty
Only digits allowed

[graham@otter pex]$ ./noise_2
Please enter a value in decibels: 2
Must specify a number 5 or greater

[graham@otter pex]$ ./noise_2
Please enter a value in decibels: 200
Must specify a number 150 or less

[graham@otter pex]$
```

Using regular expressions to accept only valid format input, the could could have read:

```
#!/usr/bin/perl

# Program to report on decibels - noise_3
#   Ask user to enter a value in decibels
#=========================================
print "Please enter a value in decibels: ";
chop ($yousaid = <STDIN>);
if ($yousaid !~ /^([01]?\d{2}|[5-9])$/) {
die ("Must specify number in range 5 to 199\n");
}
# get next value down from the data
#===================================
```

and when tested:

```
[graham@otter pex]$ ./noise_3
Please enter a value in decibels: 4
Must specify number in range 5 to 199

[graham@otter pex]$ ./noise_3
Please enter a value in decibels: 8
8 decibels is a bit noisier than the threshold of hearing

[graham@otter pex]$ ./noise_3
Please enter a value in decibels: 27
27 decibels is a bit noisier than a Bedroom at Night

[graham@otter pex]$ ./noise_3
Please enter a value in decibels: 101
101 decibels is a bit noisier than Inside a Tube Train

[graham@otter pex]$ ./noise_3
Please enter a value in decibels: 161
161 decibels is a bit noisier than the threshold of Pain

[graham@otter pex]$ ./noise_3
Please enter a value in decibels: 202
Must specify number in range 5 to 199

[graham@otter pex]$ ./noise_3
Please enter a value in decibels: fifty
Must specify number in range 5 to 199

[graham@otter pex]$
```

You'll notice that we've extended our testing here since the regular expression match itself has many possibilities, conditions and loops, and we need to ensure that it will do what we require in all circumstances.

Exercise

7.2 As a job gets larger

Our decibel example was a small application, but as applications grow there can be many levels of flowcharting involved, and the overall results would be a huge block of code, with the various sections separated by nothing more substantial than a comment.

A better approach is often to mimic the technique we used in the flowcharting using Perl subroutines,[1] a subroutine call and, within that subroutine, we define the detail.

Thus our initial flowchart can be coded as:

```
#!/usr/bin/perl

# Program to report on decibels - noise_4

get_user_value();
get_near_match();
print_results();

###############################################
sub get_user_value {
#   Ask user to enter a value in decibels
print "Please enter a value in decibels: ";
chop ($yousaid = <STDIN>);
if ($yousaid !~ /^([01]?\d{2}|[5-9])$/) {
die ("Must specify number in range 5 to 199\n");
 }
}

###############################################
sub get_near_match {

# get next value down from the data

# while data is available ... read it
while ($preset = <DATA>) {

# extract value and text
($val,$text) = split(/\s+/,$preset,2);

# is it larger than the user's value?
next if ($val > $yousaid);

# have we already got a higher value?
next if ($val < $best_so_far);
# save as best!
```

1 a box that performs a quite complex task is reduced to a single Perl statement

```
$best_so_far = $val;
$best_text = $text;

# end of loop
}
}

###############################################
sub print_results {
# print out results
print "$yousaid decibels is a bit noisier than $best_text";
}

__END__
5 the threshold of hearing
15 a Broadcasting Studio
25 a Bedroom at Night
35 a Library
45 a Family Living Room
55 a Typical Office
65 Conversational Speech
75 Average Roadside Traffic
85 Inside a Bus
95 Inside a Tube Train
105 a Pop Group at 20 metres
115 a Loud Car Horn at 1 metre
125 a Pneumatic Drill
135 the threshold of Pain
```

Good, but we can do better. We've split our code into a series of subroutines, but all the data held in variables is strung across the code. Things will be much clearer in our main code if we pass variables in and out of subroutines. Let's rework our main program with this in mind:

```
#!/usr/bin/perl

use strict;
# Program to report on decibels - noise_5
my ($userval,$text);
$userval = get_user_value();
$text = get_near_match($userval);
print_results($userval,$text);

###############################################
sub get_user_value {

#   Ask user to enter a value in decibels
print "Please enter a value in decibels: ";
chop (my $yousaid = <STDIN>);
if ($yousaid !~ /^([01]?\d{2}|[5-9])$/) {
die ("Must specify number in range 5 to 199\n");
}
$yousaid;

# returns the value type in if it's valid
}

##########################################

sub get_near_match {

my $yousaid = $_[0]; # collect incoming parameter
my ($best_so_far, $best_text);

# get next value down from the data

# while data is available ... read it
while (my $preset = <DATA>) {

# extract value and text
my ($val,$text) = split(/\s+/,$preset,2);

# is it larger than the user's value?
next if ($val > $yousaid);

# have we already got a higher value?
next if ($val < $best_so_far);

# save as best!
$best_so_far = $val;
$best_text = $text;

# end of loop
}

$best_text; # return value
}

###########################################
sub print_results {
my ($yousaid,$best_text) = @_; # print out results
print "$yousaid decibels is a bit noisier than $best_text";
}

__END__
5 the threshold of hearing
15 a Broadcasting Studio
25 a Bedroom at Night
35 a Library
45 a Family Living Room
55 a Typical Office
65 Conversational Speech
75 Average Roadside Traffic
```

```
85 Inside a Bus
95 Inside a Tube Train
105 a Pop Group at 20 metres
115 a Loud Car Horn at 1 metre
125 a Pneumatic Drill
135 the threshold of Pain
```

Let's look at some of the detail there:

```
use strict;
```

We've told Perl that we want all variables to be declared so that they're only shared between subroutines if we take explicit actions to share them. Remember that by default, Perl variables are global and you must declare them as `my` variables to limit their scope.

`strict` doesn't actually add any extra facilities; in fact all it does is gives you compile-time errors if any of your variables aren't declared. It is good to get into the habit of using `strict` since it picks up subtle, difficult-to-locate programming errors.

Within our main code, we then wrote:

```
my ($userval,$text);
```

to declare the variables `$userval` and `$text` as local (`my`) variables.

Our subroutine calls became:

```
$userval = get_user_value();
```

`get_user_value` requires no input parameters, but it returns a parameter which we have chosen to store in `$userval`. This is the decibel level that we're interested in comparing to.

```
$text = get_near_match($userval);
```

We're passing `$userval` in to `get_near_match`, which uses it to look up the data. It passes back a text string in `$text`.

```
print_results($userval,$text);
```

We're passing in the original value entered by the user, and the response text, for formatting and printing. There's no value returned to us by `print_results`.

Within the subroutines, variables must be `my`ed, and parameters picked up and returned.

Here's the code of `get_user_value`:

```
sub get_user_value {
print "Please enter a value in decibels: ";
chop (my $yousaid = <STDIN>);
if ($yousaid !~ /^([01]?\d{2}|[5-9])$/) {
die ("Must specify number in range 5 to 199\n");
}
$yousaid;
}
```

Remember that a subroutine returns the last result of the last statement it contains; in this case the contents of the variable `$yousaid`, and no parameters are passed into the subroutine.

On the other hand, subroutine `print_results` is called with two parameters; they'll be passed in through the special `list @_`, and we'll usually copy such parameters into local variables.

```
sub print_results {
my ($yousaid,$best_text) = @_;
```

`get_near_match` is just called with a single parameter:

```
sub get_near_match { my $yousaid = $_[0];
```

7.3 Summary

In this module, you've learnt how you must analyse and understand the task you'll be undertaking. Take great care to understand what your inputs are, and how you generate your outputs from them.

Flowcharting can be of great assistance in breaking down and understanding a task, and you can break a task down into a series of more detailed boxes if you wish. This can be translated into subroutines in Perl – a good idea as it helps you develop your code section-by-section, and later will let you re-use common code in other applications.

Exercise

8 Initial String Handling

You've read character strings and written them out, but your handling of text has so far been very limited.

You can read a character string into a variable, and use the `chop` (or the `chomp`) function to remove the new-line character on the end.

But that's just the start. Perl has so many facilities for handling strings that you would feel overwhelmed if you tackled them all at once. So here's the first section.

8.1 String handling functions

Perl has facilities for looking for individual characters or groups of characters within strings, for finding out the length of strings, and for extracting part of a string.

```
#!/usr/local/bin/perl
# string - some string functions

print "Enter your name ";
chop ($name = <STDIN>);

$nchars = length($name);
$nc = "Name is % characters long\n";

$pspace = index($name," ");

if ($pspace >= 0) {
   if (index($name," ",$pspace+1) > 0) {
       die "Can't handle more than 2 names\n";
       }
   $first = substr($name,0,$pspace);
   $rest = substr($name,$pspace+1);
   print "that's $rest, $first for indexing\n";
} else {
   $first = $name;
   $rest = "not stated";
}

$banner = "Hello ".$first." and welcome\n";
$say = "Your surname is $rest\n";
substr($nc,8,1) = $nchars;

print $banner,$say,$nc;
```

```
seal% string
Enter your name Graham Ellis
that's Ellis, Graham for indexing
Hello Graham and welcome
Your surname is Ellis
Name is 12 characters long
seal% string
Enter your name Graham
Hello Graham and welcome
Your surname is not stated
Name is 6 characters long
seal% string
Enter your name Graham J Ellis
Can't handle more than 2 names
seal%
```

Figure 33

Running Perl program "string".

We've used the following functions:

•`length`	to find out the length of a string
•`chop`	to remove the last character of a string
•`index`	to find the first occurrence of one string in another
•`index`	(different call) to find the next occurrence
•`substr`	to extract part of a string
•`substr`	(different call) to edit a string

and there are a number of other functions you might wish to use as well:

•`rindex`	find the last occurrence of one string in another
•`lc`	convert string to lower case
•`lcfirst`	convert 1st character of string to lower case
•`uc`	convert string to upper case
•`ucfirst`	convert 1st character of string to upper case
•`sprintf`	format a string (we come back to this later)

Thus:

```
#!/usr/local/bin/perl
# proper

print "name: "; $name = <STDIN>;
print ("hello ", ucfirst lc $name);
```

```
seal% proper
name: graham
hello Graham
seal% proper
name: GRAHam
hello Graham
seal%
```

Figure 34
Running Perl program "proper".

8.2 String handling operators

As well as the functions listed above, the following operators are designed for use on strings:

`.`	join strings together
`x`	duplicate string on left by number of times on right (e.g. `"allo" x 3` is "alloalloallo")
`".."`	convert string in quotation marks into a text string

Double-quoted strings

The double-quoted string interpretation is very clever. We've already used it to interpret variables, but it can do much much more! The following special codes are recognised by the " operator:

`\n`	new-line character
`\t`	tab character
`\a`	alert character
`\r`	carriage return
`\f`	form feed
`\b`	back space
`\e`	escape character

Octal, hexadecimal and control codes may also be specified:

`\243`	pound sign[1]
`\xa3`	pound sign
`\cJ`	line feed

[1] On Microsoft Windows fonts, the pound sign is \234 or \1x8a

Variables are also interpreted in line.

`$...`	scalar variable
`@...`	list (later!)

If you actually want one of these special characters to appear in a string, you should precede it with a `\`, thus:

`\$`	dollar character
`\@`	"at" character
`\"`	double-quote character
`\\`	backslash character

And finally, you can force upper case and lower case within the string:

`\l`	next character lower case
`\L`	subsequent characters lower case
`\u`	next character upper case
`\U`	subsequent characters upper case
`\E`	indicates the end of the \U or \L

Here are some in use:

```
#!/usr/local/bin/perl
# prop2

print "name: "; chop($name = <STDIN>);
print "amount: "; chop($amount = <STDIN>);
print
"hi \L\u$name\E.\nThe amount is \xa3$amount\n";
```

```
seal% prop2
name: grAham
amount: 2.46
hi Graham.
The amount is £2.46
```

Figure 35

Running Perl program "prop2".

Single-quoted strings

There will be occasions when you want to print out a character string exactly, and the double quotes would cause unwanted interpretation.

This can be achieved by using single quotes rather than double quotes.

qq and q strings

There may also be occasions that you want to use a different delimiter for your string. For example, you might have a sentence in the variable `$saying` that you want to print out.

```
    print "\"$saying\" he said\n";
```

It works, but you can select an alternative delimiter using a `qq` notation:

```
    print qq-"$saying" he said\n-;
```

You can go further and replace the alternative delimiter with brackets:

```
    print qq("$saying" he said\n);
```

A similar scheme works for single-quoted text strings, but using a single letter `q`, thus:

```
    print q("$saying" he said\n);
```

```
#!/usr/local/bin/perl
# prop3

print "name: "; chop($name = <STDIN>);
print "amount: "; chop($amount = <STDIN>);
print
"hi \L\u$name\E.\nThe amount is \xa3$amount\n";
print
'hi \L\u$name\E.\nThe amount is \xa3$amount\n';

$saying = "It's raining";
print "\n";
print "\"$saying\" he said\n";
print qq-"$saying" he said\n-;
print qq("$saying" he said\n);
print q("$saying" he said\n);
```

```
seal% prop3
name: grAHAM
amount: 12.24
hi Graham.
The amount is £12.24
hi \L\u$name\E.\nThe amount is
  £$amount\n
"It's raining" he said
"It's raining" he said
"It's raining" he said
"$saying" he said\nseal%
```

Figure 36
Running Perl program "prop3".

Here documents

Imagine that you want to print a lot of lines. Would you use a lot of print statements?

There's an alternative: `print <<"FRED";`

```
line to print
next line, addressed to $name
and another line
FRED
```

It's called a "here" document and it's very useful for blocks such as copyright notices, the headings for web pages, addresses and the like.

The double quotes on the first line are optional[1] and they define a string that must appear on a line on its own to terminate the string. The `;` to end the statement may appear on the same line as the print statement, or after the block, but not on the line with the terminating string.

Do not leave a space between the `<<` and the following character or you'll find your here document being terminated at the next blank line!

8.3 Comparing strings exactly

We've read, manipulated and printed out strings. But we haven't yet asked the question "Does string `$g` contain the same text as string `$h`?"

You might think I could write

```
if ($g == $h) ...
```

but that will not work.

Why? Because `==` is a numeric operator and it treats the two things it's comparing as numbers.

By way of example, let's say that `$g` contained the word "tomato" and `$h` contained the word "bacon". What would the numeric value of `$g` be? Zero. What's the numeric value of `$h`? Also zero. The two values are the same, so `$g == $h` is true.

1 But it is recommended that you use the double quotes.

Err ... ?

If I want to compare two strings, I write

```
    if ($g eq $h) ...
```

and that will perform a string equality test. Exactly what I want to use to compare "tomato" and "bacon".

String comparison operators:

`eq`	returns `true` if the strings are identical
`ne`	returns `true` if the strings are not identical
`lt`	returns `true` if the first string is lexically less than the second
`le`	returns `true` if the first string is lexically less than the second, or identical
`gt`	returns `true` if the first string is lexically greater than the second
`ge`	returns `true` if the first string is lexically greater than the second, or identical

Common pitfalls and things you should be aware of all hinge around the word "identical". These tests are case significant -- "cat" is NOT "Cat". If one string has a new line on the end and the other does not, the lines do NOT match. If one string has a space but the other has a tab, they do NOT match

Of course, I could force a match ignoring case:

```
     if (lc ($g) eq lc($h)) ...
```

but there is a better way.

8.4 Comparing strings to regular expressions

Let's say that I want to see if `$g` contains "bacon". Well, not exactly "`b a c o n `".

"bacon" is acceptable, or "Bacon" or either of those preceded or followed by any white space characters. That's easy when you know how:

```
    if ($g =~ /^\s*[Bb]acon\s*$/) ...
```

We read that as:

`$g`	contents of `$g`
`=~`	is matched to
`/.../`	a regular expression.

and the regular expression reads:

`^`	starting with
`\s`	white space
`*`	zero or more of previous item (white space)
`[Bb]`	Upper or lower case B
`acon`	the letters a-c-o-n, all lower case
`\s`	white space
`*`	zero or more of previous item
`$`	and that's the end of the string

There's also another pattern match operator, so the complete set is

=~	returns true if the string matches the regular expression
!~	returns true if the string does NOT match the regular expression

Let's try out that example in a program:

```
#!/usr/local/bin/perl
# pig - looks to match "Bacon"


print "Breakfast: ";
chop ($g = <STDIN>);

while ($g) {
        if ($g =~ /^\.*[Bb]acon\.*$/) {
                print "Yes\n";
        } else {
                print "No\n";
        }
        print "Next meal: ";
        chop ($g = <STDIN>);
}
```

```
seal% pig
Breakfast: bacon...
Yes
Next meal: ...bacon
Yes
Next meal: sausage.and.bacon
No
Next meal: .Bacon
Yes
Next meal: Bacon!
No
Next meal:
seal%
```

Figure 37

Running Perl program "pig". We have replaced the back-slashes with `\.` *in our example so you could see where we placed the white spaces.*

We're not going to have you write complex regular expressions yet, but we will introduce you to a few more features of them and have you try them out on a testbed program called "pattern". The program searches a database for towns and other places. For your reference, here's the code; the only new thing apart from regular expressions is the opening / reading a file.

We'll come back to it in a few minutes when we simplify some of the code!

```
#!/usr/local/bin/perl
# pattern - dialling code match testbed!

print "enter a pattern ";
chop ($plook = <STDIN>);

while ($plook) {
   open (FH,"towns");       # Open a file
   while ($town=<FH>){      # read from that file
        chop $town;
        if ($town =~ /$plook/) {
            print "$town\n";
        }
   }
   print "enter a pattern ";
   chop ($plook = <STDIN>);
}
```

Let's try some regular expression matching.

```
seal% pattern
enter a pattern Rock
Rockcliffe(Cumbria)
Castle_Rock
Rock
Rockingham
Rockcliffe(Kirkcudb.)
Rockbourne
Wetley_Rocks
Standon_Rock
enter a pattern
seal%
```

Figure 38

Running Perl program "pattern" looking for letters and digits that match exactly.

First, letters and digits are matched exactly. But by default the matching string can appear anywhere in the string being tested.

```
seal% pattern
enter a pattern ^Rock
Rockcliffe(Cumbria)
Rock
Rockingham
Rockcliffe(Kirkcudb.)
Rockbourne
enter a pattern Rock$
Castle_Rock
Rock
Standon_Rock
enter a pattern ^Rock$
Rock
enter a pattern
seal%
```

Figure 39

Running Perl program "pattern" looking for an explicit match using an anchor.

We can use anchors -- a `^` on the start or a `$` on the end -- to request an explicit match to the start or the end of the string. If we use both, we're looking to match the whole string.

As well as matching explicit characters, we can specify groups of characters in square brackets. These groups can also use a special **-** character to indicate a range.

```
seal% pattern
enter a pattern [Rr]ock$
Castle_Rock
Rock
Grays_Thurrock
Gourock
Standon_Rock
Barrock
Brock
enter a pattern [s-z]ock$
Dervock
Burton_Bradstock
Powerstock
Kessock
Ibstock
Brigstock
Crantock
Corsock
Beattock
Radstock
Tavistock
Hemyock
Cotterstock
Martock
Woodstock
enter a pattern
seal%
```

Figure 40

Running Perl program "pattern" looking for explicit characters.

If a group of this sort starts with a `^` it means "not" Thus ...

```
seal% pattern
enter a pattern [^A-Za-kq-z]ock$
Challock
Cumnock
New_Cumnock
Banknock
Greenock
Dymock
Cannock
Kilmarnock
Matlock
Tredunnock
Porlock
Gargunnock
Much_Wenlock
Gillock
Speaking Clock
enter a pattern
seal%
```

Figure 41

Running Perl program "pattern" looking for words that do not contain certain letter combinations.

There's a number of pre-defined groups available:

`\s`	any white space character
`\d`	any digit
`\w`	any word character (letter, digit, underscore)

and the opposite of those:

`\S`	and character that is NOT white space
`\D`	and character that is NOT a digit
`\W`	and character that is NOT a word character

Finally, a **.** means ANY one character.

```
seal% pattern
enter a pattern \wcall
Brinscall
Scalloway
Riccall
High_Ercall
Childs_Ercall
No_charge_to_caller
enter a pattern \Wcall
Do NOT display my number on "call
  return" (prefix)
BT Credit card call (prefix)
Who called last message
Local area call
Local area call
Local area call
Local area call
Local area call
Local area call
Local area call
Local area call
enter a pattern
seal%
```

Figure 42

Running Perl program "pattern" looking for any pre-defined groups.

And finally for this first taste of regular expressions, we'll introduce you to some counts.

?	0 or 1 of the previous character
*****	0 or more of the previous character
+	1 or more of the previous character

```
seal% pattern
enter a pattern Up*i
Uig
Uppingham-(Oakham)
Uppington
enter a pattern Up?i
Uig
enter a pattern Up+i
Uppingham-(Oakham)
Uppington
enter a patternseal%
seal
```

Figure 43
Running Perl program "pattern" using counts.

8.5 Summary

Perl variables can hold character strings and many functions are provided to manipulate those strings. There are also a number of string operators such as `.` and `x`.

Within double-quoted strings, you can use special sequences such as `\n` for a new line and `$var` for a variable. If you want to prevent this, use a single quote instead to have all characters taken literally.

Alternative notations:
`"hello"` or `qq-hello-` or `qq(hello)` or here documents

To compare strings exactly, use operators such as `eq`. To see if a string matches a pattern, use `=~` instead.

Patterns are written between `/` characters and may include

anchors

`^`	to ask for a match at the start
`$`	to ask for a match at the end

character groups

`[Bb]`	to match any character from a list
`\s`	to match any white space character
`\d`	to match any digit
`\w`	to match any alphanumeric
`\S`	to match any nonwhitespace
`\D`	to match any nondigit
`\W`	to match any nonalphanumeric
`.`	to match any 1 character

counts

`?`	to match 0 or 1 of previous character
`*`	to match 0 or more of previous character
`+`	to match 1 or more of previous character

and all upper case, lower case letters and digits otherwise match exactly.

These patterns are also known as regular expressions and we'll visit them again later.

Exercise

Run our pattern program to find all towns ...

a) All towns starting "Mat"
b) All towns including "Mat"
c) All towns including "Br" and later "oll"

Our example answer hasn't got a name

Sample

```
graham@otter:~/profile/book> pattern
enter a pattern [work out what you must type]
Matlaske
Matlock
enter a pattern [work out what you must type]
Matlaske
Matlock
Worth_Matravers
enter a pattern [Work out what you must type]
Brent_Knoll
Bridge_Sollars
enter a pattern
```

Modify the program of the previous section to exit when the user enters the word `end` (upper case or lower case). Modify it to print out a pound sign before the total amount.

Our example answer is autumn

Sample

```
graham@otter:~/profile/answers_pp> autumn
Enter first value 12.50
next value: 3.22
next value: End
2 items, totalling 15.72
graham@otter:~/profile/answers_pp>
```

Write a new program to read in a sentence, and print it out with the first character forced to a capital letter and with an exclamation mark on the end if the sentence wasn't complete.

Our example answer is winter

Sample

```
graham@otter:~/profile/answers_pp> winter
enter text: Is this line punctuated?
Is this line punctuated?
graham@otter:~/profile/answers_pp> winter
enter text: graham wrote this
Graham wrote this!
graham@otter:~/profile/answers_pp>
```

9 More Loops and Conditionals

9.1 The variety that is Perl

Up to this point in the course, you've been using `if` statements for conditional text and while loops to repeat blocks of code. And you'll continue to use these; they're fundamental structures. But sometimes they're a bit wordy and there can be a better way of doing things.

In fact, there are so many ways to do these things in Perl that we'll show you examples of the most common ones and summarise some of the "well, if you must" type stuff.

Although we're talking about writing programs on this course, you may well find that a lot of your time is spent modifying, extending and repairing code that you've written earlier, or has been written by others. You should consider this maintenance aspect as you write your programs:

- Comment the programs well
- Design them in a modular way (structure or object oriented, a topic for later!)
- Have a version numbering system
- Write your Perl in such a way that it's easy for others to follow
- Set and use programming standards within your team or organisation.

These last two points mean that you should not try to use every different construct of the language!

Because others will not have stuck to guidelines in the past, anyone who's responsible for maintaining code probably does need to have a good overview of all the facilities of the language, though.

9.2 More conditional statements

Here's a short program ... some of you may remember something similar from the Perl Basics course!

```
#!/usr/local/bin/perl
# telegram - an if statement
print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;
if ( $age > 100 )
           {
           print "well done\n";
   } else  {
         print "$togo years to your telegram\n";
           }
```

```
seal% telegram
please enter your age: 45
55 years to your telegram
seal% telegram
please enter your age: 102
well done
seal%
```

Figure 44
Running Perl program "telegram".

If -- single statement rather than a block

You've been writing

```
if ( $age > 100 )
   {
   print "well done\n";
   }
```

and if you've written C or Java in the past, you've been wishing you could leave out those braces. In Perl you can't, but you can write the `if` statement the other way around:

```
print "well done\n" if ($age > 100);
```

Good idea?

<u>For</u>: Saves coding. Makes it obvious it's a single statement block.

<u>Against</u>: Not intuitive. Conditional tends to get lost in body of code.

Unless -- an inverted if statement

You could replace

```
if ( $age > 100 )
   {
   print "well done\n";
   }
```

by

```
unless ($age <= 100)
   {
   print "well done\n";
   }
```

An `unless` statement is the same as an `if` statement, except that the block will be performed unless the condition is true. It's rather like an "if not". Yes, you could have also written

```
if (! ($age <= 100) )
   {
   print "well done\n";
   }
```

Just like `if`, you can add an `else` block after an `unless` statement. Just like `if`, you can write a single statement conditional the other way around:

```
print "well done\n" unless ($age <= 100);
```

Good idea?

<u>For</u>: Sometimes saves writing awkward negative conditions.

<u>Against</u>: Perhaps obscure for future maintainers

```
seal% tel2
please enter your age: 102
Well done ... WELL done
seal% tel2
please enter your age: 53
47 years to your telegram
seal%
```

Figure 45
Running Perl program "tel2".

```
#!/usr/local/bin/perl
# tel2 - some if alternatives

print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;

print "Well done ... " if ( $age > 100 );
print "WELL done\n" unless ($age <= 100);
unless ($age > 100) {
          print "$togo years to your telegram\n";
            }
```

Conditional operators

Consider the following statement:

```
    if ($age < 18 && $alcohol > 0) { print "no ...
```

You're checking to see if you can make a legal sale.

"If the age is less than 18, and the person has more than zero alcohol in his cart, reject the sale."

What if the person was aged 18 or over? Do you need to even look whether he has alcohol? No, it's legal anyway.

Perl knows this. It will only make the second check if necessary, in this case only if the purchaser is less than 18.

You can made use of this facility. Let's look at:

```
if ($havebasket > 0 && ($rdbasket = <STDIN>) )
   { print "sale";}
```

What does this do?

- Check and see if our customer has a basket
- If so, read basket contents and save to `$rdbasket`
- And if there was something in the basket, print "sale"

So we have the basket being read only if the customer has a basket. If we didn't want to print the word "sale" we could just have written

```
($havebasket > 0 && ($rdbasket = <STDIN>) ) ;
```

or even

```
$havebasket > 0 && ($rdbasket = <STDIN>) ;
```

We now have conditional code -- just the facility the `if` statement itself provides -- without the need for the word `if`.

- You can read `&&` as "and if true"
- You can read `||` as "and if false" or as "or".

Let's use them on the telegram example:

```
seal% tel3
please enter your age: 94
6 years to your telegram
seal%
```

Figure 46
Running Perl program "tel3".

```
#!/usr/local/bin/perl
# tel3 - avoiding an if statement

print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;

$age > 100 && print "well done\n";
$age > 100 ||
print "$togo years to your telegram\n";
```

You can use multiple `&&` and `||` operators in a single statement; usual rules for brackets apply. Thus:

```
$havebasket > 0 && ($rdbasket = <STDIN>) &&
print "sale";
```

And there are some very common uses:

```
$age >= 18 || die "No sale. Customer too young\n";
```

Good idea?

For: Short, quick and clean

Against: Confusing to newbies. Limited to single statement.

You can also use the words `and` and `or` to replace `&&` and `||`, but at a different precedence.

The ? : operator

`&&` and `||` gave you shorthands for simple `if` and unless statements, but not a quick and easy way of writing `if else`. It's very common to write code such as

```
if ( $age > 100 )
    {
    print "well done\n";
    } else  {
         print "$togo years to your telegram\n";
    }
```

Perl provides an operator in three parts to help you with this.

(condition) `?` (true action) `:` (false action)

so that you could have written

```
print ($age > 100) ? "well done\n" :
    "$togo years to your telegram\n" ;
```

or even

```
($age > 100) ?  print "well done\n":
    print "$togo years to your telegram\n";
```

Another very common use:

```
        $largest = ($g>$h)?$g:$h;
```

and yet another:

```
print "You asked for ",$n,($n!=1)
    ?" places\n":" place\n";
```

In use:

```
#!/usr/local/bin/perl
# tel4 - an if shorthand

print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;

( $age > 100 ) ?
            print "well done\n":
           print "$togo years to your telegram\n";

print "please enter number of places: ";
chop ($n = <STDIN>);
print "You asked for ",$n,($n!=1)?" places\n":
    " place\n";
```

```
seal% tel4
please enter your age: 34
66 years to your telegram
please enter number of places: 2
You asked for 2 places
seal% tel4
please enter your age: 101
well done
please enter number of places: 1
You asked for 1 place
seal%
```

Figure 47

Running Perl program "tel4".

Good idea?
For: Short, quick and clean
Against: Confusing to Perl newbies. Limited to single statement. ":" can get mistaken for ";" by the human reader

9.3 More loop statements

The while loop has, and will continue, to serve you well, but there are other loops too.

The until loop

Just as you could invert `if` by using `unless`, you can invert `while` using `until`.

Single statement while and until loops

Just as you can remove the { and } from a single statement `if` or `unless` (provided that you also write the condition on the end), so you can with `while` and `until`.

```
#!/usr/local/bin/perl
# power - next power of 2 above a number

print "Enter a number: ";
$num = <STDIN>;
$power = 1;
$power *=2 until ($power >= $num);
print "next power of 2 is $power\n";
```

```
seal% power
Enter a number: 4
next power of 2 is 4
seal% power
Enter a number: 100
next power of 2 is 128
seal% power
Enter a number: 178
next power of 2 is 256
seal%
```

Figure 48
Running Perl program "power".

The for loop

Very frequently, you'll want to run a loop a certain number of times. Five working days in the week, or `$ndays` days on a course:

```
    $now = 1;
    while ($now <= 12) {
         print $now," times ",$number,
         " is ",$now*$number,"\n";
         $now++;
         }
```

It works. But it's such a common requirement and the handling of the loop -- the variable `$day` -- is in three different statements. In this simple case they're all within five lines of code, but imagine a much longer program in which they are a l-o-n-g way apart. Easy to follow? No!

The `for` loop lets you take the three elements that control a loop such as this:

- Initial setting (s)
- Condition
- Action(s) before retesting the condition

and put all those elements into a single statement:

```
    for ($now = 1;$now <= 12;$now++) {
         print $now," times ",$number,
         " is ",$now*$number,"\n";
         }
```

The round brackets of a for loop must contain exactly two semi-colons. The conditional section (between them) must give a true or false result. The first and third sections may be empty, or may even contain a comma separated list. It's possible to write some very confusing for loops ... best to keep them simple rather than trying to be too clever. The first loop in this example is good; the second may give you a headache!

```
#!/usr/local/bin/perl
# forloop - for loop
#

print "Enter a number = ";
chop ($number = <STDIN>);

for ($now = 1;$now <= 12;$now++) {
        print $now," times ",$number,
                " is ",$now*$number,"\n";
        }

print "and also\n\n";

for (;$div < 12;
        print (++$div," divided by ",$number),
        print (" is ",$div/$number,"\n")){};
```

```
seal% forloop
Enter a number = 4
1 times 4 is 4
2 times 4 is 8
3 times 4 is 12
4 times 4 is 16
5 times 4 is 20
6 times 4 is 24
7 times 4 is 28
8 times 4 is 32
9 times 4 is 36
10 times 4 is 40
11 times 4 is 44
12 times 4 is 48
and also

1 divided by 4 is 0.25
2 divided by 4 is 0.5
3 divided by 4 is 0.75
4 divided by 4 is 1
5 divided by 4 is 1.25
6 divided by 4 is 1.5
7 divided by 4 is 1.75
8 divided by 4 is 2
9 divided by 4 is 2.25
10 divided by 4 is 2.5
11 divided by 4 is 2.75
12 divided by 4 is 3
seal%
```

Figure 49
Running Perl program "forloop".

9.4 Breaking a loop

There will be times that you're in a loop, but as a result of testing a condition, you'll want to do one of these:

- Get out of the loop and carry on beyond
- Go around the loop again, with the same initial condition
- Move on to the next iteration of the loop

You can do each of these, using

`last;`	to leave the loop
`redo;`	to rerun with the same initial condition
`next;`	to move on to the next iteration

`next` is an example using all three. This program asks the user to enter the number of course lunches required for each day of a five-day week. If the user enters a number over 12, an error is flagged and the user is prompted again -- `redo;`

If the user enters 0, the rest of that loop iteration is skipped as there's no need to run the code to place the order -- `next;`
If the user enters "end" then he's signalling that the course is finished and there are no more days to ask for -- this is the `last;` time through the loop.

```
seal% jumps
Number of lunches, day 1: 4
Ordering 4 dinners for day 1
Number of lunches, day 2: 20
Too many. Try again
Number of lunches, day 2: 2
Ordering 2 dinners for day 2
Number of lunches, day 3: 0
Number of lunches, day 4: end
Total dinners - 6
seal%
```

Figure 50
Running Perl program "jumps".

```
#!/usr/local/bin/perl
# jumps - loop

for ($day = 1; $day <=5; $day++) {
   print "Number of lunches, day $day: ";
   chop ($ndins = <STDIN>);

   $ndins =~ /^\s*end\s*$/ && last;
   next if ($ndins == 0);

   if ($ndins > 12) {
       print "Too many. Try again\n";
       redo;
       }

   print
   "Ordering $ndins dinners for day $day\n";
   $ndtot+=$ndins;
}
print "Total dinners - ",$ndtot+0,"\n";
```

Although illustrated with a `for` loop, these statements can also be used with other loops such as `while` and `until`, and with the `foreach` loop we'll meet later.

9.5 Labels

"`last` gets you out of a loop". Yes, but which loop?

The normal answer is "the loop that you're in"' but what if you're in a loop within a loop?

The exact rule is that "`last` gets you out of the innermost loop which you are in".

But what if that's not what you want? What if you want to get out of two nested loops?

- Label (name) the block you want to jump out of.
- Specify the label after the word `last`.

Labels must start at the beginning of a line and be the only thing on that line.

They do not follow the normal white space rules. The text of a label comprises a series of alphanumeric characters followed by a colon.

When you reference a label -- in this example in the `last` statement -- they follow the normal syntax rules. For example, they may appear anywhere on a line. The `:` should be omitted.

Next is an example (our course lunches again!).

If you enter `end` (lower case) it's the end of entry for this week. If you enter `END` (upper case) it's the end of the whole thing.

```
#!/usr/local/bin/perl
# jump2 - exit 2 loops

week:
for ($week =1; $week <=4; $week++) {

for ($day = 1; $day <=5; $day++) {
   print
   "Number of lunches, week $week, day $day: ";
       chop ($ndins = <STDIN>);

       $ndins =~ /^\s*END\s*$/ && last week;

       $ndins =~ /^\s*end\s*$/ && last;

       $ndtot+=$ndins;
}
print "Dinners to date - ",$ndtot+0,"\n";
}
print "Total dinners - ",$ndtot+0,"\n";
```

```
seal% jump2
Number of lunches, week 1, day 1: 3
Number of lunches, week 1, day 2: 4
Number of lunches, week 1, day 3: 2
Number of lunches, week 1, day 4: end
Dinners to date - 9
Number of lunches, week 2, day 1: 3
Number of lunches, week 2, day 2: 2
Number of lunches, week 2, day 3: 9
Number of lunches, week 2, day 4: END
Total dinners - 23
seal%
```

Figure 51
Running Perl program "jump2".

Perl has no switch or case statements, using a label and a block they're unnecessary:

```
#!/usr/bin/perl

while (1) {

print "please enter a word: ";
chop ($word = <STDIN>);
$co++;

choose:
{
($word =~ /quit/) and die "Leaving the program\n";
($word =~ /count/) and print ("count is $co\n") and last choose;
($word =~ /echo/) and print ("hello\n") and last choose;
print "eh?\n";
}

}
```

9.6 The goto statement

Get two programmers together, and suggest that "you don't need a goto statement" and you're likely to see a major argument break out. Some people find it very useful to be able to jump around their programs, but others say it makes the code very hard to test, hard to follow and hard to modify later.

But the philosophy of Perl is "if it might be useful, provide it".

Syntax:

```
goto label
```

You may jump out of a block, but never into a block

You'll learn later in this course that `goto` can do other things as well. You can get it to jump to a whole variety of different labels depending upon the value of some variable, and it does something which even the official texts call "highly magical" in certain other circumstances.

9.7 Summary

Should you only want to perform a single statement after an `if`, you can write it back-to-front and leave out the `{` and `}`. You can also replace `if` with `unless` to invert the condition.

An expression to the right of a `&&` (or the word `and`) is evaluated only if the condition to the left is true.

An expression to the right of a `||` (or the word `or`) is evaluated only if the condition to the left is false.

The `?...:` operator gives you a "shorthand" `if ... else` construct.

The `for` loop can be used as an elegant replacement for `while` loops.

```
for (initial; condition; subsequent) { ...
```

Within a loop, you can use

`last`	to exit the loop
`redo`	to rerun the current pass
`next`	to move on to the next pass

Normally, these affect your innermost loop but you can use labels so that they affect an outer block.

A label appears on a line on its own, followed by a colon.

The `goto` statement is available, but is poor practice and inefficient.

Exercise

Please try to do the following exercise WITHOUT using the words "while" or "if" in your program.

Write a program to ask the user to enter four numbers each between the value of 1 and 6. If the user enters a number below 1 or over 6, ask him to enter that number again. If the user enters the word `END`, stop reading numbers.

- Print out the count, average and total of all the numbers entered. Take care to print an appropriate error message for the average if no numbers were entered.

Our example answer is dice

Sample

```
graham@otter:~/profile/answers_pp> dice
Value of die 1: 5
Value of die 2: 7
Invalid
Value of die 2: 3
Value of die 3: 1
Value of die 4: 2
count: 4
total: 11
average: 2.75
graham@otter:~/profile/answers_pp> dice
Value of die 1: 1
Value of die 2: 2
Value of die 3: 4
Value of die 4: end
Invalid
Value of die 4: END
count: 3
total: 7
average: 2.33333333333333
graham@otter:~/profile/answers_pp> dice
Value of die 1: END
count:
total:
average: infinity
graham@otter:~/profile/answers_pp>
```

10 File Handling

You've read from the keyboard, written to the screen. `STDIN` and `STDOUT` have been explained to you and, if you're familiar with the operating system on your computer, you should be able to read or write files instead of using your keyboard or screen. But what if you want to do both? What if you want several files at the same time?

10.1 File input and output

You've only got one keyboard, or one form source, so when you start a Perl program the file handle `STDIN` is available to you without you having to ask for any special connection.

But on your disk, there will be thousands of files.

- No chance that you'll want more than a few for any application
- Impractical to have them all selected at once

Therefore, before you use a file you need to select it.

File Handles

Today you might want to read from a file called "france.txt" and tomorrow from a file called "sweden.txt" but you don't want to have to go all through your program making changes. So we won't keep referencing a file by its name. Rather, we'll create a special variable called a "file handle" and we'll use that as our route to the file.

`STDIN` is a file handle variable.

File handle variables start with a letter, followed by any number of letters, digits and underscores. Conventionally, they're written in all upper case. As with regular variables, you can use almost any name you want, but if you use one of the following, you'll conflict with Perl and remove access to a default facility:

```
DATA
STDIN
STDOUT
STDERR
```

The open function

Let's open the file named in the variable `$country` for read access:

```
   open (PORTS,$country);
or open (PORTS,"$country");
or open (PORTS,"<$country");
```

That will create a new file handle called "PORTS" and it will open the file through it. In this case, this means that the start of the file will be read into memory, not yet passed back to your Perl program, but there ready. It's more efficient to buffer data than to read in dribs and drabs.

If we had wanted to open the file for write, instead of read, we would have written:

```
open (PORTS,">$country");
```
to open for overwrite

```
open (PORTS,">>$country");
```
to open for append

That will create a new file handle called `PORTS` and open the file whose name is held through that variable. If the file didn't exist, it would be created. If it existed and you used `>`, the old contents would be deleted. If it existed and you used `>>`, the file would be set for you to add information at the end.

`open` is an operator and it returns a value. True for success, false for failure. Therefore

```
open (PORTS,"$country") || die
    "Can't open file\n";
```

is very common.

Reading from a file handle

You've already seen the "read from" operator `<...>` on `STDIN`. Simply use it on any other input file handle.

Each time you read from the `PORTS` file handle, a line of text (up to and including a new line character) is read and, if you give a variable name, saved into that variable.

Reading starts at the beginning of the file. You'll get the next line each time you call for another read.

If you get nothing back, not even a new line character, you've reached the end of the file.

Writing to a file handle

You've already used print to print to `STDOUT`.

Although you've not been explicit, when you've written

```
print  $var,.....;
```

you've really meant

```
print  STDOUT  $var,.....;
```

In other words, the print function takes an extra parameter which is the name of the file handle. So, to write to a file:

```
print PORTS  $var,.....;
```

Closing a file

If you're about to exit from your Perl program when you finish with a file, you don't need to worry about closing the file. It will be done automatically and any remaining data will be written.

Normally, though, you should close your files using the `close` function:

```
close PORTS;
```

Let's see a simple example of handling a number of files:

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

# fan - fan out towns
# based on vowels in name

open (IN,"towns") || die
("No town file to read\n");

open (A,">a.towns");
open (E,">e.towns");
open (I,">i.towns");
open (O,">o.towns");
open (U,">u.towns");


while ($line = <IN>) {
$m = 0;
    (++$m && print A $line) if ($line =~ /[aA]/);
    (++$m && print E $line) if ($line =~ /[eE]/);
    (++$m && print I $line) if ($line =~ /[iI]/);
    (++$m && print O $line) if ($line =~ /[oO]/);
    (++$m && print U $line) if ($line =~ /[uU]/);
    print "No vowels in $line" unless ($m);
}
```

```
seal% fan
No vowels in Crymych
No vowels in Dyffryn
No vowels in Ynysybwl
No vowels in Glyndwr
No vowels in Cwm
No vowels in Dryslwyn
No vowels in Lydd
No vowels in Rhyl
No vowels in Bwlch
No vowels in Lymm
No vowels in Tydd
No vowels in Lyth
seal%  ls -l *.towns
-rw-r--r--  1 graham  wellho  36605 Jan 20 20:36 a.towns
-rw-r--r--  1 graham  wellho  39065 Jan 20 20:36 e.towns
-rw-r--r--  1 graham  wellho  30473 Jan 20 20:36 i.towns
-rw-r--r--  1 graham  wellho  34257 Jan 20 20:36 o.towns
-rw-r--r--  1 graham  wellho  13804 Jan 20 20:36 u.towns
seal%
```

Figure 52
Running Perl program "fan".

Other things you can handle through the file interface

As well as files, most operating systems handle devices as if they were files. So if you're looking to read / write directly from a serial port or a tape drive, or to access other windows on your screen (in text mode), you'll follow the instructions as above. When we come on to file testing, you'll learn that you can even check a file name to see if it really is a file, or if it's a device of some sort.

With operating systems that support multiple processes at the same time (very few these days do not), you can start up another process from within your Perl program and read or write to or from that as if it was a file. This procedure is called "piping" and will probably make your code operating-system dependent.

Let's write a Perl process that starts two more processes -- one to give a disk utilisation report, and the other to email the report onwards.

You'll notice that we use a vertical bar character at the start (write to process) or end (read from process) of our file name parameter.

THIS IS THE TYPICAL GLUEWARE ROLE OF PERL!!

Let’s email Lisa a report:

```
#!/usr/local/bin/perl
# piper - email a df report!
#

open (GETSYS,"df -k |") || die
   "Can't read pipe\n";

open
   (MESSAGE,
   "|mail -s \"Disk Usage\" lisa\@wellho.net")
       || die "No email\n";

while ($line = <GETSYS>) {
        print MESSAGE $line;
        }
```

```
seal% piper
seal%
```

Figure 53

Running Perl program "piper". But there’s no screen output.

Of course, there's no screen output ... but in the end of Lisa's mailbox:

```
From graham Wed Jan 20 21:14:48 1999
Return-Path: <graham@wellho.net>
Received: by wellho.net (SMI-8.6/SMI-SVR4)
   id VAA09029; Wed, 20 Jan 1999 21:14:48 GMT
Date: Wed, 20 Jan 1999 21:14:48 GMT
From: graham@wellho.net (Graham Ellis)
Subject: Disk Usage
Message-Id: <199901202114.VAA09029@wellho.net>
Content-Type: text
Apparently-To: lisa@wellho.net
Content-Length: 811

Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t3d0s0      61615   56225    5329    92%    /
/dev/dsk/c0t3d0s6     432839  379122   53285    88%    /usr
/proc                      0       0       0     0%    /proc
fd                         0       0       0     0%    /dev/fd
/dev/dsk/c0t3d0s7     422223  408607   13194    97%    /export/home
swap                  130532      16  130516     1%    /tmp
/dev/dsk/c0t0d0s0     192783   19090  173501    10%    /extra/disc0.slice0
/dev/dsk/c0t0d0s4     481983   74851  406651    16%    /extra/disc0.slice4
/dev/dsk/c0t0d0s5     481983   11687  469815     3%    /extra/disc0.slice5
/dev/dsk/c0t0d0s6     963895  382751  579538    40%    /extra/disc0.slice6
/dev/dsk/c0t0d0s7    6404166 5555652  784473    88%    /extra/disc0.slice7
```

As well as handling other processes like this, file handles can be used if you want a Perl program to split in two and have both halves continuing to work on separate tasks. That is known as "forking" and we'll do this later in the course.

You can also use file handles to talk to other processes on other machines on your network ... or indeed on the internet as a whole. Once again, we do this later on this course and also on our Perl Advanced - Network Applications course.

In both these last two cases, please note that you use something else rather than `open` to create and connect the file handle in the first place. And you have other issues you need to consider as well.

10.2 File testing

Did we manage to `open` our file successfully earlier on in this section?

Well, yes, we did. Otherwise the `open` function would have returned a false value and the `die` function would have printed a message to `STDERR`.

But what if we had wanted to check if a file existed before we opened it for write, or whether it was a device, or ...

Let's take the question "Does the file "towns" exist?". It's clearly some sort of equality test.

Numeric equality?	No!
String comparison?	No.
Pattern match?	No!!

It a new type of operator.

Just like we can write `-` in front of a variable to negate it, so we can write `-e` in front of a file name to see it the named file exists. Let's see some further tests:

`-e`	does it exist?
`-d`	is it a directory?
`-z`	is it empty?
`-r`	can I read it?
`-w`	can I write it?
`-x`	can I execute it?
`-f`	is it a plain file?

There are also a number of other tests which return more than just true or false:

`-M`	when was it last modified (time ago, in days)
`-s`	how big is it (in bytes)

```
Can't execute file re-ask. Size is 536 bytes
Can't execute file read_write. Size is 410 bytes
Can execute file reporter. Size is 256 bytes
Can execute file rw. Size is 414 bytes
Can execute file rw2. Size is 414 bytes
Can execute file rw3. Size is 480 bytes
Can't execute file sharecode. Size is 474 bytes
```

Figure 54

Only just a part of the output from running Perl program "reporter".

```
#!/usr/local/bin/perl
# reporter - tell us about files ...

open (FILES,"ls|") || die "Can't read pipe\n";

while ($line = <FILES>) {
   chop $line;
   print ((-x $line) ? "Can":"Can't",
   " execute file $line. Size is ",
   -s $line," bytes\n");
   }
```

10.3 Formatted printing

Earlier in this course, you saw numbers been printed out with absurd accuracy.

Whilst 1/7 does work out to 0.142857142857143, you'll often want to print it out as just 0.143.

It's also been hard so far to print information in neat columns as the number of characters in a value varies, all the other information to the right gets messy. You only need look up to the last example to see the problem!

The `print` function was great for "quick and dirty" printing, but if we need more control we should use the `printf` function. Although the two functions have similar names, they do different things, take different parameters and work in different ways.

However, just as you can specify a file handle with `print`, so you can with `printf`.

Here's an example `printf`:

```
  printf "%.2f degrees f converts to %.2f degrees c\n",$far,$cent;
```

or

```
  printf  STDOUT "%.2f degrees f converts to %.2f degrees c\n",$far,$cent;
```

The first parameter is different to all the others -- it's a template to describe how the information should be printed. In this case:

`%.2f`	a floating point number, 2 digits after the decimal point
`degrees f converts to`	text - print as it is in the format
`%.2f`	another floating point number, same format
`degrees c`	more text, to print as in the format
`\n`	a new-line character

The format is what's printed out every time the `printf` statement is run; the rest of the parameters tell `printf` what to place into the variable fields that start with a `%` character. If too few parameters are specified, `printf` fills in with nulls. Too many and it ignores the excess!

Here are some examples from that `printf` statement:

```
28.00 degrees f converts to -2.22 degrees c
29.00 degrees f converts to -1.67 degrees c
30.00 degrees f converts to -1.11 degrees c
31.00 degrees f converts to -0.56 degrees c
32.00 degrees f converts to 0.00 degrees c
33.00 degrees f converts to 0.56 degrees c
```

Figure 55
Examples from the `printf` *statement.*

If you think this looks similar to the printf function of the C language, or the one that's in nawk, or the one that's built into some operating systems these days, you're correct!

There are many more formats available, for example ...

Floating point formats

`%f`	just printed out the number in a default way
`%.2f`	printed it out to 2 decimal places
`%5.2f`	printed it out a TOTAL of 5 columns wide, two decimal places
`%05.2f`	printed it the same way, except if 0 filled leading spaces
`%-5.2f`	Total of 5 columns, 2 after the "." now LEFT justified.
`%e`	prints the value in scientific (exponential) form
`%5.2e`	prints (or tried to) print it out in scientific notation in a total width of 5 columns and to 2 decimal places. The width isn't enough, so Perl will add more columns for you.
`%8.2e`	8 columns wide, scientific
`%15.2e`	15 columns wide, scientific

You can replace the "e" or "f" with a "g" if you would like to select which notation's best for any particular number

Formats for whole numbers (integers)

`%x`	print a number in base 16 (hexadecimal)
`%o`	print a number in base 8 (octal)
`%d`	print a number in base 10 (decimal)

Decimal is, of course, the most common, but we must stress that `%d` prints a whole number (an integer) to base 10, and not a number that includes a decimal point. Very confusing if you forget!!

As with floating point formats, you can specify:

numbers	to give a minimum column width
leading 0	to zero fill
leading -	to left justify

Formats for variable text strings

`%s`	print a string of characters

Column widths and left justification apply. You can also specify a `.` (full stop) to ask for a total field width and a length at which to truncate the incoming string.

`%c`	print out a single character (from a variable that contains its ASCII value)

Constant text

Finally, anything that does not start with `%` in a format string will be handled as literal text by `printf`. The usual double quote rules apply: `\n` for new line, `\\` for a backslash, etc.

It may surprise you that you should use `%%` if you want to print out an actual percent character.

sprintf

In our earlier section on string handling, we briefly mentioned `sprintf` and that it formats text using exactly the same rules as `printf`. However, rather than sending the resulting string to a file handle, it's returned to the programmer as a string. Therefore you could save it into a variable or perform further manipulation on it.

10.4 Summary

File handles are special variables that don't start with a `$` sign and are usually written in upper case. `STDIN`, `STDOUT` and `STDERR` are provided as standard, otherwise you must use an `open` statement to associate a handle with a file.

By default, Perl opens files for read but you can also specify

`">$abc"`	to open to overwrite
`">>$abc"`	to open to append
`"|$abc"`	to pipe to a command
`"$abc|"`	to pipe from a command

Reading from a file may be done using

```
<FH>
```

Writing may be done using

```
print FH $var
```

You can test whether a file exists by writing

```
if (-e $abc) { ...
```

and many other tests are available.

For more control over printing, use `printf` instead of `print`.

The first parameter to `printf` is a format

Subsequent parameters are values to be printed

Within the format, you can use

`%f`	a floating point number
`%s`	a string
`%d`	an integer (decimal)

and so on.

Modifiers can be placed between the `%` and the letter, thus

`%6.2f`	6 columns minimum width, 2 figs after decimal place
`%02d`	2 column minimum width integer, zero filled
`%-12s`	12 column minimum width string, left justified

Exercise

Write a program to read in the file "towns" file line by line, and print out each line ending with the word "Bridge", with a line number across to the right, like the sample below.

Our example answer is bridges

Sample

```
       Dunsop_Bridge # town no.   1
     Hubberts_Bridge # town no.   2
       Lowick_Bridge # town no.   3
       Appley_Bridge # town no.   4
        Spean_Bridge # town no.   5
       Hebden_Bridge # town no.   6
       Haydon_Bridge # town no.   7
   Bonchester_Bridge # town no.   8
     Skeabost_Bridge # town no.   9
         Carr_Bridge # town no.  10
      Dulnain_Bridge # town no.  11
         Gara_Bridge # town no.  12
       Steens_Bridge # town no.  13
    Johnstone_Bridge # town no.  14
      Newnham_Bridge # town no.  15
       Whaley_Bridge # town no.  16
         Mayo_Bridge # town no.  17
      Ettrick_Bridge # town no.  18
     Stamford_Bridge # town no.  19
       Blythe_Bridge # town no.  20
       Pooley_Bridge # town no.  21
       Tummel_Bridge # town no.  22
   Collingham_Bridge # town no.  23
      Whitley_Bridge # town no.  24
      Wootton_Bridge # town no.  25
```

The "Perl World" theme park opened at 11 in the morning yesterday, and admitted 17 visitors every minute for the first hour. Visitors each paid 4.75 to enter the park. Produce a file containing a table of takings at 5-minute intervals. Display what the grand total would be after an hour. The file should contain results like our sample below.

Our example answer is thepark

```
11:00 - takings so far     0.00        11:35 - takings so far  2826.25
11:05 - takings so far   403.75        11:40 - takings so far  3230.00
11:10 - takings so far   807.50        11:45 - takings so far  3633.75
11:15 - takings so far  1211.25        11:50 - takings so far  4037.50
11:20 - takings so far  1615.00        11:55 - takings so far  4441.25
11:25 - takings so far  2018.75        12:00 - takings so far  4845.00
11:30 - takings so far  2422.50
```

11 Lists

In almost every program, you'll require to store a large amount of information and then look through that data, perhaps manipulating it and processing it on the way. That could be as simple a task as sorting the lines of a file so that they appear in a different order, or as looking up a series of towns in our "town" file. But so far there has been no practical way for us to do this. Of course, Perl has the answer: The "list".

If you're familiar with other programming languages, you'll probably be familiar with arrays, which are closely akin to Perl's lists.

11.1 Basics

Creating a list

A list is just that. It's a list of items written between round brackets and separated by commas. We didn't tell you at the time, but actually the parameters to your `print` and `printf` statements were lists.

```
print
("There are",$persons," people in the room");
```

That list is created, used and released within the `print` statement.

Let's say, though, that you want to store the elements of the list under a single name. You could write:

```
@line =
("There are",$persons," people in the room");
```

That:

- creates a list (array) called "@line"
- creates three elements within that list
- assigns values to each of the elements

You'll notice:

- there's a single name for the list
- list names start with `@` rather than `$`
- you don't have to state how long the list will be
- you don't have to state how big each element of the list will be

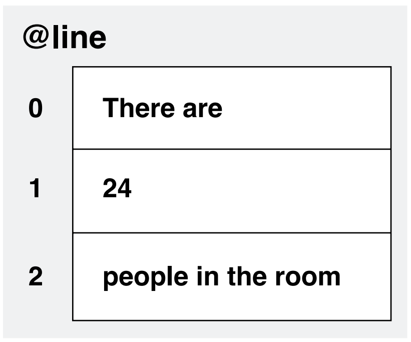


Figure 56

The @line in use

Referencing an element in a list

Although you refer to the list as a whole using `@`, you refer to each individual item using:

- `$` first character
- list name
- element number in square brackets

The first element is number zero, so our three-element list above has element numbers `0` `1` and `2`.

```
#!/usr/local/bin/perl
# ar1 - first list

$persons = 10 + 2 * 7;
@line = ("There are",$persons," people in the
  room");

print "Item 1: ",$line[1],"\n";
print "Item 2: ",$line[2],"\n";
print "Item 0: ",$line[0],"\n";
```

```
seal% ar1
Item 1: 24
Item 2:  people in the room
Item 0: There are
seal%
```

Figure 57
Running Perl program "ar1".

You can also refer to list elements using an expression to give the element number. Within double-quoted strings, you can reference list elements just in the same way that you can access scalars.

```
#!/usr/local/bin/perl
# ar2 - second list

$persons = 10 + 2 * 7;
@line = ("There are",$persons,
    " people in the room");

for ($k = 2; $k>=0; $k--) {
print "Item $k: $line[$k]\n";
}
```

```
seal% ar2
Item 2:  people in the room
Item 1: 24
Item 0: There are
seal%
```

Figure 58
Running Perl program "ar2".

Changing a list

To change the contents of a list element, just assign a new value to that element. That's exactly the same thing you would do with a regular scalar.

Conversion between strings and numbers happens in just the same way it does with a scalar. The list element grows longer to cope with longer strings, and gets shorter when you reduce a string length, just in the same way it does with a scalar.

The same type of flexibility also occurs with extending a list. If you assign a value to an element beyond the end of a list, the list will be extended automatically. Any intermediate values that you don't set will have an undefined value.

```
seal% linelen
File to analyse: france.txt
Line with   5 characters:   2 occurrences
Line with   8 characters:   2 occurrences
Line with  10 characters:   2 occurrences
Line with  11 characters:   1 occurrences
seal% cat france.txt
Plymouth
Poole
Southampton
Portsmouth
Newhaven
Folkestone
Dover
seal%
```

Figure 59

Running Perl program "linelen".

```
#!/usr/local/bin/perl
# linelen - line length counter

print "File to analyse: ";
chop ($fn = <STDIN>);

open (ANALYSE,$fn) ||
die "Can't read file\n";

while ($line = <ANALYSE>) {
 chop $line;
 $nc = length $line;
 $lltab[$nc]++;
 }

close ANALYSE;
for ($k=0;$k<120;$k++)
 {
next unless ($lltab[$k]);
printf
"Line with %3d characters: %3d occurrences\n",
$k, $lltab[$k];
}
```

When the program starts running, the list `@lltab` does not exist. It's created when the program reaches line 24 for the first time. The first of the data file is eight characters long, so the list is created with nine elements in it (numbered 0 through 8) and the last element is incremented from an undefined value to a value of 1. The other eight elements remain undefined.

The next time the program reaches line 24, the `$nc` variable contains the value 5. So element number 5 of the list is incremented from undefined to 1.

The third time through the loop, `$nc` contains the value 11 -- past the current end of the list -- so the list is extended.

Since the longest line in the incoming file is 11 characters long, our list ends up containing 12 elements (numbered 0 to 11) when the analysis is completed.

The loop to print out the results goes through each element of the list in turn, reporting the element number and contents where a value has been stored.

Since we did not keep track of how long our longest line was, we have put a value high enough for all reasonable circumstances in the printout loop, but our program will miss reporting of any lines over that 120 character length.

11.2 The length of a list

Rather setting an artificially high ceiling on an list as we did in the last section, we have two alternatives:

- we could keep a record of the list length ourselves (messy!)
- we can use a variable that Perl provides for each list

```
$#listname
```

The `$#listname` variable tells us the number (often referred to as the index number or ordinate) of the last element of the list. So the list in the previous example was 12 elements long which means

`$#lltab` ends with a value of11

`$#lltab` can be used just like any other variable. The printing loop in our last example may be rewritten from

```
 for ($k=0;$k<120;$k++)
```

to

```
 for ($k=0;$k<=$#lltab;$k++)
```

And our bug that "lines over 120 characters are not reported" has been fixed. Furthermore, if all the lines in our file are short, as in our test case, we're not left looping around many more times than necessary after all the list contents have been printed.

You'll have noticed the use of the `#` character, previously used to indicate the start of a comment, within this special variable. In detail, a `#` only indicates the start of a comment where it follows after a white space character.

11.3 Context

Some operators in Perl can only mean something sensible when used on a scalar. For example, you can add 5 to a scalar, but adding 5 to a list would be a meaningless request.

There are other operators we'll meet which would only be sensible when used on a list. For example, you couldn't sort a scalar into order but you can sort a list.

And there are some operators which mean different things when they're used on a scalar to when they're used on a list.

We're introducing the subject of context. Operations in Perl can be performed in a list or a scalar context. How does Perl know which to use? If something is valid in a list context, then the list context meaning takes precedence.

Let's see an example ... first the program which we'll then study line-by-line.

```
#!/usr/local/bin/perl
# context demonstration

@mylist = ("a","b","c");

@one = @mylist;
$two = @mylist;
(@three) = @mylist;
($four) = @mylist;

print "one ",@one,"\n";
print "two ",$two,"\n";
print "three ",@three,"\n";
print "four ",$four,"\n";
```

```
seal% context
one abc
two 3
three abc
four a
seal%
```

Figure 60
Running Perl program "context".

Let's look one-by-one:

```
@one = @mylist;
```

List context. The = sign assigns the list on the right to the list on the left. You are copying the whole list.

```
$two = @mylist;
```

Scalar context. We're assigning to a scalar on the left-hand side, after all. How does Perl handle `@mylist` in a scalar context? It uses the length of the list[1] as that's about the most sensible thing it can do. That's why 3 was printed out.

```
(@three) = @mylist;
```

List context. The = sign assigns the list on the right to the list on the left. You are copying the whole list. The brackets are redundant.

```
($four) = @mylist;
```

List context. Because you have brackets around the scalar on the left-hand side of the assignment, the left-hand side has become a list. So this is list context as well. As the list on the left can only take one value, the first value of `@mylist` is copied into `$four`. That's why `a` is printed out.

Getting the idea? Let’s try some more:

```
#!/usr/local/bin/perl
# cont2 - context demonstration

@mylist = ("a","b","c");
print "five ",@mylist,"\n";
print "six @mylist\n";
print "seven ",@mylist+0,"\n";
print "eight @mylist+0\n";
```

```
seal% cont2
five abc
six a b c
seven 3
eight a b c+0
seal%
```

Figure 61
Running Perl program "cont2".

Analysed as:

```
print "five ",@mylist,"\n";
```

List context. The entire contents of the list are printed out

```
print "six @mylist\n";
```

Double-quote context. The entire contents of the list are printed out, but with a space character between each element (you'll learn later how to change the separator character)

```
print "seven ",@mylist+0,"\n";
```

Scalar context . You can't add 0 to a list, but you can add it to a scalar. So `@mylist` was taken as the length of the list. Another very common use of `@mylist` in a scalar context is:

```
 if (@mylist > 20) ....
```

```
print "eight @mylist+0\n";
```

Double quote context. `@mylist` is expanded with a space between each element. In double-quote context, mathematical expressions aren't interpreted, they're just a part of the text, so the +0 is printed.

[1] one greater than the value of $#mylist

11.4 Summary

A list is a number of values, written separated by commas.

Lists may be saved into variables, which have names starting with `@` charcaters rather than `$` characters. Individual elements of the list can be accessed using `$list_name[$el_no]`

The first element in a list is numbered 0, the second is numbered 1, and so on. If you refer to a negative number, Perl will count backwards from the end.

It's important to remember that Perl can work in both a list and a scalar context, and some operators do different things depending in which context you call them.

`$#list_name` always gives you the number of the top element of `list_name`. `@list_name` in a scalar context gives you the number of elements in the list.

Exercise

Rewrite your dice program. This time, store all the values entered in a list, and print out just the average and then echo back all the numbers that were correctly entered.

Our example answer is dilist

Sample

```
graham@otter:~/profile/answers_pp> dilist
Value of die 1: 3
Value of die 2: 2
Value of die 3: 4
Value of die 4: 5
average: 3.5
values: 3 2 4 5
graham@otter:~/profile/answers_pp>
```

11.5 Functions that operate on lists

Functions that let you manipulate the elements within a list

`push`	add new element(s) to end of list
`pop`	return last element of list and remove it from list
`shift`	return first element from list, remove it, move rest of list up
`unshift`	add new elements(s) to start of list, move list up
`splice`	remove and return part of a list, replace with new list
`push` and `pop`	implement a stack on the end of a list
`shift` and `unshift`	implement a stack on the start of a list
`push` and `shift` (or `pop` and `unshift`)	can be used to implement a queue
`splice`	is a generic function than can be used for any of the above, and more!

Functions that let you re-order a list (returning a new list)

`reverse`	returns a list in the opposite order
`sort`	returns a list sorted lexically

Function to return the length of a list

`scalar` returns length of a list.

(an alternative to `$#`xxx and to `@`xxx in a scalar context)

Operators that relate to lists

<>	in a list context, read until end of file into a list
`..`	generate a list of scalars
`$#`xx=	truncate or extend a list
`foreach`	process each element of a list
`qw`	quote word

Operators than manipulate strings and lists

`grep`	filter a list for all elements matching a regular expression
`join`	join the elements of a list into a single string
`split`	split a string into a list of fields
`pack`	pack a list into a scalar
`unpack`	unpack a scalar into a list
`chomp`	remove last character in each element of list if it's end-of-line

That's a big list of operators. Let's see some of them in use in some example programs.

```
#!/usr/local/bin/perl
# fanlist - fan out towns based on vowels in name

open (IN,"towns") ||
die ("No town file to read\n");
chomp(@tlist = <IN>);

@vowels = qw(a e i o u);

foreach $vow(@vowels) {
 foreach $town(@tlist) {
 push @matched_town,$town if (lc($town) =~ /$vow/
  );
 }
 print "vowel $vow: matches "
 ,scalar(@matched_town),"\n";
 $#matched_town = -1;
}
```

```
seal% fanlist
vowel a: matches 2957
vowel e: matches 3123
vowel i: matches 2281
vowel o: matches 2712
vowel u: matches 1082
seal%
```

Figure 62
Running Perl program "fanlist".

- `chomp(@tlist = <IN>);`

  Read from the file, putting each line into the next element of the list `@tlist`. Remove the last character off each element if it's a new-line character (it will be).

- `@vowels = qw(a e i o u);`

  Set up a list called `@vowels` containing the text strings "a","e","i","o","u". This is a shorthand -- the statement could just as easily have been written

  `@vowels = ("a","e","i","o","u");`

- `foreach $vow(@vowels) {`

  Perform the following block with the variable `$vow` taking each value from the list `@vowels` in turn.

  You can use any list, not just a single named-list variable, with a `foreach` statement. Additionally, all features that you know on `while`, `until` and `for` loops (`next`, `redo`, `last`) are available to you.

- `push @matched_town,$town`

  Extend the list `@matched_town` by one element and save the contents of the variable `$town` into that element.

- `scalar(@matched_town)`

  Return the length of the list `@matched_town`

- `$#matched_town = -1;`

  Reduce the length of the `@matched_town` list to zero, but don't actually destroy the list itself.

Here's another example:

```
#!/usr/local/bin/perl
# booklook

open (BOOX,"books") || die "no book file\n";
chomp (@books = <BOOX>);

@randolph = grep(/Schwartz/,@books);

print "Total count: ",@books+0,"\n";
print "By Schwartz: ",@randolph+0,"\n";

foreach $book(@randolph) {
    @fields = split (/\|/,$book);
    unshift @isbns,$fields[3];
    }

$report = join(", ",@isbns);
print $report,"\n";
```

```
seal% booklook
Total count: 20
By Schwartz: 5
1-56592-149-6, 1-56592-042-2,
1-56592-284-0, 0-937175-64-1,
1-56592-324-3
seal%
```

Figure 63
Running Perl program "booklook".

· `@randolph = grep(/Schwartz/,@books);`

Build a list called `@randolph` which contains all the elements of `@books` matching the regular expression `/Schwartz/`

· `print "Total count: ",@books+0,"\n";`

· `print "By Schwartz: ",@randolph+0,"\n";`

Confirms in our example that the original book list contains 20 elements and that the one generated by `grep` contains 5.

· `foreach $book(@randolph) {`

Have `$book` take each value from `@randolph` in turn.

· `@fields = split (/\|/,$book);`

Divide the string in `$book` at vertical bar characters[1] and save each of the resulting fields into the next element of the `@fields` list.

· `unshift @isbns,$fields[3];`

Move all the elements of `@isbns` up by one and push `$fields[3]` into the newly vacated first position.

· `$report = join(", ",@isbns);`

Joins together all elements of the `@isbns` list into a single scalar, placing a ", " separator between each.

11.6 Iterating through a list

You can loop through a list (we're supposed to call them "lists" these days!) using a foreach loop running a counter, or using the second form of foreach that doesn't provide a counter.

[1] the `\` protects the `|` character from the regular expression handler

Thus:

```
#!/usr/bin/perl
@demo1 = (20,30,50,60,80,200);
@demo2 = (40,70,90,40,20,10,0);
foreach ($k=0; $k<@demo1; $k++) {
        $demo1[$k] += 2;
        }
print ("@demo1\n");
foreach $item (@demo2) {
        $item += 3;
        }
print ("@demo2\n");
```

```
$ perl prog.pl
22 32 52 62 82 202
43 73 93 43 23 13 3
$
```

Figure 64
Using a foreach loop to run through a counter

In the first form, you're providing a counter and know the element that you're on, thus you could make a change based on the element number. In the second form, you don't have a counter;[1] however, many folks don't realise that any changes you make to the "loop variable" are saved back in the list.

In other words, the second form is a very neat way of altering every element of a list. There is no need to actually know the element number, which will allow you to simplify code in many (but not position number-dependent) places.

Note that this trick only works if you specify just an array name in the `foreach` statement. If you wrote

    foreach $item(@this,@that)

then changes you make to `$item` are not reflected back in the *`@this`* and *`@that`* lists.

Further note: the words "for" and "foreach" are interchangable, so you could save four more bytes if you really want.

Another possibility is to use the `map` function, or keep your own counter and use the second form of "for".[2]

```
!#/usr/bin/perl
@demo1 = (20,30,50,60,80,200);
@demo2 = (40,70,90,40,20,10,0);
@demo1 = map($_+7,@demo1);
print ("@demo1\n");
foreach $item (@demo2) {
        $demo2[$n] += 9;
        $n++;
        }
print ("@demo2\n");
```

```
$ prog2.pl
27 37 57 67 87 207
49 79 99 49 29 19 9
$
```

Figure 65
Using your own counter

The `map` function is worthy of further study. Each element of a list is put into a special variable called `$_` in turn. You can then perform any operation that you wish on that item.

[1] and, no, there is not one in any special variable that this author knows of
[2] I don't know why you would do this

11.7 List slices

The notation `@abc` referred to the whole of the list `abc`, and the notation `$abc[4]` referred to the scalar which is element 4 of the list. `$#abc` referred to the index[1] of the last element of the list `abc`.

There's one more notation: `@abc[4,5]` refers to a slice of the list – elements 4 and 5 in this case. Within the square brackets you give a list of ordinates.

By the way, `$abc` would be a separate scalar. It is recommended that you don't use both `$abc` and `@abc` in the same program[2] as this could be very confusing when you come to maintain your program later. Here's some array slices in use:

```
#!/usr/local/bin/perl
# vgrep - matches word in file;
# prints surrounding slice

print "file: ";
chop ($fn = <STDIN>);
open (FI,$fn) || die "Can't read that\n";
@source = <FI>;
@end = @source[-3..-1];
close FI;

print "-------------file
  starts\n",@source[0..2];
print "-------------file ends\n",@end;
print "Look for: ";
chop ($look = <STDIN>);

for ($k=0;$k<@source;$k++) {
   next unless ($source[$k] =~ /$look/);
   $n++ &&
   print "==============================\n";
   print "from line ",$k+1,"\n";
   print @source[(($k>2)?($k-2):0)..($k+2)];
   }
```

- `@end = @source[-3..-1];`

  `@end` is the last three elements of the `@sources` list
- `print "-------------file starts\n",@source[0..2];`

  Printing a slice
- `print @source[(($k>2)?($k-2):0)..($k+2)];`

  More complex specification of a slice!

Five elements of a list -- from `$k-2` to `$k+2`, but taking care that the calculation `$k-2` doesn't give a negative result which would select the `END` of the list.

```
seal% vgrep
file: towns
-------------file starts
International
Canada
USA
-------------file ends
Premium
Product Purchase
Adult Services
Look for: llom
from line 3060
Porthcawl-4-Figs.
Wick-(Mid_Glam)
Millom
Broughton-in-Furness
Ravenglass
==============================
from line 3870
Dolton
Langtree
Sullom_Voe
Brae
Hillswick
==============================
from line 4238
Limpsfield_Chart
Craddock
Cullompton
Tiverton-(Devon)
Bickleigh
seal%
```

Figure 66
Running Perl program "vgrep".

[1] ordinate
[2] though there will be exceptions

A very common use of slices:

```
printf
"%s %s %d %.2f %d %d\n" @fields[0,4..6,1,3];
```

To print out six fields from a record, formatted, and in a different order to which they are held in the list.

11.8 Anonymous lists

Although we've given names to most of the lists we've used so far, there are times you may want to use a list within just a single line, and there's no need to name it.

Here's an example:

```
#!/usr/local/bin/perl
# dow - day of week

print "day No.: ";
$val = <STDIN>%7;

$name =
  ("Sun","Mon","Tue","Wed","Thu","Fri","Sat")[$val];

print "Day $val is $name\n";
```

```
seal% dow
day No.: 3
Day 3 is Wed
seal% dow
day No.: 1
Day 1 is Mon
seal% dow
day No.: 7
Day 0 is Sun
seal%
```

Figure 67

Running Perl program "dow".

You may ask "Why?" We'll be coming back to lists later and you'll see that anonymous lists are much more than just a curiosity!

11.9 Summary

Functions that operate on lists include `push` and `pop`, `shift` and `unshift`, `splice`, `reverse` and `sort`.

`Grep`, `join`, `split`, `pack`, `unpack` and `chomp` manipulate strings and lists.

In a list context, `<FH>` reads the rest of the file `open` on `FH` into a list, one line per list element.

`foreach` allows you to assign the contents of each element of a list in turn into a variable.

You may set `$#list` to elongate or shorten a list, although Perl normally elongates lists automatically as necessary.

The `..` operator allows you to generate a counter list.

You may use list slices using a notation such as

```
        @list[0,1,4..7]
```

If you want to use a list at just one place in your program and don't need to name it, you can use an anonymous list.

Exercise

Read the whole of the file access_log supplied in the course profile. Put it into a list, taking care to eliminate the first line which is in a different format.

- Ask the user to enter a number of host names and report any lines that report "404" in the 9th field of the line for each host in turn.

Our example answer is web_ac

Sample

```
graham@otter:~/profile/answers_pp> web_ac
hosts of interest? perch plaice
perch - - [12/Nov/1998:05:25:37 -0500]"GET /jrl/applets/search.zip HTTP/1.0" 404 -
plaice - -[11/Nov/1998:06:40:19 -0500]"GET /penguin HTTP/1.0" 404 -
plaice - -[11/Nov/1998:06:40:30 -0500]"GET /penguin/in_work HTTP/1.0" 404 -
plaice - -[11/Nov/1998:06:40:53 -0500]"GET /penguin/in_work/JA HTTP/1.0" 404 -
plaice - -[11/Nov/1998:06:41:19 -0500]"GET /penguin/in_work/JA/JA1.2.traineeware HTTP/1.0" 404 -
plaice - -[11/Nov/1998:06:41:28 -0500]"GET /penguin/in_work/JA/JA1.2.traineeware/folder HTTP/1.0"
404 -
plaice - -[11/Nov/1998:06:41:41 -0500]"GET /penguin/in_work/JA/JA1.2.traineeware/folder/
hawthorn.html HTTP/1.0" 404 -
graham@otter:~/profile/answers_pp>
```

Modify the application to produce a report that tabulates only date and time, page accessed and return code.

Our example answer is web_ac2

Sample

```
graham@otter:~/profile/answers_pp> web_ac2
hosts of interest? perch plaice
Host perch
 [12/Nov/1998:05:25:37    404 /jrl/applets/search.zip
Host plaice
 [11/Nov/1998:06:40:19    404 /penguin
 [11/Nov/1998:06:40:30    404 /penguin/in_work
 [11/Nov/1998:06:40:53    404 /penguin/in_work/JA
 [11/Nov/1998:06:41:19    404 /penguin/in_work/JA/JA1.2.traineeware
 [11/Nov/1998:06:41:28    404 /penguin/in_work/JA/JA1.2.traineeware/folder
 [11/Nov/1998:06:41:41    404 /penguin/in_work/JA/JA1.2.traineeware/folder/hawthorn.html
graham@otter:~/profile/answers_pp>
```

12 Subroutines in Perl

12.1 What are subroutines and why?

The limitations of "single block code"

You won't be the first person in the world to want to ...

- read options from the command line
- interpret form input in a CGI script
- pluralize words in English

You won't be the first person in your organisation to want to ...

- output your organisation's copyright statement
- validate an employee code
- automatically contact a resource on your web site

You may need to handle the same data in several programs, or to handle in your programs the same data that your colleagues handle in theirs.

And you may want to perform the same series of instructions at several places within the same program.

With the Perl programs you've written so far, all your code has been in a single file and indeed has "flowed" from top to bottom.

- You have not been able to call the same code in two different places
- You have not been able to share code between programs -- copying is not normally an option as it gives maintenance problems
- You have not used your colleague's code, nor code that's available for everyone on the CPAN, nor additional code that's so often needed that it's shipped with the Perl distribution.

First use of subroutines

The first computer programs were written rather like the ones that we've written so far. Each one for its own specific task.

In time, programmers (said to be naturally lazy people) noticed that they could save effort by placing commonly used sections of code into separate blocks which could be called whenever and wherever they were needed. Such separate blocks were variously known as functions, procedures or subroutines. We'll use the word "subroutine" because Perl does!

Structured programming

The subroutine approach was then taken to extreme so that all the code was put into separate blocks, each of which could be described as performing a single task.

For example, the program I run might be described as performing the task of "reporting on all towns with names matching a pattern".

That single task divides down into separate tasks, each of which would be a subroutine:

- get the pattern to match
- get the database of towns
- filter out the towns wanted
- report the results

Getting the match pattern divides down into:

- check the command line for a pattern
- read a pattern from STDIN[1]

and so on!

Even in this simple example, you'll start to discover that there are tasks that are likely to be shared between different applications. Also:

- It's easier to identify which block errors are in
- For maintenance, as a task changes, a new version can slot in easily
- No tasks are too long, so the code is easier to follow

Perl was founded on subroutines and all versions that you'll come across have full support for structured programming. We'll encourage you to use at least some of the principles of structured programming for all the programs you write.

Object oriented programming

Let's go a stage further. What's our program example above dealing with? Data about something, about an object of some type. In this case, the objects are towns.

How are we going to store information about a town?

... An array? Something else?

On disk?

In memory??

We could guess at some answers, but if someone else has already written the code for handling towns, frankly, we'd rather not be concerned about it. What do we need to be able to do?

- Create a town
- Check a town and see if it matches

and perhaps some other functions later.

Object oriented programs put objects at the centre of the design model.

The programmer who wishes to make use of objects of a particular type[2] need only know how to access the objects via subroutines[3] and to be able to tell the methods which particular object of the class to which he is referring. He identifies the particular object by using an "instance variable".

1 to run if it wasn't on the command line

2 a particular class in object oriented terms; a particular package or module in Perl terms

3 methods in object oriented terms

In Perl, you've already come across file handles, which are rather like an instance variable. You create them in your `open` call, then use them to say which file in which you're interested. But the fact remains that you don't know what goes on internally.

When you start using other classes in Perl, you'll discover that instance variables can be held in scalars, just like any other variable, and passed to methods to indicate which instance.

From version 5.00, Perl has had object oriented capabilities. Unlike languages such as Java, which force you to use object oriented techniques and to stick to the rules, Perl assumes you know what you're doing.

12.2 Calling a subroutine

If you're going to call a subroutine that's already been written, you need to do two things:

a)Include a `use` statement in your code, telling the Perl compiler to pull in the file that contains the subroutine and

b)actually call the subroutine.

Of course, this implies you know which file contains the subroutine you want, what the subroutine does, and how to call it.

12.2.1 Calling subroutines

An example program that shows some first subroutine calls:

```
#!/usr/local/bin/perl
# Sample use of subroutines

use valread;

print "Demonstration program\n";
&stamper;
sleep 2;
stamper();
sleep 2;
stamper;
sysrep;
```

```
C:\graham\wash>perl substamp
Demonstration program
Output written Mon Apr 17 15:38:45 2000
Output written Mon Apr 17 15:38:47 2000
Output written Mon Apr 17 15:38:49 2000
Perl version 5.006 running on MSWin32

C:\graham\wash>
```

Figure 68

Sample of first subroutine calls

Explanation:

```
use valread;
```

This requests the compiler to include all the subroutines in a module called *valread* (actually a file called *valread.pm*) and make them available to the rest of the program.[1]

This sample program then calls the *stamper* subroutine three times:

```
&stamper;
```

The most generic call, just as `$` means a scalar variable, so `&` means a subroutine. You must provide the `&` in Perl 4, but using Perl 5 we usually leave it out:

```
stamper();
```

The brackets indicate that this is a subroutine. If Perl already knew this, we could leave the brackets out as well:[2]

[1] In Perl 4 you would require "valread.pm"

```
          stamper;
```

In our example program, we also called the *sysrep* subroutine in the *valread* module, and we've used the `sleep` function which is built into Perl to add some delays.

12.2.2 Passing values out of subroutines

Our first subroutines didn't take any information in from the calling program, nor pass any information back. Most times, though, you'll want to have a subroutine give you an answer, or at the very least tell you that it worked.

Here's a program that calls a subroutine to read a number. The subroutine does rather more than the <> operator. It checks that you really have given digits and re-prompts and re-checks if necessary

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

# Sample use of subroutines

use valread;

stamper;
print "please enter a number ";
$yoused = readnumber();
print "You entered $yoused!\n";
```

```
C:\graham\wash>perl subgetnum
Output written Mon Apr 17 15:51:00 2000
please enter a number ten
Number format error. Please retry 10
You entered 10!

C:\graham\wash>
```

Figure 69
Sample of calling a subroutine to read a number

12.2.3 Passing values in to subroutines

Can we extend our example so that our subroutine can act on values passed to it? Yes; all we have to do is specify those values after the subroutine name in the call:

```
#!/usr/local/bin/perl
# Sample use of subroutines

use valread;

$trainees = readrange("Students booked",0,10);
$duration = readrange("Days long",1,5);
print "A $duration day course with $trainees
  students\n";
print $duration*($trainees+1)," lunches\n";
```

```
C:\graham\wash>perl subgnrange
Students booked = eight
Number format error. Please retry 8
Days long = 8
Must be in range 1 to 5 ...
Days long = 5
A 5 day course with 8 students
45 lunches

C:\graham\wash>
```

Figure 70
Extending the sample subroutine to act on values passed to it

In this example, our *readrange* subroutine is actually doing quite a bit of work. It

- Prompts the user
- Reads the reply
- Checks that it's a number and in range
- Re-prompts on error until the user makes a valid entry.

Short program, but very effective.

[2] In this example, it knew it was a subroutine because of the `use` statement

12.3 Writing your own subroutine

There's a wide range of subroutines already available to you - built in to Perl, provided with the Perl distribution, available from the CPAN, and perhaps also available from other work that's been done at your company. But the time will come when you want to write your own subroutine, placing it within the same file as your program.

Here's a program which prompts for a player's name and validates that a single name was entered:

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

# calling a subroutine in the same file

announce();

foreach $quadrant("North","West","South","East")
  {
        push @names,getplayer($quadrant);
        }
print "Players are @names.";

#### A comment to separate off subroutines ######

sub announce {
  $now = localtime();
 print "Bridge game, call for players at $now\n";
  }

sub getplayer {
        $angle = $_[0];

        do {
        $rpt = 0;
       print "Player sitting at $angle is ... ";
        chomp ($response = <STDIN>);
        ($response !~ /^\S+$/) and
                  print "One name please\n" and
                        $rpt=1;
        }
                while ($rpt);

        return ($response);
        }
```

```
C:\graham\wash>perl mysub
Bridge game, call for players at
  Mon Apr 17 17:01:58 2000
Player sitting at North is ...
  John
Player sitting at West is ...
  Julie
Player sitting at South is ...
  Julian Brown
One name please
Player sitting at South is ...
One name please
Player sitting at South is ...
  Julian
Player sitting at East is ...
  Jenny
Players are John Julie Julian
  Jenny.
```

Figure 71
Prompting for single-word entries, then validating

You'll notice that we don't have a `use` statement in our program, since there is no other file to use; the subroutines are included in-line further down the file.

The subroutines are called as follows:

```
        announce();
```

and

```
          push @names,getplayer($quadrant);
```
although it would have been just as effective to write
```
          $plname = getplayer($quadrant);
          push @names,$plname;
```
for the second subroutine call.

The subroutines themselves are declared as a separate block, starting with "sub" and the name and enclosed in curly braces. Although subroutines can be placed anywhere in the source, it's usual to place them at either the start or the end of the file, and to put some sort of bold comment statement to separate them from the rest of the code.

Subroutine *announce* doesn't receive any parameters from the main program, nor does the main program take any notice of what it passes back, but a parameter is passed in and out, to and from subroutine *getplayer*.

12.3.1 Passing parameters in

Parameters are passed into subroutines in a list with a special name – it's called `@_` and it doesn't conform to the usual rules of variable naming. This name isn't descriptive, so it's usual to copy the incoming variables into other variables within the subroutine. Here's what we did at the start of the *getplayer* subroutine:
```
          $angle = $_[0];
```
If multiple parameters are going to be passed, you'll write something like:
```
          ($angle,$units) = @_;
```
Or if a list is passed to a subroutine:
```
          @pqr = @_;
```
In each of these examples, you've taken a copy of each of the incoming parameters; this means that if you alter the value held in the variable, that will not alter the value of any variable in the calling code. This copying is a wise thing to do; later on, when other people use your subroutines, they may get a little annoyed if you change the value of an incoming variable!

12.3.2 Returning values

Our first example concludes the subroutine with a return statement:
```
          return ($response);
```
which very clearly states that the value of `$response` is to returned as the result of running the subroutine.

Note that if you execute a `return` statement earlier in your subroutine, the rest of the code in the subroutine will be skipped over. For example:
```
          sub flines {
                  $fnrd = $_[0];
                 open (FH,$fnrd) or return (-1);
                  @tda = <FH>;
                  close FH;
                  return (scalar (@tda));
                  }
```
will return a `-1` value if the file requested couldn't be opened for

`read`, otherwise the file will be read and the number of lines read passed back as the returned value.

You may often see Perl subroutines that don't end with a `return` statement. Is anything passed back? Yes, Perl always returns something; if there's no explicit `return` statement, it passed back the result of the last operation in the subroutine! Frequently you'll see a subroutine concluding:

```
        $response;
        }
```

or even

```
        <STDIN>;
        }
```

or

```
        1;
        }
```

12.4 Writing subroutines in a separate file

Subroutines are often reused between programs. You really won't want to rewrite the same code many times, and you'll certainly not want to have to maintain the same thing many times over.

Plan of action:

a) Place the subroutines in a separate file, using a file extension *.pm*

b) Add a `use` statement at the top of your main program, calling in that file of subroutines

c)Add a `1;` at the end of the file of subroutines. This is necessary since `use` executes any code that's not included in subroutine blocks as the file is loaded, and that code must return a true value – a safety feature to prevent people `use`-ing files that weren't designed to be `use`-ed.

main program (file ms2):

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

# calling subroutines in a different file

use bridges;

announce();

foreach $quadrant("North","West","South","East")
  {
        push @names,getplayer($quadrant);
        }
print "Player's names are @names.";
```

```
C:\graham\wash>perl ms2
Bridge game (2), call for players
  at Mon Apr 17 18:06:40 2000
Player sitting at North is ...
  David
Player sitting at West is ...
  Celine
Player sitting at South is ...
  Bernice
Player sitting at East is ...
  Albert
Player's names are David Celine
  Bernice Albert.
C:\graham\wash>
```

Figure 72

Prompting for single-word entries, then validating

Subroutines in file *bridges.pm*:

```
# Perl "use"d file bridges.pm

sub announce {
        $now = localtime();
        print "Bridge game (2), call for players at $now\n";
        }

sub getplayer {
        $angle = $_[0];

        do {
        $rpt = 0;
        print "Player sitting at $angle is ... ";
        chomp ($response = <STDIN>);
        ($response !~ /^\S+$/) and
                        print "One name please\n" and
                        $rpt=1;
        }
                while ($rpt);

        return ($response);
        }

1;
```

12.5 Scope

Although we've been passing variables around between subroutine in Perl, you don't always need to; you might take advantage of the fact that (by default) variables are shared between subroutines.

12.5.1 Global Scope

In this example, both `$k` and `@vals` are used in both the subroutine and the main program:

```
#!/usr/local/bin/perl
# variables are global by default
sub getvals {
        for ($k=0;$k<5;$k++) {
                print "Enter value ",$k+1,": ";
                chomp ($vals[$k] = <STDIN>);
                }
        }

###############################################
print ($k = "======================","\n");
getvals();
print "values: @vals\n";
print $k,"\n";
```

We've made use of the global nature of the list called `@vals` by printing it out in the main code, but we were hoping that printing out `$k` would give another row of `=` signs. It didn't. It gave the value 5 since the variable `$k` is also used in the subroutine.

```
C:\graham\wash>perl globvar
=======================
Enter value 1: 4
Enter value 2: 6
Enter value 3: 4
Enter value 4: 3
Enter value 5: 6
values: 4 6 4 3 6
5

C:\graham\wash>
```

Figure 73

Prompting for single-number entries, then validating

12.5.2 my variables

It wouldn't be practical to share subroutines between as many programs if variables were all global. There would be long lists of banned variable names supplied with each module file, and occasions when you couldn't use two modules that you wanted because they both happened to use the same variable internally.

To prevent variables that you use within subroutines being visible outside, declare them as being `my` variables. A `my` variable is created afresh each time that the declaration is encountered, and the `my` variable is discarded when the program execution reaches the end of the block on which the variable was declared.

Thus, we could correct our previous example as follows:

```
#!/usr/local/bin/perl
# variables are global by default - use of my
sub getvals {
        my $k;
        for ($k=0;$k<5;$k++) {
                print "Enter value ",$k+1,": ";
                chomp ($vals[$k] = <STDIN>);
                }
        }

###############################################

print ($k = "======================","\n");
getvals();
print "values: @vals\n";
print $k,"\n";
```

```
C:\graham\wash>perl myvar
=======================
Enter value 1: 5
Enter value 2: 3
Enter value 3: 4
Enter value 4: 7
Enter value 5: 8
values: 5 3 4 7 8
=======================

C:\graham\wash>
```

Figure 74
Correcting a prior mistake by declaring a my variable

The solution is good, though the author of the subroutines needs to be well disciplined to ensure that a `my` declaration is used for every variable that's to be localised. In practice, it's very easy for a programmer to make an error and leave out a few `my`s. Perhaps no great harm is done at first, but later on as others use the subroutines it's likely that one of the missing `my` statements will cause a variable conflict.

A module called *strict* is provided with the Perl distribution itself. If you `use strict;` you are asking the compiler to be strict with you and reject any variables that are not declared `my`. Most authorities will tell you that you should always

```
        use strict;
```

if you're writing files of subroutines.

We started this module by showing you how to call subroutines provided by others using a file called *valread.pm*. This file contains examples of `strict`, `my`, parameter passing and `return` – a good revision of the subjects we have covered so far in this module.

```
# File of subroutines - module P209

use strict;

sub stamper {
        my $now = localtime();
        print "Output written $now\n";
        }

sub sysrep {
        print "Perl version $] running on $^O\n";
        }

sub readnumber {
        my $input = <STDIN>;
        while ($input !~ /^\s*[-+]?\d*\.?\d*\s*$/ or
                $input !~ /\d/) {
                print "Number format error. Please retry ";
                $input = <STDIN>;
                }
        $input+0;
        }

sub readrange {
        my ($prompt,$low,$high) = @_;
        die "readrange - wrong number of parameters\n"
                if ($#_ != 2);
        print $prompt," = ";
        my $given = readnumber();
        while ($given < $low or $given > $high) {
                print "Must be in range $low to $high ...\n$prompt = ";
                $given = readnumber();
                }
        $given;
        }

1;
```

Exercise

1. Write a subroutine that will calculate the hypotenuse of a triangle. Here's an example of how you would call it:

```
        $h = hypot(5,12);
```

and the result to be returned into `$h` is the square root of the sum of the squares of the two parameters (using Pythagoras's theorum).

Write a test program in the same file as the subroutine to test it. Make two calls to the subroutine, using 5,12 and 4,5 as the parameters. Check that the results you get back are as follows:

"answers are 13 and 6.40312423743285"

Our example answer is py

Place your hypot subroutine in a separate file, and have Perl load that file into your test code at run time, via a `use` statement.

Our example answer is py2

12.6 packages

The subroutines that we've studied so far in this module have all operated well in isolation from one another, and there are times where the facilities provided are all we need.

There are, though, many times that you'll want information to be retained from one call to a subroutine to another call to the same subroutine, or from a call to one subroutine through to a call to another subroutine.

Here's an example:

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

# using a module to retain information

use mailfilter;

mf_init("inbox");
$count = 0;

while ($fromline = mf_get()) {
        print $fromline;
        $count++;
        }
print "total of $count emails\n";
```

```
C:\graham\wash>perl maillister
Subject: Re: Coffee, mobile
  phones, etc
Subject: Java course - timing and
  text of proposal
Subject: Hi there!
Subject: US contact point
Subject: RE: Java, September
Subject: hello
Subject: whtas been going on?
Subject: hello
Subject: PASS MCSE
total of 9 emails

C:\graham\wash>
```

Figure 75

Information retained from one call to a subroutine to another call to a subroutine

We can't use only `my` variables within the subroutines in the *mailfilter* modules since each call to `mf_get` relies on the effect of previous calls to `mf_get` and/or `mf_init`.

We couldn't use truly global variables in the calling program or in other modules in case such variables conflicted with variables.

In fact, variables in Perl are not truly global. Each variable belongs to a package. Subroutines are a special type of variable, and they belong to packages too.

By default, variables belong to *package main*. Package statements may be used to switch the current package; the package statement stays in force from where it is placed to the end of the current block, or to the end of the current file of subroutines, or until it is over-ridden.

Individual variables may be referenced within a different package to the current package by qualifying the variable name with the package name followed by `::`. For example:
`$count` in package *mailfilter* is `$mailfilter::count`

To show you how this works, here's the *mailfilter.pm* module that we called from our program earlier in this section:

```
# mailfilter - reads a pure text mailbox and
# returns subjects one by one

package mailfilter;

sub main::mf_init{
        my $filename = $_[0];
        open (FH,$filename) or die "No mail file $filename\n";
        my $strh = <FH>;
        $count = 1;
        }

sub main::mf_get {
        my $retline,$nextline;
        $count++;
        while ($nextline = <FH>) {
                last if ($nextline =~ /^\+OK/i);
                $count++;
                $retline = $nextline if ($nextline =~ /^Subject:/i);
        }
        $retline =~ s/^Subject:\s+/Subject: /i;
        $retline;
        }

1;
```

Although we've still used a number of my variables, the following variables are global through the package:

```
          FH
          $count
```

and the following subroutine names have been forced into the main package (or namespace):

```
          mf_init
          mf_get
```

Much better, but still not perfect. What would happen if two modules both contained subroutines called `mf_init`?

It's best practice to include the subroutine names within the package name space. The only reason we didn't do so earlier was so that we could show you how to call packages without having to start with a long explanation.

Here's that last example, rewritten using that best practice and now printing out the number of lines in the email box as well as the number of emails, both using a variable called `$count`.

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

# using a module to retain information

use mf2;

mf2::init("inbox");
$count = 0;

while ($fromline = mf2::get()) {
        print $fromline;
        $count++;
        }
print "total of $count emails\n";
print "total of $mf2::count lines\n";
```

```
# mailfilter - reads a pure text mailbox and
  returns
# subjects one by one

package mf2;

sub init{
        my $filename = $_[0];
       open (FH,$filename) or die "No mail file
  $filename\n";
        my $strh = <FH>;
        $count = 1;
        }

sub get {
        my $retline,$nextline;
        $count++;
        while ($nextline = <FH>) {
                last if ($nextline =~ /^\+OK/i);
                $count++;
              $retline = $nextline if ($nextline
  =~ /^Subject:/i);
        }
        $retline =~ s/^Subject:\s+/Subject: /i;
        $retline;
        }

1;
```

```
C:\graham\wash>perl maillister
Subject: Re: Coffee, mobile
  phones, etc
Subject: Java course - timing and
  text of proposal
Subject: Hi there!
Subject: US contact point
Subject: RE: Java, September
Subject: hello
Subject: whtas been going on?
Subject: hello
Subject: PASS MCSE
total of 9 emails
total of 896 lines
```

Figure 76
Using best practice

12.7 Calling objects

You've just seen how we can place a lot of data and subroutines relating to one particular topic, in this case emails, into a single package. You probably noticed how the author of the main program now need only be aware of how to call the subroutines, and need not concern himself with internal variables. Good news, since it allows the subroutines to be looked after by one programmer, and the main program by another.

Let's see if we can take this further.

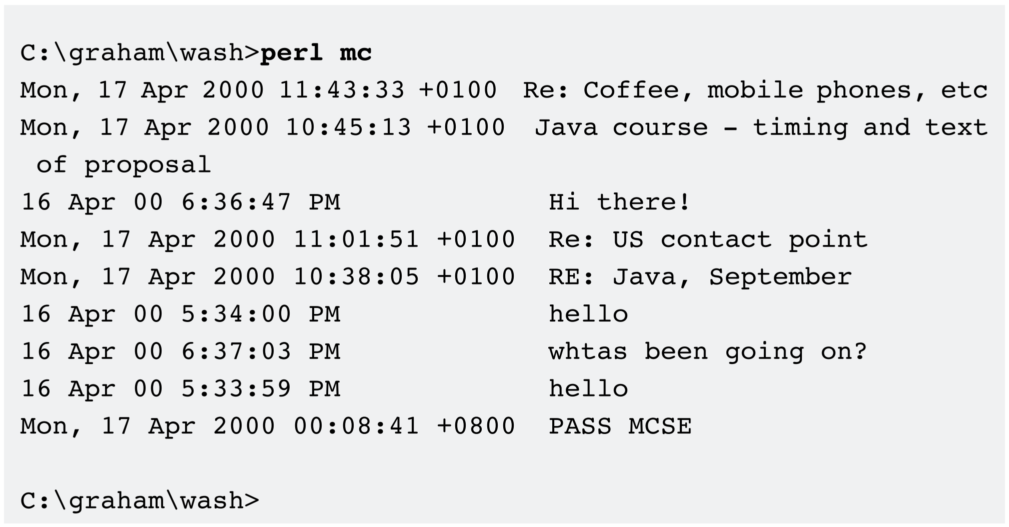
Imagine that you want to write a whole series of programs that each analyses multiple emails. What shall we do as the application programmer?

Firstly, we'll want to read all the emails. We don't care how they're stored internally, we just want to have a reference to them (rather like with a file handle, where we don't want to know which particular sectors and blocks of our disk drive are involved!).

Then we'll step through each of the emails in turn finding the date stamp and subject, and listing them out.

All the while, as the application programmer, not knowing the format of the email files themselves!

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

# Towards object orientation!

use email;

while ($mailref = email::new("email","inbox")) {
        push @mailq,$mailref;
        }

foreach $letter(@mailq) {
        $da = email::get($letter,"date");
        $su = email::get($letter,"subject");
        printf ("%-32.32s %s\n",$da,$su);
        }
```



```
C:\graham\wash>perl mc
Mon, 17 Apr 2000 11:43:33 +0100  Re: Coffee, mobile phones, etc
Mon, 17 Apr 2000 10:45:13 +0100  Java course - timing and text
 of proposal
16 Apr 00 6:36:47 PM              Hi there!
Mon, 17 Apr 2000 11:01:51 +0100  Re: US contact point
Mon, 17 Apr 2000 10:38:05 +0100  RE: Java, September
16 Apr 00 5:34:00 PM              hello
16 Apr 00 6:37:03 PM              whtas been going on?
16 Apr 00 5:33:59 PM              hello
Mon, 17 Apr 2000 00:08:41 +0800  PASS MCSE

C:\graham\wash>
```

Figure 77
Listing out emails

This code does indeed do the trick. It calls the new subroutine in the email package and gets some sort of reference (!) back. These references are stored in a list until there are no more of them available.

Each of the references is then passed to another subroutine in the email package which extracts and returns the appropriate parameter.

The author of the code above is totally unaware of how the data is being stored internally, but the program works:

Our requirement so far has been fulfilled, but we've repeated the package name *email* numerous times. It turns out that Perl knows when a reference points to a thing (an object) of type `email`, and so we can simplify our coding somewhat. Note that we're not doing this just out of laziness; there's no reason why we couldn't have a list containing a mixture of emails and postallet-ters, and the new notation will let us dynamically call the appropriate `get` subroutine within our reporting loop.

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

# Towards object orientation!

use email;

while ($mailref = new email("inbox")) {
        push @mailq,$mailref;
        }

foreach $letter(@mailq) {
        $da = $letter -> get("date");
        $su = $letter -> get("subject");
        printf ("%-32.32s %s\n",$da,$su);
        }
```

Object orientation has a whole language to itself, but we've tried to steer clear of all the jargon in this section. As you progress, though, you'll want to know that:

a class is a package

a methodis a subroutine

andan instanceis a reference

And, yes, Perl does support polymorphism, nested multiple level inheritance, and most of the other things that those of you experienced in OO techniques will be asking about. It turns out that Perl's OO model is much more flexible than the OO model used in many other languages, but then this is Perl, so you won't be surprised!

12.8 Writing a class – an introduction

Many Perl users use a considerable number of classes that have been written by others before they start writing their own. Many of the standard modules that are provided with Perl use the object oriented approach, and many of the modules available on the CPAN also do so.

In order to complete the last example, we've added a listing of the *email.pm* module onto the end of this section of the course.

```
# email - reads a pure text mailbox and returns
# email object references one by one

package email;

$firstcall = 1;

sub new{
        my ($class, $filename) = @_;
        if ($firstcall) {
                open (FH,$filename) or die "No mail file $filename\n";
                $firstcall = 0;
                my $strh = <FH>;
                }
        my @thismail;
        while ($nextline = <FH>) {
                last if ($nextline =~ /^\+OK/i);
                push @thismail,$nextline;
                }
        (@thismail == 0) and return (0);
        bless \@thismail,$class;
        }

sub get {
        my ($inst,$what) = @_;
        foreach $line(@$inst){
                if ($line =~ /^$what:/i) {
                        $line =~ s/^\S+\s+//;
                        $line =~ s/\s*$//;
                        return $line;
                        }
                }
        return (0);
        }

1;
```

13 Special Variables

You'll recall how parameters to subroutines are passed in using the list `@_`.

`@_` is just the first example of a whole range of special variables that Perl provides for you.

13.1 The Command line

If you use your computer's operating system from the command line, how many commands do you know that never take parameters? And, yes, I mean NEVER. Very few. On Linux, perhaps just `pwd` and `logout`.

And yet your programs so far haven't been able to read from the command line.

Command line parameters

All the parameters from the command line actually are easily available to you. They're in a list called `@ARGV`. You can, of course, find out how many parameters there are using `$#ARGV`.

```
#!/usr/local/bin/perl
# clp - command line parameters

for ($k=0;$k<=$#ARGV;$k++) {
 print "Parameter :",$k+1," value $ARGV[$k]\n";
 }

print "total of ",$#ARGV+1," parameters\n";
```

```
seal% clp
total of 0 parameters
seal% clp -v test.txt
Parameter :1 value -v
Parameter :2 value test.txt
total of 2 parameters
seal% clp tel*
Parameter :1 value tel2
Parameter :2 value tel3
Parameter :3 value tel4
Parameter :4 value telegram
total of 4 parameters
seal%
```

Figure 78

Running Perl program "clp".

You'll notice how it's very often the calling program (the shell) that expands metacharacters such as `*` and `?`, so that there was no need for our program to do so. Of course, had we wanted to expand such things within Perl, there are two different ways.

The name of your program.

Your program can find its own name using the variable `$0` which, like `@ARGV`, is always there for the referencing.

```
#!/usr/local/bin/perl
# myname - print program name

print "This program is $0\n";
```

```
seal% myname
This program is myname
seal% /extra/disc0.slice7/perl/profile/book/myname
This program is /extra/disc0.slice7/perl/profile/
  book/myname
seal%
```

Figure 79

Running Perl program "myname".

As we carry on through this section, you'll learn of many more variables with short, obscure names.

In case you prefer to use longer names, a Perl module is provided. If you

```
    use English;
```

at the top of your code, you'll be able to map all the very short variable names onto longer ones. `$0` becomes

`$PROGRAM_NAME`

So:

```
#!/usr/local/bin/perl
# myname2 - print program name

use English;
print "This program is $PROGRAM_NAME\n";
```

```
seal% myname2
This program is myname2
seal% ./myname2
This program is ./myname2
seal%
```

Figure 80
Running Perl program "myname2".

Throughout this section, we'll use the short default names for special variables, but we'll include the alternative(s) in brackets after each as we first introduce it.

13.2 Information variables

`$0` (or `$PROGRAM_NAME`) was an information variable. You can make use of it, but you would not normally change it. There are a number of other information variables that you might wish to use.

```
#!/usr/local/bin/perl
# info - special information variables

print "please enter data ";
$b = <STDIN>;

open(FH,"hgsaghsadghj");

print "Input line number:     $.\n";
  # $INPUT_LINE_NUMBER or $NR
print "Latest Error:       $! or ",$!+0,"\n";
  # $OS_ERROR or $ERROR
print "Process ID:          $$\n";
  # $PROCESS_ID or $PID
print "Real User ID:        $<\n";
  # $REAL_USER_ID or $UID
print "Effective User ID:     $>\n";
  # $EFFECTIVE_USER_ID or $EUID
print "Real Group ID:         $(\n";
  # $REAL_GROUP_ID or $GID
print "Effective Group ID:    $)\n";
  # $EFFECTIVE_GROUP_ID or $EGID
print "Perl Version:          $]\n";
  # $PERL_VERSION
print "Operating System:      $^O\n";
  # $OSNAME
print "Script Start time:     $^T\n";
  # $BASETIME
print "Program Name         $0\n";
  # $PROGRAM_NAME
print "Executable Name        $^X\n";
  # $EXECUTABLE_NAME
```

```
seal% ./info
please enter data asdasd
Input line number: 1
Latest Error:     No such file or
                   directory or 2
Process ID:        10772
Real User ID:      2000
Effective User ID: 2000
Real Group ID:     1999 14 1999
Effective Group ID:1999 14 1999
Perl Version:      5.003
Operating System:  solaris
Script Start time: 917035598
Program Name       ./info
Executable Name   /usr/local/bin/
  perl
seal%
```

Figure 81
Running Perl program "info".

```
C:\perlcourse>perl a:info
please enter data dffgdfgdfgfgd
Input line number:      1
Latest Error:          No such file
  or directory or 2
Process ID:             -254985
Real User ID:           0
Effective User ID:      0
Real Group ID:          0
Effective Group ID:     0
Perl Version:           5.00502
Operating System:       MSWin32
Script Start time:      917040048
Program Name            a:info
Executable Name
  C:\PERL\BIN\PERL.EXE

C:\perlcourse>
```

Figure 82
Running Perl program "info" on "coypu".

Some of the fields in this report are system dependent, so it'll look rather different on "coypu" -- our Microsoft Windows System.

Example - use of the $^O variable

Although Perl originated in the Unix world, these days many of our users are running on Linux, Mac, or Microsoft based windows systems, and the use of Perl has grown as employers want their staff to use a language which can easily be ported.

One of the specific "issues" that arises with code that is to be portable is how to terminate your program. On a Linux or Unix system, simply dropping out of the end of the script is fine, as your output is left displayed on your window. Unfortuanatly, when you drop out of the bottom of your Perl program on a Windows system, the operating system closes the window for you automatically and you results will disappear before you have a chance to read them. By adding

```
        $^O =~ /^MS/ and <STDIN>;
```

to the end of your program, you'll cause you program to pause when run on Windows to wait for the user to press the [Enter] key before the window is removed ... but there won't be any need for the user to do this on Linux.

13.3 Behaviour changing variables

The variables in this section all start with default values which can be examined. However, these variables can be changed, and changing them will affect the subsequent operation of certain features of Perl.

Format control

Let's take `$"` (or `$LIST_SEPARATOR`) as an example. This variable controls the string that is placed between each element of a list as the list is expanded within double quotes.

You may recall earlier in the course that expanding a list within double quotes put a space between each element. That's because `$"` defaults to a single space.

```
#!/usr/local/bin/perl
# list_sep - demo of $"

@sample = (1,2,"three",4);

prs("default");

$" = ", ";
prs("comma-space");

$" = "";
prs("empty");

$" = "\n";
prs("newline");

###########################
```

```
seal% list_sep
default: 1 2 three 4
comma-space: 1, 2, three, 4
empty: 12three4
newline: 1
2
three
4
seal%
```

Figure 83
Running Perl program "myname2".

```
sub prs {
print "$_[0]: @sample \n";
}
```

`$,` (or `$OFS` or `$OUTPUT_FIELD_SEPARATOR`) affects the expansion of strings, but this time in the `print` statement. It specifies the string that's output between each element of `print`'s list.

```
#!/usr/local/bin/perl
# print_sep - demo of $,

@sample = (1,2,"three",4);

prs("default");

$, = " ";
prs("space");

$, = ", ";
prs("comma-space");

$, = "\n";
prs("newline");

###########################

sub prs {
print "$_[0]: ",$sample[0],@sample[1..3],"\n";
}
```

```
seal% print_sep
default: 12three4
space:  1 2 three 4
comma-space: , 1, 2, three, 4,
newline:
1
2
three
4

seal%
```

Figure 84
Running Perl program "print_sep".

`$\` (or `$ORS` or `$OUTPUT_RECORD_SEPARATOR`) lets you set a delimiter to be output at the end of each `print` statement. Normally, this variable is empty and you use and explicit `\n` on each `print` statement.

Variables that control input

Remember:

```
 $abc = <STDIN>
```

reads from `STDIN` up to and including a new-line character. What is a new-line character for these purposes? It's the character defined in `$/` (or `$RS` or `$INPUT_RECORD_SEPARATOR`), and it can even be a string of characters for systems with multiline separators. Two special cases:

`undef $/;` will cause the whole input to read as 1 record

`$/ = "";` will delimit at each blank line (paragraph mode)

Variables that control buffering

Under normal circumstances, your computer collects information you print into buffers, and then actually outputs the buffer when it gets a certain amount of data in it. That's much more efficient that outputting character-by-character, or after each `print` statement.

Perl uses this behaviour, and unless you take alternative actions, output will actually be sent:

- When a new-line character is encountered in the output stream or
- When the output is to `STDOUT` and a `<STDIN>` is encountered (so that you have the request for input visible when you type)

By changing `$|` (or `$OUTPUT_AUTOFLUSH`) to a non-zero value, you change Perl's behaviour so that the buffer is sent at the end of each `print` statement. Uses of this include:

- Stay-alive web pages from CGI scripts
- "I'm working" "..." indicators from a heavy program
- Interaction with other processes via pipes
- Interaction with other computers via sockets

This example looks the same in the book, but when you run it

- the first line of dots comes out one-per-second
- the second line, after a LONG pause, appears all at once!

```
#!/usr/local/bin/perl
# buffer - show buffering

$|=1;
dot();
$|=0;
dot();

sub dot {
 foreach $k(1..15) {
 sleep 1;
 print ".";
 }
 print "\n";
 }
```

```
seal% buffer
...............
...............
seal%
```

Figure 85
Running Perl program "buffer".

13.4 The default input and pattern match space

Let's look at the way we work. We take on a project and we work at that project to the exclusion of other projects we have rather than jumping around doing one task on each project. We do all of the washing up, then all of the ironing, rather than alternately washing a plate and ironing a shirt.

Programming's a bit the same.

You'll read something into a variable, chop it, check that it's not a comment, make some changes to it, then print it out. Let's see a program that does that:

```
#!/usr/local/bin/perl
# capital - make all upper case

$\="\n";

while ($line = <STDIN>) {
 chop $line;
 next if ($line =~ /^\s*#/);
 $line =~ tr/a-z/A-Z/;
 print $line;
}
```

```
seal% capital < sweden.txt
HARWICH
HULL
NEWCASTLE
seal%
```

Figure 86
Running Perl program "capital".

What a lot of references to `$line` -- but there don't need to be!

Many Perl functions assume you're using the variable `$_` (or `$ARG` [1]) if you don't tell them what to work on. Here's that same program, but using `$_`.

```
#!/usr/local/bin/perl
# cap2 - make all upper case

$\="\n";

while (<>) {
 chop ;
 next if (/^\s*#/);
 tr/a-z/A-Z/;
 print ;
}
```

```
seal% cap2 < sweden.txt
HARWICH
HULL
NEWCASTLE
seal%
```

Figure 87
Running Perl program "cap2".

Where is `$_` used? Many MANY places! In this example:

- It's where information read from a file handle is placed if (and ONLY if) it's the sole item in a `while` condition
- `chop` assumes you mean `$_` if you don't tell it
- Regular expressions match to `$_` if there's no `=~` present
- `tr` [2] defaults to work on `$_`
- Even `print` without any list prints ... `$_!`

So for the uninitiated, we have a series of apparently isolated lines passing information from one to the next to the next. Of course, you know that the answer is "`$_`".

As we come to `tr`, and `s`, we'll see `$_` appear again.

Reading from <>

If no parameters are given on the command line, <>[3] reads from `STDIN`.

If parameters are given on the command line, <> reads all the lines from each file in turn, returning an end of file upon completion of reading the last file.

1 not to be confused with `@ARGV`

2 we come to that later

3 < > is different to `<STDIN>`

13.5 More options on Perl's command line

Recall so far:

`-c`	compile only
`-w`	give warning messages
`-v`	give version number of Perl

Let's now add:

`-n`	add a `while` loop, reading from <> into `$_` around your code
`-p`	as `-n`, but the loop also includes a `print` statement.

So here's a diagram of those parts of the last program we can replace with a `-p` on the command line:

```
#!/usr/local/bin/perl
# cap2 - make all upper case

$\="\n";
while (<>) {
 chop ;
 next if (/^\s*#/);
 tr/a-z/A-Z/;
 print ;
}
```

So the program could become:

```
#!/usr/local/bin/perl -p
# cap3 - make all upper case
tr/a-z/A-Z/;
```

```
seal% cap3 < sweden.txt
HARWICH
HULL
NEWCASTLE
seal%
```

Figure 88
Running Perl program "cap3".

13.6 Others

There are a number of special variables associated with the handling of regular expressions, which we'll look at when we come back to that topic.

There is also a group that concerns formatted printing (whole-page-at-a-time stuff; the sort of thing you would do with pre-printed stationary) which we cover only briefly towards the end of this course.

13.7 Summary

Perl provides you with many built-in variables that you can use as needed. Many are known by very terse combinations, and `use English;` can be used to provide additional, more understandable names.

Amongst the special variables:

`@ARGV`	contains the command line parameters
`$0` or `$PROGRAM_NAME`	the name of your program
`$^O` or `$OSNAME`	the operating system name

Some special variable effect behaviour:

`$"`	changes the separator for ".." expansion
`$/`	changes the input line delimiter
`$|`	turns buffering off and on

`$_` or `$ARG` is used in many places as the default variable for input, pattern matching and printing if you don't specify a variable explicitly.

<> reads from files named on the command line, or if there are no files named on the command line, it reads from `STDIN`.

Command-line options `-n` and `-p` can be used to wrap your whole program in an implicit loop (`-p` prints out `$_` each time) and are useful if you want to run something on every line of a file.

Exercise

Write a one-liner to print all lines containing the string "cliad" from the file "access_log" (specify the file name on the command line)

Our example answer is clfind

Sample

```
graham@otter:~/profile/answers_pp> clfind access_log
o_whelk - -[23/Jul/1998:05:25:25 -0400] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
sardine - -[23/Jul/1998:05:41:05 -0400] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
mussel - - [23/Jul/1998:05:41:13 -0400] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
skate - - [10/Sep/1998:05:03:03 -0400] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
whale - - [10/Sep/1998:05:03:14 -0400] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
trout - - [10/Dec/1998:11:50:33 -0500] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
seal - - [28/Jan/1999:12:38:09 +0000] "GET /pub/graham/cliad.html HTTP/1.0" 200 323
graham@otter:~/profile/answers_pp>
```

14 Hashes

Hashes are also known as associative arrays. We'll use the word "hashes" in this course, just as we used the word "lists" in preference to arrays in an earlier section.

Hashes are like lists except that you don't start counting the elements at 0 and go on up from there. Instead of numbers for the indexes (also known as the keys), you can use any unique value, usually a string. In other words, they're ideal for storing any table of information where you aren't concerned by the line numbers.

Here’s a file of data that matches a description:

```
Dover       Calais, Oostende
Folkestone  Bologne
Newhaven    Dieppe
Portsmouth  Caen, Le Havre, Cherbourg
Poole       Cherbourg
Plymouth    Roscoff
```

The questions asked might include:

- Where can I travel to from Dover?
- Which ports are in your list?

but would NOT include:

- What's the third port in your list?

14.1 Setting up a hash

Just as you can build up a list element-by-element, so you can build up a hash element by element.

The first time you refer to the hash, Perl creates the structure of the hash. When you write a new element, it is create. And when you write an element with an existing name, the old contents are lost.

Elements in hashes can contain anything a scalar can contain, including numbers, strings and instance variables, and can expand and contract in size as necessary to accommodate the data you save in them.

The difference as you create an element in a hash is:

1. You refer to key (index) in `{}`s rather than in `[]`s

   (That's how Perl knows you want a hash)

and of course

2. The key you give will probably be a string

   This is how we set up a hash for the data file above:

```
open (CH,"channel") || die "No channel file\n";

while ($line = <CH>) {
 ($uk,$france) = split(/\s+/,$line,2);
 $port{$uk} = $france;
 }
```

14.2 Accessing a hash

Individual elements

We can now look up an element of the hash by using its key; if we try to look up an element that's not defined, we'll get a false value returned.

```
while (1) {
 print "Port of interest: ";
 chop ($poi = <STDIN>);
 last unless ($poi);
 if ($port{$poi}) {
 print "Travel to $port{$poi}";
 } else {
 print "Don't know that one\n";
 }
 }
```

```
seal% ukports
Port of interest: Portsmouth
Travel to Caen, Le Havre, Cherbourg
Port of interest: Portsfoot
Don't know that one
Port of interest: Dover
Travel to Calais, Oostende
Port of interest:
seal%
```

Figure 89
Running Perl program "ukports".

The whole hash

Just as you referred to the whole of a list using a special character -- an @ -- you refer to the whole of a hash using a %.

You can set up a complete hash from a list, specifying alternate elements, passing the whole of a hash to a subroutine, etc, this way.

If you refer to a hash in a scalar context, you are returned some internal information about the storage used by the hash.

If you refer to a hash in a list context, you are returned alternate elements of the hash. The elements will not appear to be in any particular order. You'll look and scratch your head and ask "why?"

```
#!/usr/local/bin/perl
# niports - set up a hash

%port =
  ("Campbeltown","Ballycastle","Cairnryan",
 "Belfast",
 "Stranraer","Larne","Liverpool","Belfast");

portid(%port);

print "Port hash:\n",%port,"\n";

$stats = %port;

print "Statistics: $stats \n";

@plist = %port;
for ($k=0;$k<$#plist;$k+=2) {
 print $plist[$k],"   ",$plist[$k+1],"\n";
 }

################################################
```

```
seal% niports
Port of interest: Liverpool
Travel to Belfast
Port of interest:
Port hash:
StranraerLarneLiverpoolBelfast
 CampbeltownBallycastle
 CairnryanBelfast
Statistics: 3/8
Stranraer   Larne
Liverpool   Belfast
Campbeltown   Ballycastle
Cairnryan   Belfast
seal%
```

Figure 90
Running Perl program "niports".

```
sub portid {
 my %from = @_;
 my $poi;
 while (1) {
 print "Port of interest: ";
 chop ($poi = <STDIN>);
 last unless ($poi);
 if ($from{$poi}) {
 print "Travel to $from{$poi}\n";
 } else {
 print "Don't know that one\n";
 }
 }
}
```

In the initial setup in this example, where we assigned a list of constants to a hash, it can become hard to see which values are the keys and which are the contents of the hash.

An alternative notation: in Perl 5, you can rewrite:

```
%port =
("Campbeltown","Ballycastle","Cairnryan",
"Belfast","Stranraer","Larne","Liverpool",
"Belfast");
```

as

```
%port =
("Campbeltown"=>"Ballycastle","Cairnryan"=>
"Belfast","Stranraer"=>"Larne","Liverpool"=>
"Belfast");
```

Do be aware that => is just an alternative to a comma; you still need to be careful not to get out of step!

You can go a step further and leave the string quoting off the keys if you wish when using the => notation, thus

```
%port = (Campbeltown => "Ballycastle",Cairnryan =>
"Belfast", Stranraer => "Larne", Liverpool =>
"Belfast");
```

(if your keys aren't just single words, don't try this final step as you'll confuse the Perl compiler)

You can go a step further and leave the string quoting off the keys if you wish when using the => notation, thus:

```
%port = (Campbeltown => "Ballycastle",Cairnryan =>
"Belfast", Stranraer => "Larne", Liverpool =>
"Belfast");
```

If your keys aren't just single words, don't try this final step as you'll confuse the Perl compiler.

14.3 Processing every element of a hash

So far you've looked up explicit elements of a hash and copied the whole hash into a list from which you could:

- extract each key in turn by examining alternate elements
- count the elements `(@list/2);`

Keys and values

There are, though, other functions built in to Perl that let you go through each pair in turn without having to copy the whole hash to a list explicitly.

- `keys` returns a list of keys
- `values` returns a list of contents

Each

In this example with a short hash, the solution is fine. With a hash containing 1000 elements, perhaps it would be fine as well. How about a very large hash -- let's say 10,000 elements?

No reason at all for the program to fail even then, but the efficiency could be questioned. When you use `keys` or `values`, a list is returned and that may have a marked impact on system performance if it's huge. A Perl program that has already grabbed a couple of megabytes of memory might grab another megabyte, swap disks go wild ...

Perl is such a powerful language that even a single use of a large hash without care in that way can affect performance! Of course, Perl also has the solution for you -- the `each` function, which returns the next (key, value) pair from your list each time you call it.

```
#!/usr/local/bin/perl
# nip2 - reading a hash

%port =
  ("Campbeltown"=>"Ballycastle","Cairnryan"=>
   "Belfast",
   "Stranraer"=>"Larne","Liverpool"=>"Belfast");

foreach $from(keys %port) {
 print "From: $from\n";
 }

foreach $to(values %port) {
 print "To: $to\n";
 }

while (($from,$to) = each (%port)) {
 print "From: $from\n";
 print "To: $to\n";
 }
```

```
seal% nip2
From: Stranraer
From: Liverpool
From: Campbeltown
From: Cairnryan
To: Larne
To: Belfast
To: Ballycastle
To: Belfast
From: Stranraer
To: Larne
From: Liverpool
To: Belfast
From: Campbeltown
To: Ballycastle
From: Cairnryan
To: Belfast
seal%
```

Figure 91
Running Perl program "nip2".

How does `each` know which pair to return from the hash?

The first time you use it, it returns the first pair, but it then keeps note internally of how far it has got, so that on subsequent calls it will return the next element. Once it runs off the end of the hash, it returns null -- just once -- then starts all over again.

Think that one through! It's a natural behaviour, but it does mean that if you don't complete a whole cycle through each element of a hash, a subsequent loop may start halfway. We've done this intentionally in the next example, but sometimes it's not what you want and it can be an obscure problem to fix if you overlook it! [1]

```
#!/usr/local/bin/perl
# nip3 - reading a hash

%port =
  ("Campbeltown"=>"Ballycastle","Cairnryan"=>
 "Belfast",
 "Stranraer"=>"Larne","Liverpool"=>"Belfast");

print "keys: ";
foreach $from(keys %port) {
 print " $from";
 }
print "\n";

while (($from,$to) = each (%port)) {
 print "From: $from\n";
 last if ($from eq "Liverpool");
 }

while (($from,$to) = each (%port)) {
 print "and from: $from\n";
 }
```

```
seal% nip3
keys: Stranraer Liverpool
Campbeltown Cairnryan
From: Stranraer
From: Liverpool
and from: Campbeltown
and from: Cairnryan
seal%
```

Figure 92
Running Perl program "nip3".

Whilst you may safely alter the values of elements of a hash as you pass through them with a loop of `each` functions, and you're safe to delete elements as well, you should not add in new elements.

14.4 Ordering a hash (sorting)

With a list of just four ports, the order they're printed in probably isn't important to you. With a list of 20, it would be. But so far the order appears random.

Perl has a built-in sort function. It takes the elements of a list and returns them in standard string-comparison order.

```
#!/usr/local/bin/perl
# nip4 - reading a hash

%port =
  ("Campbeltown"=>"Ballycastle","Cairnryan"=>
  "Belfast",
  "Stranraer"=>"Larne","Liverpool"=>"Belfast");

foreach $from(sort keys %port) {
 print "From: $from\n";
 }
```

```
seal% nip4
From: Cairnryan
From: Campbeltown
From: Liverpool
From: Stranraer
```

Figure 93
Running Perl program "nip4".

[1] You'll find similar behaviour is available to you in regular expression matching, which we'll come to in a later chapter.

We could have written

```
  sort (keys (%port))
```

to clarify the order of operation if your program will be looked at by those not too familiar with Perl.

Let's sort a new, and slightly more complex, data set

```
#!/usr/local/bin/perl
# sorter

%codes = (
 "Albury" => "0127974",
 "Aldbury_Common" => "0144285",
 "Aylesbury" => "01296",
 "Avebury" => "016723",
 "Abbotsbury" => "01305",
 "Almondsbury" => "01454",
 "Ashbury" => "0179371",
 "Amesbury" => "01980" );

foreach $ex (sort keys %codes) {
 printf "%14s %s\n",$ex,$codes{$ex};
 }
```

```
seal% sorter
 Abbotsbury01305
 Albury 0127974
 Aldbury_Common0144285
 Almondsbury01454
 Amesbury 01980
 Ashbury0179371
 Avebury 016723
 Aylesbury01296
seal%
```

Figure 94
Running Perl program "sorter".

Sorting using your own subroutine for comparison

Perhaps you'll want to sort alphabetically primarily. But other times you might also want to sort in different ways, for example:

- numerically
- by length of string
- by value rather than by key
- by value, then if both values are the same, by the key

This looks like it's going to be messy. How can you possibly specify all these different options to sort in a clean, easily understood way?

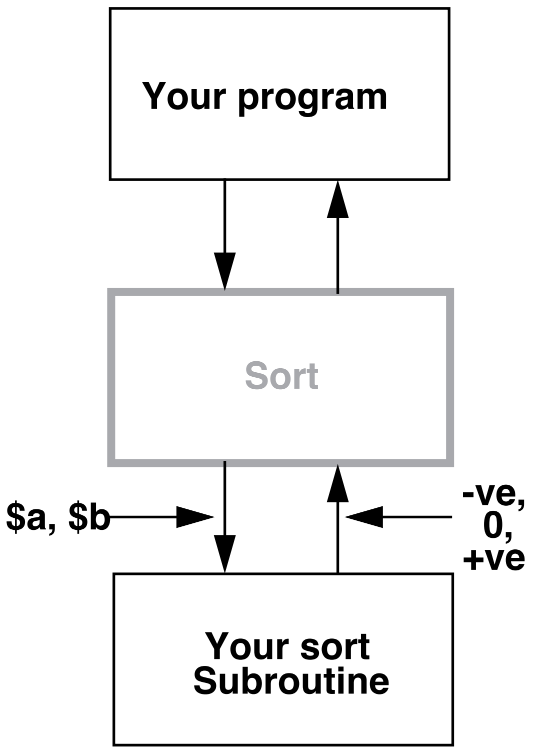
When you call the `sort` function that's built in to Perl, it sorts by making a large number of comparisons between two elements of the incoming list and deciding which comes first and which comes second. There's a huge amount of work involved in working out the order of comparison, but the actual comparisons can be very simple.

It's made easy in Perl this way:

- You call `sort`, passing it the list and also the name of a subroutine which compares two elements
- `sort` works out the order of comparison, provides management, returns information when completed and calls your subroutine

In order to interface correctly, your subroutine must:

- compare records `$a` and `$b`
- return a negative integer if the first is greater, 0 if they are the



same, and a positive integer if the second is greater.

Here's how we would sort by the length of the place name:

```
#!/usr/local/bin/perl
# so2 - sort using subroutine

%codes = (
 "Albury" => "0127974",
 "Aldbury_Common" => "0144285",
 "Aylesbury" => "01296",
 "Avebury" => "016723",
 "Abbotsbury" => "01305",
 "Almondsbury" => "01454",
 "Ashbury" => "0179371",
 "Amesbury" => "01980" );

foreach $ex (sort bylength keys %codes) {
 printf "%14s %s\n",$ex,$codes{$ex};
 }

sub bylength {
length($a) - length ($b);
}
```

```
seal% so2
 Albury   0127974
 Avebury   016723
 Ashbury  0179371
 Amesbury   01980
 Aylesbury  01296
 Abbotsbury 01305
 Almondsbury01454
 Aldbury_Common0144285
seal%
```

Figure 95

Running Perl program "so2".

Operators cmp and <=>

In that last example, we returned the difference between the lengths to give us our negative, zero or positive indicator. But what if we were comparing text strings? Our subroutine code would have to make two checks:

```
($a eq $b) ? 0 : (($a lt $b) ? -1 : 1);
```

Perl has two extra comparison operators to help:

`cmp`	compares two strings
`<=>`	compares two numbers

both return:

`-ve`	1st less
`0`	equal
`+ve`	1st greater

Used in normal code when you're looking for "true / false", they would function like the `ne` or `!=` operator respectively. But here in the `sort` subroutine they let you reduce

```
($a eq $b) ? 0 : (($a lt $b) ? -1 : 1);
```

to

```
($a cmp $b)
```

A more complex sort selector routine

In this example, we've sorted initially by the length of the value contained in the hash rather than by the key. And if the two values had the same length, we've sorted the key strings.

```
#!/usr/local/bin/perl
# so4 - sort using subroutine

%codes = (
 "Albury" => "0127974",
 "Aldbury_Common" => "0144285",
 "Aylesbury" => "01296",
 "Avebury" => "016723",
 "Abbotsbury" => "01305",
 "Almondsbury" => "01454",
 "Ashbury" => "0179371",
 "Amesbury" => "01980" );

foreach $ex (sort myway keys %codes) {
 printf "%14s %s\n",$ex,$codes{$ex};
 }

sub myway {
 # length of stdcode
(length ($codes{$a}) <=> length ($codes{$b}))
 # if the stdcode is the same length
||
 # by the exchange name
($a cmp $b);
}
```

```
seal% so4
 Abbotsbury01305
 Almondsbury01454
 Amesbury  01980
 Aylesbury 01296
 Avebury  016723
 Albury  0127974
 Aldbury_Common   0144285
 Ashbury 0179371
seal%
```

Figure 96
Running Perl program "so4".

Sorting with a comparison block

Instead of specifying a subroutine name as the first parameter after the word `sort`, you can also put a block of text in at that point, and that is the block that will be used; it's in effect an inline defined subroutine that has no name. We'll conclude our section on sorting with an example:

```
#!/usr/local/bin/perl
# so5 - sort using anonymous block

%codes = (
 "Albury" => "0127974",
 "Aldbury_Common" => "0144285",
 "Aylesbury" => "01296",
 "Avebury" => "016723",
 "Abbotsbury" => "01305",
 "Almondsbury" => "01454",
 "Ashbury" => "0179371",
 "Amesbury" => "01980" );

foreach $ex (sort {$codes{$b}<=>$codes{$a}} keys
  %codes) {
 printf "%14s %s\n",$ex,$codes{$ex};
 }
```

```
seal% so5
 Ashbury 0179371
 Aldbury_Common0144285
 Albury  0127974
 Avebury  016723
 Amesbury  01980
 Almondsbury01454
 Abbotsbury01305
 Aylesbury 01296
seal%
```

Figure 97
Running Perl program "so5".

14.5 Programming techniques

You're here to learn about Perl rather than programming techniques that apply to all programming languages. If you attended the Perl Basics day, you will have had an introduction to some of the most basic techniques, but if you skipped that day we assume that you have programmed before and are aware.

Hashes, though, are not necessarily familiar even to the most expert of programmers since they aren't necessarily available in all languages. So we'll take a brief step to one side and look at a couple of techniques that you might wish to apply, not only in Perl, but in other languages as well.

Non-unique keys

"The keys of a hash must be unique". Yes, and the operative word of that statement is "must".

Let's try to convert an std code file into a hash:

```
#!/usr/local/bin/perl
# stdhash - read stdcodes into a hash
# won't work - some keys are not unique

open (DIAL,"stdcodes") ||
die "No stdcodes file \n";

while (<DIAL>) {
 chop;
 ($dial,$place) = split(/\s+/,$_,2);
 $codes{$place} = $dial;
 $nread++;
 }
@places = keys %codes;

print $nread," lines read\n";
print @places+0," entries in hash\n";
print "Florida: ",$codes{"Florida"},"\n";
```

```
seal% stdhash
4898 lines read
4742 entries in hash
Florida: 001954
seal%
```

Figure 98

Running Perl program "stdhash".

What happened?

There were some lines in the incoming data file that repeated the keys from previous lines. The first entry for each such key in the hash correctly created a new element, but subsequent entries overwrote the first element.

If you might have duplicate keys being created, you MUST check every time. And you are then left with two options:

- merge both record into a single key-ed record OR
- create a new unique key

Let's see examples of both methods:

```
#!/usr/local/bin/perl
# stdhash2 - read stdcodes into a hash
# append info if element already exists

open (DIAL,"stdcodes") ||
die "No stdcodes file \n";

while (<DIAL>) {
 chop;
 ($dial,$place) = split(/\s+/,$_,2);
 if ($codes{$place}) {
 $codes{$place} .= ", $dial";
 } else {
 $codes{$place} = $dial;
 }
 $nread++;
 }
@places = keys %codes;

print $nread," lines read\n";
print @places+0," entries in hash\n";
print "Florida: ",$codes{"Florida"},"\n";
```

```
seal% stdhash2
4898 lines read
4742 entries in hash
Florida: 001305, 001352, 001407,
  001561, 001813, 001904, 001941,
  001954
seal%
```

Figure 99
Running Perl program "stdhash2".

In that example, there were still less entries in the hash than lines read, and the programmer, when he uses the contents of the hash later in his program, must be aware of the possibilities of multi-hit records like the one for Florida.

Let's now set up new, unique keys:

```
#!/usr/local/bin/perl
# stdhash3 - read stdcodes into a hash
# create unique keys if element already exists

open (DIAL,"stdcodes") || die "No stdcodes file
  \n";

while (<DIAL>) {
 chop;
 ($dial,$place) = split(/\s+/,$_,2);
 $place .= "-" while ($codes{$place}) ;
 $codes{$place} = $dial;
 $nread++;
 }
@places = keys %codes;

print $nread," lines read\n";
print @places+0," entries in hash\n";
print "Florida: ",$codes{"Florida"},"\n";
```

```
seal% stdhash3
4898 lines read
4898 entries in hash
Florida: 001305
seal%
```

Figure 100
Running Perl program "stdhash3".

And we now have a complete hash and a "nice" single code for Florida. But it's a pity only the one record was listed out! Of course, we could solve that by adding a loop to our `print` statement to keep adding "-" to the key and seeing if another record exists, and that's a good technique in some circumstances.

Looking for matching keys

There will be times you wish to select all records that match a regular expression rather than an explicit record. For example:

- All records with an ISBN of `/.-56592-...-./`
- Records for Florida, upper or lower case!
- Records for Florida, perhaps with - signs on the end

You can't directly match the key to a regular expression, but you could write:

```
   @plkeys = keys (%codes);
   @plmatches = grep(/Florida-*/,@plkeys);
```

and loop though any / all matches.

```
#!/usr/local/bin/perl
# stdhash4 - read stdcodes into a hash
# create unique keys if element already exists
# read back checks for all matching keys

open (DIAL,"stdcodes") ||
die "No stdcodes file \n";

while (<DIAL>) {
 chop;
 ($dial,$place) = split(/\s+/,$_,2);
 $place .= "-" while ($codes{$place}) ;
 $codes{$place} = $dial;
 $nread++;
 }
@places = keys %codes;

print $nread," lines read\n";
print @places+0," entries in hash\n";

@plkeys = keys (%codes);
@plmatches = grep(/^Florida-*$/,@plkeys);

foreach (@plmatches){
print "$_: ",$codes{"$_"},"\n";
}
```

```
seal% stdhash4
4898 lines read
4898 entries in hash
Florida------: 001941
Florida-: 001352
Florida---: 001561
Florida--: 001407
Florida: 001305
Florida-----: 001904
Florida----: 001813
Florida-------: 001954
seal%
```

Figure 101
Running Perl program "stdhash4".

Ah, back to `sort` to put them in a sensible order, I think!

Use hashes for stock numbers

When do you use hashes?

When you want to look up the line of a table and you don't care which line number it is, but rather what is in one particular field.

I'm handling telephone numbers, perhaps working for BT, and I want to enter some information about 813520. It's a number, so can I use

`$phone[813520]` Right?

No, I can't because Perl lists are not sparse. They're fully populated, which out of jargon, means that space would be allocated all the way from `$phone[0]` through `$phone[813519]` before `$phone[813520]` could be created!

You don`t want to look down a table with that number of lines for the 813520th entry, so don't use a list. Use a hash!

Deleting elements, clearing out hashes

To delete an element from a hash:

```
delete $phone{"813520"};
```

To clear out an entire hash:

```
undef %phone;
```

It's quite common to need to clear out a hash.

Scenario: You've been working on a set of phone numbers for one exchange, and you're going to move on and reuse the same hash for the next exchange. What happens if you don't clear the hash? The numbers from the old exchange appear on the new exchange too.

Initialising hashes

If you create an element in a hash, it takes a default value of null / zero.

Remember the "x" operator that replicated a string? You can create a whole series of elements in a hash using that same operator in a hash context:

```
#!/usr/local/bin/perl
# bankout - initial monopoly money

@players = qw(Tyler Lisa Graham Kimberly Chris);

@cash{@players} = (1200) x @players;

while (($player,$money) = each (%cash)) {
 print "$money  $player\n";
 }
```

```
seal% **bankout**
1200  Graham
1200  Tyler
1200  Chris
1200  Lisa
1200  Kimberly
seal%
```

Figure 102
Running Perl program "bankout".

The `x` operator can also be used to initialise a list. Here's a list of 1200s:

```
@hands = (1200) x 6;
```

14.6 Special hashes

We introduced you to the special list `@ARGV`, and there are others that you'll meet later. There are also special hashes.

%ENV

You can read your environment using the `%ENV` hash. The keys are the names of the environment variables, and the values are the content.

```
seal% envrep
CC              gcc
CLASSDIR        .
DISPLAY         :0.0
EDITOR          vi
GS_DEVICE       ljet3
GS_LIB          /usr/local/lib/ghostscript
HELPPATH        /usr/openwin/lib/locale:/usr/openwin/lib/help
HOME            /export/home/graham
HZ              100
LANG            C
LOGNAME         graham
MAIL            /var/mail/graham
MANPATH         /usr/local/share/man:/usr/openwin/share/man:/usr/s
MOZILLA_HOME    /extra/disc0.slice4/net4
NOSUNVIEW       0
OPENWINHOME     /usr/openwin
PATH            /usr/openwin/bin:/usr/bin:.:/usr/ccs/bin:/usr/sbin
PWD             /penguin/perl/profile/book
SHELL           /bin/csh
TERM            sun-cmd
TERMCAP         sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
TZ              GB
USER            graham
WINDOW_TERMIOS XFILESEARCHPATH   /usr/openwin/lib/locale/%L/%T/%N%S:/usr/openwin/li
XINITRC         /usr/openwin/lib/Xinitrc
seal%
```

Figure 103

Running Perl program "envrep".

```
#!/usr/local/bin/perl
# envrep - environment report

foreach $vn(sort keys %ENV) {
 printf "%-15.15s %-50.50s\n",
 $vn,$ENV{$vn};
 }
```

With the module

```
use Env;
```

you can import all the environment variables into Perl variables of the same name. Specifying a list of names in the `use` statement allows you to be selective. Thus:

```
seal% env2
Termcap: sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
HelpPath: /usr/openwin/lib/locale:/usr/openwin/lib/help
Openwinhome:
Openwinhome: /usr/openwin
seal%
```

Figure 104

Running Perl program "env2".

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

# env2 - use Env;

use Env qw(TERMCAP HELPPATH);

print "Termcap: $TERMCAP\n";

print "HelpPath: $HELPPATH\n";

print "Openwinhome: $OPENWINHOME\n";
print "Openwinhome: ",$ENV{"OPENWINHOME"},"\n";
```

14.7 Summary

Hashes are a little bit like lists except the elements they contain are not numbered but named.

You refer to the whole of a hash using a name like `%hname`, and individual elements using `$hname{"elname"}`.

You can set up the whole of a hash from a list; alternate elements being taken as keys and values. To aid readability, you may replace each comma by => in this situation.

To find out the name of each element in a hash, use the `keys` function, or use the `each` function to step through each key, value pair in turn. The functions return the elements in what appears to be a random order.

Any list (including keys) can be sorted using the `sort` function, which may be given the name of a subroutine that can place record `$a` in relation to record `$b`. This subroutine may be provided in an anonymous subroutine format.

Keys must be unique. Reusing a key will overwrite a record, and you must take great care to ensure this is what you mean to do in a program. Various other programming techniques will also need consideration if you're new to hashes.

You may reference your environment variables through the `%ENV` hash, although you may also

```
          use Env;
```

to give them all direct variable names in your program.

Exercise

Using a hash, count the number of hits in the access log file for each host computer.

Our example answer is web_count

Sample

```
graham@otter:~/profile/answers_pp> web_count
   catfish 5
    dogfish 22
       seal 313
    sardine 348
        dab 1
   pilchard 1
  localhost 1
    o_whelk 458
    sealion 324
etc
      trout 302
     walrus 310
    seaweed 4
graham@otter:~/profile/answers_pp>
```

Modify your example to list out the top ten hits first, then all other systems that have accessed us, just by name.

Our example answer is web_c2

Sample

```
graham@otter:~/profile/answers_pp> web_c2
Top hits:
      skate 1995
      whale 1044
    o_whelk 458
      perch 391
    sardine 348
    sealion 324
    flipper 322
   aviemore 318
       seal 313
     magnet 312
Also accessed by:
     walrus       trout     mussel     plaice        tuna
       clam       lecht    dogfish    dolphin     catfish
    seaweed     whiting        dab   pilchard   localhost
    manatee       whelk
graham@otter:~/profile/answers_pp>
```

15 More on Character Strings

You learnt something of character string handling quite a while ago. Since then you've learnt about a whole lot of other things. But there are still a number of very important string handling facilities of which you should be aware.

Let's start by summarising what we've covered to date.

15.1 Summary to date

Strings of any length can be held in a scalar variable, and that string can contain any character including non-printables.

Built-in functions can be used to manipulate strings, for example:

`length`	to find out the length of a string
`chop`	to remove the last character of a string
`index`	to find the first occurrence of one string in another
`index`	(different call) to find the next occurrence ...
`substr`	to extract part of a string
`substr`	(different call) to edit a string
`rindex`	find the last occurrence of one string in another
`lc`	convert string to lower case
`lcfirst`	convert 1st character of string to lower case
`uc`	convert string to upper case
`ucfirst`	convert 1st character of string to upper case
`sprintf`	format a string

String handling operators we have used include:

`x`	duplicate string on left number of times on right (e.g. "allo" x 3 is "alloalloallo")
`".."`	Convert string in brackets into a text string

Strings may be compared using string comparison operators:

`eq`	returns true if the strings are identical
`ne`	returns true if the strings are not identical
`lt`	returns true if the first string is lexically less than the second
`le`	returns true if the first string is lexically less than the second, or identical
`gt`	returns true if the first string is lexically greater than the second
`ge`	returns true if the first string is lexically greater than the second, or identical
`cmp`	returns -ve / 0 / +ve if the first string is greater than / equal to / less than

A string may be checked to see if it matches a pattern (regular expression) using:

`=~`	returns true if the string matches the regular expression
`!~`	returns true if the string does NOT match the regular expression

The regular expression is specified between `/` characters, and comprises of:

- characters

`A` to `Z`, `a` to `z`, `0` to `9`	and more literal characters
`\$ \^ \\`	and more literal special characters
`\n`	new line
`\t \f \e \a`	tab, form feed, escape, alarm
`\243`	octal 243 (pound sign `£`)
`\xa3`	hex a3 (also pound sign `£`)

- any one character from a group

`[abcdh-z]`	any one character from the brackets
`[^abcdh-z]`	any one character NOT from the brackets
`\s`	any white space character
`\S`	any character that's not white space
`\d`	any digit
`\D`	any non-digit
`\w`	any word character
`\W`	any non-word character
`.`	any character

- anchors

`^`	match at start of string
`$`	match at end of string

- counts

`?`	0 or 1 of the previous character
`*`	0 or more of the previous character
`+`	1 or more of the previous character

You also learnt in the section about the default pattern matching space -- `$_` -- that Perl matches regular expressions against the contents of that variable if you don't use an explicit =~ or `!~`.

15.2 Extracting information from a match

You can ask the question "Does this string match?" and get a yes/no answer.

This first example looks for two words that form a name. And it can confirm whether or not the format is correct:

```
#!/usr/local/bin/perl
# names - extract forename and surname

print "please enter your name ";
chop ($name = <STDIN>);

if ($name =~ /^\s*\S+\s+\S+\s*$/) {
 print "Forename, Surname sequence";
} else {
 print "no match";
}
print "\n";
```

```
seal% names
please enter your name asasd
no match
seal% names
please enter your name Graham
  Ellis
Forename, Surname sequence
seal%
```

Figure 105
Running Perl program "names".

Good. But what if I want to make use of the matching words?

Perl will gather and store groups for you if you use round brackets in a regular expression, so you can write:

```
   if ($name =~ /^\s*(\S+)\s+(\S+)\s*$/)
```

$1, $2, etc

The groups are stored in the special variables `$1`, `$2`, etc. So:

```
#!/usr/local/bin/perl
# n2 - extract forename and surname

print "please enter your name ";
chop ($name = <STDIN>);

if ($name =~ /^\s*(\S+)\s+(\S+)\s*$/) {
 print "Hi $1. Your Surname is $2.";
} else {
 print "no match";
}
print "\n";
```

```
seal% n2
please enter your name Graham
  Ellis
Hi Graham. Your Surname is Ellis.
seal%
```

Figure 106
Running Perl program "n2".

Assign to a list

So far you've used the =~ operator in a scalar context. If you use it in a list context, you'll be returned a list of all the matched groups:

```
#!/usr/local/bin/perl
# n3 - extract forename and surname

print "please enter your name ";
chop ($name = <STDIN>);

if (($first,$second) = ($name =~
 /^\s*(\S+)\s+(\S+)\s*$/)) {
 print "Hi $first. Your Surname is $second.";
} else {
 print "no match";
}
print "\n";
```

```
seal% n3
please enter your name Graham
  Ellis
Hi Graham. Your Surname is Ellis.
seal%
```

Figure 107
Running Perl program "n3".

$', $& and $`

Whenever you're matching a string, the incoming string can be thought of as being in three parts

- the part before the match
- the match itself
- the part after the match

Perl can make all three of those available to you in `$'` `$&` and `$``.

If you are matching large amounts of data in your application, you are warned that you may not want to use these variables as they can have a significant effect on performance. Used even once in some initial code and they'll be calculated for every regular expression match in your program!

```
#!/usr/local/bin/perl
# n4 - extract forename and surname

print "please enter your message ";
chop ($name = <STDIN>);

if (($first,$second) = ($name =~
 /([A-Z][a-z]*)\s+([A-Z][a-z]*)/)) {
 print "To $first. Surname is $second.\n";
 print "Prematch: $`\n";
 print "Match: $&\n";
 print "Postmatch: $'\n";
} else {
 print "no match\n";
}
```

```
seal% n4
please enter your message to Lisa
  Ellis from Graham Ellis
To Lisa. Surname is Ellis.
Prematch: to
Match: Lisa Ellis
Postmatch:  from Graham Ellis
seal%
```

Figure 108
Running Perl program "n4".

15.3 More about regular expressions

When we were just matching regular expressions to ask the question "is there a match?" it really didn't matter if there were two possible ways that the match could be made. But now it does. Look at that last example and you'll see that the match was made to "Lisa Ellis" rather than to "Graham Ellis".

Have a look at this match:

```
#!/usr/local/bin/perl
# html1 - extract an HTML tag

$page = "<TITLE>This is a Web page</TITLE>";

if ($page=~ /<(.*)>/) {
 print "Tag: $1\n";
 print "Prematch: $`\n";
 print "Match: $&\n";
 print "Postmatch: $'\n";
} else {
 print "no match\n";
}
```

```
seal% html1
Tag: TITLE>This is a Web page</
  TITLE
Prematch:
Match: <TITLE>This is a Web page</
  TITLE>
Postmatch:
seal%
```

Figure 109
Running Perl program "html1".

Hoping to be given a single tag back? Sorry ...
Perl matches

- left-most
- longest

unless instructed otherwise.

What else can I put in regular expressions?

`|`	or
`+?`	None-greedy counts
`*?`	
`??`	
`{2,}`	
`{2}`	
`{2,4}`	
`\b`	Anchor - word boundary
`\B`	Anchor - non-word-boundary
`\L ... \E`	
`\l`	
`\Q`	

```
#!/usr/local/bin/perl
# html2 - extract an HTML tag

$page = "<TITLE>This is a Web page</TITLE>";

if ($page=~ /<(.*?)>/) {
 print "Tag: $1\n";
 print "Prematch: $`\n";
 print "Match: $&\n";
 print "Postmatch: $'\n";
} else {
 print "no match\n";
}
```

```
seal% html2
Tag: TITLE
Prematch:
Match: <TITLE>
Postmatch: This is a Web page</
  TITLE>
seal%
```

Figure 110
Running Perl program "html2".

More brackets

The grouping brackets that we've been using to capture interesting parts of the regular expression can also be followed by a count. For example:

```
        (\w+\.)+
```

will match a series of word characters followed by a literal `.` one or more times. This is a useful facility.

To match an IP address we might write:

```
        (\d{1,3}\.){3}\d{1,3}
```

If we happen to be capturing (via `$1` and `$2`, or via a list) the interesting parts of a match, this can give us a problem. An unwanted part of the IP address will be captured and we'll have to use a list slice or a junk variable name to get rid of it. Better to use `(?: .... )` for the grouping. These alternative brackets don't capture and they're not the "package deal" of `( ... )`. Thus:

```
        (?:\d{1,3}\.){3}\d{1,3}
```

15.4 Match modifiers

Match Modifiers

`i`	ignore case
`x`	any space character is a comment
`g`	global match
`s`	single line mode
`m`	multiline mode
`o`	once only evaluation

How do I match ignoring case?

```
 [Hh][Ee][Ll]{2}[Oo]
```

No. There are modifiers that may be added AFTER the final delimiter:

`i`	ignore case
`x`	any space character is a comment

`x` requires noting. By default, spaces are literally matched. As a result, regular expressions tend to be long and complex strings. The `x` modifier means that spaces are taken as comments and comments are allowed.

```
  if ($instring =~
   m/^\s*(\d{4,4})(\d{2,2})(\d{2,2})\s*$/) {
   $day = $3; $year = $1; $month = $2; }
```

can become

```
if ($instring =~ m/^\s*# leading white space allowed
    (\d{4,4})# 4 digits - year
    (\d{2,2})# 2 digits - month
    (\d{2,2})# 2 digits - day
    \s*$/x) {# trailing while space allowed
    $day = $3; $year = $1; $month = $2; }
```

There are other modifiers which affect the operation of the match.

`g` "global" match
in a scalar context, return next match
in a list context, return all matches

```
#!/usr/local/bin/perl
# html3 - extract all HTML tags

$page = "<TITLE>This is a Web page</TITLE>";

while ($page=~ /<(.*?)>/g) {
 print "Tag: $1\n";
 print "Prematch: $`\n";
 print "Match: $&\n";
 print "Postmatch: $'\n";
}
```

```
seal% html3
Tag: TITLE
Prematch:
Match: <TITLE>
Postmatch: This is a Web page</
  TITLE>
Tag: /TITLE
Prematch: <TITLE>This is a Web
  page
Match: </TITLE>
Postmatch:
seal%
```

Figure 111
Running Perl program "html3".

```
#!/usr/local/bin/perl
# html4 - extract all HTML tags

$page = "<TITLE>This is a Web page</TITLE>";
@tags = ($page =~/<(.*?)>/g);

print $#tags+1," tags\n";
foreach (@tags){
 print;
 print "\n";
 }
```

```
seal% html4
2 tags
TITLE
/TITLE
seal%
```

Figure 112
Running Perl program "html4".

Global v Greedy

It's important to specify global matching when you want to get back more than one match from within a single string. It's also important to specify sparse (rather than defaulting to greedy) counts where necessary within the regular expression. The next example shows you three different matches to a piece of HTML.

```
$abcd = "<h1>This is a heading</h1>above <b>this</b> text";

@tag = ($abcd =~ /<(.*)>/);

print ("Whole string: $abcd\n");
print ("Greedy: ",join(" %% ",@tag),"\n");

@tag = ($abcd =~ /<(.*?)>/);
print ("Sparse: ",join(" %% ",@tag),"\n");

@tag = ($abcd =~ /<(.*?)>/g);
print ("Sparse and global: ",join(" %% ",@tag),"\n");
```

The first match will return a single result since it's not global. All the text from the first < character to the last > character will be the result as the match is greedy.

The second match will return a single result, again, since it's not global. It will result in the text from the first < to the subsequent > character since the match is not greedy.

The final match will return multiple results as it's global. The results will not overlap, because a global match resumes matching where the previous match left off. In this example, the match is sparse and the result is a list of matches, each of which is the text from a < to the subsequent >.

Here are the results when we run the program:

```
[localhost:~/jan03] graham% perl greedyvglobal.pl
Whole string: <h1>This is a heading</h1>above <b>this</b> text
Greedy: h1>This is a heading</h1>above <b>this</b
Sparse: h1
Sparse and global: h1 %% /h1 %% b %% /b
[localhost:~/jan03] graham%
```

`s`	single line mode `.` matches `\n` (otherwise it does not!)
`m`	multiline mode `^` and `$` match at embedded line starts and ends `\A` and `\Z` still match true string start and end only
`o`	once only evaluation (for efficiency; use this if the regular expression never changes within a single run)

15.5 Alternative delimiters

Remember that "Hello" could be written

```
qq!Hello!
```

In the same way,

```
/sdfsdfsdsdfsdf/
```

can be rewritten

```
m-sdfsdfsdsdfsdf-
```

Here's an example where we required `/` characters within our match

```
            if ($instring =~ m!^\s*(\d{1,2})[/\\\s](\d{1,2})[/\\\s](\d{2,4})\s*$!){
$day = $2; $month = $1; $year = $3 ; }
```

15.6 Some favourite regular expressions

We thought you might find these useful, they're all written using Perl-style regular expressions.

To match an email address

```
/(?:^|\s)[-a-z0-9_.]+@([-a-z0-9]+\.)+[a-z]{2,6}(?:\s|$)/i
```

To match a UK Postcode

```
/\b[a-z]{1,2}\d{1,2}[a-z]?\s*\d[a-z]{2}\b/i
```

To match an American Zip code

```
/\b\d{5}(?:[-\s]\d{4})?\b/
```

To match a date (UK Style)

```
m!\b[0123]?\d[-/\s\.](?:[01]\d|[a-z]{3,})[-/\s\.](?:\d{2})?\d{2}\b!i
```

To match a time

```
/\b\d{1,2}:\d{1,2}(?:\s*[aApP]\.?[mM]\.?)?\b/
```

To match a complete URL for a web page

```
m!https?://[-a-z0-9\.]{4,}(?::\d+)?/[^#?]+(?:#\S+)?!i
```

To match a Visa number

```
/\b4\d{3}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}\b/
```

To match a Mastercard number

```
/\b5[1-5]\d{2}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}\b/
```

To match a UK Phone number

```
/\b0[-\d\s]{10,}\b/
```

To match a UK car registration plate

```
/\b[A-Z]{2}(?:51|02|52)[A-Z]{3}\b/# current series
/\b[A-HJ-NP-Y]\d{1,3}[A-Z]{3}\b/# previous series
```

```
/\b[A-Z]{3}\d{1,3}[A-HJ-NP-Y]\b/# previous series
/\b(?:[A-Z]{1,2}\d{1,4}|[A-Z]{3}\d{1,3})\b/# old series - letters first
/\b(?:\d{1,4}[A-Z]{1,2}|\d{1,3}[A-Z]{3})\b/# old series - digits first
```

To match a UK national insurance number

```
/\b[a-z]{2}\s?\d{2}\s?\d{2}\s?\d{2}\s?[a-z]\b/i
```

To match a book's ISBN number

```
/\b(?:[\d]-?){9}[\dxX]\b/
```

Notes:

a) We've used non-capture groups throughout. You're free to use our expressions and add in extra brackets if you wish to extract parts of the match.

b) Some of these examples are fairly rudimentary tests, and it's up to you to check that they're suitable for use in your application

c) Many of these examples can only check that a string of text is in the correct format. There's no way that we can check if a particular car registartion was issued, unless of course we have access to the DVLC database.

Let's try those out:

```
$ ./favex.pl testdata
from "My car registration is J407LCM and my neighbour's in WN02PNG." we can get...
Car registration #1 WN02PNG
Car registration #2 J407LCM
from "There's book 1-56592-257-3 here beside me as I write." we can get...
US Zipcode 56592
ISBN Number 1-56592-257-3
from "My phone number is 01225 708225 and my fax is 01225 707126" we can get...
US Zipcode 01225
phone number 01225 708225
from "My email address is graham@wellho.net" we can get...
email address  graham@wellho.net
from "You can read about me at http://www.grahamellis.co.uk/index.html" we can get...
Complete URL http://www.grahamellis.co.uk/index.html
from "My national insurance number is YX 17 11 94 A" we can get...
UK date 17 11 94
UK N.I. number YX 17 11 94 A
from "My date of birth is 16 July 1954, and it is now 10/8/2002" we can get...
UK date 16 July 1954
from "The time is 21:55" we can get...
time 21:55
from "The time is 9:55 p.m. (if you still use that style)" we can get...
time 9:55 p.m
from "My postcode is SN12 6QL, and my Visa is NOT 4567 6552 5532 7761!" we can get...
UK Postcode SN12 6QL
Visa number 4567 6552 5532 7761
Car registration #4 SN12
Car registration #5 6QL
from "My Mastercard is not 5353 6789 0087 6534 and our car is not 404WHC." we get...
Mastercard number 5353 6789 0087 6534
Car registration #5 404WHC
from "90011-1774 would be a zip code from California" we can get...
US Zipcode 90011-1774
from "So would 90011 without a further 4 digits" we can get...
US Zipcode 90011
$
```

Here's the test data we used:

```
My car registration is J407LCM and my neighbour's in WN02PNG.
There's book 1-56592-257-3 here beside me as I write.
My phone number is 01225 708225 and my fax is 01225 707126
My email address is graham@wellho.net
You can read about me at http://www.grahamellis.co.uk/index.html
My national insurance number is YX 17 11 94 A
My date of birth is 16 July 1954, and it is now 10/8/2002
The time is 21:55
The time is 9:55 p.m. (if you still use that style)
My postcode is SN12 6QL, and my Visa is NOT 4567 6552 5532 7761!
My Mastercard is not 5353 6789 0087 6534 and our car is not 404WHC.
90011-1774 would be a zip code from California
So would 90011 without a further 4 digits
```

and here's the Perl program itself:

```
#!/usr/bin/perl -n
# Some favourite regular expressions
chop;
print "from \"$_\" we can get .... \n";
/(?:^|\s)[-a-z0-9_.]+@([-a-z0-9]+\.)+[a-z]{2,6}(?:\s|$)/i and
       print "email address $&\n";
/\b[a-z]{1,2}\d{1,2}[a-z]?\s*\d[a-z]{2}\b/i and print "UK Postcode $&\n";
/\b\d{5}(?:[-\s]\d{4})?\b/ and print "US Zipcode $&\n";
m!\b[0123]?\d[-/\s\.](?:[01]\d|[a-z]{3,})[-/\s\.](?:\d{2})?\d{2}\b!i and
       print "UK date $&\n";
/\b\d{1,2}:\d{1,2}(?:\s*[aApP]\.?[mM]\.?)?\b/ and print "time $&\n";
m!https?://[-a-z0-9\.]{4,}(?::\d+)?/[^#?]+(?:#\S+)?!i and print "Complete URL $&\n";
/\b4\d{3}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}\b/ and print "Visa number $&\n";
/\b5[1-5]\d{2}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}\b/ and
       print "Mastercard number $&\n";
/\b0[-\d\s]{10,}\b/ and print "phone number $&\n";
/\b[A-Z]{2}(?:51|02|52)[A-Z]{3}\b/ and print "Car registration #1 $&\n";
/\b[A-HJ-NP-Y]\d{1,3}[A-Z]{3}\b/ and print "Car registration #2 $&\n";
/\b[A-Z]{3}\d{1,3}[A-HJ-NP-Y]\b/ and print "Car registration #3 $&\n";
/\b(?:[A-Z]{1,2}\d{1,4}|[A-Z]{3}\d{1,3})\b/ and print "Car registration #4 $&\n";
/\b(?:\d{1,4}[A-Z]{1,2}|\d{1,3}[A-Z]{3})\b/ and print "Car registration #5 $&\n";
/\b[a-z]{2}\s?\d{2}\s?\d{2}\s?\d{2}\s?[a-z]\b/i and print "UK N.I. number $&\n";
/\b(?:[\d]-?){9}[\dxX]\b/ and print "ISBN Number $&\n";
```

Exercise

Write a subroutine as follows:	name	urlsplit
	input	a text string representing a URL
	output	a list of elements from the URL:
		0 protocol
		1 server name
		2 port no
		3 page name
		4 location on page

For example, if you passed the subroutine *http://www.wellho.net:80/course/index.html#june* your return list would contain:

```
0    http
1    www.wellho.net
2    80
3    /course/index.html
4    june
```

Allow for protocols `http`, `https` and `ftp`. Allow the user to omit the protocol from the input URL (they should default to 80, 443 and 21 for `http`, `https` and `ftp`). Allow the user to omit a location on the page

Also write a short test program to call your subroutine and ensure that it works correctly.

Our example answer is regextra

Sample

```
$ ./regextra
Please enter a URL: http://www.wellho.net/course/index.html
  /course/index.html :80 www.wellho.net http
$ ./regextra
Please enter a URL: https://cedar.he.net/internal.html#javabooks
 #javabooks /internal.html :443 cedar.he.net https
$
```

15.7 Substitutions

You've matched. You've matched and extracted the string or parts of it. Now you want to match and replace.

- Instead of the `m` operator, use the `s` operator
- Add a replacement expression

Here's an example:

```
#!/usr/local/bin/perl
# totext - convert & to &amp;  < to &lt;  > to &gt;

print "text: ";

$sample .= $_ while (<STDIN>);
$whole = $sample;

$sample =~ s/&/&amp;/;
$sample =~ s/</&lt;/;
$sample =~ s/>/&gt;/;

print "First changes: $sample";

$_=$whole;
s/&/&amp;/g;
s/</&lt;/g;
s/>/&gt;/g;
s/\n{2,}/<P>/g;
s/\n/<BR>\n/g;

print "Second changes: $_";
```

```
seal% totext
text: if ($a <=> $b && $c < $d) {
$j = ($k < $p;) }
First changes: if ($a &lt;=&gt; $b
  &amp;& $c < $d) {
$j = ($k < $p;) }
Second changes: if ($a &lt;=&gt;
  $b &amp;&amp; $c &lt; $d) {<BR>
$j = ($k &lt; $p;) }<BR>
seal%
```

Figure 113
Running Perl program "totext".

The first set of changes matches a regular expression and makes a single change. =~ is used so the change is made to the named variable.

With the second set of changes, no =~ is used so the change is made to `$_`. The `g` modifier causes multiple (global) changes to be made. Note that the `\n` in the output string is not re-matched. In other words, we don't end up with `<BR><BR><BR><BR><BR><BR>...` and so on.

Some common substitutions:

`s/^\s*//;`# remove white space off string start

`s/\s+/ /g;`# replace all blocks of white space withsingle space

`s/\s*$//;`# remove trailing white space

Further notes:

- Alternative delimiters may be used
- Count of changes made is returned
- Output string is NOT a regular expression.
- It may use `$1`, etc
- `\1` (for 1st match) etc
- Other special characters as if they were literals

```
#!/usr/local/bin/perl
# phone - internationalise a number

print "entry: ";
$_=<STDIN>;

$nmatch = s:\b0([1-9]):+44 (0) \1:g;

print "Count: $nmatch\n";
print;
```

```
seal% phone
entry: call 01380 813520
Count: 1
call +44 (0) 1380 813520
seal% phone
entry: call 01380 813520, or fax
  01380 818281
Count: 2
call +44 (0) 1380 813520, or fax
  +44 (0) 1380 818281
seal% phone
entry: to call from Germany, dial
  0044 1380 813520
Count:
to call from Germany, dial 0044
  1380 813520
seal%
```

Figure 114

Running Perl program "phone".

Substitute and execute

Consider:

```
#!/usr/local/bin/perl
# name - capitalise!

print "Message: ";
$_=<STDIN>;

/\b(graham)\b/i;
$proper = ucfirst(lc($1));
s/\bgraham\b/$proper/;

print;
```

```
seal% name
Message:  this is a message for
  graham to read
 this is a message for Graham to
  read
seal%
```

Figure 115

Running Perl program "name".

Can we write the following?

```
#!/usr/local/bin/perl
# name2 - capitalise!
#

print "Message: ";
$_=<STDIN>;

s/\b(graham)\b/ucfirst(lc($1))/;

print;
```

```
seal% name2
Message: This is a message for
  graham to read
This is a message for
  ucfirst(lc(graham)) to read
seal%
```

Figure 116
Running Perl program "name2".

Yes, but it doesn't do what we want!

The `e` modifier on the `s` operator says "execute the second part as a Perl expression":

```
#!/usr/local/bin/perl
# name3 - capitalise!

print "Message: ";
$_=<STDIN>;

s/\b(graham)\b/ucfirst(lc($1))/e;

print;
```

```
seal% name3
Message: This is a message for
  graham to read
This is a message for Graham to
  read
seal%
```

Figure 117
Running Perl program "name3".

Note that the `e` modifier, like the `eval` function, causes Perl to have to go back to the compiler. If this is called deep within a loop it can have a serious detrimental effect on performance.

Perl also has an `eval` function that lets you evaluate a string in a similar way. You're recommended to `eval` a whole loop once, rather than have an `eval` within a loop, if possible!

15.8 Regular expression efficiency

- If you use `$&` `` $` `` or `$'` anywhere in your program, they'll be saved at every regular expression match.
- Use the `o` modifier on the end of regular expressions if the pattern does not change over the life of the process.
- Two short regular expression matches are usually faster than one big one.
- Reject common short cases early.
- Try to avoid too many quantifiers and optional matches in regular expressions.
- Try to increase the length of non-optional pieces of text in regular expressions.
- If you are rematching the latest string, just specify `//`.
- If splitting, it's more efficient to use a fixed string rather than a pattern.
- Use `study` where appropriate.

15.9 tr

`tr` lets you translate all occurrences of one character to another. It does not use regular expressions, but it does use a similar-looking syntax:

```
#!/usr/local/bin/perl
# pwline - change : to ,

$line =
"berlioz:x:2002:2000::/trainee/berlioz:/bin/csh"
;

print $line,"\n";
$line =~ tr /:/,/;
print $line,"\n";
```

```
seal% pwline
berlioz:x:2002:2000::/trainee/
  berlioz:/bin/csh
berlioz,x,2002,2000,,/trainee/
  berlioz,/bin/csh
seal%
```

Figure 118
Running Perl program "pwline".

`tr` can be given multiple characters to change, and ranges of characters, using square brackets. And the letter "y" can be used in place of `tr`. An alternative delimiter may be used, as with `m`, `s`, `qq`, etc.

Modifiers:

`c`	compliment - change all characters that don't match
`s`	squeeze - compress resulting multiple characters to 1 character

```
#!/usr/local/bin/perl
# pwline2 - change : to ,

$line =
"berlioz:x:2002:2000::/trainee/berlioz:/bin/csh"
;
print $line,"\n";
$line =~ y/a-zA-Z/A-Za-z/;
print $line,"\n";
$line =~ tr!a-zA-Z0-9/!,!cs;
print $line,"\n";
```

```
seal% pwline2
berlioz:x:2002:2000::/trainee/
  berlioz:/bin/csh
BERLIOZ:X:2002:2000::/TRAINEE/
  BERLIOZ:/BIN/CSH
BERLIOZ,X,2002,2000,/TRAINEE/
  BERLIOZ,/BIN/CSH
seal%
```

Figure 119
Running Perl program "pwline2".

First change -- upper case to lower case, and vice versa
Second change -- all non-alphanumerics (and non-slashes) to commas, and squeeze out resultant multiple commas to single commas.

15.10 Handling binary text

On a number of operating systems, there are few utilities available to handle binary data. Perl plugs this gap!

C programmers will already be aware that the "null" character terminates a string in C. DOS programmers will know of the havoc a CTRL-Z can cause. But Perl has no such limitations. ANY one of the 256 possible combinations of 8 bits are allowed anywhere in a scalar.

There are a number of "bit by bit" operators:

`&`	bitwise AND operator
`|`	bitwise OR operator

^	bitwise XOR operator
~	bitwise NOT operator
<<	Left shift operator
>>	Right shift operator

and also functions such as `pack` and `unpack`.

Take, for instance, a GIF file. Let's say we have a number of files with names ending ".gif" and we want to check whether they really are GIF files, and if so, report on the image size. In ".gif" files, we have the following header:

```
GIFbytes 0 to 2 87a or 89abytes 3 to 5 X sizebytes 6 & 7 - byte swapped Y size   bytes 8 & 9 - byte swapped
```

Here is the sort of output we want:

```
seal% ystwyth.pl *.gif
======= beach.gif ========
GIF file - version 87a
195 pixels wide by 210 pixels high
======= fronds.gif ========
GIF file - version 87a
118 pixels wide by 90 pixels high
======= gorse.gif ========
This is NOT a valid GIF file
======= grass.gif ========
GIF file - version 87a
192 pixels wide by 157 pixels high
seal%
```

Figure 120
Running Perl program "ystwyth.pl".

```
#!/usr/local/bin/perl
# ystwyth.pl - unpack / binary data

foreach $fyle(@ARGV) {
 print ("======= $fyle ========\n");
 unless (open (GIFFILE,"$fyle")) {
 print ("Cannot open file\n");
 next;
 }
 read (GIFFILE,$header,10);
 ($gifword,$giflevel,$xlo,$xhi,$ylo,$yhi)=
 unpack("a3a3C4",$header);
 if ($gifword ne "GIF") {
 print ("This is NOT a valid GIF file \n");
 } else {
 print ("GIF file - version $giflevel\n");
 print ($xhi*256+$xlo," pixels wide by ",
 $yhi*256+$ylo," pixels high\n");
 }
}
```

How do we achieve this? We open each file in turn. We use `read` to read a specified number of bytes into a regular variable. We unpack the (binary) bytes in that variable, specifying an unpacking format. This puts the results into a list. We can then use the unpacked data in our list.

`unpack` takes what can be quite a complex template (but we'll start you with an easy one!) and gives the order and type of values. Each may be followed by a number to specify how many.

Common value types:

a	unstripped ASCII string
C	unsigned character value
s	signed short value
i	signed integer value
f	single precision floating point number

and you also have controls such as:

x	skip forward a byte
X	go back a byte
@	go to absolute byte position

Let's examine the heart of the code:

```
read (GIFFILE,$header,10);
($gifword,$giflevel,$xlo,$xhi,$ylo,$yhi)=
unpack("a3a3C4",$header);
```

10 bytes are read into `$header`, which is then split:

- 3 bytes into a string `$gifword`
- 3 bytes into a string `$giflevel`
- 1 byte (as an unsigned number) to `$xlo`
- 1 byte (as an unsigned number) to `$xhi`
- 1 byte (as an unsigned number) to `$ylo`
- 1 byte (as an unsigned number) to `$yhi`

Note that "a3" uses only one item from our list of variables, but "C4" uses four. Everything except "a" uses one list item per specifier.

It may sound rather obvious that the opposite of `unpack` is `pack`. Remember, for length-dependant / non-binary applications, functions like `split` and `join` and operators like `.` are an easier solution.

15.11 Summary

Regular expressions comprise:

- literal characters

`a` to `z` `A` to `Z` `0` to `9`	and some specials
`\$ \^ \& \* \.`	and others
`\n \r \t \e \a \f`	control codes
`\xa3` and `\243`	for hex and octal values

•any one character from a group

`.`	any character
`\S`	any non-white space character
`\s`	any white space character
`\w \W`	any word / non-word character
`\d \D`	any digit / non-digit character
`[a-k]`	any character from a to k
`[^JP-Z]`	any character except J or P to Z

•anchors

`^`	match at start of line
`$`	match at end of line
`\b \B`	match on / not on word boundary
`\A`	match at start of string
`\Z`	match at end of string

•counts

`? * +`	(0 or 1) (0 or more) (1 or more) greedy
`?? *? +?`	same counts (non-greedy)
`{2,4}`	2 to 4
`{2,}`	2 or more

•also

`|`	or
`()`	grouping
`(?: )`	group but don't capture

•modifiers

`i`	ignore case
`g`	global
`x`	spaces are comments
`s`	single line mode
`m`	multiline mode
`o`	once only evaluation

Unless you use the non-greedy counts, Perl will match the longest and left-most the first time you match. Subsequent matches to the same string will return the next match if you've specified global.

Matching groups are captured into variables `$1`, `$2`, etc. They can also be saved into a list and used in the form `\1 \2` etc in the substitute operator.

If you don't use `=~`, Perl assumes you're matching against `$_`. You can use the `s` operator to substitute rather than just match, in which case you must give an output string and a third delimiter. An `e` modifier on a substitute will cause the output to be executed as a piece of Perl, the result of which will be substituted.

`tr` can be used to perform a character-by-character translation.

Exercise

Our example program "dmatch" includes a subroutine to match dates against five different regular expressions, but only tests three of the formats.

a) Add examples to the program to test out the remaining two match patterns.

b) This example was originally written for American dates.

Convert it where appropriate to match dates in the English form (e.g. 24/07/99 instead of 07/24/99).

Sample

```
graham@otter:~/profile/answers_pp> dmatch
   99    7   24    1
 1999    7   24    3
   99    7   24    5
graham@otter:~/profile/answers_pp>
```

Find all the lines in the "access_log" file relating to "catfish" and report them in the format shown in the sample.

Our example answer is catshow

Sample

```
graham@otter:~/profile/answers_pp> catshow
1998 Aug 28 - /index.html
1998 Aug 28 - /index.html
1998 Aug 28 - /perl/index.html
1998 Dec 11 - index.html
1998 Dec 11 - /index.html
graham@otter:~/profile/answers_pp>
```

Extend that exercise to convert the month into a month number.

Our example answer is cats2

Sample

```
graham@otter:~/profile/answers_pp> cats2
1998 08 28 - /index.html
1998 08 28 - /index.html
1998 08 28 - /perl/index.html
1998 12 11 - index.html
1998 12 11 - /index.html
graham@otter:~/profile/answers_pp>
```

16 HTML – Quick Reminder

HTML (HyperText Markup Language) is the text formatting language that's used to describe the content of web pages. Whilst your browser can read and display a simple text file, most web page authors prefer to use HTML which lets them suggest some formatting, supply links to other pages, etc.

Although you may be familiar with products such as FrontPage to generate HTML, you'll need to know a little bit more about the basic structure when you come on to our CGI or Web Client modules; this section is a very brief review or introduction.

16.1 Tags

HTML directives are written in tags, between `<` and `>` signs. The first word after the `<` character is the type of tag and may be followed by a series of parameter `=` value pairs before the `>` sign.

Many tags mark the start of an area, and a matching tag with a `/` character before the type name marks the end of the area. Thus

```
<FONT COLOR=red>Some text in Red</FONT>
```

Tag and parameter names are not case sensitive, but some of the parameter values are.

16.2 Structure of a page

Web pages comprise a block of HTML, usually written between

`<HTML>` and `</HTML>`

tags.[1]

Within the HTML, the page is split into two parts.

- The Head (`<HEAD>` to `</HEAD>`) which contains information that does not refer to the main display of the browser; for example, details of the page author, information for search engines and probably a title that will be used for bookmarking and labelling the displayed window.

  That Title will be between `<TITLE>` and `</TITLE>` tags.
- The body (`<BODY>` to `</BODY>`) which contains the text to be displayed in the main browser window and other information about how it is to be displayed and what else (e.g. images) is to be displayed as well. Colours for background, text and links can be specified in the `<BODY>` tag.

16.3 Special Characters and new lines

If you include a `<` sign in your text, the browser will think that a tag is coming. In order to prevent this, you can specify

```
&lt;
```

instead. You must also use

[1] You can omit these tags and the browser will usually be OK about it.

```
        &amp;
```
if you want an ampersand in your text, and you can use a whole range of other specials too, such as
```
        &pound;
        &copy;
```
You'll be writing programs later on this course which may include any text in their output, and you'll need to filter your output to make these substitutions.

Within HTML, any (and multiple) white space characters are replaced by single spaces, and lines are re-folded as necessary. If you want to force a line break, you should use a
```
        <BR>
```
tag, and if you want to leave a gap to a new paragraph you should use
```
        <P>
```
Don't eliminate new line characters completely, though; a few `\n`s will be indispensable when you come to edit or read the text of your page.

16.4 Some common tags

If you're looking for a "quick and dirty" display of a table of text in a fixed width font, precede the text with `<PRE>` and end it with `</PRE>`. This is the preformatted tag and within the block, the compression of spaces is suppressed. It's a very useful way of keeping data in columns, but you still need to filter `<` and `&` characters!

An `<HR>` tag gives you a horizontal ruling.

An `<H1> </H1>` pair lets you put in a headline size 1 (largest); you can also use `<H2>` through `<H6>` in a similar way. `<EM>` and `</EM>` can be used to mark a part of your text as emphasised. There are various other things one can do. See the "FONT" example above as a sample.

Anything written between `<CENTER>` and `</CENTER>` tags will be entered rather than left-justified in your browser's window.

If you want to specify a user-selectable link to another page, use an anchor tag, for example:

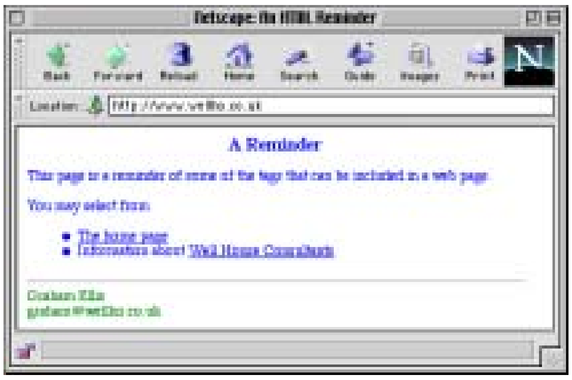
Go to `<A HREF=previous.html>`last newsletter`</A>`.

The `HREF` parameter can be any URL definition you choose – an html file in the current directory in this example – but it could call another directory, another site, or even refer to a different communication protocol:

Download `<A HREF=ftp://ftp.wellho.co.uk/pub/diary.txt>`calender`</A>`

Here's a page with some of these tags in use:

```
<HTML>
<HEAD>
<TITLE>An HTML Reminder</TITLE>
</HEAD>
<BODY BGCOLOR=WHITE TEXT=BLUE>
</BODY><CENTER>
<H3>A Reminder</H3>
</CENTER>
This page is a reminder of some of the tags that can be included
in a web page.
<P>
You may select from
<UL>
<LI><A HREF=index.html>The home page</A>
<LI>Information about
<A HREF=http://www.wellho.net>
Well House Consultants</A>
</UL>
<HR>
<FONT COLOR=GREEN>Graham Ellis<BR>
graham@wellho.net</FONT>
</HTML>
```



http://www.wellho.co.uk

A Reminder

This page is a reminder of some of the tags that can be included in a web page.

You may select from:

- The home page
- Information about Well House Consultants

Graham Ellis
graham@wellho.co.uk

Figure 121
HTML file as seen through Netscape

16.5 Lists, Tables, Frames, Forms, Images etc

Within a page of HTML, you can specify an ordered list (<OL>) or an unordered list (<UL>), ending with </OL> and </UL> respectively. Individual list items should be prefixed with <LI>.

To give better control over formatting than a list, you may want to arrange your data into a table. Tables are enclosed in <TABLE> and </TABLE> tags; within tables, rows are enclosed in <TR> and </TR> tags, and within each row, data elements are enclosed in <TD> and </TD> tags. Many parameters can be specified to tables to control their looks.

It's possible to divide the browser window into a series of frames. In this case, you'll have multiple HTML documents. The window as a whole will be defined in one document (a FRAMESET), with each of the resulting frames being defined in its own piece of HTML. Documents called up can then replace all the frames, or one frame, or even open a new browser window. There's nothing to stop you using frames with the other facilities you'll learn about on this course, but in order to keep the learning focused we won't be using them during this course.

Forms are a vital part of this course. They define a set of user-enterable boxes between a <FORM> and </FORM> tag; within the <FORM> tag an ACTION parameter gives the URL to be called when the user indicates that the form has been completed. Don't worry if you're not familiar with forms at the moment; we'll be coming back and studying the elements in more depth later.

Executable content and graphic images (in the form of Java applets, and GIF and JPG images) can also be included on your web page. They use the `<APPLET>` and `</APPLET>`, or the `<IMG>` tag respectively. In each case, the tag requires you to give a parameter specifying which applet is to be run or which image is to be displayed, and if that applet or image isn't presently loaded by the browser it will request it of the host.

16.6 Which HTML standard?

Netscape and Microsoft have been locked in a browser war for a considerable period, with both companies adding their own enhancements to the HTML language at each release.

There are other browsers around such as Lynx and HotJava too. And many users are quite contented to use an older browser that does not support the very latest bells and whistles.

Users can elect to turn Java off, change the font style and background and resize their windows at will, and can even choose not to show graphics.

Which, when all put together, means that any tags you use within a web page can be considered to be just hints to the browser which may or may not be acted on!

You should be very careful to consider your target audience for HTML, whether you're writing the HTML directly, having it generated by a software package such as FrameMaker or FrontPage, or by a Perl program.

Suggestions:

- Stick to the HTML 3.2 standard, old though it is, for vital pages
- Do not use Netscape or Microsoft extensions
- Do not rely on images to convey vital content
- If you must use Frames, also provide an alternative
- If you must use Applets, also provide an alternative
- Check your pages on Linux and Windows platforms
- Check your pages on a number of different browsers
- See how your page looks when scaled right up or down

And don't assume that your users will have some special plugin, or a high resolution screen, or a fast line to the internet just because YOU do ;-)

In the majority of applications, users will be looking for professionally presented content and not magic bells and whistles. Even restricting yourself as described above, you should have plenty of scope to impress. Of course, if you're going to employ someone with the skills to present an effective web site, they'll need Graphic Design, Library Science, Journalism, Time and motion, marketing and computer science skills as well as a thorough understanding of the business being represented on the site. Subject for a different course!

17 Perl on the Web

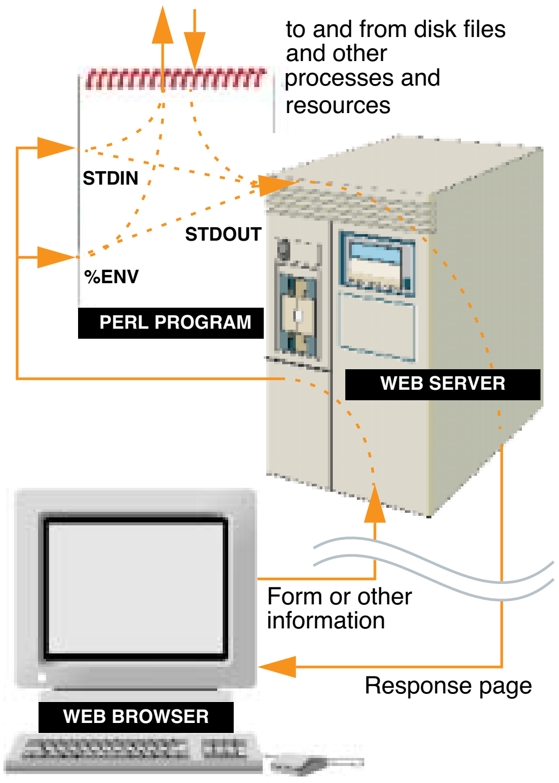

to and from disk files and other processes and resources
STDIN
STDOUT
%ENV
PERL PROGRAM
WEB SERVER
Form or other information
Response page
WEB BROWSER

Figure 122

How a Perl program interacts with the web

The example Perl programs that you have written so far read from `STDIN` and write to `STDOUT`. `STDIN` is often the keyboard and `STDOUT` is often a window. Those Perl programs also make use of environment variables.

You still use `STDIN`, environment variables and `STDOUT` when you write Perl programs that run on web servers. It's just that the environment and `STDIN` contain information about your user and what he's written in a form; `STDOUT` gets sent back to the user.

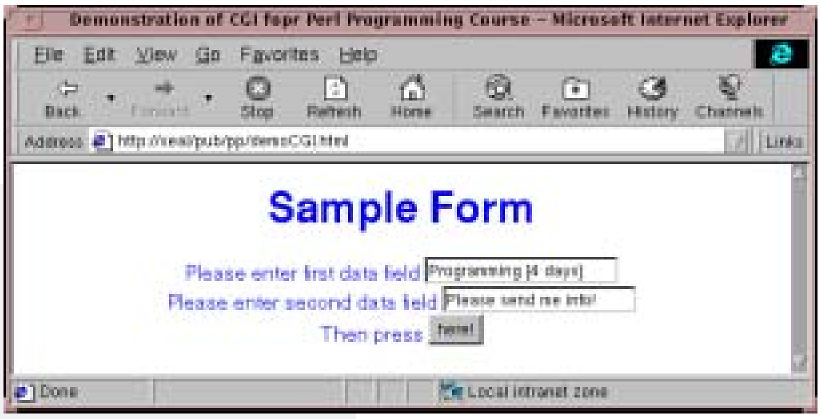
The data formats for input and output differ from each other and are amongst the topics we will cover on the Perl Advanced - Network Application course.

The interface between the web and your application (written in Perl on this course, though other languages can be used) is known as the Common Gateway Interface (or CGI).

17.1 The HTML form

Although you could call up a program without filling in a form, most applications do require the user input that a form provides.

The illustration below shows a sample form:


Demonstration of CGI fopr Perl Programming Course – Microsoft Internet Explorer

Address: http://seal/pub/pp/demoCGI.html

Sample Form

Please enter first data field Programming [4 days]

Please enter second data field Please send me info!

Then press here!

Figure 123

A sample HTML form, **right**. *The HTML code,* **below**.

```
<HTML>
<HEAD>
<TITLE>
Demonstration of CGI
for Perl Programming Course
</TITLE>
</HEAD>
<BODY BGCOLOR=white text=blue link=green vlink=red>
<CENTER><H1>Sample Form</H1>
<FORM method=get action="http:/cgi-bin/pub/pp/demo.pl">
Please enter first data field <INPUT name=cat><BR>
Please enter second data field <INPUT name=dog><BR>
Then press <INPUT TYPE=submit Value="here!">
</FORM>
</HTML>
```

If you want to study HTML, our Web Presence course covers that topic. For this course, you need to note:

- The `action` URL which says where the Perl program is.
- The various field names --- we used "cat" and "dog".
- The `method` -- we used `Get` on this course.

17.2 Inputs

When you complete our form (the URL on this course is:
http://seal/pub/pp/demoCGI.html)
and press the submit button, the contents of the form and the URL to call up, are uploaded to the server.

In this instance, this URL is
http://seal/cgi-bin/pub/pp/demo.pl
at which point we have placed a Perl program.

Using the `GET` method, the fields from the form arrive in the format

```
var=value&var=value&.......
```

in the variable

```
$ENV{"QUERY_STRING"}
```

and they can be extracted (in this example) using:

```
($cat,$dog) = /^cat=(.*)&dog=(.*)$/;
```

URL encoding

What if the user enters an `=` sign, or an `&` into the form?

Of course, it can't be uploaded as that character. After all, how would our Perl program tell field ends and data apart? Therefore

- special characters become `%xx` (where xx is a hex code)
- space characters become `+` characters

The following code takes (as an example) the first field you entered into the form and removes the URL encoding:

```
$catcode=$cat;
$catcode=~ tr/+/ /;
$catcode=~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
```

17.3 Outputs

You have now read form input into your CGI program, which is running on the web server. It's a Perl program, therefore you can do anything you like (subject to the permissions given by the administrator, etc.) on that system! How do you get results back to the user? You simply `print` (or `printf`) to `STDOUT` ...

Headers

When outputting through the CGI interface, you must generate a header as well as the HTML page, and the format of the header must be correct.

If you make an error in the header, or if the Perl program has a syntax error, your user will get a server error message:

"The Server encountered an internal error or
misconfiguration, and was unable to complete
your request ....."

If your reply page is in HTML, send

```
Content-type: text/html
```

followed by two new-line characters, since a blank line is required to finish the header.

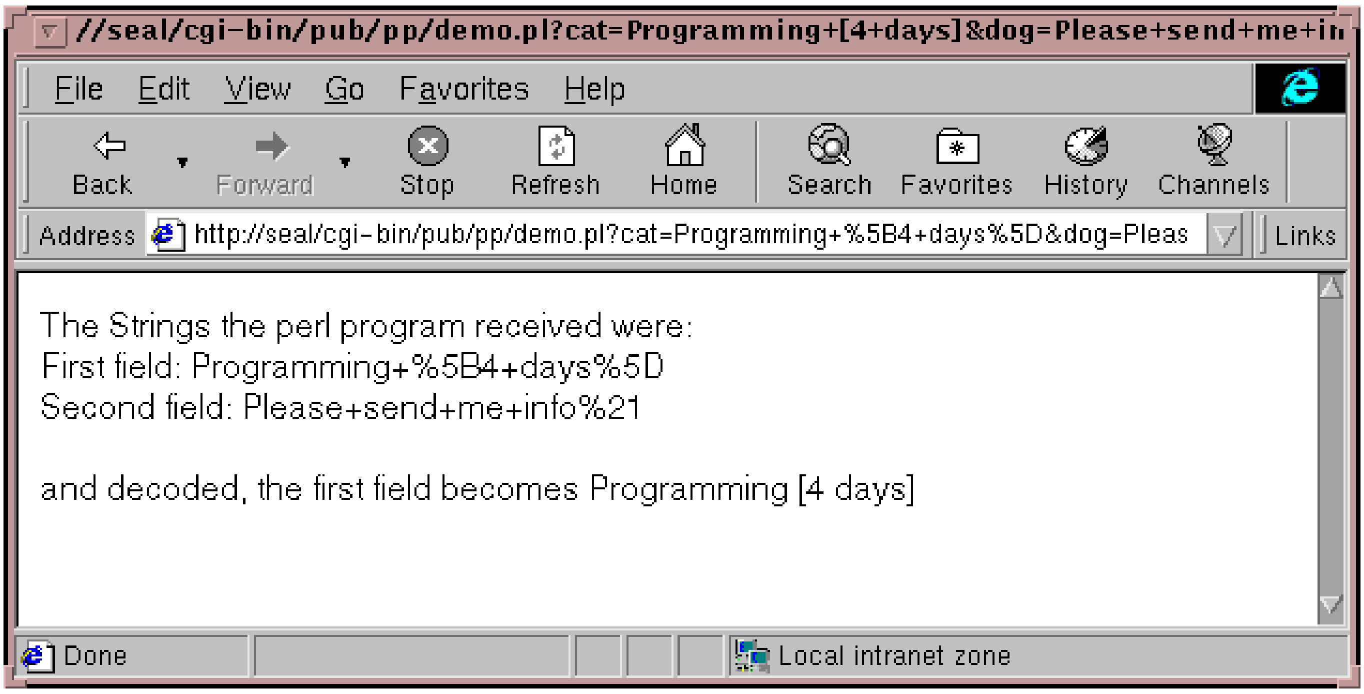
The reply page

You may then send the text of your page.

You should send a complete page, from <`HTML`> to <`/HTML`>.

17.4 All together!

Here's the complete Perl program and the outcome viewed on a browser that uses the CGI that we've quoted from above:


//seal/cgi-bin/pub/pp/demo.pl?cat=Programming+[4+days]&dog=Please+send+me+in
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels
Address http://seal/cgi-bin/pub/pp/demo.pl?cat=Programming+%5B4+days%5D&dog=Pleas
Links
Done
Local intranet zone

```
The Strings the perl program received were:
First field: Programming+%5B4+days%5D
Second field: Please+send+me+info%21

and decoded, the first field becomes Programming [4 days]
```

Figure 124
The HTML displayed

```
#!/usr/local/bin/perl
# Reads a form filled in by the user ...
# turns it into a web page which is then
  displayed!

# get the fields

$_ = $ENV{"QUERY_STRING"};
($cat,$dog) = /^cat=(.*)&dog=(.*)$/;

# decode one of them to show special char
  handling!

$catcode=$cat;
$catcode=~ tr/+/ /;
$catcode=~ s/%([a-fA-F0-9][a-fA-F0-9])/
  pack("C",hex($1))/eg;

# send a reply back!

print <<"PAGE" ;
Content-type: text/html

<HTML><BODY BGCOLOR=white text=black>
The Strings the perl program received were:<BR>
First field: $cat<BR>Second field: $dog<P>
and decoded, the first field becomes $catcode<P>
</BODY></HTML>
PAGE
```

17.5 The power of using Perl on the Web

The Common Gateway Interface that we've introduced in this module provides a wrapper around Perl that allows user inputs from a form and outputs to a browser. It's not a part of the Perl language itself, but rather it's an application of Perl. There's a wide variety of uses that Perl can be put to on the web through CGI and other uses. For example:

- Other elements available on a form
- Linking a series of forms into a complete application
- Telling who's using your CGI program
- Sending an email from your CGI program
- Modules available to help you
- Uploading and downloading files
- Providing a search engine for a small site
- Looking up information from other systems
- Handling images through CGI
- Using your browser as an interface to already-written applications
- Using the web to submit long tasks and reap the results later
- Crawlers and Spiders
- Sending a series of reply pages
- Sending a dynamic / changing page
- Reading back from a Java applet
- Sending data from CGI to a Java applet
- Security aspects
- How to debug and maintain web sites using Perl

17.6 A real example of Perl on the Web

Here's a real example of Perl in use on a web site. In this example, the application is to filter out and search for matching data in a very large file so that the user of the web site is only presented with the information he requires. The data file is in fact a web server's access log file. We've used the Perl script to filter records from six days of accesses totalling 86,000 records.

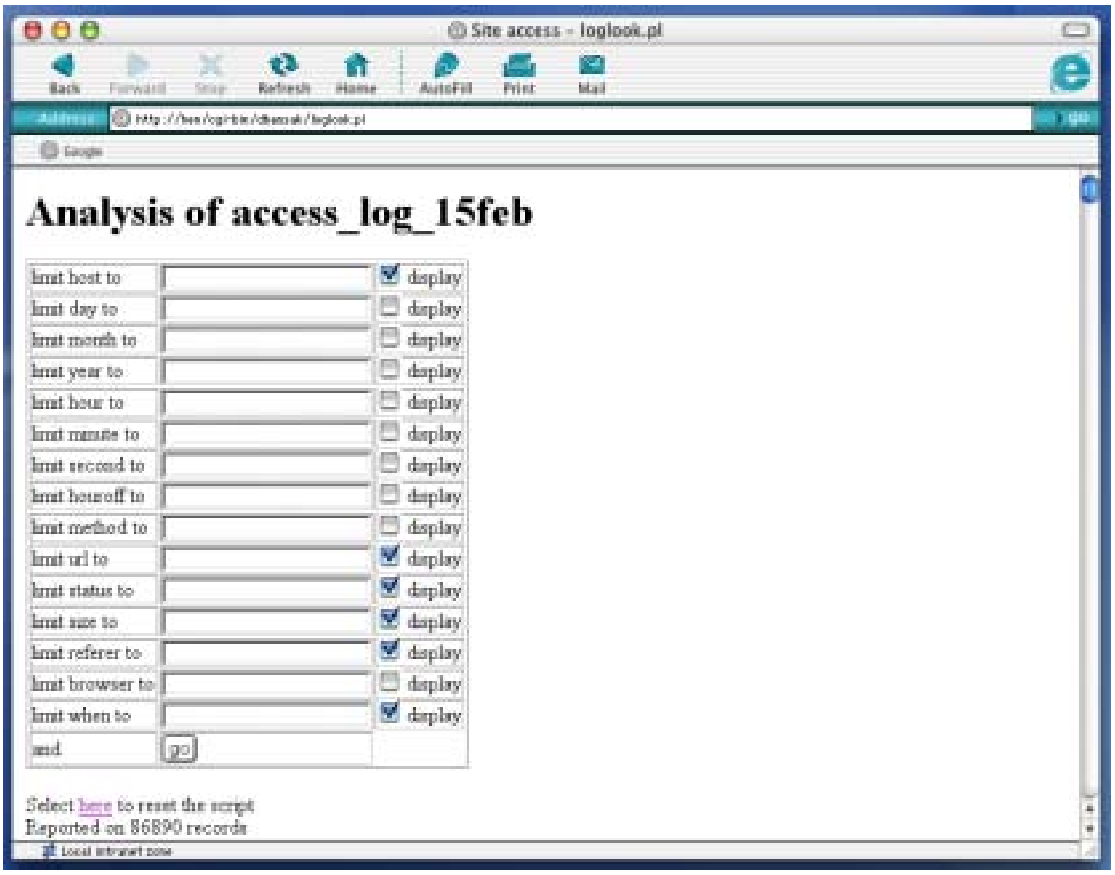
The example contains some meaty regular expression handling to handle the incoming record format, and it keeps its reply page in a separate template file (*nice.htp*) so that the program and the web page can be maintained by different staff members with different skills. The program simply grabs the *.htp* file and substitutes all strings `%\w+%` with a result from `%fill` in the Perl program.

After you start the program, it analyses the data file and offers:

a) A form in which you can select the records and fields to display if you wish

b) A report of all the domain names whose accesses are recorded in the log file, with links to allow the user to select specific hosts

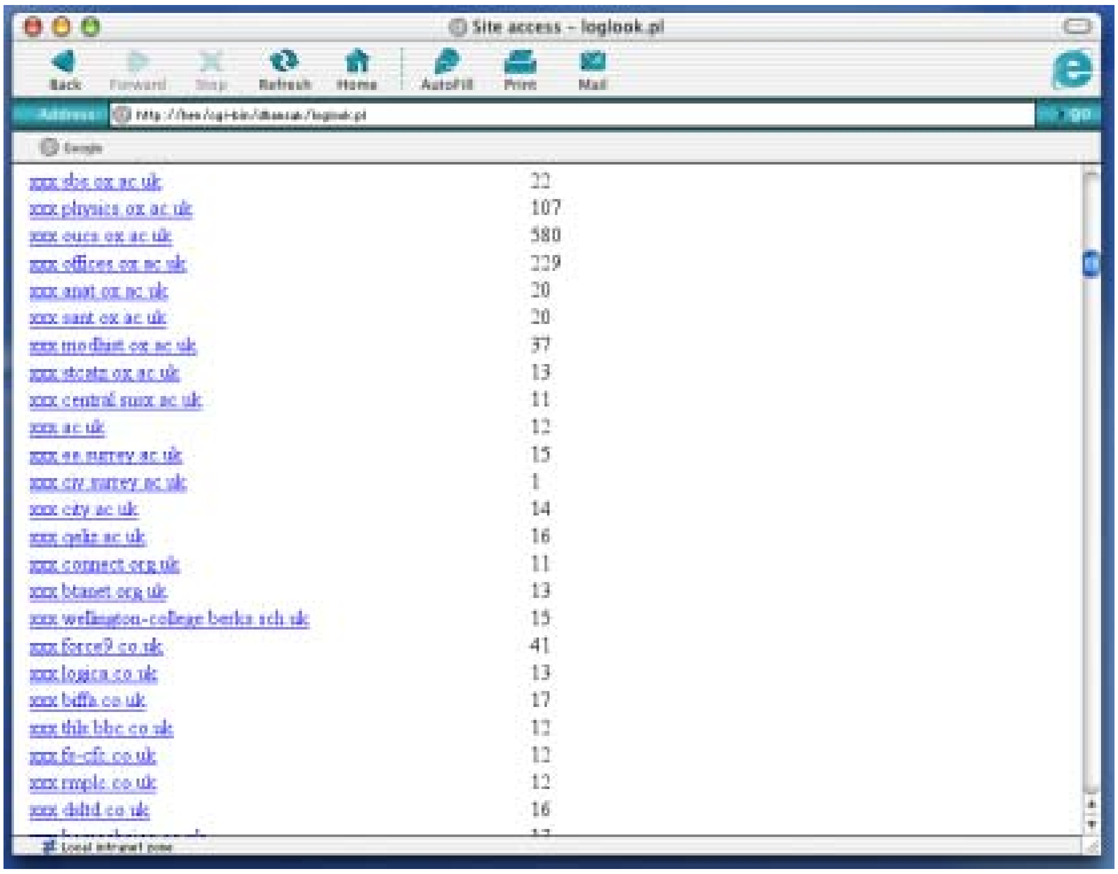
This is quite a long initial page; here are only parts of it:



Analysis of access_log_15feb

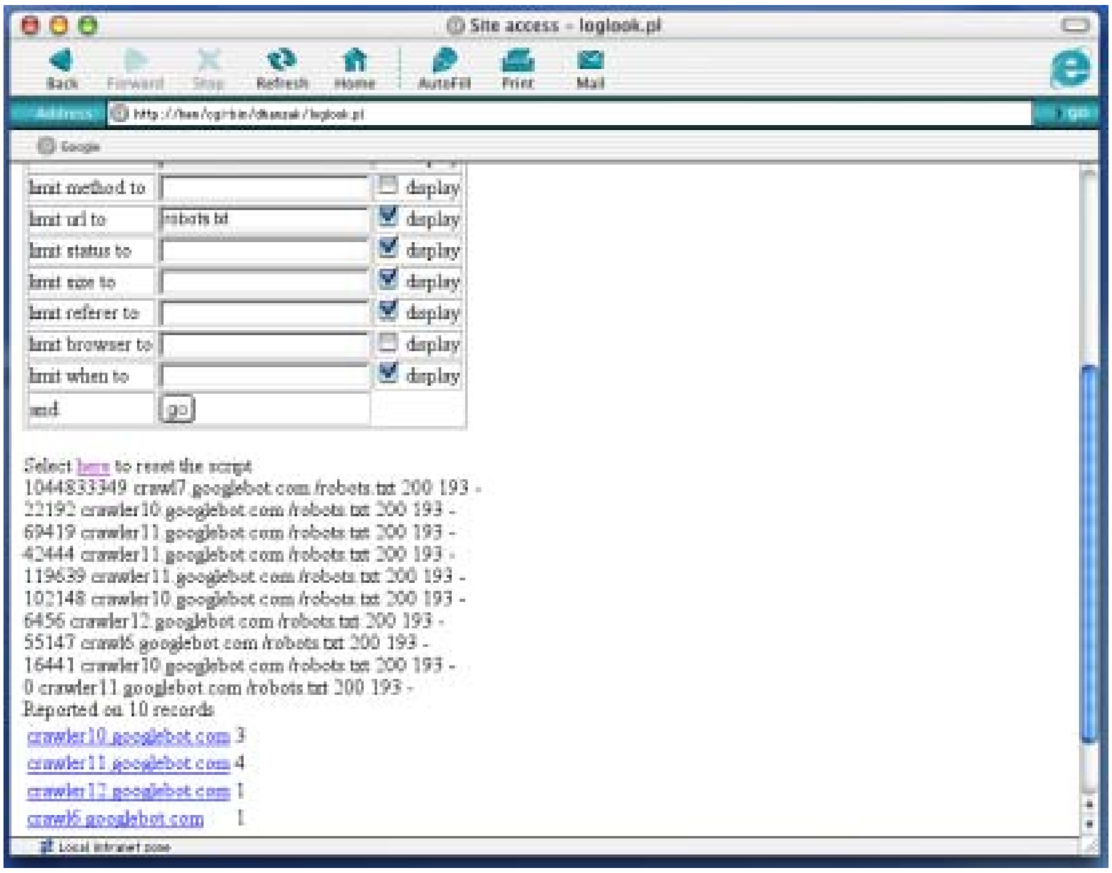
limit host to		[✓] display
limit day to		[ ] display
limit month to		[ ] display
limit year to		[ ] display
limit hour to		[ ] display
limit minute to		[ ] display
limit second to		[ ] display
limit houroff to		[ ] display
limit method to		[ ] display
limit url to		[✓] display
limit status to		[✓] display
limit size to		[✓] display
limit referer to		[✓] display
limit browser to		[ ] display
limit when to		[✓] display
and	go	

Select here to reset the script
Reported on 86890 records



xxx.dbs.ox.ac.uk	22
xxx.physics.ox.ac.uk	107
xxx.oucs.ox.ac.uk	580
xxx.offices.ox.ac.uk	229
xxx.anat.ox.ac.uk	20
xxx.sant.ox.ac.uk	20
xxx.modhist.ox.ac.uk	37
xxx.stcatz.ox.ac.uk	13
xxx.central.susx.ac.uk	11
xxx.ac.uk	12
xxx.ee.surrey.ac.uk	15
xxx.civ.surrey.ac.uk	1
xxx.city.ac.uk	14
xxx.qmkc.ac.uk	16
xxx.connect.org.uk	11
xxx.btanet.org.uk	13
xxx.wellington-college.berks.sch.uk	15
xxx.force9.co.uk	41
xxx.logica.co.uk	13
xxx.biffa.co.uk	17
xxx.thk.bbc.co.uk	12
xxx.fr-cfi.co.uk	12
xxx.rmplc.co.uk	12
xxx.dsltd.co.uk	16

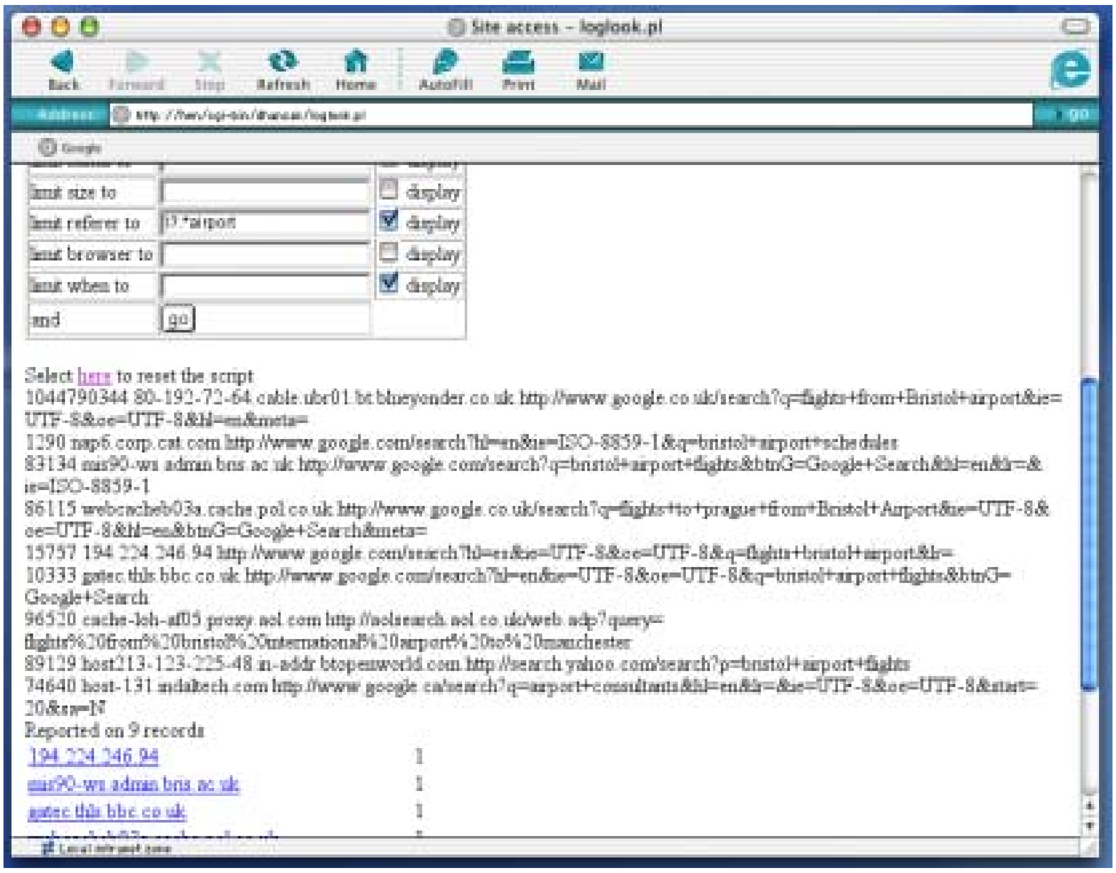
Once a form has been completed, the user will be presented with requested results and another form (already pre-filled with what he has just chosen so that he can refine the search).



limit method to		☐ display
limit url to	robots.txt	✓ display
limit status to		✓ display
limit size to		✓ display
limit referer to		✓ display
limit browser to		☐ display
limit when to		✓ display
and	go	

```
Select here to reset the script
1044833349 crawl7.googlebot.com /robots.txt 200 193 -
22192 crawler10.googlebot.com /robots.txt 200 193 -
69419 crawler11.googlebot.com /robots.txt 200 193 -
42444 crawler11.googlebot.com /robots.txt 200 193 -
119639 crawler11.googlebot.com /robots.txt 200 193 -
102148 crawler10.googlebot.com /robots.txt 200 193 -
6456 crawler12.googlebot.com /robots.txt 200 193 -
55147 crawl6.googlebot.com /robots.txt 200 193 -
16441 crawler10.googlebot.com /robots.txt 200 193 -
0 crawler11.googlebot.com /robots.txt 200 193 -
Reported on 10 records
crawler10.googlebot.com 3
crawler11.googlebot.com 4
crawler12.googlebot.com 1
crawl6.googlebot.com    1
```

And if the user selects a link to choose all the records from a particular host, then those records are indeed presented in full.



limit size to		☐ display
limit referer to	17.*airport	✓ display
limit browser to		☐ display
limit when to		✓ display
and	go	

```
Select here to reset the script
1044790344 80-192-72-64.cable.ubr01.bt.blueyonder.co.uk http://www.google.co.uk/search?q=flights+from+Bristol+airport&ie=
UTF-8&oe=UTF-8&hl=en&meta=
1290 nap6.corp.cat.com http://www.google.com/search?hl=en&ie=ISO-8859-1&q=bristol+airport+schedules
83134 mis90-ws.admin.bris.ac.uk http://www.google.com/search?q=bristol+airport+flights&btnG=Google+Search&hl=en&lr=&
ie=ISO-8859-1
86115 webcacheb03a.cache.pol.co.uk http://www.google.co.uk/search?q=flights+to+prague+from+Bristol+Airport&ie=UTF-8&
oe=UTF-8&hl=en&btnG=Google+Search&meta=
15757 194.224.246.94 http://www.google.com/search?hl=en&ie=UTF-8&oe=UTF-8&q=flights+bristol+airport&lr=
10333 gatec.thls.bbc.co.uk http://www.google.com/search?hl=en&ie=UTF-8&oe=UTF-8&q=bristol+airport+flights&btnG=
Google+Search
96520 cache-loh-af05.proxy.aol.com http://aolsearch.aol.co.uk/web.adp?query=
flights%20from%20bristol%20international%20airport%20to%20manchester
89129 host213-123-225-48.in-addr.btopenworld.com http://search.yahoo.com/search?p=bristol+airport+flights
74640 host-131.indatech.com http://www.google.ca/search?q=airport+consultants&hl=en&lr=&ie=UTF-8&oe=UTF-8&start=
20&sa=N
Reported on 9 records
194.224.246.94              1
mis90-ws.admin.bris.ac.uk   1
gatec.thls.bbc.co.uk        1
```

Here's the Perl program:

```
#!/usr/bin/perl

use Time::Local;

# Version 0.2 - Analysis of log files

initvars();
%form = collect_form();
$limitrecs = 500; $nrec = 0;

open (FH,$source);

# Search through all records in the data file that's being analysed

while ($line = <FH>) {
        %info = get_ncsaparts($line);
        next unless ($info{status}) ;

# If a selection form has been completed, look for records we need

        if ($form{select} or $form{host}) {
                $selected = 1;
                foreach $field (@names) {
                        $form{$field} and
                                $info{$field} !~ /$form{$field}/i and
                                        $selected = 0;
                        }
                if ($selected) {
                        $nrec++;
                        if ($nrec <= $limitrecs) {
                                my $thisline;
                                $nexttime = $info{when};
                                $info{when} = $info{when} - $previoustime;
                                $previoustime = $nexttime;
                                foreach $field("when",@names) {
                                        ($form{"_$field"} or ! $form{select})
                                                and $thisline .= $info{$field}." ";
                                        }
                                $stuff .= "$thisline<br>";
                        }
                        $visitors{$info{host}}++;
                }

# If no selection form has been completed, summarise all hosts

        } else {
                $summaryhost = $info{host};
                if ($summaryhost =~ /[a-z]$/) {
                        $summaryhost =~ s/^[^.]+/xxx/;
                } else {
                        $summaryhost =~ s/\.\d+$/.xxx/;
                }
                $nrec++;
                $visitors{$summaryhost}++;
                }
        }

# All records read.  Report on the number of matches and any truncation

$stuff .= ($nrec>=$limitrecs and ($form{select} or $form{host})) ?
        "REPORT TRUNCATED TO $limitrecs out of $nrec<br>":
        "Reported on $nrec records<br>";

# Set fields for display on next form if necessary

unless ($form{select}) {
        foreach $field(@show) {
                $form{"_$field"} = 1;
        }
}

($me) = ($0 =~ m!.*/(.*)!);
$fill{myname} = $me;
$fill{source} = $source;

# Make up a table of all hosts in case user wants to select by host

$stuff .= "<table>";
foreach $v (sort {reverse($a) cmp reverse($b)} keys %visitors) {
        $v1 = $v;
        $v1 =~ s/xxx//;
        $stuff .= "<tr><td><a href=$me?host=$v1>$v</a></td>".
                "<td>$visitors{$v}</td></tr>";
        }
$fill{stuff} = "$stuff</table>";

# Make up form through which to offer the user next options

foreach $field (@names,"when") {
        $checked = $form{"_$field"} ? " CHECKED" : "";
        $fill{formbody} .= "<tr><td>limit $field to</td><td>".
                        "<input name=$field value=\"$form{$field}\"></td>".
                       "<td><input name=_$field type=checkbox$checked> display</td>".
                        "</tr>";
        }

# Read and complete template, send it out to browser

open (FH,"nice.htp");
read (FH,$html,-s "nice.htp");
$html =~ s/%(\w+)%/$fill{$1}/g;
```

```
print ("content-type: text/html\n\n$html");

##################################################################

sub get_ncsaparts {
        my ($inline) = @_;
        my %record;
# Extract parts from an NCSA extended log file record (format is next 4 lines)
#   202.187.80.126 - - [09/Feb/2003:00:37:31 -0800]
#   "GET /forum/3935408201.html HTTP/1.1" 200 7250
#   "http://www.google.com/search?hl=en&ie=UTF-8&oe=UTF-8&q=text+file+and+php"
#   "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
        if (my @parts = ($inline =~
                /^(\S+)                  # IP Address or host name
                \s+\S+\s+\S+\s+          # Ignore rarely used security flds
                \[(\d+)\/(\w+)\/(\d+):  # Date
                (\d+):(\d+):(\d+)       # Time
                \s+([-+]?\d\d)\d\d\]\s+ # time offset
                "(\w+)\s+(\S+)          # method and URL
                \s+\S+"\s+              # Ignore HTTP level
                (\d+)\s+([-0-9]+)\s+    # Status and size
                "(\S+)"\s+              # Referer
                "(.+)"\s*$              # Browser
                /x)) {
        for (my $k=0; $k<@names; $k++) {
                $record{$names[$k]} = $parts[$k];
                }
                $record{when} = timegm($record{second},
                        $record{minute},$record{hour},
                        $record{day},$mnames{$record{month}},
                        $record{year}%100) - $record{houroff} * 3600;
        } else {
                $record{status} = 0;
        }
        return %record;
        }

sub collect_form {
        my %ret;
        if ($ENV{REQUEST_METHOD} eq "POST") {
                read (STDIN,$qs,$ENV{CONTENT_LENGTH});
                $ret{rqm} = "POST";
        } else {
                $qs = $ENV{QUERY_STRING};
                $ret{rqm} = "GET";
                }
        my @els = split(/&/,$qs);
        foreach (@els) {
                my ($nam,$val) = split(/=/);
                $val =~ tr/+/ /;
                $val =~ s/%(..)/pack("C",hex($1))/ge;
                $ret{$nam} = $val;
                }
        return %ret;
        }

sub initvars {
        $source = "access_log_15feb" ;
        @names = ("host","day","month","year","hour","minute",
                        "second","houroff","method","url"
                        ,"status","size", "referer","browser");
        @show = ("when","host","url","status","size", "referer");
        %mnames = (Jan => 0, Feb => 1, Mar => 2, Apr => 3,
                May => 4, Jun => 5, Jul => 6, Aug => 7,
                Sep => 8, Oct => 9, Nov => 10, Dec => 11);
        }
```

And the template file:

```
<html>
<head><title>Site access - %myname%</title></head>
<body bgcolor=white>
<h1>Analysis of %source%</h1>
<form method=POST action=%myname%>
<table border=1>%formbody%
<tr><td>and</td><td><input type=submit name=select value=go></td></tr>
</table></form>
Select <a href=%myname%>here</a> to reset the script<br>
%stuff%
<hr>
Copyright Well House Consultants
</body>
</html>
```

17.7 Summary

Perl programs are often used on the web to provide server-side interaction. Inputs still come from the command line, environment variables and `STDIN` and are still written out to `STDOUT`.

In the example we saw in this section, data that a web user entered into a form was read by our Perl program from an environment variable. It was encoded in the form

```
name=value&nextname=nextvalue
```

and many special characters had been replaced by three-character hex codes. This is known as URL encoding.

After processing and accessing data, files, etc, our perl / web application wrote out a header (the format MUST be correct):

```
Content-type: text/html
```

followed by a blank line and then a page of HTML.

This method of using a browser to access a program rather than just a data file is known as CGI or the Common Gateway Interface.

Exercise

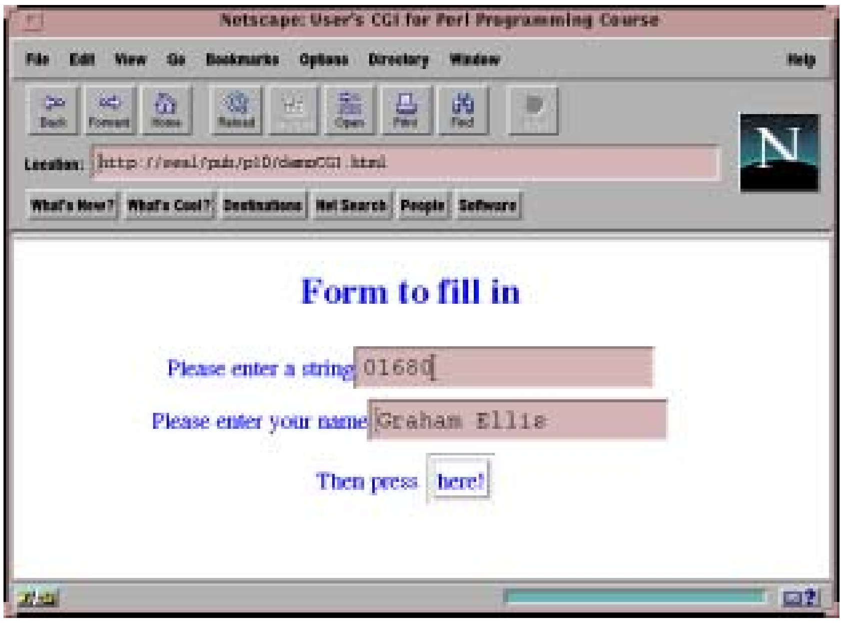
There's a form at
http://seal/pub/xxx/demoCGI.html (where "xxx" is your account name)

View the form, view the source and work out the field names and the name of the Perl program that it calls. You should also note that there will be a copy of the "stdcode" file available at `../stdcode.uk` when you're running your program via the web.

Write a Perl program to collect the data from the form and match the string. You may place the program into the correct directory on the server by typing

`place xxxxxxx` (where "xxxxxxx" is the name of the Perl program)

Our example answer is demo.pl



Location: http://seal/pub/p10/demoCGI.html

Form to fill in

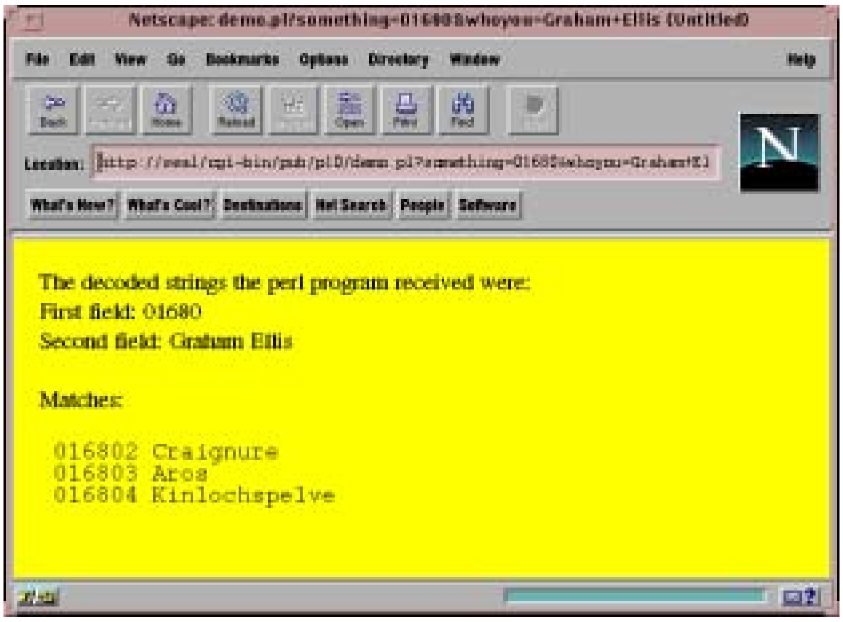
Please enter a string 01680

Please enter your name Graham Ellis

Then press here!

Test by filling in and submitting your form. Work with the student beside you.

Ask him to visit your page; you visit his.



Location: http://seal/cgi-bin/pub/p10/demo.pl?something=01680&whoyou=Graham+E

The decoded strings the perl program received were:
First field: 01680
Second field: Graham Ellis

Matches:

```
016802 Craignure
016803 Aros
016804 Kinlochspelve
```

18 System Dependencies

18.1 The Philosophy

If you've learnt the Java language, you'll be aware that it is designed to run exactly the same no matter what piece of hardware or underlying operating system is in use. You'll also be aware of two other things:

- It's very hard indeed to do things like find out how much disk space is in use and perform systems administration tasks in Java.
- Even then, some Java programs only run on some machines. Do you know which release of Java is it? Does it use the Microsoft extensions? Etc ...

Perl sets out to give you the flexibility to do whatever you like, and it assumes you know what you're doing. You don't have the system administration limits that Java applies. Indeed, Perl is becoming a favourite language for systems administration work on Linux and on Windows NT, as well as on Unix.

And, an even bigger bonus, Perl is remarkably portable!

Of the topics covered thus far, only the section that covered the environment around your Perl program differed greatly between systems, and even there, code can be written that includes both a line to make it execute as a program on Linux and Unix, and a name which means it will work on Win32 systems.

How is it so portable?

Perl originated from the Unix world and you'll find that it's very "Unix-ish" in how it does things. But that Unix-ness is part of the language itself.

Therefore, the authors of implementations for Macintosh and Windows systems had (and took advantage of) the ability to make it do the same thing on the PC.

Let's see an example of that:

If you're a Solaris[1] guru, you'll know that to translate a host computer from a name to an IP address, you would look at

- `/etc/nsswitch.conf` to tell you whether to look at
- `/etc/hosts`
- A NIS map (domain from `/etc/defaultdomain`)
- A NIS+ table
- The Domain Name Service (via `/etc/resolv.conf`)

Err ... too much like hard work, isn't it?

It's all been built into Perl for a long time, so you don't even need to know the stuff above. Instead, you just call

```
gethostbyname("walrus");
```

1 Sun's version of Unix

And -- here's the beauty -- it looks at all the files for you. On Solaris, it'll use the sequence above. On other Unixes, the sequence will vary a bit. On Windows NT it will be very different ... but still you'll just call

```
   gethostbyname("walrus");
```

Of course, the news isn't 100% good. Where the operating system has capabilities beyond what other operating systems have, Perl will tend to support it, and then ... code author beware. You may get a whole load of nulls and undefined's back from other operating systems.

We've grouped together in this section some of the facilities which you should be careful of using if you intend to write portable code.

Finding out what system your script is running on

There's a special variable `$^O (or $OSNAME)` which tells you what the operating system is. There's also a `Config` module which provides a very large amount of extra information

```
#!/usr/local/bin/perl
# info_more - config

use Config;

print "Operating System:    $^O\n";
  # $OSNAME

foreach $name (sort keys %Config) {
 printf "%10s %s\n",$name,$Config{$name};
 }
```

```
seal% info_more
 Operating System:solaris
   Author
 CONFIGtrue
 Date $Date
 Header
 Id     $Id
 Locker
  Log  $Log
 Mcc    Mcc
 PATCHLEVEL3
 RCSfile$RCSfile
 Revision$Revision
 SUBVERSION0
 Source
 State
 afs  false
 alignbytes8
 aphostname
etc ...
 useposixtrue
 usesafetrue
 usevforkfalse
 usrinc/usr/include
  uuname
 vi
 voidflags15
 xlibpth/usr/lib/386 /lib/386
 zcat
```

Figure 125

Running Perl program "info_more" (only a portion of the output is shown).

Let's look at some more of the features of Perl that will make your program system dependent:

18.2 Shell access

You want to include an operating system command within your Perl program? You can do so using backquotes!

```
   `cp one.txt one.bak`;
```

will copy a file on Linux and Unix systems, from "one.txt" to "one.bak".

This is also very useful for utilities which are not built in to Perl

```
   `tar cf /tmp/dbn.tar .`;
```

But what happens to any output on `STDOUT`?

The ` returns the `STDOUT` from the commands you run, so in the examples above, any output is lost. Assign it to a variable, and you have the text to manipulate. Simply print it, and it's seen on the screen.

```
#!/usr/local/bin/perl
# back - system dependent backquotes

$directs = `du -sk /export/home/www/*`;
print "Disc Utilisation\n";
print "================\n";
print `df -k`;
print "Web directories > 1 Mb\n";
print "======================\n";
@lines = split(/\n/,$directs);
foreach (sort {$b <=> $a} @lines) {
 s/$/\n/;
 last if ($_<1000);
 print ;
 }
```

```
seal% back
Disc Utilisation
================
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t3d0s0      61615   56730    4824    93%    /
/dev/dsk/c0t3d0s6     432839  379122   53285    88%    /usr
/proc                      0       0       0     0%    /proc
fd                         0       0       0     0%    /dev/fd
/dev/dsk/c0t3d0s7     422223  408704   13097    97%    /export/home
swap                  147448      76  147372     1%    /tmp
/dev/dsk/c0t0d0s0     192783   19090  173501    10%    /extra/disc0.slice0
/dev/dsk/c0t0d0s4     481983   74851  406651    16%    /extra/disc0.slice4
/dev/dsk/c0t0d0s5     481983   11687  469815     3%    /extra/disc0.slice5
/dev/dsk/c0t0d0s6     963895  766802  195487    80%    /extra/disc0.slice6
/dev/dsk/c0t0d0s7    6404166 5558119  782006    88%    /extra/disc0.slice7
Web directories > 1 Mb
======================
6639/export/home/www/website.version.4
6518/export/home/www/website
6265/export/home/www/pub
4820/export/home/www/website.version.3
3129/export/home/www/website.version.2
2474/export/home/www/website.version.1
1912/export/home/www/perl
1210/export/home/www/wedding
seal%
```

Figure 126
Running Perl program "back".

Variables are expanded within backquotes prior to the execution of the command in just the same way that they would be expanded within double quotes.

Alternative delimiters are available. Use the `qx` operator to indicate "quote" and "execute".

```
$back =  qx!$first *\n!;
```

`Here` documents are available if you want to place a large block of commands within your Perl program

```
   <<`SHELL`
echo "Disc Utilisation at the present time" > /tmp/$$
   df -k >> /tmp/$$
   rusers -a >> /tmp/$$
   SHELL
   ;
```

18.3 System database enquiries

We mentioned getting the host information at the start of this section. If you want to do something but it's not an obvious one-liner

- someone's probably done it already and
- their code's probably available to you

This applies especially to accessing system databases. The following functions are built in to Perl:

```
gethostbyname
gethostbyaddr
```

in a scalar context, these convert from name to IP address or vice versa. In a list context, they return much more information:

```
#!/usr/local/bin/perl
# hostinfo

print "+++++ Checking on 192.168.200.130\n";
@ipdata = (192,168,200,130);
$ippacked = pack("C4",@ipdata);
@about = gethostbyaddr($ippacked,2);
prhost();

print "+++++ Checking on sealion\n";
@about = gethostbyname("sealion");
prhost();

sub prhost {
 @ip = unpack("C4",$about[4]);
 print "Main name: $about[0]\n";
 print "Other names: $about[1]\n";
 print "Address Type: $about[2]\n";
 print "Length: $about[3]\n";
 @ip = unpack("C4",$about[4]);
 print "IP address: ",join(".",@ip),"\n";
 }
```

```
seal% hostinfo
+++++ Checking on 192.168.200.130
Main name: lecht
Other names: www.wellho.co.uk
Address Type: 2
Length: 4
IP address: 192.168.200.130
+++++ Checking on sealion
Main name: sealion
Other names: www.wellho.com
Address Type: 2
Length: 4
IP address: 192.168.200.127
seal%
```

Figure 127
Running Perl program "hostinfo".

Although the introduction seemed pessimistic and too system-dependent, Perl hides all the "look at this file or that file" stuff so, wherever possible, it works on any operating system.

Many of the following, though, are system-dependent. Other system enquires are already covered, too (this isn't a full list)

`getpwnam`	log in by name
`getpwuid`	log in by user id
`getprotoname`	protocol by name
`getprotonumber`	protocol by number
`getnetbyname`	network by name
`getnetbyaddr`	network by address
`getppid`	parentprocess id

```
seal% userinfo
User id is 2000
Group id is 1999
home directory is /export/home/
  graham
log in program is /bin/csh
seal%
```

Figure 128
Running Perl program "userinfo".

```
#!/usr/local/bin/perl
# userinfo

($name,$pwcode,$uid,$gid,$quota,$comment,$gcos,
$home,$logprog) = getpwnam("graham");

print ("User id is $uid\n");
print ("Group id is $gid\n");
print ("home directory is $home\n");
print ("log in program is $logprog\n");
```

18.4 How Perl helps on crossplatform requirements

The topics we've looked at so far in this module have shown you how to write your Perl to be as host operating system independent as possible, and have taken the attitiude of "warning you off" certain elements of the language if at all possible. There are some important times, though, where Perl can be used as a powerful tool to assist you with cross-platform applications.

Text files – end-of-line characters

When you're typing in a plain text file, be it in notepad or edit on a Windows machine, vi or kedit on a Linux or Unix system, or simpletext on a Mac, you'll press the [Enter] key at the end of each line of typing for a new line. Did you realise that your operating system actually puts different control characters into your text on each of the three different groups of platforms?

Windows	`\r\n`	Carriage Return, Line Feed
Mac	`\r`	Carriage Return
Linux / Unix	`\n`	Line Feed

That's all very well (and automatically taken care of) when you're running in a single operating system environment, but when you're using a mixture of systems, some strange results can occur. Some applications (notepad for example) simply can't cope with input that has the wrong delimiters.

Here's a convertor program, written in Perl, that takes a file named on the command line, and converts its end-of-line characters to the Linux / Unix standard:

```
#!/usr/bin/perl

open (FH,$ARGV[0]);

read (FH,$buffer,-s $ARGV[0]);
$buffer =~ s/\r\n/\n/g;
$buffer =~ s/\r/\n/g;
print $buffer;
```

Output from this program is to STDOUT, so that the user can redirect it to a file, or to another application, as appropriate.

If you wish to run a program such as this to produce output that's suitable for a PC or Mac, take care with the order of your substitute commands; we recommend that you use the above program and add before the print function:

```
     $buffer=~s/\n/\r/g;          for Mac
or   $buffer =~ s/\n/\r\n/g;      for Windows
```

Handling Microsoft Word and Excel files in Perl

Good news. There are modules supplied with the Perl distribution from ActiveState that allow you to read in and handle Microsoft Word, Excel and other files. The modules actually make use of some of the Microsoft dll files, so you'll need to run your Perl program on a Windows box that has the appropriate Microsoft product installed.

Why would you want to handle a Word file in Perl?

Here's an example of Perl in use as "Glueware". An Estate Agent produces his detail sets on his office PC, using Microsoft Word. Into the details, he adds photographs of the property for sale (from his digital camera or scanner, at a higher resolution so that he can print out quality details), and floor plans which he enters in to his handheld iPAQ when surveying the property, and come in a wmf (Windows Metafile) format.

In order to upload his details to the Web, he needs to extract the text from the Word document (so that the details can be in a different format on the web site, and sorted by price of house), and reformat the images so that they're gifs or jpgs (changing their resolution).

What follows is a simplified version of the program that's used on the PC that has Word installed; the Image conversion is done using "Advanced Batch Convertor" which is a PC application, and the output is a plain text file which is then uploaded to the Estate Agent's web site using a "file" box on the form, and there it is further interpreted. For ease of support / debugging, the file for upload is hex encoded so that it's all printable characters which can be easily manuipulated.

```
# Perl program - extracting from a Word document ready for upload!

use Win32::OLE;
use Win32::OLE::Enum;

print "Name of Word document: ";
chomp ($doc = <STDIN>);

$document = Win32::OLE -> GetObject($doc);
print "Name of upload file: ";
chomp ($upload = <STDIN>) ;
open (FH,">$upload");
    print "Extracting Text ...$document \n";
    $paragraphs = $document->Paragraphs();
    $enumerate = new Win32::OLE::Enum($paragraphs);
    while(defined($paragraph = $enumerate->Next()))
    {
        $style = $paragraph->{Style}->{NameLocal};
        print FH "+$style\n";
            $text = $paragraph->{Range}->{Text};
            $text =~ s/[\n\r]//g;
            $text =~ s/\x0b/\n/g;
            print FH "=$text\n";
    }
      for ($k=2; $k<10; $k++) {
      $prog = "\"C:\\Program\ Files\\Advanced\ Batch\ Converter\\abc.exe\"";
      print ("Name of image file to use: ");
      chomp ($imgname = <STDIN>);
      last unless ($imgname);
      $instruct = $outstruct = "$imgname";
      #$outstruct = "/convert=X$k.gif";
      if ($instruct =~ /wmf$/) {
        $also = "/resize=(640,0,1)";
        $outfile = "X$k.gif";
      } else {
        $also = "/resize=(640,0,1)";
        $outfile = "X$k.jpg";
      }
      print "Converting $instruct ...\n";
      $result = `$prog $instruct $also /convert=$outfile`;
      open (FHI,$outfile);
      binmode FHI;
      read (FHI,$buffer,-s "$outfile");
      $buffer =~ s/(.)/sprintf("%02x",ord($1))/sge;
      $buffer =~ s/(.{1,68})/=$1\n/g;
      print FH "+pic$k\n=$imgname\n";
      print FH "+pic_$k\n";
      print FH "$buffer\n";
      }
      close (FH);
      print "Job completed.  Press [return] to exit ";
      $n = <STDIN>;
```

Note that in order to be able to use these "Win32" modules, you need to be runnng Perl on a PC with the particular application installed. Because the modules actually make use of the files within the Microsoft application, the example above will only run on a machine that has Word installed, and the following (which uses an Excel spreadsheet) requires Excel to be available to Perl.

```
use Win32::OLE;
$file = "C:\\temp\\MyTest.xls";
$excel = Win32::OLE->GetActiveObject('Excel.Application');
unless($excel)
{
    $excel = new Win32::OLE('Excel.Application', \&QuitApp)
                 or die "Could not create Excel Application object";
}
```

```
$excel->{Visible} = 1;
$excel->{SheetsInNewWorkBook} = 1;
$workbook = $excel->Workbooks->Add();
$worksheet = $workbook->Worksheets(1);
$worksheet->{Name} = "Directory listing";

@files = glob('*');

$range=$worksheet->Range('A1:C1');
$range->{Value} = ['Filename', 'Size', 'Time'];

my $cellrow = 2;

foreach $file (@files)
{
    my ($size,$mtime) = (stat($file))[7,9];
    $range=$worksheet->Range(sprintf("%s%d:%s%d",'A',$cellrow,'C',$cellrow));
    $range->{Value} = [$file,$size,scalar localtime $mtime];
    $cellrow++;
}
$workbook->SaveAs($file);

sub QuitApp
{
    my ($object) = @_;
    $object->Quit();
}
```

This example creates a new Excel spreadsheet, titles the columns "filename", "size" and "time", and puts details of all files in the current directory into that spreadsheet.

18.5 Summary

Perl sets out to give you maximum flexibility, so it's up to you to ignore features like backquotes that are system-dependent.

Built-in routines like `gethostbyname`, although they look to be very Unix-ish, are in fact implemented as widely as possible.

`$OSNAME` (or `$^O`) allows you to check on which system you're running and take appropriate steps

19 More than Simple Lists and Hashes!

When you created your own classes earlier on, your instance variables might have been references to scalars, or references to lists, or references to hashes.

And if you held the instance variables themselves in a list or in a hash, you in effect had a two-dimensional list.

19.1 Multidimensional arrays

Multidimensional lists

In Perl 5 you can write two-dimensional lists directly, just like other programming languages by using two sets of square brackets.

```
#!/usr/local/bin/perl
# deal - deal a pack of cards

@pack = (1..52);
@shpack = shuffle(1, @pack);
print "dealing: \n";
for ($k=0;$k<13;$k++) {
 for ($j=0;$j<4;$j++) {
 print $hand[$j][$k] = $shpack[$nc++]," ";
 }
 print "\n";
}
print "hands: \n";
for ($j=0;$j<4;$j++) {
 for ($k=0;$k<13;$k++) {
 print $hand[$j][$k]," ";
 }
 print "\n";
}
##############################################
sub shuffle {
my ($mode,@incards) = @_;
 return @incards unless ($mode);
srand;
for ($k=52;$k;){
 $card = int rand $k;
 push @out,$incards[$card];
 $incards[$card]=$incards[--$k];
 }
return @out;
}
```

```
seal% deal
dealing:
14 6 16 33
5 23 19 1
47 17 22 28
30 29 52 39
37 25 12 3
43 45 24 35
21 36 18 42
34 31 48 32
38 51 26 9
15 27 4 7
44 11 50 46
13 40 8 20
10 49 41 2
hands:
14 5 47 30 37 43 21 34 38 15 44 13 10
6 23 17 29 25 45 36 31 51 27 11 40 49
16 19 22 52 12 24 18 48 26 4 50 8 41
33 1 28 39 3 35 42 32 9 7 46 20 2
seal%
```

Figure 129
Running Perl program "deal".

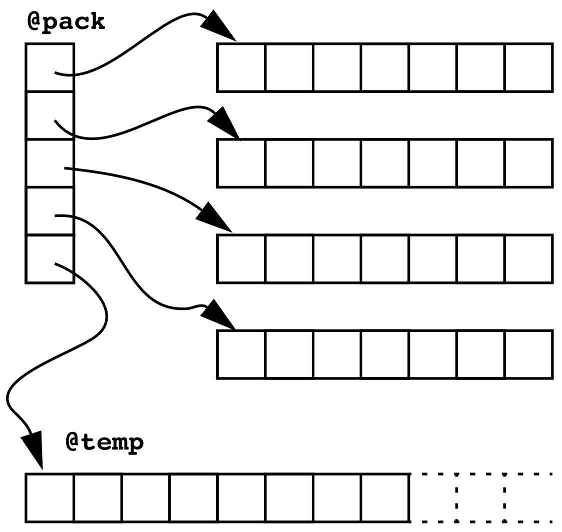
Notice that there's still no need to tell Perl how big the list will be. It's automatically sized and expanded!

Internally, Perl's using a list of lists so there's no need for the two-dimensional array to be rectangular. Here's an example of dealing seven cards to each of four players and placing the rest as a "stock" in a fifth hand:

```
#!/usr/local/bin/perl
# deal2 - deal a pack of cards and leave stock

@pack = (1..52);
@shpack = shuffle(1, @pack);

print "dealing: \n";

for ($k=0;$k<7;$k++) {
 for ($j=0;$j<4;$j++) {
 print $hand[$j][$k] = $shpack[$nc++]," ";
 }
 print "\n";
}

my (@temp) = @shpack[$nc..51];
$hand[4] = \@temp;

print "hands: \n";

for ($j=0;$j<4;$j++) {
 for ($k=0;$k<7;$k++) {
 print $hand[$j][$k]," ";
 }
 print "\n";
}

print "stock: \n";
for ($k=0;$k<24;$k++) {
 print $hand[4][$k]," ";
 }
print "\n";
################################################
sub shuffle {

my ($mode,@incards) = @_;
 return @incards unless ($mode);
srand;
for ($k=52;$k;){
 $card = int rand $k;
 push @out,$incards[$card];
 $incards[$card]=$incards[--$k];
 }
return @out;
}
```

```
seal% deal2
dealing:
1 20 16 11
42 7 27 50
35 17 29 48
45 19 5 30
22 38 26 9
34 23 24 37
25 15 14 44
hands:
1 42 35 45 22 34 25
20 7 17 19 38 23 15
16 27 29 5 26 24 14
11 50 48 30 9 37 44
stock:
21 10 28 49 6 12 3 32 52 47 46 2
31 39 36 18 40 41 13 4 43 8 51 33
seal%
```

Figure 130
Running Perl program "deal2".



Mixing the dimensions

Although that first example used a list of lists, there's no reason why you can't use

- Hashes of lists
- Lists of hashes
- Hashes of hashes

as appropriate

19.2 Something more complex

You're not restricted to the conventional!

Let's say that you want to access information about host computers based on any one of their names or their IP address.

Design first

Firstly, let's design the record for each computer. A simple list will do. The first element being the IP address, the second being the real name of the computer and then subsequent elements being any aliases.

Then let's create a hash, with entries for the IP addresses and for each name, with those entries all pointing to the list that holds the data itself.

No matter whether the user enters an IP address or a name, we can look it up in the hash.

Setting up the structure

Let's look at the code to create the hash:

```
while ($record = <SOURCE>) {
chop $record;
my (@fields) = split(/\s+/,$record);
foreach $item(@fields) {
$db{$item} = \@fields;
}
}
```

Each line is read in and split into "@fields" -- a `my` list -- in other words, the name "@fields" will be released each time the `while` loop completes, so each time through it will be a new list.

The address of each "@fields" list, though, is stored into the hash `%db`. Indeed, it's stored several times since this is a multi-keyed application.

When we show you the complete code at the end of this section, you'll notice that we have other code present to remove comments and check that keys really are unique!

Referencing the structure

To keep the example short, we've just asked the user to enter any key for which we look up the data and exit.

```
print "item of interest: ";
chop ($ask = <STDIN>);
@extract = @{$db{$ask}};
print "Info: @extract\n";
```

```
seal% multikey
item of interest: lecht
Info: 192.168.200.130 lecht www.wellho.net
```

Figure 131
Running Perl program "multikey" showing partial output.

We've simply copied the information out of the list that's held in the `%db` hash, then printed the whole of that copy list.

Although you may be a little surprised that you're allowed to write things like this ... why not?? Literally, "the extract list is to become a copy of the list at `$db` of `$ask`".

Here is an alternative way of extracting information:

```
 print ${$db{$ask}}[0],"\n";
```

Reading as before ... "scalar element 0 of the list at `$db` of `$ask`".

If that's getting a bit tricky, here's an alternative notation:

```
 print $db{$ask}->[0],"\n";
```

or I could even write

```
 print $db{$ask}[0],"\n";
```

after all, we created our original structure in a new way, but it can be followed through just like a normal two-dimensional array!

Finally, here are lines that print the whole list relating to one particular key, and the number of elements in that list:

```
 print @{$db{$ask}},"\n";
 print $#{$db{$ask}}+1,"\n";
```

Let's put the whole together and run it.

```
#!/usr/local/bin/perl
# multikey - hash of lists
# keyed on IP and on names

open (SOURCE,"systems") ||
die "no system info\n";

while ($record = <SOURCE>) {
 chop $record;
 $record =~ s/#.*//;
 $record =~ s/^\s*//;
 next unless ($record);

 my (@fields) = split(/\s+/,$record);

 foreach $item(@fields) {
 if ($db{$item}) {
 print "conflict on $item\n";
 $err ++;
 next;
 }
 $db{$item} = \@fields;
 }
 }

$err && die
```

```
seal% multikey
item of interest: lecht
Info: 192.168.200.130 lecht www.wellho.net
other ways of extracting:
192.168.200.130
192.168.200.130
192.168.200.130

192.168.200.130lechtwww.wellho.net
3
seal%
```

Figure 132
Running Perl program "multikey".

```
"data not suitable for this structure\n";

print "item of interest: ";
chop ($ask = <STDIN>);
@extract = @{$db{$ask}};
print "Info: @extract\n";

print "other ways of extracting:\n";
print ${$db{$ask}}[0],"\n";
print $db{$ask}->[0],"\n";
print $db{$ask}[0],"\n\n";
print @{$db{$ask}},"\n";
print $#{$db{$ask}}+1,"\n";
```

19.3 Grouping in Perl

By this point, you may be getting confused between round brackets and square brackets, braces and <> pairs. There are only three types of brackets in the ASCII character set but, in languages like Perl, you need to group elements together for many more than just three reasons. So brackets have to mean different things at different times.

The following table shows you various uses of `()` `[]` `{}` and <> in Perl. You'll probably be familiar with most of them by now, but the odd one might be new.

`()` Round brackets

1. Precedence in expressions
   e.g. `$temp = ($other - 32) / 9 * 5;`
2. Force list context
   e.g. `($var) = split /\s+/,$in,2 ;`
3. Tagging ("interesting bits") in regular expressions
   e.g. `$ENV{"QUERY_STRING"} =~ /(.*)=(.*)/;`

`[]` Square brackets

1. An element in a list
   e.g. `$table[16]++;`
2. Reference to an anonymous list
   e.g. `@chinese = (["Soup","Salad","Seaweed"], ["Curry","Kung Po","Chow Mein"]);`
3. A list slice
   e.g. `@start = @info[0..3];`
4. Regular expression – any one of
   e.g. `/^[A-Za-z]+$/;`

`{}` Curly braces

1. An element in a hash
   e.g. `$name = $table{"name"};`
2. A block of code
   e.g. `{ $i = 16;  $k = 25; }`
3. Delimiting a variable name
   e.g. `print ("It weighs ${pounds}lbs\n");`
4. general counts in regular expressions

e.g. `/\.\w{2,6}$/;`

<> Less than and greater than

1. Read from a file handle
   e.g. `@lines = <FH>;`
2. Read from matching file names
   e.g. `@files = <*.html>;`

The use of square brackets to give you a reference to an anonymous list may come as a surprise. You might have expected to write:

```
        $abc = \(40,60,75);
```

but that will generate a list of references and not a reference to a list. Thus (in a scalar context) `$abc` will be assigned the value 3.

What you want to write instead is:

```
        $abc = [40,60,75];
```

19.4 Interpreting a complex reference to a variable

If you look at complex variable names with lots of `$`s, `@`s, `%`s, \s, `{}`s, `[]`s and ->s in them and wonder how to read that, here are some tips:

1. Start at the extreme left. If the variable name starts with a

`$`	it's a scalar
`%`	it's a hash
`@`	it's a list

   If it starts with a `\`, then it's a scalar which is:

`\$`	a reference to a scalar
`\@`	a reference to a list
`\%`	a reference to a hash

   For references to anonymous variables, you can also have:

`\{`	a reference to a hash
`\[`	a reference to a list
`\(`	a list of references

2. Curly braces immediately after a `%`, `$` or `@` are used to delimit the variable name.
3. An extra leading `$`, or a `->`, mean "referenced by" or "contents of", and the `->` can be left out if it comes between subscript elements.
4. A `[]` subscript means a list member, and a `{}` subscript means a hash member.

You might want to work through some examples, perhaps with a pencil and paper to draw diagrams.

```
        @{$db{$ask}}
```

The list which is referenced by the element of the hash `%db` whose key is in the scalar `$ask`.

```
        ${$db{$ask}}[0]
```

The scalar which is in element 0 of the list referenced by the element of the hash `%db` whose key is in the scalar `$ask`. The following who examples refer to the same scalar:

```
        $db{$ask}->[0]
        $db{$ask}[0]
```

19.5 Design MATTERS

If you're going to get involved in applications that require this sort of thing ...

START with the design of your data and
USE a class of METHODS

The integrity of the data structure is vital. By isolating access in a package, you can reduce the danger of user code causing damage, as well as making the user code easier to follow.

19.6 Summary

You can create two-dimensional lists ... and lists of hashes, and hashes of lists, and other more exotic structures if you wish. An example we studied was a double-keyed hash of lists.

It is vital that you design your application carefully if you are going to use such structures, and you are strongly encouraged to use an object-oriented design and implementation.

Exercise

Read the file "rgb.txt" (each line consists of 4 values) into a two-dimensional list. Print out all the entries which have a value over 239 in the third position.

Our example answer is rgb

Sample

```
graham@otter:~/profile/answers_pp> ./rgb
255 250 250 snow
248 248 255 ghost white
248 248 255 GhostWhite
245 245 245 white smoke
245 245 245 WhiteSmoke
etc.
```

For Advanced Students

You may prefer to examine our sample answer rather than write your own unless you had considerable programming experience prior to this course!

- Read the "access_log" file and create a hash with keys being the name of each host computer. Have each hash element point to a list of access records.
- Print out the first and last access record for each host in alphabetic order.

Our example answer is wstruct

Sample

```
aviemore
- - [15/Jul/1998:08:32:55 -0400] "GET /perl/interact/wash.html HTTP/1.0" 200 686
- - [02/Feb/1999:11:52:20 +0000] "GET /perlman/READMEs/README HTTP/1.1" 200 4716
catfish
- - [28/Aug/1998:09:47:39 -0400] "GET /index.html HTTP/1.0" 200 1476
- - [11/Dec/1998:09:27:25 -0500] "GET /index.html HTTP/1.0" 200 1476
clam
etc.
```

20 Handling Dates and Time

"Which is the most recent file of .....?"
"How long has this program been running?"
"What will the date be three weeks from today?"
Common questions handled in Perl requiring comparisons of date and time. Hardly an easy job with the tools you've seen thus far. Of course, Perl can make it easy!

20.1 So far

File status operators

You've written

```
 $modded = -M "subtwo"
```

to tell you how long ago the file "subtwo" was last modified

```
#!/usr/local/bin/perl
#  dt1 - file last modified

print "file: ";
chop ($name = <STDIN>);

$modded = -M $name;

print "$name was last altered
$modded days ago\n";
```

```
seal% dt1
file: subtwo
subtwo was last altered 9.45079861111111 days ago
seal%
```

Figure 133
Running Perl program "dt1".

That figure is reported in decimal days.
The following options are also available:

`-A`days since file was last accessed

`-C`days since header last changed[1]

stat on a file

If you `stat` a file, you can get back the same information, but in seconds from Midnight, 1st January 1970 (this time is also known as the epoch)

```
#!/usr/local/bin/perl
#  dt2 - file last modified

print "file: ";
chop ($name = <STDIN>);

@finfo = stat($name);

print "$name was last altered at $finfo[9]\n";
```

```
seal% dt2
file: subtwo
subtwo was last altered at 1053071981
seal%
```

Figure 134
Running Perl program "dt2".

[1] not WIN32

Via system commands

You've learnt how you can get the current date and time by using system commands. You've also been told not to do this if you want portable code!

```
 #!/usr/local/bin/perl
#  dt3 - current date and time

$current = `date`;
print "Current date and time is $current";

$day = `date +%A`;
print "Today is $day";
```

```
seal% dt3
Current date and time is Sun Jan 31 19:03:50 GMT 1999
Today is Sunday
seal%
```

Figure 135
Running Perl program "dt3".

20.2 How Perl handles dates and times

You've already seen a range of units and facilities ... and there's more to come. But if you work with dates and times, you'll want to convert them to more convenient units than days, months, years, hours, minutes and seconds. In other words, to easily answer a question like:

"What will the date and time be three weeks, two days and seven hours from today?"

It's easy!

- Get the current date and time in whatever units
- Convert to seconds from 1.1.1970[1]
- Do your arithmetic
- Convert back to whatever units you want to report in

In other words, all dates and times are reduced to seconds from the epoch, and converters are provided to work each way.

Amazingly, you can also work in seconds before the epoch, so the scheme is valid from 1904 through to 2038 before nasty things start happening.

Other date information available

`stat`[2]	
`time()`	current time and date
`$^T` (or `$BASETIME`)	time the script started

Finally, you can set the times that a list of files were last accessed and the time they were last modified using the `utime` function.

```
  utime ($acc,$mod,@files);
```

In Linux and Unix terms, this is the equivalent of the "touch" command although empty files are not created as you give a new name.

1 for the Macintosh, operating systems prior to OSX, the time in seconds is from 1.1.1904; with OSX, time is compatible with other systems - i.e. from 1.1.1970

2 covered earlier

20.3 Convertors

Convert from epoch seconds into "human readable" form:

`gmtime`epoch seconds to Greenwich Mean Time
`localtime` epoch seconds to local time
Let's look forward six weeks and 100 weeks:

```
#!/usr/local/bin/perl
#  dt4 - look forward  weeks

print "How many weeks? ";
chop ($weeks = <STDIN>);

$now = time();
$forward = $now + $weeks * (60 * 60 * 24 * 7) ;

treport("Today it is ",$now);
treport("In $weeks weeks it will be ",$forward);

################################################

sub treport {

my ($text,$timing) = @_;

($sec,$min,$hour,          # second, minute, hour
 $mday,$month,$year,        # day of month, month
   # (0-11), year
 $wday,$yday,$dst) =       # day of week, day of
   # year, daylight
 localtime($timing);

printf ("%s %d/%d/
  %d\n",$text,$mday,$month+1,$year+1900);
}
```

```
seal% dt4
How many weeks? 20
Today it is  16/5/2003
In 20 weeks it will be  3/10/2003
seal% dt4
How many weeks? -200
Today it is  16/5/2003
In -200 weeks it will be 16/7/1999
seal%
```

Figure 136
Running Perl program "dt4".

You'll notice:

- Months come back as 0 to 11, not 1 to 12
- Years are returned from 1900. Watch years after 1999!
- We can tell the day of the week
- We can tell whether daylight saving is in effect

In a scalar context, `localtime` and `gmtime` will return a string containing the date in a nice format

Convert from human readable form to epoch seconds

Not such a common requirement, so you need to use a module which is supplied as standard with the Perl distribution

```
use Time::Local;
```

and you can then use function
`timelocal`to convert local time to seconds
`timegm`to convert GM Time to seconds

```
#!/usr/local/bin/perl
#  dt5 - compare a stated date and time with now!

use Time::Local;
print "When do you want? ";
chop ($timeline = <STDIN>);
($day,$month,$year,$hour,$min) =
 ($timeline =~
  /^\s*(\d+)\/(\d+)\/(\d+)\s+    # date
 (\d+):(\d+)\s*$/x);             # time
$now = time();
$then = timegm(0,$min,$hour,$day,$month-
  1,$year);

$diff = $then -$now;
@tsplit = splittime($diff,60,60,24,7);
treport("It is ",$now);

print "You looked ",($diff>0)?"forward ":
 "backward ",
 "$tsplit[4] weeks ",
 "$tsplit[3] days ",
 "$tsplit[2] hours ",
 "$tsplit[1] minutes \n";
################################################
sub treport {
my ($text,$timing) = @_;

($sec,$min,$hour,          # second, minute, hour
 $mday,$month,$year,        # day of month, month
   # (0-11), year
 $wday,$yday,$dst) =        # day of week, day of
   # year, daylight
 gmtime($timing);

printf (
 "%s %d/%d/%d %02d:%02d\n"
 ,$text,$mday,$month+1,$year+1900,
 $hour,$min);
}
################################################
sub splittime {
my ($val,@list) = @_;

$val = abs($val);
foreach $factor (@list){
 push @rv,$val%$factor;
 $val/=$factor;
 }
push @rv,int($val);
return @rv;
}
```

```
seal% dt5
When do you want? 28/12/79 06:15
It is  16/5/2003 08:04
You looked back 1220 weeks 0 days
  1 hours 49 minutes
seal% dt5
When do you want? 09/08/03 11:30
It is  16/5/2003 08:05
You looked forward 12 weeks 1 days
  3 hours 24 minutes
seal%
```

Figure 137

Running Perl program "dt5".

20.4 Handling centuries

Whilst it is easily possible to write code that will work with old data for years starting 19xx, and also for current 20xx data, you need to take care as you program.

In particular, please note:

- `gmtime` and `localtime` return the year since 1900 so that you'll get back the number "103" for the year 2003, for example. If you want a two-digit year, you could format it using:

```
      $yr = sprintf("%02d",$yrfromgmtime % 100);
```

or for a four-digit year:

```
      $yr = sprintf("%4d",$yrfromgmtime + 1900);
```

- `timegm` and `timelocal` take a two-digit year code. If the number you pass in is 00 to 37, it's assumed you mean 2000 to 2037. Enter a number from 39 to 99, and Perl will use 1939 to 1999.

20.5 Elapsed time sleep

All of the functions and codes above work with dates and time. What if you want to refer to a number of seconds within your program?

`sleep(10)`	sleep 10 seconds
`sleep()`	sleep "forever"

What's the point in sleeping forever? Until an external signal is received from another process. We'll look at that more on our advanced courses.

alarm

Sets an alarm clock, so that a signal goes off after a given number of seconds. You'll study the full mechanism on the Perl Advanced course, but here's a taster if you just have a simple requirement:

```
#!/usr/local/bin/perl
#  chivvy - hurry the user up!

@waken = (20,15,10,10,5);

$SIG{"ALRM"} = "comeon";

print "Enter your name: ";
until ($yousaid or not $waken[$kp])
 {
 alarm ($waken[$kp]);
 $yousaid = <STDIN>;
 $kp++;
 }

$yousaid ?

 print "You entered $yousaid":
 print "You failed to respond in time\n";
###############################################
```

```
seal% chivvy
Enter your name:
40 seconds left ...
25 seconds left ...
15 seconds left ...
5 seconds left ...
0 seconds left ... You failed to respond in time
seal% chivvy
Enter your name: Graham Ellis
You entered Graham Ellis
seal% chivvy
Enter your name:
40 seconds left ...Graham Ellis
You entered Graham Ellis
seal%
```

Figure 138
Running Perl program "chivvy".

```
sub comeon {
$at += $waken[$kp];
print "\n",60-$at," seconds left ... ";
}
```

- the `%SIG` is an array of signals, into which is placed the name of subroutines to be called when a particular signal is received - in this case, `ALRM`
- `alarm` is used to set an interval timer. Perl will break out of any waiting statement when the timer goes off, firstly performing the "comeon" subroutine, then continuing with the next statement.
- the `<>` operator knows that it hasn't returned any of your typing if a signal occurred, so the text remains in the input buffer.

20.6 Summary

All date mathematics are handled in epoch seconds. Such numbers are given directly to you by the `stat` function, by the `time` function, and in the `$^B` variable.

You can convert from epoch seconds to date and time using `localtime` or `gmtime`, and from date and time to epoch seconds using `timelocal` or `timegm` from the `Time::Local` module.

Epoch seconds work fine from 1904 to 2038, but you must take care with conversions, etc, to ensure that the programs you write are cross-century compliant.

An alarm clock can be set up to signal back to you after a certain interval, allowing your process to sleep or wait. This also allows you to time out users who are slow in replying to a prompt.

Exercise

Write a program to print out the time elapsed between each web site access from the machine "catfish". (You might like to start from our example program "wrt2", or your equivalent.)

Our example answer is webgap

Sample

```
graham@otter:~/profile/answers_pp> webgap
28/Aug/1998:09:47:39
seconds gap: 655
0 days 0 hours 10 minutes 55 seconds
28/Aug/1998:09:58:34
seconds gap: 67
0 days 0 hours 1 minutes 7 seconds
28/Aug/1998:09:59:41
seconds gap: 9073563
105 days 0 hours 26 minutes 3 seconds  << Is this correct?  Yes!!! Why?
11/Dec/1998:09:25:44
seconds gap: 101
0 days 0 hours 1 minutes 41 seconds
11/Dec/1998:09:27:25
graham@otter:~/profile/answers_pp>
```

21 Practical Example – Perl in use

This module takes a practical programming problem that was posed to us, and provides a solution in Perl. We have intentionally used a wide range of fundamental and intermediate Perl topics in our answer, but we have not used anything that's really obscure or advanced. Our intent is to provide you with a case study in which you can find snippets of code showing how Perl's features are used and how they work together.

21.1 The requirement

Introduction

A large department has a number of employees, each of whom are to attend one of a series of courses of their choice. There are a limited number of places on each course, and some courses are more popular than others. Although we would love to give everyone their first choice, we can't!

Our requirement is to place people onto courses in such a way that there are as few "downgrades" as possible. We define a downgrade as someone having their second choice rather than their first, or their third choice rather than their second (i.e. someone who is given a place on his third choice has been downgraded twice).

Input and output

Notice that it's very important in the specification of a problem to start off asking "what do we start with?" and "where to we end up?".

We begin with a file containing everyone's name, and a list of the courses that they rate, starting with their first preference. Here's the file:

```
antonia Perl XML PHP Tcl/Tk MySQL
barbara Tcl/Tk ASP Ruby Java
cherry Perl Java Ruby MySQL
delia XML PHP Java ASP
ethel MySQL Perl Tcl/Tk ASP
florence Ruby PHP Java ASP
gloria XML Perl Tcl/Tk MySQL
hazel PHP Python Perl Ruby ASP
iris  Perl MySQL Java Tcl/Tk
jenny XML Perl Ruby ASP
kerry Perl Tcl/Tk Ruby MySQL
leane PHP Python ASP Perl Java
margaret XML Perl Ruby MySQL Tcl/Tk
nina Tcl/Tk Perl ASP Ruby
olivia  MySQL Python ASP PHP
```

```
petra XML Tcl/Tk ASP Perl Ruby
queenie Ruby Perl ASP MySQL
rita PHP ASP Ruby Perl
sally Tcl/Tk Perl XML MySQL
tina Tcl/Tk Ruby Java
uva MySQL Perl Java PHP
venus Java Perl Ruby ASP
wendy Perl Tcl/Tk ASP MySQL
xena  Java Perl PHP ASP XML
yollanda Ruby MySQL Tcl/Tk
zoe Ruby ASP Perl PHP
adam Tcl/Tk Perl Python MySQL
barry Python XML Java Perl PHP
charles Perl Ruby MySQL Tcl/Tk
david Perl Tcl/Tk Java
ed Ruby Perl Java PHP
fred MySQL Perl Java XML
graham Java Perl Tcl/Tk
harry PHP Python Java
ivan Ruby Java Perl Tcl/Tk MySQL
john PHP XML Java Perl
ken Tcl/Tk Python Java Perl
len Perl Java MySQL Ruby
morris Perl Java PHP Tcl/Tk
nigel PHP Python Java Perl
orpheus MySQL Ruby Tcl/Tk XML
peter PHP Java Perl
quentin Tcl/Tk Perl PHP Ruby
rupert Java Python MySQL
steve Tcl/Tk Perl PHP Ruby
tommy Perl Java XML
ulsyees Java PHP Perl
victor Ruby Perl Tcl/Tk MySQL
william Ruby Perl PHP
xavier PHP Java Perl
yuri XML PHP Perl Tcl/Tk
zachary MySQL Java Tcl/Tk
```

We're looking for an output that tells us who is on which course, and indicates how many downgrades he has taken to be there. We've enhanced the output to include a review of the incoming data (how many people want each course, number of places available, etc.), and also to report on the operation of the program as it proceeds.

Here is our complete output:

```
Incoming trainees:           52
Different courses requested: 9
Places per course:           6
Total places available:      54

16 expressed interest in ASP
```

```
0 had top choice of ASP

28 expressed interest in Java
5 had top choice of Java

22 expressed interest in MySQL
6 had top choice of MySQL

23 expressed interest in PHP
8 had top choice of PHP

42 expressed interest in Perl
10 had top choice of Perl

9 expressed interest in Python
1 had top choice of Python

24 expressed interest in Ruby
8 had top choice of Ruby

26 expressed interest in Tcl/Tk
8 had top choice of Tcl/Tk

14 expressed interest in XML
6 had top choice of XML

...Improved to -16
...Improved to -14
....Remains at -14 (only achieved -20)
...Remains at -14 (only achieved -21)
...Remains at -14 (only achieved -23)
..Remains at -14 (only achieved -18)
...Remains at -14 (only achieved -17)
...Remains at -14 (only achieved -16)
...Remains at -14 (only achieved -17)
...Remains at -14 (only achieved -17)
      Adam is on     Tcl/Tk - their choice 1
   Antonia is on       Perl - their choice 1
   Barbara is on        ASP - their choice 2
     Barry is on     Python - their choice 1
   Charles is on       Perl - their choice 1
    Cherry is on       Java - their choice 2
     David is on       Perl - their choice 1
     Delia is on        XML - their choice 1
        Ed is on       Ruby - their choice 1
     Ethel is on      MySQL - their choice 1
  Florence is on        PHP - their choice 2
      Fred is on      MySQL - their choice 1
    Gloria is on        XML - their choice 1
    Graham is on       Java - their choice 1
     Harry is on        PHP - their choice 1
     Hazel is on     Python - their choice 2
```

```
      Iris is on      MySQL - their choice 2
      Ivan is on       Ruby - their choice 1
     Jenny is on        XML - their choice 1
      John is on        PHP - their choice 1
       Ken is on     Python - their choice 2
     Kerry is on       Perl - their choice 1
     Leane is on     Python - their choice 2
       Len is on       Java - their choice 2
  Margaret is on        XML - their choice 1
    Morris is on       Perl - their choice 1
     Nigel is on        PHP - their choice 1
      Nina is on     Tcl/Tk - their choice 1
    Olivia is on     Python - their choice 2
   Orpheus is on      MySQL - their choice 1
     Peter is on        PHP - their choice 1
     Petra is on        XML - their choice 1
   Queenie is on       Ruby - their choice 1
   Quentin is on     Tcl/Tk - their choice 1
      Rita is on        ASP - their choice 2
    Rupert is on     Python - their choice 2
     Sally is on     Tcl/Tk - their choice 1
     Steve is on     Tcl/Tk - their choice 1
      Tina is on     Tcl/Tk - their choice 1
     Tommy is on       Perl - their choice 1
   Ulsyees is on       Java - their choice 1
       Uva is on      MySQL - their choice 1
     Venus is on       Java - their choice 1
    Victor is on       Ruby - their choice 1
     Wendy is on        ASP - their choice 3
   William is on       Ruby - their choice 1
    Xavier is on        PHP - their choice 1
      Xena is on       Java - their choice 1
  Yollanda is on       Ruby - their choice 1
      Yuri is on        XML - their choice 1
   Zachary is on      MySQL - their choice 1
       Zoe is on        ASP - their choice 2
Measure of success in place people - -14 points
ASP    Barbara * Rita *    Wendy **  Zoe *
Java   Cherry *  Graham    Len *     Ulsyees   Venus     Xena
MySQL  Ethel     Fred      Iris *    Orpheus   Uva       Zachary
PHP    Florence *Harry     John      Nigel     Peter     Xavier
Perl   Antonia   Charles   David     Kerry     Morris    Tommy
Python Barry     Hazel *   Ken *     Leane *   Olivia *  Rupert *
Ruby   Ed        Ivan      Queenie   Victor    William   Yollanda
Tcl/Tk Adam      Nina      Quentin   Sally     Steve     Tina
XML    Delia     Gloria    Jenny     Margaret  Petra     Yuri
```

In this final table, each "*" represents a downgrade point. In our sample data, no one expressed a top preference for ASP, and only Barry preferred Python above anything else, so you see a lot of downgrades on those courses. There are only two free places anywhere in the plans, so someone's not going to be satisfied.

The plans

This is a surprisingly difficult task. Have you ever struggled to get a whole lot of people into groups or classes, or a number of items into a limited number of shopping bags each with limited capacity, and where you don't want to mix the cleaning chemicals with the food? It's very easy to place the first few items and people, but then it starts getting awkward when you start re-arranging. Very rarely are you truly satisfied with the result, or have it optimal. How on earth are we going to automate this?

We'll write our application to read in our data into memory by using a Perl hash of lists so that we can keep coming back to it.

We'll get a list of all the people who are looking for courses, and we'll go through that list in a random order, placing each person on his top choice of course that's still got places available. Anyone who's unfortunate enough to have had all his/her choices filled up already is placed on a list for later placement.

Once all people who can be given a course of their choice are placed, we simply take the people we couldn't place and assign them to a remaining vacancy.

We then work out the score – the number of downgrades for our solution.

Next, we'll try swapping over every pair of trainees in turn, and see if that improves the score. If it does, we leave the change in place. If not, we swap them back. We keep doing this until no further improvement can be made by any two-way swap.

The results probably look good, but are not necessarily the best that can be achieved. This is a "linear programming" problem, and it's rather like climbing a mountain and always taking the steepest track to get up quickly. You may get to the top, find there's no way up any more, and look across to see another, higher peak of the same mountain.

In practice, our initial placement algorithm has given us a pretty good start ("well up the mountain already"), so we might not be too far out. But in order to ensure best placement, our program remembers the solution and the score that it has found, and repeats the whole process, with the names in a different, re-randomised order, to see if it can do better. This loop runs 10 times.

The detail

Here's our program, with line numbers to help the tutor (and you) track down elements that we have described above.

```
  1  #!/usr/bin/perl
  2
  3  # Course placer.   Demonstration program showing the use of
  4  # many Perl fundamentals, written and Copyright Well House
  5  # Consultants Ltd.  Phone +44 (0) 1225 708225
  6
  7  $coursemax = "@ARGV" || 6;# maximum trainees per course
  8  open (FH,"requests") or# file of course requests
  9      die "No incoming file of place requests\n";
 10
 11  # set up list of names, hash of requirements and hash of courses
 12  ################################################################
 13
 14  while (<FH>) {
 15      my ($name,@want) = split (/[\s,]+/);
 16      $request{$name} = \@want;
 17      foreach $course (@want) {
 18          $c{$course}++;
 19          $tops{$course}+=0 ;
 20          }
 21      $tops{$want[0]}++ ;
 22      $ntrainees++;
 23      }
 24
 25  # Our input data is now in
 26  #   %c  hash of course names, containing number of
 27  #       expressions of a preference for each
 28  #   %topshash of coures names, containing a count of
 29  #       top preferences
 30  #   %request  hash of lists of preferences. The main data
 31  #       for the forthcoming sections
 32  # Above data is NOT altered during iterations
 33
 34  # List out incoming stats. This section does not alter the
 35  # data.
 36
 37  print "Incoming trainees:           $ntrainees\n";
 38  print "Different courses requested: ",$nc=scalar(keys %c),"\n";
 39  print "Places per course:           $coursemax\n";
 40  print "Total places available:      ",$nc*$coursemax,"\n\n";
 41  foreach $course (sort keys %c) {
 42      print "$c{$course} expressed interest in $course\n";
 43      print "$tops{$course} had top choice of $course\n\n";
 44      }
 45
 46  ($ntrainees > $nc * $coursemax) and die "Not enough places\n";
 47
 48  # place people one by one on to their most available course
```

```
############################################################

# Put code from here down in a loop to keep trying to get
# a better result; otherwise, we may end up with a result
# that cannot be improved but is in an iteration "cul de sac"

$iterationcount = 10;   # number of iterations to run

for ($iteration=0; $iteration<$iterationcount; $iteration++) {

my %assign;

# Take all the trainees (in a random order) and place each in
# turn on their highest choice course that's still available

@names = shuffle(keys %request);

foreach $person (@names) {
    my @want = @{$request{$person}};
    $placed = 0;
    foreach $try (@want) {
        if ($assign{$try} > ($coursemax-1)) {
            next;
        } else {
            $oncourse{$person} = $try;
            $assign{$try}++;
            $placed = 1;
            last;
        }
    }
# If none of their choices is available, store them for the
# next stage - see if you can place other people on a
# choice thay have made first!

    unless ($placed) {
        # warn ("Can't place $person yet\n") ;
        push @filler,$person;
        }
}

# spread the unsatisfied around amongst courses that are slack!
# Start with most popular courses for better results later?
# But this will lead to some dreadful slack courses!
# Doesn't make much difference if courses are really tight
################################################################

while (1) {
foreach $course (sort {$a <=> $b} keys %c) {
    next if ($assign{$course} > ($coursemax-1));
    $force = pop @filler;
    last unless ($force);
    $oncourse{$force} = $course;
    $assign{$course}++;
    }
last unless ($force);
}


# print out the initial assignments and scores for this iteration
# if you're in debug mode; also get a score.    0 is a perfect
# score (everyone on their top choice) and 1 is taken off that
# score for each time someone is moved one place down their list
# of preferences

# At this point, %oncourse is a hash of people containing the
# name of the course they are provisionally selected to attend.
# %assign contains the course names and the number of people
# assigned to each. So far, it is only used to avoid overbooking
# of a course (already done), but it might be useful later as
# we enhance the algorithm.

$value = reportscore (0);  # change to 1 to test improvement loop!

# See if we can improve scores by doing a crossover swap
#########################################################

# keep trying this loop until no further improvements are made

do {
$swapped = 0;
$| = 1; # do not buffer the following output
print ".";   # for trace / debug / interest purposes
for ($k=0; $k<@names; $k++) {
    for ($j=$k+1; $j<@names; $j++) {

        $temp = $oncourse{$names[$k]};
        $oncourse{$names[$k]} = $oncourse{$names[$j]};
        $oncourse{$names[$j]} = $temp;

        $newvalue = reportscore(0);
        # print "$newvalue, $value ";

        if ($newvalue > $value) {
            # print "*";
            $value = $newvalue;
            $swapped++;
            next;
        } else {
            $temp = $oncourse{$names[$k]};
            $oncourse{$names[$k]} = $oncourse{$names[$j]};
            $oncourse{$names[$j]} = $temp;
            }
    }
}
```

```
} while ($swapped);
$|=0;# turn buffereing back on

# "reportscore" works out the score; with a parameter of 2, it also
# sets up a hash called final that is keyed on course names, and each
# element contains a list of people on that course

$value = reportscore (2);
# tabulate();   # do this if tracing

# If this is the best iteration so far, or the fist iteration, store it

unless ($iteration) {
    $bestvalue = $value -1;
    }
if ($bestvalue < $value) {
        print "Improved to $value\n";
        $bestvalue = $value;
        %bestcourse = %oncourse;
    } else {
        print "Remains at $bestvalue (only achieved $value)\n";
    }

undef %oncourse;
undef %final;

}  # End of 10 x iteration loop

# Restore the settings that we saved during the best iteration, recalculate
# the scores and provide a full report of who is on what course.

%oncourse = %bestcourse;
reportscore (1);
reportscore(2);
tabulate();



############################################################################

sub reportscore {
    my ($trace) = $_[0];
    $skipper = 0;
    foreach $p (sort keys %oncourse) {
        $chnum = 0;
        for ($i=0; $i<=$coursemax; $i++) {
            $chnum++;
            last if ($request{$p}[$i] eq $oncourse{$p});
            $skipper--;
        }
        printf ("%10s is on %10s - their choice %d\n",ucfirst($p),
                     $oncourse{$p},$chnum) if ($trace == 1);
        $pstack = $p." ".("*" x ($chnum-1));
        push @{$final{$oncourse{$p}}},$pstack if ($trace == 2);
    }
    print "Measure of success in place people - $skipper points\n"
                     if ($trace == 1);
    return $skipper;
}

################################################################################

sub shuffle {
    my @input = @_;
    my @output;
    while (@input) {
        $posn = int(rand(@input));
        push @output,splice(@input,$posn);
        }
    return @output;
    }

################################################################################

sub tabulate {
foreach $course (sort keys %final) {
    printf "%-6s ",$course;
    foreach $student (sort @{$final{$course}}) {
        printf "%-10s",ucfirst($student) ;
        }
    print "\n";
    }
}

__END__

=head1The place_people program

This is a program which allows you to place people on their
preferred course and uses an I<Iteration technique> to improve
on its first results is everyone cannot be given their
first choice.

Inputs:

=over 4

=item 1

A file called B<requests> containing each persons name and
list of course ordered by their preference

=item 2

```

```
A command line parameter which is the maximum number of
people that can be taken on each course
```

21.2 Plain Old Documentation (POD)

You'll notice that we've commented our program fairly fully, and also included POD documentation at the end. Comments are for future programmers who have to access the code, and POD documentation is for the user. Here's how we extract the user documentation:

```
$ pod2text place_people
The place_people program
    This is a program which allows you to place people on their preferred
    course and uses an *Iteration technique* to improve on its first results
    is everyone cannot be given their first choice.

    Inputs:

    1   A file called requests containing each persons name and list of
        course ordered by their preference

    2   A command line parameter which is the maximum number of people that
        can be taken on each course

$
```

Other pod utilities could be used to output in man page, postscript, web page or other forms.

21.3 Possible enhancements

As with any short demonstration program, there are plenty of ideas and enhancements that are possible and probably advisable; here are just some of them:

On data validation

It would be sensible to check that the requested data file contains only the names of valid courses. If course names come in the wrong case (e.g. Perl not perl), our program really should cope with it!

We should check that we don't have two lines for the same person, or two people with the same name.

On data input and output

We should allow course names to include spaces and some form of comment to be added in to the requests file.

On extending the application

We should allow for courses of different capacities. This would probably be done via another data file that listed course and capacities. References to `$coursemax` in our code would all be replaced by references to a hash. The total capacity calculation we do early on would become slightly more complex.

Web interface input, input from an SQL database, output to a web page, output back to a database, output to the email system to let everyone know what they're on ... Perl is excellent glueware!

On the optimising algorithm

The placement of people who can't be given any course they have asked for could be modified. You might want to balance course sizes, or do the very best you can, or cancel slack courses.

In linear programming terms, you could add three-way swaps and see if you get any improvements from that. You could also do level swaps, that don't improve the score, in the hope that they will let you then find an improvement elsewhere in the possible solution matrix.

There are some efficiency issues. The current program will keep trying to improve on a score of "0" which is clearly impossible, and it will also try to improve on a score of "1" if one course is one place overpopular. We should test for at least some of these cases. We might also be able to find a way of knowing that no swapping is going to improve anything without having to go through a complete cycle of our loop and drawing a blank. The issues in this paragraph affect running speed only, and if it's fast enough for you anyway, do you really want to bother?

22 Libraries and Resources

There's a very wide range of modules, associated programs, documentation, web sites and so on and so forth, available that can help you make the most of Perl. This part of the training course is a road map of some of these resources.

Although the Perl language itself is very stable, web sites (in particular) seem to come and go. We checked all the URLs in this section as we wrote it, and we try to update them frequently, but this section of the course will go out of date much quicker than the rest.

22.1 Standard Perl modules

The Perl language has grown out of all recognition since Larry Wall wrote the first version in 1988. In the early days, most features that were added were "core" facilities that merged naturally into the language itself. Progressively, as Perl grew, added features became more and more specialised, and these days most new features are added as "perl modules", which are separate code from the main language itself, but included with the main distribution.

If you want to use a standard module, you'll load it in with the `use` statement (just like you load in your own modules), and then make use of the facilities it provides via method calls (if the module uses OO principles), subroutine calls, global and/or exported variables.

Pragmata

Some standard modules are "Pragmatic"; a module that's a pragma affects the compilation phase of your program and (indirectly) may also affect the execution. Pragma modules can be recognised by the fact that their names are lower case; they're invoked using `use` and can be switched back off using `no`. Example:

```
#!/usr/bin/perl

use integer;

$first = 17 / 7;

no integer;

$ second = 17 / 7;

print "first is $first\n";
print "second is $second\n";
```

```
$ ./wholenumbers
first is 2
second is 2.42857142857143
$
```

Figure 139
The integer pragma forces whole number calculations

Perhaps the most commonly used pragma is `strict`, which changes the rules concerning what the compiler considers to be legal code. By default, Perl is a language that assumes you know what you're doing and it allows many dubious coding practices. This is fine for a smaller program, but as a program grows into a bigger project you want to impose more rules. Apart from anything else, this helps protect you against coding accidents.

What does `strict` reject?

- Any variables which are not predeclared (using `my` or `our`), or are predefined by Perl, fully qualified (using `::` notation) or imported.
- Strings that are bare words. By default, you can get away with writing `$xxx = yyy;` in Perl and the string `yyy` will get placed into the variable `$xxx`.
- Symbolic references[1]

The strict pragma can be limited to apply only one or two of the extra tests, for example:

```
use strict "vars"; use strict qw(subs refs);
```

Other useful pragmata:

`constant`	define a constant (e.g. `use constant VAT_RATE=> 17.5;`)
`diagnostics`	control warning messages at compile and run time
`lib`	set libraries for loading modules
`overload`	redefine operators in a package (e.g. make + do something different!)
`warnings`	further control over warning messages

Standard modules

Having spoken about the rather special type of module known as a pragma, we'll go on to list some of the more normal standard modules supplied with Perl.

`Benchmark`	Time testing of code
`Carp`	`die` and `warn`, but reporting on line in calling module
`Config`	System information
`Cwd`	Find the current working directory
`Data::Dumper`	Get a printable string from Perl variables (see also `Storable` and `FreezeThaw` on the CPAN)
`English`	provide alternative English-like names for Perl special variables
`Exporter`	Make variables in a module available in the calling code
`File::Compare`	compare two files
`File::Copy`	copy and move files
`File::Find`	recursively find a file
`Getopt::Std`	Handle `- style` options on the command line (see also `Getopt::Long` for even more features)

[1] an advanced topic this one!

`IO::Socket`	Client programming; talk to a server (see also `Socket`)
`POSIX`	Provide POSIX standard identifiers
`Sys::Hostname`	Find the name of the current host computer
`Time::Local`	Convert second, minute, hour, day, month, year to seconds from 1.1.1970

Documentation is available from the standard sources on all these modules; once you've started to use modules, you'll find it easy. Here's a sample showing a few of them in use. Can you work out which subroutine or variable is associated with which module?

```
#!/usr/bin/perl

use Sys::Hostname;
use Cwd;
use Getopt::Std;
use File::Find;
use File::Compare;
use English;
use Data::Dumper;
use Benchmark;

(getopts('vd:') and @ARGV == 1)
        or die ("Usage $PROGRAM_NAME [-v] [-d directory] filename\n");
$dir = $opt_d || ".";

print ("Running on ",hostname()," from ",getcwd(),"\n") if ($opt_v) ;

timethese (1, {'perlcode' => sub {

find (sub {push @got,$File::Find::name if /$ARGV[0]/},$dir);
@got or push @got,"nothing!";
print Dumper(\@got) if ($opt_v);
print ("Looked for $ARGV[0] in $dir and found\n",join ("\n",@got),"\n");

$f = shift @got;
foreach $file(@got) {
   compare($f,$file) and print "WARNING - $file differs from $f\n";
   }
}});
```

And the result of running that is:

```
$ ./mod_demo -v -d /home/graham P219
Running on penguin from /home/graham/mar02
Benchmark: timing 1 iterations of perlcode...
$VAR1 = [
          '/home/graham/mar02/P219',
          '/home/graham/mar02/P219.txt',
          '/home/graham/mar02/.P219.txt.swp',
          '/home/graham/mar02/P219.bak',
          '/home/graham/manuals.bak/PERL/MODULES/P219',
          '/home/graham/couklogs/sitelist/subject.hold/topic/P219.html',
          '/home/graham/couklogs/sitelist/subject/topic/P219.html'
        ];
Looked for P219 in /home/graham and found
/home/graham/mar02/P219
/home/graham/mar02/P219.txt
/home/graham/mar02/.P219.txt.swp
/home/graham/mar02/P219.bak
/home/graham/manuals.bak/PERL/MODULES/P219
/home/graham/couklogs/sitelist/subject.hold/topic/P219.html
/home/graham/couklogs/sitelist/subject/topic/P219.html
WARNING - /home/graham/mar02/P219.txt differs from /home/graham/mar02/P219
WARNING - /home/graham/mar02/.P219.txt.swp differs from /home/graham/mar02/P219
WARNING - /home/graham/manuals.bak/PERL/MODULES/P219 differs from /home/graham/mar02/
  P219
WARNING - /home/graham/couklogs/sitelist/subject.hold/topic/P219.html differs from /
  home/graham/mar02/P219
WARNING - /home/graham/couklogs/sitelist/subject/topic/P219.html differs from /home/
  graham/mar02/P219
  perlcode:  1 wallclock secs ( 0.52 usr +  0.10 sys =  0.62 CPU) @  1.61/s (n=1)
            (warning: too few iterations for a reliable count)
$
```

22.2 The CPAN

Standard Perl stuff is built in to the language, and then things which are wanted pretty often are supplied as standard modules. Interesting items that are useful to some people, but outside the needs of most people, are to be found in the CPAN – the Comprehensive Perl Archive Network.

The CPAN is a library of modules contributed by Perl programmers (you and I could contribute if we wanted!) and freely available for download and use by anyone who wishes. Although anyone can upload any module to the CPAN, indexing of the modules is done by a couple of moderators who ensure that only useful, tested material will be found by people searching for a module to undertake a specific task.

Once you find a module that you want to use, you'll download it and install it on your system:

`cd` (into its directory)

`perl Makefile`

```
make
make test
make install
```

It might be that you need a C compiler such as gcc, or some other modules, or some other library or piece of code, as there are a lot of dependencies, but this will be explained in the "INSTALL" file.

There are thousands of modules available on the CPAN. Several years ago, O'Reilly published a "Perl Resource Kit" that included a CD with the then-contents of the CPAN on it, and it was hundreds of megabytes of source even in those days. No doubt these days it would be a volume of CDs. Some CPAN modules have so many capabilities that there are complete books on them, such as:

`CGI`	Manage HTML forms, Cookies, etc. (this is also a standard module in the latest release)
`DBI` and `DBD`	Interfacing to SQL databases
`Tk`	The Tk Graphics toolkit – a complete GUI

Other modules that we find of great interest to trainees include:

`XML::Expat`	XML Parser
`libXML`	Another XML Parser
`libXSLT`	XML Style Sheet
`libNET`	FTP, Telnet and other protocol support
`LWP`	Writing your own client (useful to grab something from a web page in your program)

Also available are a number of modules associated with handling Windows operating systems structures and other Microsoft information, such as `Win32::OLE` and `Win32::OLE::Enum`, which allow you extract text from (or modify and save away) a Word document or Excel spread sheet.

22.3 Utility programs

Moving slightly away from the Perl language itself, there are a number of useful utilities included with the distribution, such as

`a2p`	convert awk to Perl
`s2p`	convert sed to Perl
`pod2html`	convert in-line documentation in a Perl program into HTML.

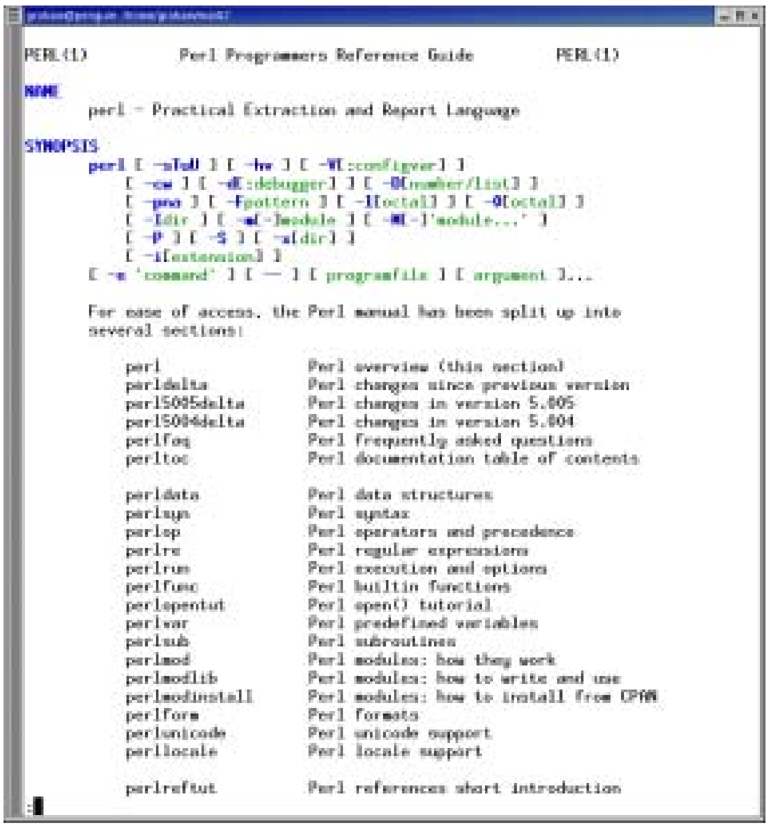
Also

`podchecker`	to check documentation
`pod2latex`	to convert to latex
`pod2man`	to convert to man pages
`pod2text`	to convert to plain text
`perlcc`	Perl to C convertor (experimental)
`perldoc`	Perl manual reader
`perlbug`	Script to report Perl bugs

Outside the distribution, you'll find a whole raft of extra programs available, ranging from text editors such as UltraEdit and KWrite which are Perl-aware, through to ActiveState's development environments.

22.4 Documentation

The Perl distribution comes with a complete electronic documentation set which can be installed as HTML pages or man pages on your system. If you're online all the time, you can use a central resource. Here are some examples:

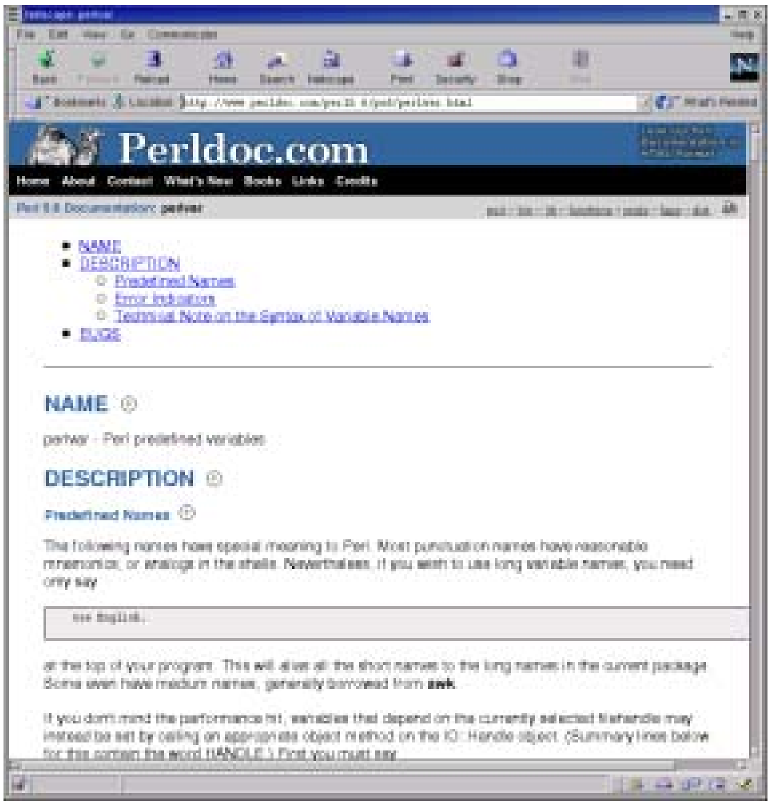


```
PERL(1)         Perl Programmers Reference Guide         PERL(1)

NAME
       perl - Practical Extraction and Report Language

SYNOPSIS
       perl [ -sTuU ] [ -hv ] [ -V[:configvar] ]
            [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
            [ -pna ] [ -Fpattern ] [ -l[octal] ] [ -0[octal] ]
            [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
            [ -P ] [ -S ] [ -x[dir] ]
            [ -i[extension] ]
       [ -e 'command' ] [ -- ] [ programfile ] [ argument ]...

       For ease of access, the Perl manual has been split up into
       several sections:

            perl                Perl overview (this section)
            perldelta           Perl changes since previous version
            perl5005delta       Perl changes in version 5.005
            perl5004delta       Perl changes in version 5.004
            perlfaq             Perl frequently asked questions
            perltoc             Perl documentation table of contents

            perldata            Perl data structures
            perlsyn             Perl syntax
            perlop              Perl operators and precedence
            perlre              Perl regular expressions
            perlrun             Perl execution and options
            perlfunc            Perl builtin functions
            perlopentut         Perl open() tutorial
            perlvar             Perl predefined variables
            perlsub             Perl subroutines
            perlmod             Perl modules: how they work
            perlmodlib          Perl modules: how to write and use
            perlmodinstall      Perl modules: how to install from CPAN
            perlform            Perl formats
            perlunicode         Perl unicode support
            perllocale          Perl locale support

            perlreftut          Perl references short introduction
:
```

Figure 140

The main perl "man" page

The page on special variables, as seen on the Internet:



Perldoc.com
Home About Contact What's New Books Links Credits
Perl 5.8 Documentation: perlvar

- NAME
- DESCRIPTION
  - Predefined Names
  - Error Indicators
  - Technical Note on the Syntax of Variable Names
- BUGS

NAME

perlvar - Perl predefined variables

DESCRIPTION

Predefined Names

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogs in the shells. Nevertheless, if you wish to use long variable names, you need only say

```
    use English;
```

at the top of your program. This will alias all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**.

If you don't mind the performance hit, variables that depend on the currently selected filehandle may instead be set by calling an appropriate object method on the IO::Handle object. (Summary lines below for this contain the word HANDLE.) First you must say

Figure 141

Special variables

Valuably, the documentation also includes the Perl FAQ.

Here's just a snippet from the contents list of the FAQ to show you how wide-ranging it is!

- `How can I remove duplicate elements from a list or array?`
- `How can I tell whether a list or array contains a certain element?`
- `How do I compute the difference of two arrays? How do I compute the intersection of two arrays?`
- `How do I test whether two arrays or hashes are equal?`
- `How do I find the first array element for which a condition is true?`
- `How do I handle linked lists?`
- `How do I handle circular lists?`
- `How do I shuffle an array randomly?`
- `How do I process/modify each element of an array?`
- `How do I select a random element from an array?`
- `How do I permute N elements of a list?`
- `How do I sort an array by (anything)?`
- `How do I manipulate arrays of bits?`
- `Why does defined() return true on empty arrays and hashes?`
- `How do I process an entire hash?`

22.5 Web resources

There are many web sites associated with Perl; we'll mention just a few here:

http://www.perl.com	Home of Perl - News, downloads, etc.
http://www.cpan.org	Comprehensive Perl Archive Network – modules library
http://www.perldoc.com	Perl Documentation and FAQs
http://www.perl.org	Perl Mongers – Perl Advocacy, user groups, etc.
http://www.pm.org	User groups – find nearby Perl users!

The following are commercial sites which, however, offer freeware and shareware downloads:

http://www.activestate.com	Perl for Microsoft Windows
http://www.indigostar.com	Perl for Windows, integrated with Apache

You'll find some resources on our web site too. For example, if you want to download the source of one of the examples on this course, go to *http://www.wellho.net/resources/index.html* and put in the name of the program that you want. Please read the disclaimer and limitation of use message that's on each example.

22.6 Newsgroups

UseNet News was one of the earliest internet services. It provides you with the ability to post a message to a public notice board, and to answer, or follow up on, messages that others have posted, either by adding another message after the original (and forming a "thread"), or by replying in email. There's a huge number of posting made every day to UseNet News, so messages are split into separate newsgroups by topic. It's up to you when you post or read to select the most relevant group.

Groups are hierarchically named, so that all computer-related topics should be in groups with names that start "comp.". Since Perl is a programming language, follow next on to "lang.", etc.

Most major service providers have a news server that keeps a copy of postings from the last week or two, and exchanges posts with other news servers. Thus, you should have nearby access to news no matter where you are. Traffic levels are so high, though, that ISPs limit the groups they carry, and they also tend to run filtering software; many postings don't make it all around the world.

Originally, messages were posted to a newsgroup via email, and then read by anyone who wished, using a mail client. These days, many users have moved on to accessing newsgroups via a web portal.

Perl information in newsgroups

Chances are that the question you want answered has been asked and answered already. But if this wasn't in the last week or two, your ISP's news server may no longer has what you want.

A number of years back, a company called "DejaNews" started to keep an archive of all postings to the majority of groups so that you could search; DejaNews became Deja, and was then taken over by Google, and this archive resource is now available via *http://groups.google.com*.

Here are some of the active, relevant groups that Google holds:

URL direct to group	**Group name**	**No. of threads**[1]
http://groups.google.com/groups?hl=en&group=comp.lang.perl	comp.lang.perl	44200
http://groups.google.com/groups?hl=en&group=comp.lang.perl.misc	comp.lang.perl.misc	257000
http://groups.google.com/groups?hl=en&group=comp.lang.perl.moderated	comp.lang.perl.moderated	9850
http://groups.google.com/groups?hl=en&group=comp.lang.perl.modules	comp.lang.perl.modules	36300
http://groups.google.com/groups?hl=en&group=comp.lang.perl.tk	comp.lang.perl.tk	21500
http://groups.google.com/groups?hl=en&group=comp.lang.perl.announce	comp.lang.perl.announce	1840
http://groups.google.com/groups?hl=en&group=alt.perl	alt.perl	22000

Google also holds a number of groups in languages other than English:

http://groups.google.com/groups?hl=en&group=de.comp.lang.perl.misc	de.comp.lang.perl.misc	27500	German
http://groups.google.com/groups?hl=en&group=de.comp.lang.perl.cgi	de.comp.lang.perl.cgi	30200	German
http://groups.google.com/groups?hl=en&group=fr.comp.lang.perl	fr.comp.lang.perl	34200	French
http://groups.google.com/groups?hl=en&group=it.comp.lang.perl	it.comp.lang.perl	17800	Italian
http://groups.google.com/groups?hl=en&group=fido7.ru.perl	fido7.ru.perl	21800	Russian

Groups are also available in Finnish, Dutch, Norwegian, and no doubt other languages, but these are typically not carried by the Google service and you should look initially at the ISPs in the countries where these languages are spoken.

If you follow the URL that we've given, you'll get the most recent posts, but Google also provides an excellent search facility so that you can comb through the data to find what you want. Although the information is a little raw, it's often very good, and you're far more likely to learn about down-to-earth matters with Perl through postings than through web sites.

[1] The number of threads column in the table indicates how many Google held in the group as of 1st March 2002

A note of caution

We strongly suggest that you read the newsgroups and look for the information you want there and elsewhere before you even consider actually posting your question. Remember that any message you post may be read by hundreds or thousands of people, and many of them won't hesitate to tell you that the question was asked last week, or that it's on page 474 of "The Camel Book".

You should also bear in mind that by posting to a newsgroup, you're putting your email address in the public domain, and you'll start getting a load of junk email as spammers have programs that harvest email addresses from newsgroups. You might wish to use a modified address and include instructions in the posting as to how to get your real address, or perhaps use a separate email address (e.g. Hotmail) for posting.

There's more to getting posts right ("netiquette") too. Remember you're publishing information and you should bear in mind taste, copyright, libel, etc. Read your ISP's acceptable user policy (AUP).

Here's an example of what to expect if you post off-topic, or fail to read the manual first.

```
> > > I am wondering if it is possible to disable mod_perl in a specific
> > > directory tree on a server that otherwise is running mod_perl?
> >
> > Yes simply switch to using the cgi-script handler rather than
> > perl-script handler.  (This, of course, has nothing to do with Perl).
>
> PLEASE expand on this answer.... my host is an idiot, and I am
> ignorant in the ways of apache server configuration.

There is a manual you know. We are all ignorant until we make the effort
to learn. If you are ignorant of how to do things like this yourself it
does not make sense to buy hosting from someone who is also ignorant. You
either need to know how to run your website yourself or buy your hosting
from someone who does.
Please do not use Usenet groups as a "read the manual to me" service;
especially don't use Perl newgroups as a "read the Apache manual to me"
service.
```

22.7 Chat

Internet Relay Chat (IRC) provides chatrooms; users log on to an IRC server (via an IRC client program), and then select the chatroom that they wish to visit. IRC servers are linked together, and you're onto a worldwide talking network.

The chat room associated with Perl is know as #perl, so:

`(`*`Start your IRC client`*`) /join #perl`

Anything that you type with a leading `/` is a command, and anything else is text that gets displayed on the screen of whoever else has joined `#perl` at the time.

IRC is an interactive way of getting specific help. Remember to be polite to the experts who are often to be found in this chatroom. This is a free service, and the least you can do is to be polite!

Other commands

`/who *`who is in this chat room?

`/quit`leave the IRC client

`/help`give a list of commands
`/rules`read the IRC rules
`/list`list channels

22.8 Books

Perl is well covered in books; we have about 60 in the library at our training centre, and there's a list and a description of most books at our web site: *http://www.wellho.net/resources/library.html*.

You won't find some of these Perl books in your local book-store, but almost any of them still in print can be ordered online. You'll find links on our website to take you to an appropriate store.

22.9 Meeting users, getting local support and training

User Groups

There are Perl user groups all over the world; once you start looking, you'll be amazed at just how many. Near us, there are groups in Bath and Bristol and Oxford and Southampton....

Start looking at *http://www.pm.org* and you'll find links. Some of them are very active, others less so.

List Servers

Some of the local groups run list servers to keep you up-to-date on their activities and other matters of interest. There are also other specialist list servers if you're going to be involved in Perl development or other topics beyond the scope of regular newsgroups.

Well House Consultants

You'll be reading these notes because you are or have been on one of our training courses (or a colleague has passed you our training notes so that you can teach yourself).[1]

We offer a wide range of Perl and other Open Source courses and we can probably help with your more advanced Perl training requirements. For one or two trainees, it's most cost effective for you to attend a public course at our training centre; for larger groups we can run a private course at our place or yours.

After your course, other questions may crop up. Please look in your notes, look in the books, do a quick search on Google as we described above – you'll find that most of your questions are answered. Do you have any colleagues who might know the answer? If you're still left with a question, you're very welcome to email me (*graham@wellho.net*), and I'll get back to you 24 to 48 hours later with at least a few pointers. Please bear in mind that this is a free service that we offer to trainees, and don't overuse it!

1 We KNOW this happens ;-))

23 Perl 6 Look Ahead

The rewrite of Perl 5 into the new Perl 6 language is under way. Now that Perl is a teenager, it's showing signs of age that would make it hard to develop it forward while maintaining compatibility with code that was written in its early childhood. So the very brave decision has been taken that this will be a new language. There will be good tools to help you convert your source, automatically in most cases. Perl 6 will have "Perl 5 modes" in one or two areas, and the underlying philosophies of Perl remain. The P of Perl will still stand for "practical". The language will still be eclectic, and will still assume you know what you're doing as you code. If you don't know what some structure does, it still probably does the sensible thing. Indeed a term for this, DWIM ("Do what I mean"), has been coined.

As I revise this module, early summer 2003, the first Perl 6 book is about to hit the shelves but the language itself isn't yet released even in a testing form. There has been a whole series of conversations and discussions via the RFC ("Request for Comment") mechanism, and the Perl community has made good input. Now, Larry Wall and team are working on the minutiae, and the coding of the language itself. In some areas (such as "Parrot"), there's already been code around for quite a while.

This doesn't mean that you should wait for Perl 6 before you do any more coding (you would have a long holiday!), nor that Perl 5 development has stopped. Currently, the 5.8 release is available for production use, a 5.9 development thread is starting which will include a few of the aspects also slated for Perl 6, and there's every intention that there will be a 5.10 production release which will be a final, stable Perl 5. Perl 5.10 will probably survive for years, just like Perl 4.036 did (or should I say "does"?). Rather, marvel at the sheer genius of what Larry's come up with, then go forward with Perl, confident that it's not about to be usurped by some language such as Java or C#.

23.1 Objects

In Perl 6, everything is an object, and has *PROPERTIES*. That doesn't just mean that what you create is an object; it means that the Perl built-in types are objects too, and you use methods to set or discover things about them, or get them to do things. You want to get the number of elements in a list?

```
@aaa.getprop(length)
```

should work for you, but then being Perl, that will be shortened down to:

```
@aaa.length
```

if you like or even:

```
+@aaa
```

(OK - ask about that last one later.)

There's a proper `class` keyword to define classes, you can still define Perl 5 packages if you wish, and you still bless the main structure using the `bless` function. You will be forced to use the two-parameter form of `bless`, by the way. You can also define extra properties that will be maintained for each object separate from the blessed structure, using the `attr` (for attribute) keyword.

Inheritance will be defined using the `is` keyword on the class declaration, rather than the kludged `@ISA`; thus:

```
       class dog is animal
```

Since everything's an object, with attributes, you can read and write some surprising settings on something as simple as a scalar. Let's say that you want to set a variable to contain a numeric value, and have that value be true.

```
  $demo =16;
```

Yes, that's true in Perl 5 and also in Perl 6.

```
  $demo = 0;
```

Oops, that has a value of zero, but if you test it in Perl 5 it will give you a false value. That might not be what you want. You might want it to be true even though it's zero. So in Perl 6:

```
  $demo = 0 but true;
```

and the attribute is set. If you wonder about things like this, think how useful it will be to return a value from a subroutine, and also to return a flag to say "yes, this is a valid number" even if the number is 0.

23.2 Operators

String handlers

`$(....)` will evaluate an expression in a scalar context, and `@(....)` will evaluate an expression in a list context. So:

```
  print ("Please enter number ",$n+1,": ");
```

becomes

```
  print ("Please enter number $($n+1): ");
```

The old concatenation operator, "`.`", has been taken away to become the method operator. Really the only sensible option as it's become such a standard in most other languages. A new operator "_" replaces the "`.`" for string concatenation. Since _ is valid in a variable name, it'll have to be used with white space just as has always been the case with operators such as "`x`".

Here documents always had to be coded up against the left margin. This is no longer so, as you can inset the final delimiter by a certain number of spaces, and then each line will be considered inset that same amount. Other changes allow you to place the `;` on the terminator line, and force you to quote the initial terminator. Thus:

```
  while ($j = pop (@stuff)) {
        print <<DONE
  There is a record called $j
  which is printed here
  DONE
           ;
        }
```

```
print "And that's your lot\n";
```

becomes

```
while ($j = pop (@stuff)) {
      print << "DONE"
           There is a record called $j
           which is printed here
           DONE ;
      }
print "And that's your lot\n";
```

Comparison operators

Ever wanted to write `if (1 < $x < 10)` ...?

Well, it will work in Perl 6. But what will it do? It will do what I mean – DWIM. In this example, it'll check that the contents of `$x` has a value (in a numeric context) that's between 1 and 10. You could also write:

`if ($a != $b != $c)` ...

but you'll need to be careful, that will check that neither `$a` nor `$c` is equal to `$b - $a` and `$c` *could* have the same value and the whole expression will still be true. By the way, `$a` and `$b` are no longer special variables.

Ever got yourself mixed up between == and `eq`? Ever found yourself comparing numbers when you should be comparing strings? Enter the smart match operator:

`if ($a =~ 20)`	That's numeric
`if ($a =~ "Twenty")`	That's an exact string match
`if ($a =~ /dog/)`	That's "contains"

You might have been worried until you saw our third example. The =~ operator has widened from pattern match; it still does pattern matches, but it does 34 other things too, depending on what it's asked to compare to what ...

`if (@a =~ 29)`	Is there a value "29" in the list `@a`?
`if (@a =~ @b)`	Are lists `@a` and `@b` identical?
`if (%a =~ "dog")`	Is there an element keyed "dog" in `%a`?

Perl 5's `or` operator (and the `||` operator too) work a treat for giving answers "a" or "b" or "c" except when one of the input values is false. You may well have written:

`$abc ||= 16;`

to set `$abc` to 16 if it didn't already have a value. You may well have been caught by the trap that if `$abc` already existed. But with a false value, that value would be overwritten with the number 16. Enter the `err` or `//` operator[1]:

`$abc //= 16;`

to set `$abc` to 16 if it doesn't have a value yet.

`err` and =~ may look exotic, but we rather suspect that they'll be the operators of choice in Perl 6 coding once it's established, and operators such as ==, `eq` and `or` will gently fade into the backwaters.

[1] `or` with a slant to it

Vectorised operators

You want to add one to every element of a list? Write a loop? Use the `map` function? Sure, you can do either, or you can use a vectorised operator. If you add a `^` in front of an operator, it will apply the operation to every element of the list, and if either of the operands is shot of values, as it would be in the case of a scalar, that operand is stretched.

Thus:

```
@result = @x ^- @y;
```

gives you a result list where every element of y is subtracted from every element of x. and:

```
@result = (@x ^+ @y) ^/ 2;
```

computes a list of averages and:

```
@result ^+= 1;
```

adds one to each element of the result list.

23.3 Data Types

File handles aren't a special limited type of variable any more. They're objects that can be held in scalars. You want to open a file?

```
$fh = open("abc.txt");
```

Typeglobs are gone too, replaced by bindings (see *23.4 Bindings*).

Internally, Perl 6 will still use long and double integers, but it will no longer get into trouble when you burst the limits of these data types. Internally, it will switch to using arbitrary precision and accuracy numbers of type `BIGINT` or `BIGNUM`. Significance for the programmer? Usually none; the change is automatic and invisible. For sure, code will slow down when big numbers come into play, but at least you'll get the correct answers.

Hashes are implemented as tables of pairs. Don't worry too much about that one at the moment, but note that you can write

```
@stuff = %hash.kv;
```

to get a list of key, value, key, value, where in the past you would just have written

```
@stuff = %hash;
```

23.4 Bindings

The `:=` operator allows you to bind an (extra) name to a variable.

In any version of Perl, a variable is comprised of two parts – the memory that's used to hold the value(s) associated with the variable; and the name itself, held in a symbol table. The memory management system keeps note of how many references there are in the symbol table, directly or indirectly, to each variable's data, so that memory can be "garbage collected" from time to time.

In addition to the `$` and `\` operators, which will still be available, Perl 6 introduces a new operator `:=`. It's known as the binding operator, and it means "is also a name for...". Let's see an example:

```
$x = 16;
$y := $x;
```

```
  $x++;
  print $y;
```

will print out the value 17.

Great theory. Practical uses? Many, many, many! Try this for starters:

```
  (@a, @b) := (@b, @a);
```

Swap over the names of lists `@a` and `@b`. Very efficient, no need to start copying great swathes of data around in memory.

Subroutine bindings

Yes, you can still use `@_` if you want to write job protection code, but it’s much better to use named parameters. Remember:

```
    sub somename {
         my ($first, $second) = @_;
         $third = $first + $second;
    etc.
```

Well, do you prefer this?

```
    sub somename ($first, $second) {
         $third = $first + $second;
    etc.
```

Much better. But you can go further:

```
    sub somename {
         $third = $^first + $^second;
    etc.
```

Perhaps we had best explain. If you start a variable name `$^`, you're referring to a calling parameter for the block your in,[1] and Perl will Asciibetically sort all such variable names so that you don't need to declare them at all. Perl will work out the order.

If you're porting your own sort subroutine from Perl 5, just replace `$a` with `$^a` and `$b` with `$^b`. Perl 6 now uses parameters to collect the two elements it's to compare, and no longer has that wart of the special variables `$a` and `$b`. This also means that you can run a sort which involves another sort internally; old global conflicts are gone!

You can now call subroutines with multiple lists, such as:

```
  myjob(@list1,@list2);
```

and they won't get slurped together into a single list in the subroutine. If you do need to have a "slurpy list" type parameter, an extra `*` preceding the variable name will give you that:

```
  sub example ($one,$two,*@three) {
```

declares a subroutine with two scalar parameters, and then all the rest of the parameters will get slurped into the `@three` list. By contrast:

```
  sub example ($one,$two,@three) {
```

declares a subroutine that takes three parameters; calling it with more would be an error. Yes, this does open the way for Perl 6 to define a number of subroutines all with the same name, and have different ones called depending on the calling sequence. Notice a similarity to Java's overloading? Syntax and rules for this capability are still being worked on.

[1] all blocks are subroutines now

You may have come across examples of methods being called in Perl 5 where a hash is passed in, the keys being used to instruct which parameters are set. Perl 6 has a direct => operator that can be used in the call to pass in a pair, which is a new data type. A hash becomes a table of pairs.

"All blocks are subroutines" - remember? You can declare:

```
$var = { code ...........}
```

and then

```
if ($j !~ 10) $var;
```

Finally, you can bind a subroutine with an assumption. Let's say that you have a subroutine that returns one number divided by another:

```
sub div {$^x / $^y}
```

and you want to define another subroutine that returns you a reciprocal, without the need to rewrite all the code.

```
$rec := &$div.prebind(x => 1);
```

or

```
$rec := &$div assuming x => 1;
```

23.5 Conditionals and loops

Topicalisation

"What?" you ask, with a shudder. OK, we're talking `$_`. You're used to leaving out a variable name in Perl 5, and having it work with the contents of `$_`. We make a bit of a game of this on our beginner's courses for Perl. Like everything else in Perl 6, an excellent concept has been breathtakingly extended.

Switch statements

No, you still don't have a Switch keyword in Perl 6. Rather, you have a `given`. Let's try an example:

```
given $val {
      when 17 { code to perform}
      when 19 { code to perform}
      when "n/a" { code to perform}
      when m:i/end/ {code to perform}
      default {code to perform}
      }
```

The `given` statement topicalises `$val`,[1] and then each when statement does a smart match. If `$val` works out to the number 17, the first block is performed; if it works out to 19, then the second block is selected instead. If it contains the text "n/a", then the third block is performed, and if it matches the regular expression `/end/`, ignoring case, then the penultimate block is performed. No match at all, and the final block after the world "default" is performed.

You'll note that there's no need to bracket the condition after the "when"; that's now a common feature of all conditional and loop statements. For sure, you can write

```
when ($ab)  {print "It matches \$ab\n";}
```

but you can also write:

```
when $ab  {print "It matches \$ab\n";}
```

1 write `$_ = $val` if you like

There does have to be a space before the `{` character; that's how Perl 6 can tell a member of a hash apart from a code block. Seems a small price to pay; after all, whoever wants to leave spaces within variable names?

At the end of a successfully completed `when` block, Perl 6 will jump out of the given construct. There's no need to specify a break,[1] or a `breaksw` or anything like that. Larry Wall has provided you with a `continue` statement that lets you continue on to the next "when" if you really want to.

Have a look at this:

```
given $val {
        when < 10 { print "Much "; continue}
        when < 21 { print "Reduced Rate"}
        $fullfare++;
        when > 65 { print "Senior"}
        print "Regular"
        }
```

for and loop

Perl 6's `loop` command replaces the traditional `for` loop of Perl 5, and C and many other languages. At its simplest:

```
$value = 1;
loop  {
       $value *= 2;
       print "$value \n";
       $count ++;
       $count < 10 or break;
       }
```

or a little for complex:

```
$value = 1;
loop ($count=0; $count<10; $count++) {
       $value *= 2;
       print "$value \n";
       }
```

The `for` loop is now an iterator that lets you step through something (at its simplest, a list):

```
for @data { print }
```

topicalises each member of `@data` in turn, and passes each to the subroutine that's declared as the thing to be performed. If you don't say anything about the parameters to a subroutine, the first parameter is taken to be `$_`, and `print` defaults to printing out `$_` as it always has done!

If I want to name my variables in a `for` loop, I can do so:

```
for @data -> $scvar { print $scvar}
```

and:

```
for @data { print $^scvar}
```

will both work, and I can also write a more complex structure:

```
for @x,@y -> $m,$n {
        print "$m $n\n"}
```

[1] you can if you wish, though

to iterate through `@x` and `@y` at the same time. This latter example will print out element 0 of both lists, then element 1 of both lists, and so on. You want to see another?:

```
for @x;@y -> $r {
      print "$r\n"}
```

will print out alternate elements of `@x` and `@y`. You may spot some similarity to iterating through a list in Tcl if you're familiar with that language.

Finally:

```
for %demohash { print .key}
```

will let you print out all the keys of a hash. If you're wondering, the `for` statement topicalises each element of the hashtable in turn. Then any method that isn't told what to run on (in this case the key method, told to run, but not told what to run on), runs on the current topic.

23.6 Exception handling

Perl 6 will return any errors as objects in `$!`. That's much simpler than the use of `$! $@ $^E` and – oh I forget the other ones – in Perl `%`). You can then run methods on the object to learn more about the error. Some of these methods use the "." notation, and others use a "Sigil" character in front of the variable to force a context. Thus:

`+$! $!`	as a number
`_$! $!`	as a text string
`^$! $!`	as a boolean

These notations are available elsewhere in Perl 6 too; the whole language is much tidier with far fewer special cases.

Try blocks

When you run a piece of code, it may work or it may fail. If it fails, you want to trap the failure. You can do this in Perl 6 using a `catch` block.

```
@demo = (10,20,30,0,50);
try {
      CATCH { print "Oops - division failure\n"}
      for @demo {print 1/$^each}
}
```

The loop attempts to print the reciprocal of each of the members of a list, but if it fails, the error is caught and an error message is printed instead. If the whole `try` block runs successfully (if, say, our demo list didn't include a zero element), then the catch block would never be run.

You'll notice that the `catch` block is inside the `try` block; using that structure, the `CATCH` block has access to locally scoped variables within the `try`. This is a much better solution than you'll find in Java, for example. The word "CATCH" is capitalised; by way of explanation, any block that is capitalised is set aside for possible later use – it's a trap, an error handler, an initiator or whatever. You may have come across `BEGIN`, `END` and `DESTROY` in Perl 5; there were also `CHECK` and `INIT` blocks. In Perl 6, not exclusively associated with `Try`, you also have:

`PRE`	Condition that must be true on block entry
`POST`	Condition that must be true on block exit

(These two are very useful during code development)

`KEEP`	To be performed on block exit if it succeeded
`UNDO`	To be performed on block exit if it fails
`FIRST`	First time through a block only
`LAST`	Last time through a block only

Blocks such as `CATCH` can take a condition, so that they'll only catch certain things. You can supply multiple `CATCH` blocks within a single try. Where you place the blocks within the `try` is up to you.

23.7 Rules and grammar

If you're a newcomer to Perl, you'll find regular expressions powerful but obtuse and hard to learn. If you're an old hand, you'll swear by regular expressions but you'll want so much more.

Perl 6 doesn't describe things as "regular expressions"; they are "rules". A series of rules can be combined to make up a grammar. Perl 6 itself can be defined as a grammar, and you can amend and alter that grammar if you want, thus changing Perl with minimal coding. Don't try that yet.

In rules (previously known as regular expressions), some things remain unchanged. Literal letters and digits still match exactly, such as `\` in front of a special character will cause that to match exactly. You'll have grouping with `()`, alternation with `|`, and counts such as `* +` and `?` (greedy) and `*? +?` and `??` (sparse) just as you're familiar with.

Modifiers

Rule modifiers are now written at the beginning of the rule, and not at the end. So that:

```
if ($x =~ /end/i) {....
```

becomes

```
if $x =~ m:i/end/ {....
```

The old `/x` modifier has gone[1] and so have the `/s` and `/m` modifiers; they should really have been defined within regular expressions in the first place, and not as modifiers.

The `/e` for execute modifier on `s` has gone. Just look at this elegant solution:

```
$j =~ s/(\w+)/ucfirst(\1)/eg;
```

becomes

```
$j =~ s:g/(\w+)/$(ucfirst($1))/;
```

using the new `$(...)` notation that we've already seen elsewhere in Perl 6.

Other modifiers:

`:c`	continue (carry on, rather like `g` in a scalar context)
`:o`	once only
`:w`	any white space in the expression is `\s*` or `\s+`, this latter if the white space appears between two words
`:p5`	Please use Perl 5 Regular expressions!

[1] by default, and white space in a regular expression is now a comment

On `s`, you might want to use:

`:4x`	do 4 substitutes
`:3rd`	substitute the third match
`:e3rd`	substitute every third match

Elements with a rule

Assertations

`^` and `$` match the beginning and end of the string, just as they always did unless you had used the `/m` modifier. You're also provided with `^^` and `$$` which will also match at the beginning and end of embedded lines. `\a \z` and friends have gone.

Grouping

`( ...)` is still a grouping. Did you like `(?: .... )` as "group but don't capture" in the past? Thought you didn't! You can now use `[.....]` for group but don't capture.

You can also use `{ ...... }` to embed some code within a regular expression. Thus:

```
when /(\d+){$1 < 256 or fail}/
```

if you want to check for a numeric value, or series of digits, that works out whether or not a number less than 256 is present.

Variables within rules

A scalar variable within a rule is matched literally, thus:

```
$var ="Hello?";
if (/$var/) {
```

will match `$_` for the literal text H-e-l-l-o-? in Perl 6, whereas it would have matched H-e-l-l followed by 0 or 1 letter o's in Perl 5. If you include a list in a Perl 6 regular expression:

```
if (/@lis/) {
```

you'll look for a match to any member of the list.[1] If you include a hash, you'll check for the existence of an element with the given key.

These extra facilities for variables will save you a lot of `\Q$var\E` type stuff, but in any case the `\E` has gone in favour of non-capture bracket groups:

```
\Q[Hello?]
```

for example.

Metasyntax

If you're wondering what has happened to the old character groups that used square brackets, welcome Metasyntax, written between `<` and `>` characters. Metasyntax includes a wide range of possibilities, including

`<sign>`	a sign character
`<'literal'>`	a literal piece of text
`<$var>`	interpolated variable (if you *really* want to!)
`<ws>`	white space - just like `\s+`
`<sp>`	a space character
`<(....)>`	a code assertion

[1] a very neat way of searching to see if something's present in a list

Capturing

When you want to capture groups in a rule, you can do so using syntax such as:

```
(\d+){let $num := $1}
```

which you can simplify down to

```
$num := (\d+)
```

Very neat – naming and capture all within the rule (no assignment of a list necessary), and the ability to capture nested brackets if you wish.

Concluding rules and grammar

There's so much that's new in Perl 6's rules and grammar, (we haven't described `<commit>`, `<null>`, `<prior>`, `<before ...>`, `:`, `::`, `:::`) that for most users the best advice is "it's good, but wait until you have a rule engine in your hands to experiment with before you try and learn the whole thing". Uniquely amongst all the features that we've described in this document, the features and changes are so far-reaching that it's probably best to learn rules as a new topic rather than to try and convert your existing regular expression knowledge.

23.8 Under the bonnet

With Perl 5, you wrote your source code into a file, and said "run that file". You'll do the same thing with Perl 6.

Internally, Perl 5 converted your script into a series of operations and opcodes via a compiler, and it then used the resulting Btree to run your program. Although it wasn't initially called a "virtual machine", that's really what it was.

Perl 6 will use the new "Parrot" virtual machine, which is being designed and written in parallel with the language specification and development. There are already parrot assemblers and test code available, that's been the case for quite a while now, and Parrot will support other languages other than Perl.

As well as having a shiny new car in Perl 6, you'll have a fresh, clean-burn engine that can run on other fuels too.

23.9 Conclusion

Perl 6 is an exciting rewrite of the Perl language, adding facilities to take it forward for the next generation. It's well thought out, clever, powerful and follows the philosophies of earlier Perls that we've grown to know and love. It bravely adds facilities, and it bravely breaks compatibility where that was seriously necessary for the future.

Perl 6 is exceptionally feature-rich, and it's not going to be the sort of language you can learn and become fluent in overnight. It's going to take time to learn, and you're going to need to use it with care. As Damian Conway, one of the key players in writing Perl 6 said at a recent lecture in London: "We're giving you all this power in Perl, now go out there and use it carefully."

[Disclaimer. Although every attempt has been made to correctly interpret the information that we have on hand on Perl 6, we cannot take any liabilities for any errors or omissions. You should also be aware that Perl 6 is currently under development, and there is a chance that some elements described may change prior to its production release.]

24 A Quick Look Ahead

24.1 Fundamental and advanced topics

There are many aspects of Perl that every Perl programmer needs to know about and understand to some extent. The fundamental elements of the language all mesh together to provide a tool with a wide variety of uses. Among these fundamental aspects, we list:

- scalar variables, calculations and assignments
- commenting your program
- string handling functions and operators
- conditional and loop statements and operators
- input and output functions and file handling
- collection variable types (lists and hashes) and associated functions
- regular expressions and their use for matching and extracting data
- subroutines for splitting code into manageable sections and multiple files
- special variables and topicalisation
- documentation and CPAN sources, books and where to look things up

Once you get beyond these fundamental aspects of Perl, you'll learn that it can do much more too. However, the extras needed by a programmer wishing to extract data from an XML data file will differ greatly from the extra required by a programmer who's using Perl as part of a really large project. And there will be a third set of extra for the programmer who wants to provide a GUI (Graphic User Interface) on a Perl application, and a fourth for the programmer who wants to use Perl to present information in a browser window.

The purpose of this module is to give you a roadmap ahead, making you aware of some of the more advanced features of Perl but not providing you with full training on them. To give you an idea of the size of some of these extra facilities, there are complete books on subjects such as *Perl and XML*, and *Perl and the DBI*[1] just to quote two examples.

How do these further facilities talk to Perl?

Some of the additional facilities that we'll tell you about are built in to the Perl language itself. Others are supplied as standard modules or modules that you can download from the CPAN (Comprehensive Perl Archive Network). Further programmer's facilities are available from third parties, and you can build Perl

[1] See Appendix: WHC Library for further details on Perl books

into your own C application, and your own C code into Perl. Remember, there's always more than one way of doing it with Perl.

24.2 Other facilities in the Perl language

File handling and system administration functions

Built into Perl, you'll find all the standard facilities of any language that allow you to open, close read, write and append files. You'll also find a whole series of operators relating to file testing – operators like the `-f` operator to test if a string is the name of a plain file.

Perl users want to be able to do more with files than just read and write them; they want to be able to parse through all the files in a directory, for example. There are standard facilities, for example, `opendir` and `readdir`, to let you do this. Used in careful combination with other file test operators, you can easily roll your own program to pass recursively through a directory tree, and the file module provides facilities like `File::Find`. It's shipped as standard with Perl these days.

There are functions in Perl to change file ownerships and permissions, rename and delete files, and create and delete directories. Although these functions all look very linux-like, they do work on other operating systems as the Perl Porters have carefully implemented to perform the morally equivalent function on Windows or MacOS that they perform on Unix and Linux. You are strongly recommended to use these functions to perform your systems admin tasks from within Perl, not only for code portability but also for efficiency. It's much better than using a `qx` operator.

Object Orientation

As programs grow in size, they become harder to maintain. It becomes natural for larger applications to split them down into a series of parts, and have each part dealing with a certain aspect of the application – a particular data type, or "class of object" as you might call it. If all such objects are referenced only through a set of subroutines or methods provided in a single place, then it means that the main application programmer doesn't really need to know about the internals at all, he simply calls a limited menu of subroutines.

This approach to programming has been with us for a very long time indeed. I was using it in the 1970s in a Fortran application, and you'll already have used file handles in Perl that are really a specialised sort of object. Think about it; you don't directly use the contents of the file handle variable directly, but rather you use the file handle as a parameter to one of a limited number of functions and operators such as `open` and `< >` without knowing which tracks, sectors or cylinders are being used on the disk. That extra information is hidden (encapsulated) within the file handling class.

For the right application, object orientation is a great concept. Languages like C++ and Java have special syntaxes that are designed to enforce rules on you, preventing you from rolling up a sleeve, reaching in and playing with internal values. Perl assumes you know what you're doing, but never the less, a special syntax is provided that lets you write shorter, cleaner code to the Object Oriented idiom if you wish, and at the same time it complicates the syntax you need to use to reach those internal variables so that you're hardly likely to do it by accident.

Because objects are such a handy way of wrapping up and distributing code, you'll find many of the calls you want to make to standard or CPAN modules use the idiom, and for this reason we consider the writing of code that uses existing modules to be a Perl fundamental. Providing your own types of object is definitely a more advanced subject; it requires understanding of further technicalities of Perl, as well as considerable thought to the design process. And not everyone needs to define their own objects.

Writing distributable modules

If you want to reuse code across a variety of applications, that's great! An Object Oriented interface allows you to hide within (oops, *encapsulate* within) all the complex code, and provide your user with a simple, easy-to-use program interface. So far, so good.

Your user will then need to keep your reusable code – your module – in a directory that's apart from Perl itself and also apart from his application code. He'll need scripts to install the module, and I'm sure he'll want some documentation into how to use it. Of course, user "A" wants his documentation in postscript, and user "B" wants his documentation in HTML, so if you're not careful, you'll end up spending more time wrapping, packaging and documenting your module than you spent writing it in the first place. There are two things you would know about to help you:

POD, or Plain Old Documentation, is a format in which documentation can be embedded within Perl source code. With the standard Perl distribution, a number of utility programs are provided with names like *pod2text*, *pod2html*, and *pod2ps* which let your user generate the documentation he needs in the format he wants.

The *h2xs* utility, also provided with the Perl distribution, generates a skeleton for new modules, including the actual module file, complete with a few comments and POD headers to get you started, a makefile that gives your users quick and easy ways of installing the code, and the shell of a test program which you can provide to your users to let them check that they've installed the new module correctly.

More complex data structures

Lists and hashes are fundamental structures of Perl, but you can go further. You can have lists of lists, hashes of hashes, or you can design your own more complex structures. You do end up writing some quite interesting-looking code, strings of `$ @` and `\` characters sometimes, with plenty of `( )` `[ ]` and `{ }` delimiters, but the power is incredible. Some strong advice as you get into this:

- use a pencil and paper to draw a diagram of what you're trying to do

- encapsulate your more complex structures into objects (that way, the user calling them will be protected from the complexity and is unlikely to damage the structure)

You are limited only by your own ingenuity. And remember the saying "design *matters*".

Tieing

What happens to a scalar when you leave your program? "Its contents are lost," you answer. That's the conventional logic, and, yes, it does usually work that way. But you can change this.

With a scalar variable, deep in the bowels of Perl there are just four functions – `create`, `destroy`, `read` and `write`. If you choose to, you can provide your own subroutines to perform each of these tasks for a specific variable, thus altering the behaviour of a variable. This is known as "tieing". As an example of its use, you could tie a scalar to the contents of a file so that every `read` or `write` is a file read or write. Slow, but the variable would persist from one running of your program to the next!

Tieing is also implemented for hashes (the most common use of the facility) and also, in recent versions of Perl, for lists and even for code.

Writing network clients and servers

Perl originated from a requirement of Larry Wall's to write a utility that was network-aware, so its network programming capabilities have always been very strong. Client programs can open network connections almost as easily as they can open files. If you know the protocols involved well enough, you can hold an automated conversation with a server on any port that you wish. For those of you who don't know, and don't want to know, the bit and byte-level stuff, a host of modules has been written and is available on the CPAN to let you write network clients using protocols from HTTP to SMTP, and from FTP to SNMP.

Writing a server is a little different until a connection is established. Under normal circumstances, you need to have a background process (running as a daemon) waiting for a connection. You don't know where that connection will come from and, when it does, you need to authorise it and usually spawn a child process to handle the request so that your main process can continue to monitor for new incoming requests.

Of course, Perl can do all of this for you.

Once the connection is established, Perl will talk to the client just as described in the earlier paragraph. Once again, you'll find a number of server protocol packages available on the CPAN if you don't want to roll your own.

Binary Termio, and low-level file controls

The standard Perl language is rich in functions that provide all sorts of binary and low-level controls if you need them. Functions such as `pack`, `unpack` and `read` allow you to handle binary data with ease, and the standard strings can hold any bit combination that you like. Operators such as `&` `|` and `^` let you perform bitwise operations.

Files can be opened and accessed at a low level using functions like `sysopen`, and controlled via `select`, `ftell` and `fseek` if you want to perform what used to be called "random access". There is cooperative file locking support if you're likely to have several users accessing the same file[1] at the same time.

External signals can be trapped if you don't want a user's `^C` to kill your Perl program. There are some 30 different signals with all sorts of other useful functions such as allowing processes to alert each other in certain events. You can even set an alarm clock and have your process alerted after a certain time.

For most applications, keyboard input consists of reading a string of text up to a new line character and letting the operating system take care of user's corrections via the backspace key, etc. If you want to jump right in there, you can arrange to pick up inputs character by character up to a different terminator, do your own cooking of edit characters, and even look ahead to see if the user has typed anything so that you don't need to stick on the keyboard while your user types, but can carry on doing background processing.

Before you start writing too much Perl that uses these facilities, do think if there's something suitable on the CPAN for you and that you're not reinventing the wheel!

Data Munging

So much of the power of Perl comes in its ability to manipulate data. After all Perl is the 'Practical Extraction and Reporting Language'. The term "data munging" has grown up to cover this side of its capabilities.

Many data munging capabilities are listed in the Perl fundamentals. Its string handling is second to none, its regular expressions are incredibly powerful,[2] and its collections data structures – lists and hashes – can be handled as a whole through functions that operate on every element, rather than you having to write longer and slower code in the form of loops.

Beyond the fundamentals, functions such as `map` and `vec` are worthy of note, and it's probably worth your time studying some of the programming techniques that can be applied using them – `grep`, and `for` and all the other functions you've learnt about – to make for really effective data munging. Even if you're an experienced programmer in another language, it's worth taking time to study new tips and techniques that can make really effective use of these functions in some amazing ways. See *The Perl Cookbook*, *Effective Perl Programming*, *Perl - the Programmer's Companion*, *Data Munging with Perl* and other books of that ilk.

[1] some to write to it

[2] but just wait till you see Perl 6!

Other built-ins

There are other built-ins that we haven't mentioned, things like data formatting with the `write` function come to mind, but in the fundamentals list at the top of this module, and the other introductions above, we've brought to your attention most of the mainstream topics that are included in the base language.

24.3 Other facilities in Perl – further modules

So you want to read a database? You want to handle XML? You want to write a graphic user interface (a GUI)? Surely this can be done in Perl. After all, the philosophy of Perl is that it embraces technology.

The answer is "yes, you can", "yes, you can" and "yes, you can". BUT ...

Only a proportion of Perl users want to, for example, read a database. Further, the main authors and maintainers of Perl, brilliant though they are, are not experts at every database package and the detail of how it works. So the facilities to handle databases are not built into the core of Perl, where they would clog up the language for non-users, and where they would have to be maintained by a team with other interest and skills. Instead, they're provided in the form of modules, either distributed with the main Perl distribution if they're very stable and unlikely to change, or on the CPAN if they're of more specialist interest, or support a fast-evolving technology. By keeping such modules on the CPAN, it means that new versions don't have to wait for the next Perl release to get out there.

There are thousands of modules on the CPAN, of all different shapes and sizes. Before a module is indexed, it's reviewed by a team of moderators, so if you find something in the searches, it's usually pretty good. It will include documentation and source so you can see what it does, and even amend your copy, making suggestions back to the author. In this section, we'll introduce some of the most significant modules.

Note that almost all the modules we're going on to talk about are called through Object Oriented interfaces. This is not some fad, it's practical. The OO interface allows the definition of a specific series of method calls which allows the module author to provide a series of carefully crafted ways in – doors – to his code, while also allowing him to hide the internals from unintentional or accidental access. The OO interface also provides the mechanism to handle several instances at the same time. In other words, a database application using an OO-based module can talk to two or more databases at the same time if that's a requirement.

Interfacing to databases

In pure computing terms, a database is any structure that holds data, so even a plain text file counts. But that's not what we're talking about. Here, we're talking about data that is arranged into tables, each of those tables having rows and columns, and the table typically held in some internal format which makes it access and update much more efficiently than handling a plain text file.

There are a number of database formats supported by Perl modules, with names like "GdbM" and "NDBM". Some of these formats are core to operating systems such as Linux and Unix. NDBM files, for example, are used in mail aliases, NIS tables, and X Windows colour tables. With Perl, you can access these tables directly and, if you have appropriate permissions, actually update the live information.

Relational databases, such as Oracle, MySQL, SQL Server, and Sybase, comprise a number of tables where an entry in one table can be used as a key to another table to avoid the duplication of data entry. Such databases are usually controlled by a single daemon running as a server, thus allowing synchronised access by several users at the same time, a Structured Query Language or SQL. Alas, SQL and interfacing to database daemons is not as standard as we would like, but support is there in Perl via the DBI.

The Perl DBI modules provide a standard wrapper that lets you talk to any supported relational database via a Data Base Dependant (DBD) module. These sub-modules are available for a very large range of database engines, and there's even one you can use to treat a command-separated file as a database table.

XML and XSLT

For data that doesn't naturally fall into relational database tables, a good alternative may be to use XML (eXtended Markup Language). XML is a metalanguage in which the user defines his own data hierarchy, and then tags the data to say what type of information he's providing where in the data stream, all done within plain text files that can be edited with any appropriate editor.

When a program analyses XML, it uses a piece of code known as a "parser" and there are several good ones written in C. Rather than reinvent the wheel, Perl modules on the CPAN provide support for XML via the Expat and Gnome[1] C language libraries, thus making XML parsing fast, and easily update-able as the base libraries change.

In order to transform XML data (where the data is tagged or labelled by its function) into a presentation format such HTML, you can use the definition language XSLT (eXtended Stylesheet Language Transforms). Once again, there are Perl modules on the CPAN to support this.

Graphics and Perl

The Tk module, based on the "Tk" part of Tcl/Tk, allows Perl programmers to write a graphic user interface so that application users aren't limited to a command line- or script-based Perl program. Using the Tk module, you define components ("widgets") which are laid out in a window by a layout manager, and then you write subroutines to define what's to be done when certain events happen (such as "the *xxx* button has been pressed"). Tk is extremely powerful in the appropriate environ-

[1] you have a choice as to which one to use

ment, but its results can't be presented in a browser window. This makes it powerful on intranet or local applications, but less so on Web applications.

For Web-based requirements, Perl can support libraries, such as GD, and for image manipulation modules, such as `Image::Magick` which can be used to read and write image objects in a wide variety of formats. Do be aware that this is all server side graphics, and you may have a job persuading an ISP to install GD for you onto his server as the processor implications could be significant. There is no commonly used plugin that lets you use Perl to generate graphics client side; that's probably a Flash or Java application.

Writing Web clients

A browser is a Web client, and you may ask "why would I want to write my own browser?". Chances are that you wouldn't want to as such, but there are times that you'll want to read information from a Web page (on a remote Web site) within your Perl.

Let's see an example.

I'm writing a program to convert an amount of money in US Dollars into Australian Dollars, and I want it to use the current exchange rate. That will work fine today, but tomorrow the exchange rate will be different and I'll have a maintenance night-mare. Much better to visit a central resource Web site[1] within your program, and update the rate automatically.

The LWP module of Perl provides all sorts of write-your-own-client capabilities. It allows you to do all sorts of clever things, many of which work technically very well but, we must warn you, can upset Web site owners as your robot's traffic volumes cost them money and ties up their Internet connections.

Talking to Microsoft applications

Although Perl originated from a Unix background, it's very well supported on Microsoft platforms, to the extent that Microsoft actively sponsor the ports and maintenance. You might wonder why. It's in their commercial interest for Perl code to run well on a *nix operating system and port to a Windows system. That way, it's one less reason for people not to move to Windows!

Once you're running Perl on Windows, you'll discover a number of other things that you want to do, such as automatically update an Excel spreadsheet, and indeed there are a number of modules provided, standard in the ActiveState port, that will let you access Word, Excel and other data formats. It's interesting to note that Perl actually calls up Microsoft's *.dll*s internally within these modules, so that the Perl programs that work on the Microsoft format data have to be run on systems which have licenses for the appropriate Microsoft packages.

[1] in this case, the European Central Bank helpfully provides all the rates

24.4 Perl in other guises

The use of Perl as a stand-alone program is fundamental, but it does have other uses too, such as within scripts, for example, or as service-providing daemons which we've already talked about earlier in this module. Here are a few others:

Interfacing to the Web via cgi

Web servers such as Apache and Microsoft's IIS provide cgi - the "Common Gateway Interface" to allow the Web to be used as a front end to application code. The majority of such code has historically been written in Perl.

Using form information read from the environment (via `%ENV`), `STDIN`, and very occasionally from the command line `@ARGV`, cgi allows any language, and not just Perl, to collect user requests. Output to `STDOUT` gets sent through the Web server back to the browser, thus completing the webification of Perl.

Data on both input and output conforms to certain formatting standards, and Perl is an excellent language through which to interpret or generate these formats. That, and Perl's ability to process the underlying data strings, is why Perl has traditionally been the language of choice of cgi applications.

Instead of writing your own CGI handlers, the *CGI.pm* module can be used to add common gateway and HTML support to your Perl. It's a huge module, and some users elect to use only parts of it. For example, it has excellent cookie handling.

Interfacing to the Web via mod-perl

Mod-perl is the Perl language built into a Web server such as Apache. By being more integrated with the Web server than is the case with cgi, the need for the server to recompile the Perl every time it is run is eliminated, leading to an increase in efficiency. The closer integration also allows a range of extra facilities and connections to be made within the Perl.

Other Web interfaces

Rather than write a program that generates HTML,[1] there are times that you'll want to write a Web page that includes some programatic elements. `Embed Perl`, which works within mod-perl, is one way that this facility can be provided. In the case of `Embed Perl`, you write ordinary HTML but embed your Perl code within it in special tags. The Web server then runs the Perl code in these special tags, and the result that's returned by this code is substituted into the HTML before it's sent on to the client. If you've come across PHP, this will sound very familiar; the concept is the same, although the tags used, and the underlying language, are different.

Modules such as `HTML::Mason` provide templating systems in Perl, and there's a wide range of modules to help with content management, etc. Indeed there are also complete commercial solutions such as Mediasurface, which are Perl-based and tailored.

[1] that's the way that cgi and mod-perl both work

Perl embedded within third-party applications

You'll sometimes find Perl embedded within other software. Even commercial software such as Tivoli's management software has it. In such cases, Perl juts out rather like a rock from the sea – a spike onto which user code can be hung to tailor the underlying product to meet the individual client's requirements.

You can, if you wish, also embed Perl into your own applications in another language such as C, or embed your C code into Perl. Tools such as Xs and Swig are provided by the Perl folks to help you with this, but do be aware that you're moving into a very specialist field, and that you must consider compatibility issues as Perl moves forward between releases. Some of the Perl structures used by these interfaces change in the move from Perl 5.6 to Perl 5.8 and a recompile will be necessary. Even more, the changes will be major if and when you move to Perl 6.

24.5 And also

Huge Data

Using just the fundamentals of Perl, and extra facilities described earlier in this module, Perl can handle a huge number of different tasks.

"Perl makes the difficult easy, and the impossible possible".

If you have a huge data file (and we've chosen to define huge as meaning "so big it won't fit in memory") then, yes, Perl can cope. This is a subject for further study once you're familiar with the fundamentals of the language. We can usefully spend half a day and more on training you on this topic if it's going to be relevant to you.

Maintainable code

A final plea – PLEASE at all times consider the poor sucker who has to maintain the code that you write. Write to a standard, document well, split code into subroutines or modules as appropriate, etc., etc.

Perl is like an artist's palette. Now that you've got some knowledge of how to paint, please use it to produce code of quality, and not something that looks like it's been dragged in by the cat!

Appendix

I. Exercise sample answers

Our Perl program for exercise sample titled "pfb" from page 12

```
#!/usr/local/bin/perl
# pfb - plain file backup

($#ARGV>0) && die "Usage: $0 [dirname]\n";

unless ($bd = $ARGV[0]) {
 print "Backup directory name: " if (-t STDIN);
 chop ($bd=<STDIN>);
 }

-e $bd && die "Can't overwrite existing object\n";

mkdir $bd,0755 || die "Can't create directory\n";

while (<*>) {
 unless (open(IN,$_)) {
 print "Can't read $_. No copy.\n";
 next;
 }
 unless (open (OUT,">$bd/$_")){
 print "Can't write $bd/$_. No copy.\n";
 next;
 }
 print OUT $section while (read(IN,$section,65536));
 }
```

Our Perl program for exercise sample titled "rgb" from page 212

```
#!/usr/local/bin/perl
# rgb - red, green, blue 2D array

open (IN,"rgb.txt") || die "not colourful\n";

while (<IN>) {
 chop;
 my @line = split(/\s+/,$_,4);
 push @carr,\@line;
 }
$,=" ";
for ($k=0;$k<=$#carr;$k++) {
 next unless ($carr[$k][2] > 239);
 print @{$carr[$k]}[0..3],"\n";
 }
```

Our Perl program for exercise sample titled "wstruct" from page 212

```
 #!/usr/local/bin/perl
 # wstruct - structure of web accesses

 open (WEB,"access_log") || die "no log file\n";
 <WEB>;

 while (<WEB>) {
  s/^(\w+)\s+//;
  push @{$htab{$1}},$_;
  }
 print "first and last accesses ...\n";
 foreach $host(sort keys %htab) {
  print "$host\n";
  print $htab{$host}[0];
  print $htab{$host}[$#{$htab{$host}}];
  }
```

Our Perl program for exercise sample titled "tptest" from page 34

```
 #!/usr/local/bin/perl
 # tptest - test "trainingprogram.pm"

 use trainingprogram;

 $demo = new trainingprogram qw(version 1.0 author Lisa);
 $demo->setcourse("Perl Programming");

 print $demo->getauthor,"\n";
 print $demo->getpurpose,"\n";
 print $demo->getversion,"\n";
 print $demo->getlanguage,"\n";
 print $demo->getcourse,"\n";

 print "\nTraining program count: $ptc\n";
```

db_serverd

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

# Database Server Daemon

use Socket;
use NDBM_File;
use Fcntl;

$dbfilename = $ARGV[0] || "tubes";
$on_port = $ARGV[1] || 4444;

if (-e "$dbfilename.pid") {
  die "pid file exists. Daemon running already?\n";
  }
```

```
open (PID,">$dbfilename.pid");
print PID "$$\n";
close PID;

tie %dbfile,"NDBM_File",$dbfilename,O_RDWR|O_CREAT,0600;

# Set up listener

$proto = getprotobyname("tcp");
socket (Server, PF_INET, SOCK_STREAM, $proto) ;
setsockopt(Server, SOL_SOCKET,SO_REUSEADDR,1);
$pbind = sockaddr_in($on_port,INADDR_ANY) ;
bind(Server,$pbind) ;
listen(Server,SOMAXCONN);

# await a contact. Do NOT multi thread in this simple example.
# Requests are single-line.

while ($paddr = accept(Client, Server)) {
  $request = <Client>;
  $request =~ s/\s+$//;
  ($action,$key,$value)=split(/\s+/,$request,3);

  $#response = -1;
  $status = 900;
  $key = lc($key);
  ($action =~ /^edit$/i) && ed();
  ($action =~ /^get/i) && get();
  ($action =~ /^grep/i) && dbgrep();
  ($action =~ /^new$/i) && new();
  ($action =~ /^replace$/i) && replace();
  ($action =~ /^delete$/i) && del();

  ($action =~ /^exit$/i) && killserver();

  $retlocks = ($action =~ /lock$/i or $action =~ /^edit$/i);

  print Client "ST: $status\n";
  foreach (@response) {
  s/\*\d+,\d+\*$// unless ($retlocks);
  print Client "$_\n";
  }

  last unless ($status);
}

untie %dbfile;
unlink "$dbfilename.pid";

#############################################################################

sub get {
```

```
if ($key) {
  if ($dbfile{$key}) {
  push @response,"$key $dbfile{$key}";
  $status = 1;
  } else {
  $status = 100;
  }
} else {
  foreach (sort keys %dbfile) {
  push @response,"$_ $dbfile{$_}";
  $status = 1;
  }
  }
}

################################################################################

sub dbgrep {
foreach (sort keys %dbfile) {
  /$key/ && push @response,"$_ $dbfile{$_}";
  $status = 1;
  }
}

###############################################################################

sub new {

if ($dbfile{$key}) {
  $status = 201;
} else {
  if ($value) {
  $dbfile{$key} = $value;
  $status = 1;
  } else {
  $status = 200;
  }
  }
}
###############################################################################

sub replace {

unless ($dbfile{$key}) {
  $status = 501;
} else {
  ($lock,$value) = split(/\s+/,$value);
  if ($dbfile{$key} !~ / \*$lock\*$/) {
  $status = 502;
  } else {
  if ($value) {
```

```
  $dbfile{$key} = $value;
  $status = 1;
  } else {
  $status = 500;
  }
  }
  }
}

############################################################################

sub ed {

if ($dbfile{$key}) {
  if ($dbfile{$key} =~ / \*\d+,\d+\*$/) {
  $status = 300;
  } else {
  $lock = ($count%=1000)++.",".time();
  $dbfile{$key} .= " *$lock*";
        push @response,"$key $dbfile{$key}";
        $status = 1;
  }
} else {
        $status = 301;
        }
}

############################################################################

sub del {

if ($dbfile{$key} =~ / \*\d+,\d+\*$/) {
  $status = 400;
} else {
  delete $dbfile{$key};
  $status = 1;
  }
}


############################################################################

sub killserver {

$status = 0;
}
```

II. CPAN sites

United Kingdom	ftp.demon.co.uk
	ftp.flirble.org
	ftp.plig.org
	sunsite.doc.ic.ac.uk
	unix.hensa.ac.uk
Europe	
Austria	ftp.tuwien.ac.at
Belgium	ftp.kulnet.kuleuven.ac.be
Bulgaria	ftp.ntrl.net
Croatia	ftp.linux.hr
Czech Republic	ftp.fi.muni.cz
	sunsite.mff.cuni.cz
Denmark	sunsite.auc.dk
Estonia	ftp.ut.ee
Finland	ftp.funet.fi
France	ftp.lip6.fr
	ftp.oleane.net
	ftp.pasteur.fr
	ftp.uvsq.fr
Germany	ftp.archive.de.uu.net
	ftp.gmd.de
	ftp.gwdg.de
	ftp.uni-erlangen.de
	ftp.uni-hamburg.de
Greece	ftp.ntua.gr
Hungary	ftp.kfki.hu
Ireland	sunsite.compapp.dcu.ie
Italy	cis.uniRoma2.it
	ftp.flashnet.it
	ftp.unipi.it
Netherlands	ftp.cs.uu.nl
	ftp.EU.net
Norway	ftp.uit.no
	sunsite.uio.no
Poland	ftp.man.szczecin.pl
	sunsite.icm.edu.pl
Portugal	ftp.ci.uminho.pt
	ftp.ua.pt
Romania	ftp.dntis.ro
	ftp.dnttm.ro
Russia	cpan.npi.msu.su
	ftp.sai.msu.su
Slovakia	ftp.entry.sk
Slovenia	ftp.arnes.si
Spain	ftp.etse.urv.es
	ftp.rediris.es
Sweden	ftp.sunet.se
Switzerland	sunsite.cnlab-switch.ch
Turkey	sunsite.bilkent.edu.tr

North America	
Alberta	sunsite.ualberta.ca
California	cpan.nas.nasa.gov
	ftp.digital.com
Colorado	ftp.cs.colorado.edu
Florida	ftp.cise.ufl.edu
Illinois	uiarchive.uiuc.edu
Indiana	ftp.uwsg.indiana.edu
Manitoba	theory.uwinnipeg.ca
Massachusetts	ftp.ccs.neu.edu
	ftp.iguide.com
Mexico	ftp.msg.com.mx
New York	ftp.rge.com
North Carolina	ftp.duke.edu
Oklahoma	ftp.ou.edu
Ontario	ftp.crc.ca
Oregon	ftp.orst.edu
Utah	mirror.xmission.com
Virginia	ftp.perl.org
Washington	ftp.spu.edu
South America	
Brazil	cpan.if.usp.br
Chile	ftp.ing.puc.cl
	sunsite.dcc.uchile.cl
Australasia	
Australia	cpan.topend.com.au
	ftp.labyrinth.net.au
	ftp.sage-au.org.au
	mirror.aarnet.edu.au
New Zealand	ftp.auckland.ac.nz
	sunsite.net.nz
Africa	
South Africa	ftp.is.co.za
	ftpza.co.za
Asia	
China	freesoft.cei.gov.cn
Hong Kong	ftp.hkstar.com
Israel	bioinfo.weizmann.ac.il
Japan	ftp.dti.ad.jp
	ftp.jaist.ac.jp
Singapore	ftp.nus.edu.sg
South Korea	ftp.bora.net
Taiwan	ftp.wownet.net
	ftp1.sinica.edu.tw
Thailand	ftp.cs.riubon.ac.th
	ftp.nectec.or.th

III. Common Errors in Perl

You'll notice that in some cases, the same error may be revealed at compile or run time. This happens because an error can sometimes result in a change of meaning and, at other times, lead to a syntax error.

<u>At COMPILE time</u> (i.e. Syntax errors – things that aren't valid Perl)

Symptom: *"Permission Denied"* message
Error: Your program is not marked as executable (Unix, Linux)

Symptom: *"Command not found"* message
Error: You have given the wrong name for your program, or it isn't included in the path that the system looks at. Note: this error can also occur if you have an error in the first (`#!`) line of your program, for example, if you misspell `perl` as `pearl`

Symptom: Very strange error messages if you run the program by typing its name at the command line, but the program functions if you type `perl` followed by its name.
Error: No `#!` line or `#!` line is wrong or `#!` line is not first in the file

Symptom: Various Perl compile time error messages
Common errors:
1. Missing `;` at the end of a statement
2. Missing `,` in a list
3. Missing `$` in front of a variable name
4. Keyword such as `while` or `print` starts with a capital letter
5. Quotes not matching
6. `/s` not matching in a regular expression; remember that you need three not two `/s` if you're using the *"s"* operator
7. Brackets or braces not matched
8. A `;` placed after an `if` or `while` or similar condition but before the block that is to form the conditional or loop
9. No block after an `if` or `while` or similar condition
10. Round brackets left out where they're needed; often a problem if you're using operators such as `+=` and `&&`

Note that Perl will often report the line number as two or three greater than the actual line on which the error occurred, and the error might not be obvious from the message given.

<u>At BEGIN time</u> (i.e. Programs that compile correctly, but there are bits missing)

Symptom: *"Can't locate* xxxx *in @INC"*
Error: The Perl module you have called for in a `use` or `require` statement can't be found

Symptom: "xxxx *did not return a true value ..."*
Error: No `1;` at the end of a module that's `use`d or `require`d

Symptom: *"Can't find import in empty package ..."*
Error: You've forgotten to put a `package` statement in your module.

<u>At RUN time</u> (i.e. Programs that malfunction)

Symptom: *"Undefined subroutine called"* or similar message
Error: Subroutine in wrong package, or subroutine name misspelt

Symptom: Program runs too fast and produces no results
Common errors:
1. The input file you gave doesn't exist or failed to open an output file correctly
2. You have the logic of a loop the wrong way round, so it skipped
3. You've misspelt a variable or list name that controls a loop
4. You've left the first comma out of the print list to `STDOUT`

Symptom: Instead of a lot of output, you get either a single number or a whole lot of `1`s or `0`s
Common Errors:
1. You've used a list in scalar context
2. Use of `@` where you meant `$` to access individual list element

Symptom: Output isn't spaced out nicely or looks odd
Common errors: 1.You forgot to add new line characters onto your prints
2.You forgot to print out spaces between printed elements
3.You've confused `print` with `printf`
4. You've put or not put `""` characters around a list to print

Symptom: Program appears to do something completely different to what you wrote
Error: You've given your program the same name as something else on your system that has priority

Symptom: Program doesn't function as you expect
Common Errors: 1.There's an error in your logic
2.Missing `$` in front of a variable name

Symptom: A `for` loop always executes exactly three times
Error: You've used `,` rather than `;` in the loop "condition"

Symptom: A subroutine doesn't seem to have been run
Error: No brackets after a subroutine call

Symptom: Conditional blocks or logic returning odd results
Common Errors: 1.Use of `=` where you meant `==` (overwrites variable named on left)
2.Use of `eq` where you meant `==` (String not numeric comparison)
3.Use of `eq` where you meant `=~` (exact comparison not match)
4.Use of `/xxx/` after `eq` (you've compared `$_` to the expression)
5.Use of `"xxx"` after `=~` (you should have used `/xxx/`)

Symptom: A variable appears to be empty, yet you're sure that it's been initialised
Common errors: 1.Use of `[..]` to refer to a hash element (you've created a list as well)
2.Use of `{..}` to reference a list element (you've created a hash as well)
3.Variable name misspelt or wrong variable name used
4.`$` missing off variable name

At RUN time on a web site

Symptom: *"Not Found (Error code 404)"*
Error: The URL that you've entered into the location bar (or that you've selected via a link) doesn't find anything; check that your CGI program is correctly named and in the correct directory

Symptom: *"Forbidden (Error code 403)"*
Error: The URL you've entered points to the executable (cgi-bin) directory, but the actual file cannot be executed by the web server; correct this using `chmod`

Symptom: *"Server Error (Error code 500)"*
This is a general error code that is presented for any error that occurs in running your CGI script; if you have access to the log files on the server, they should help you
Common errors: 1.Syntax errors, as listed earlier (run your script with `perl -c!`)
2.Incorrect header generated by script
3.Script does not generate blank line between header and body
4.Something is printed before the header line, often a "tracing" `print` statement gets left in

Symptom: Program works in testing, but not when uploaded
Common errors: 1.Permissions are only set on the program for owner, not group
2.Data files are not available in the correct locations
3.Environment variables differ

Remember command line options

`-c`	compile only
`-w`	give warning messages
`-d`	run Perl in debug mode

If you can't spot a problem, these options may give you extra clues

WHC Library

During the course, you are welcome to browse through our book library. If you are looking for a particular book but cannot find it, or would like to borrow a book overnight, please ask the tutor.

Perl

title	edition	author(s)	ISBN
ActivePerl™ with ASP and ADO	1st	Tobias Martinsson	0-471-38314-7
Advanced Perl Programming	1st	Sriram Srinivasan	1-56592-220-4
Beginning Perl	1st	Simon Cozens, Peter Wainwright	1-861003-14-5
Beginning Perl for Bioinformatics	1st	James Tisdall	0-596-00080-4
CGI Programming on the World Wide Web	1st	Shishir Gundavaram	1-56592-168-2
CGI Programming with Perl	2nd	Scott Guelich, Shishir Gundavaram, Gunther Birznieks	1-56592-419-3
Computer Science & Perl Programming	1st	Jon Orwant	0-596-00310-2
Cross-Platform Perl	2nd	Eric Foster-Johnson	0-7645-4729-1
Data Munging with Perl	1st	David Cross	1-930110-00-6
DeBugging Perl	1st	Martin C. Brown	0-07-212676-0
Effective Perl Programming	1st	Joseph N. Hall, Randal L. Schwartz	0-201-41975-0
Elements of Programming with Perl	1st	Andrew L. Johnson	1-884777-80-5
Embedding Perl in HTML with Mason	1st	Dave Rolsky, Ken Williams	0-596-00225-4
Extending and Embedding Perl	1st	Tim Jenness, Simon Cozens	1-930110-82-0
Graphics Programming with Perl	1st	Martien Verbruggen	1-930110-02-2
How to Program CGI with Perl 5.0	1st	Stephen Lines	1-56276-460-8
Learning Perl	1st	Randal Schwartz	1-56592-042-2
Learning Perl	2nd	Randal Schwartz, Tom Christiansen	1-56592-284-0
Learning Perl	3rd	Tom Phoenix, Randal L. Schwartz	0-596-00132-0
Learning Perl on Win32 Systems	1st	Randal L. Schwartz, Erik Olson, Tom Christiansen	1-56592-324-3
Learning Perl/Tk	1st	Nancy Walsh	1-56592-314-6
Mastering Algorithms with Perl	1st	Jon Orwant, Jarkko Hietaniemi, John Macdonald	1-56592-398-7
Mastering Perl/Tk	1st	Stephen Lidie, Nancy Walsh	1-56592-716-8
Mastering Regular Expressions	1st	Jeffrey Friedl	1-56592-257-3
Mastering Regular Expressions	2nd	Jeffrey Friedl	0-596-00289-0
mod_perl Developer's Cookbook	1st	Geoffrey Young, Paul Lindner, Randy Kobes	0-672-32240-4
mod_perl Pocket Reference	1st	Andrew Ford	0-596-00047-2
Network Programming with Perl	1st	Lincoln D. Stein	0-201-61571-1
Object Oriented Perl	1st	Damian Conway	1-884777-79-1
Official Guide to Programming with CGI.pm	1st	Lincoln Stein	0-471-24744-8
Perl 5 Desktop Reference	1st	Johan Vromans	1-56592-187-9
Perl 5 for Dummies	1st	Paul Hoffman	0-7645-0044-9
Perl 5 Pocket Reference	3rd	Johan Vromans	0-596-00032-4
Perl 5 Quick Reference	1st	Michael Foghlu	0-7897-0888-4
Perl and CGI for the World Wide Web - Visual Quickstart	1st	Elizabeth Castro	0-201-35358-X
Perl Cookbook	1st	Tom Christiansen, Nathan Torkington	1-56592-243-3
Perl Database Programming	1st	Brent Michalski	0-7645-4956-1
Perl Developer's Dictionary	1st	Clinton Pierce	0-672-32067-3
Perl for C Programmers	1st	Steve Oualline	0-7357-1228-X
Perl for Dummies	3rd	Paul Hoffman	0-7645-0776-1
Perl for Oracle DBAs	1st	Andy Duncan, Jared Still	0-596-00210-6
Perl for System Administration	1st	David N. Blank-Edelman	1-56592-609-9
Perl for Web Site Management	1st	John Callendar	1-56592-647-1
Perl! I Didn't Know You Could Do That ...	1st	Martin C. Brown	0-7821-2862-9
Perl in a Nutshell	1st	Ellen Siever, Stephen Spainhour, Nathan Patwardhan	1-56592-286-7
Perl & LWP	1st	Sean M. Burke	0-596-00178-9
Perl Resource Kit	1st	Ellen Siever, David Futato	1-56592-370-7
Perl, The Programmer's Companion	1st	Nigel Chapman	0-471-97563-X
Perl Weekend Crash Course	1st	Joe Merlino	0-7645-4827-1
Perl & XML	1st	Erik T. Ray, Jason McIntosh	0-596-00205-X
Perl: Your Visual Blueprint for Building Perl Scripts	1st	Paul Whitehead, Eric Kramer	0-7645-3478-5
Perl/Tk Pocket Reference	1st	Stephen Lidie	1-56592-517-3
Permachart Quick Reference Guide: Perl Programming	1st	D. Gargaro	1-55080-407-3
Proceedings of the Perl Conference 4.0	1st	Jon Orwant (Editor)	0-596-00013-8
Professional Perl Development	1st	Randy Kobes, Peter Wainwright, Shishir Gundavaram	1-861004-38-9
Professional Perl Programming	1st	Peter Wainwright	1-861004-49-4
Programming Perl	1st	Larry Wall, Randal Schwartz	0-937175-64-1
Programming Perl	2nd	Larry Wall, Tom Christiansen, Randal Schwartz	1-56592-149-6
Programming Perl	3rd	Larry Wall, Tom Christiansen, Jon Orwant	0-596-00027-8
Programming Perl 5.004 Quick Reference	1st	Johan Vromans	1-56592-187-9-x
Programming Perl in the .NET Environment	1st	Yevgeny Menaker, Michael Saltzman, Robert Oberg	0-13-065206-7
Programming the Network with Perl	1st	Paul Barry	0-471-48670-1
Programming the Perl DBI	1st	Alligator Descartes & Tim Bunce	1-56592-699-4

Programming Web Graphics with Perl & GNU Software	1st	Shawn P. Wallace	1-56592-478-9
Teach Yourself CGI Programming with Perl 5 in a week	1st	Eric Herrmann	1-57521-009-6
Teach Yourself Perl 5 in 21 Days	2nd	David Till	0-672-30894-0
Teach Yourself Perl in 24 Hours	1st	Clinton Pierce	0-672-31773-7
The Perl CD Bookshelf	1st	best-selling titles on CD	1-56592-462-2
The Perl CD Bookshelf	2nd	various	0-596-00164-9
Web Client Programming with Perl	1st	Clinton Wong	1-56592-214-X
Web Development with Apache and Perl	1st	Theo Petersen	1-930110-06-5
Web, Graphics & Perl/Tk	1st	Jon Orwant	0-596-00311-0
Web Services with Perl	1st	Randy Ray, Pavel Kulchenko	0-596-00206-8
Win32 Perl Programming: The Standard Extensions	1st	Dave Roth	1-57870-067-1
Win32 Perl Scripting	1st	Dave Roth	1-57870-215-1
Writing Apache Modules with Perl and C	1st	Lincoln Stein, Doug MacEachern	1-56592-567-X
Writing Perl Modules for CPAN	1st	Sam Tregar	1-59059-018-X
XML Processing with Perl, Python, and PHP	1st	Martin C. Brown	0-7821-4021-1

PHP

title	edition	author(s)	ISBN
Advanced PHP for Web Professionals	1st	Christopher Cosentino	0-13-008539-1
Beginning PHP4	1st	Choi, Kent, Lea, Prasad, Ullman	1-861003-73-0
Core PHP Programming	2nd	Leon Atkinson	0-13-089398-6
Create Dynamic Web Pages Using PHP and MySQL	1st	David Tansley	0-201-73402-8
Dreamweaver MX: PHP Web Development	1st	Gareth Downes-Powell, Tim Green, Bruno Mairlot	1-904151-11-6
Essential PHP for Web Professionals	1st	Christopher Cosentino	0-13-088903-2
Making Use of PHP	1st	Ashok Appu	0-471-21973-8
PHP Advanced for the World Wide Web: Visual Quickstart	1st	Larry Ullman	0-201-77597-2
PHP and MySQL Web Development	1st	Luke Welling, Laura Thomson	0-672-31784-2
PHP and MySQL Web Development	2nd	Luke Welling, Laura Thomson	0-672-32525-X
PHP Cookbook	1st	David Sklar, Adam Trachtenberg	1-56592-681-1
PHP Developer’s Cookbook	1st	Sterling Hughes, Andrei Zmievski	0-672-31924-1
PHP Developer’s Cookbook	2nd	Sterling Hughes, Andrei Zmievski	0-672-32325-7
PHP Fast & Easy Web Development	2nd	Julie Meloni	1-931841-87-X
PHP for the World Wide Web - Visual Quickstart	1st	Larry Ullman	0-201-72787-0
PHP Functions: Essential Reference	1st	Zak Greant. Graeme Merrall, Brett Michlitsch, Torben Wilson	0-7357-0970-X
PHP Graphics: Generating Images on the Fly	1st	Allan Kent, Mitja Slenc, Jason Sweat	1-86100-836-8
PHP & MySQL for Dummies	1st	Janet Valade	0-7645-1650-7
PHP Pocket Reference	1st	Rasmus Lerdorf	1-56592-769-9
PHP Pocket Reference	2nd	Rasmus Lerdorf	0-596-00402-8
PHP: Your visual blueprint for creating open source, server-side context	1st	Paul Whitehead, Joel Desamero	0-7645-3561-7
PHP4 Bible	1st	Tim Converse and Joyce Park	0-7645-4716-X
Professional PHP Programming	1st	Castagnetto, Rawat, Schumann, Scollo, Velith	1-861002-96-3
Professional PHP4 Multimedia Programming	1st	Kent, O’Dell, Chase, Rosa, Abraham, Suyoto, Apshankar	1-861007-64-7
Professional PHP4 Web Development Solutions	1st	Dash, Waters, Gianotto, Endrerud, Anton, Stephens, Solin	1-861007-43-4
Programming PHP	1st	Rasmus Lerdorf, Kevin Tatroe	1-56592-610-2
Teach Yourself PHP in 24 Hours	2nd	Matt Zandstra	0-672-32311-7
Web Application Development with PHP4.0	1st	Tobias Ratschiller, Till Gerken	0-7357-0997-1
Web Database Applications with PHP & MySQL	1st	Hugh Williams, David Lane	0-596-00041-3
XML and PHP	1st	Vikram Vaswani	0-7357-1227-1

MySQL and Other Relational Databases

title	edition	author(s)	ISBN
Beginning Databases with MySQL	1st	Neil Matthew, Richard Stones	1-861006-92-6
dBase Instant Reference	1st	Alan Simpson	0-89588-484-4
Managing & Using MySQL	2nd	George Reese, Randy Jay Yarger, Tim King, Hugh E. Williams	0-596-00211-4
MySQL	1st	Paul DuBois	0-7357-0921-1
MySQL	2nd	Paul DuBois	0-7357-1212-3
MySQL and JSP Web Applications	1st	James Turner	0-672-32309-5
MySQL and Perl for the Web	1st	Paul DuBois	0-7357-1054-6
MySQL Cookbook	1st	Paul DuBois	0-596-00145-2
MySQL Pocket Reference	1st	George Reese	0-596-00446-X
MySQL & mSQL	1st	Randy Jay Yarger, George Reese, Tim King	1-56592-434-7
MySQL & PHP from scratch	1st	Wade Maxfield	0-7897-2440-5
MySQL Reference Manual	1st	Michael “Monty” Widenius, David Axmark, MySQL AB	0-596-00265-3
MySQL: Visual Quickstart Guide	1st	Larry Ullman	0-321-12731-5
MySQL/PHP Database Applications	1st	Brad Bulger, Jay Greenspan	0-7645-3537-4
Oracle & Open Source	1st	Andy Duncan, Sean Hull	0-596-00018-9
Sequence Analysis in a Nutshell	1st	Scott Markel, Darryl León	0-596-00494-X
SQL for Dummies	4th	Allen G. Taylor	0-7645-0737-0
SQL in a Nutshell	1st	Daniel Kline, Kevin Kline	1-56592-744-3
SQL Performance Tuning	1st	Peter Gulutzan, Trudy Pelzer	0-201-79169-2
SQL: Visual Quickstart Guide	1st	Chris Fehily	0-321-11803-0
Teach Yourself SQL in 10 Minutes	2nd	Ben Forta	0-672-32128-9
The Linux Database	1st	Fred Butzen, Dorothy Forbes	1-55828-491-5
Using SQL	1st	James Grodd, Paul Weinberg	0-07-881524-X

Tcl, Tcl/Tk and Expect

title	edition	author(s)	ISBN
Effective Tcl/Tk Programming	1st	Mark Harrison, Michael McLennan	0-201-634747-0
Exploring Expect	1st	Don Libes	1-56592-090-2
Graphical Applications with Tcl & Tk	2nd	Eric Foster-Johnson	1-55851-569-0
Practical Programming in Tcl and Tk	2nd	Brent B. Welch	0-13-616830-2

Practical Programming in Tcl and Tk	3rd	Brent B. Welch	0-13-022028-0
Tcl and the Tk Toolkit	1st	John K. Ousterhout	0-201-63337-X
Tcl & Tk Multimedia Training Course	1st	Brent Welch, Dave Zeltserman	0-13-080756-7
Tcl/Tk for Dummies	1st	Alex Francis, Tim Webster	0-7645-0152-6
Tcl/Tk for Real Programmers	1st	Clif Flynt	0-12-261205-1
TCL/TK in a Nutshell, A Desktop Reference	1st	Paul Raines, Jeff Tranter	1-56592-433-9
Tcl/Tk Pocket Reference	1st	Paul Raines	1-56592-498-3
Tcl/Tk Programmer’s Reference	1st	Christopher Nelson	0-07-212004-5
TCL/TK Tools	1st	Mark Harrison	1-56592-218-2
Teach Yourself Tcl/Tk in 24 Hours	2nd	Venkat Sastry, Lakshmi Sastry	0-672-31749-4
Web Tcl Complete	1st	Steve Ball	0-07-913713-X

Ruby

title	edition	author(s)	ISBN
Programming Ruby: The Pragmatic Programmer’s Guide	2nd	David Thomas, Andrew Hunt	0-201-71089-7
Ruby Developer’s Guide	1st	Robert Feldt, Lyle Johnson, Michael Neumann	1-928994-64-4
Ruby in a Nutshell	1st	Yukihiro Matsumoto	0-596-00214-9

Java (including Servlets and JSP)

title	edition	author(s)	ISBN
Advanced JavaServer Pages	1st	David M. Geary	0-13-030704-1
Ant Developer’s Handbook	1st	Williamson, Pepperdine, Gibson, Wu	0-672-32426-1
Ant: The Definitive Guide	1st	Jesse Tilly, Eric Burke	0-596-00184-3
Beginning Java	1st	Ivor Horton	1-861000-27-8
Beginning Java 2 (JDK 1.3 Edition)	1st	Ivor Horton	1-861003-66-8
Concurrent Programming in Java	2nd	Doug Lea	0-201-31009-0
Core Java	1st	Gary Cornell, Cay Horstmann	0-13-565755-5
Core Java 2 Volume I-Fundamentals	5th	Cay S. Horstmann, Gary Cornell	0-13-089468-0
Core Java 2 Volume II-Advanced Features	5th	Gary Cornell, Cay S. Horstmann	0-13-092738-4
Core JFC	2nd	Kim Topley	0-13-090581-X
Core JSP	1st	Damon Hougland, Aaron Tavistock	0-13-088248-8
Creating Effective JavaHelp	1st	Kevin Lewis	1-56592-719-2
Database Programming with JDBC and Java	2nd	George Reese	1-56592-616-1
Developing Java Servlets	2nd	James Goodwill	0-672-32107-6
Effective Java Programming Language Guide	1st	Joshua Bloch	0-201-31005-8
Enterprise JavaBeans	1st	Richard Monson-Haefel	1-56592-605-6
Enterprise JavaBeans	3rd	Richard Monson-Haefel	0-596-00226-2
Inside Java2 Platform Security	2nd	Li Gong	0-201-31000-7
Inside the Java Virtual Machine	1st	Bill Venners	0-07-913248-0
Java and XSLT	1st	Eric M. Burke	0-596-00143-6
Java by Example	2nd	Jerry Jackson, Alan McClellan	0-13-272295-X
Java Cookbook	1st	Ian F. Darwin	0-596-00170-3
Java Developer's Reference	1st	Cohn, Morgan, Morrison, Nygard, Joshi, Trinko	1-57521-129-7
Java Development with Ant	1st	Erik Hatcher, Steve Loughran	1-930110-58-8
Java Enterprise in a Nutshell	1st	David Flanagan, Jim Farley, William Crawford, Kris Magnusson	1-56592-483-5
Java Examples in a Nutshell	1st	David Flanagan	1-56592-371-5
Java Examples in a Nutshell	2nd	David Flanagan	0-596-00039-1
Java for the World Wide Web - Visual Quickstart	1st	Dori Smith	0-201-35340-7
Java Foundation Classes in a Nutshell	1st	David Flanagan	1-56592-488-6
Java in a Nutshell	2nd	David Flanagan	1-56592-262-X
Java in a Nutshell	3rd	David Flanagan	1-56592-487-8
Java in a Nutshell	4th	David Flanagan	0-596-00283-1
Java Reference Library	1st	David Flanagan	1-56592-304-9
Java RMI	1st	William Grosso	1-56592-452-5
Java Servlet Programming	1st	William Crawford, Jason Hunter	1-56592-391-X
Java Servlet Programming	2nd	William Crawford, Jason Hunter	0596-00040-5
Java Threads	2nd	Scott Oaks, Henry Wong	1-56592-418-5
Java & XML	2nd	Brett McLaughlin	0-596-00197-5
JavaServer Pages Pocket Reference	1st	Hans Bergsten	0-596-00231-9
JFC Java Foundation Classes	1st	Daniel I. Joshi, Pavel A. Vorobiev	0-7645-8041-8
JSP, Servlets, and MySQL	1st	David Harms	0-7645-4787-9
Just Java	2nd	Perter van der Linden	0-13-272303-4
Jython Essentials	1st	Samuele Pedroni, Noel Rappin	0-596-00247-5
Learning Java	2nd	Pat Niemeyer, Jonathan Knudsen	0-596-00285-8
Professional Java Server Programming: J2EE Edition	1st	various	1-861004-65-6
Programming with Java!	1st	Tim Ritchey	1-56205-533-X
Pure JFC Swing	1st	Satyaraj Pantham	0-672-31423-1
Quick Start: Borland JBuilder	1st	Inprise Corporation	part JBE1330WW21000
Special Edition Using Java 1.2	4th	Joseph Weber	0-7897-1529-5
Struts Kick Start	1st	James Turner, Kevin Bedell	0-672-32472-5
Teach Yourself Java 1.1 Programming in 24 hours	1st	Rogers Cadenhead	1-57521-270-6
Teach Yourself Object Oriented Programming	2nd	Anthony Sintes	0-672-32109-2
Teach Yourself Visual J++ in 21 days	1st	Winters, Olhasso, Lemay, Perkins	1-57521-158-0
The Java™ Language Specification, Second Edition	2nd	Gosling, Joy, Steele, Bracha	0-201-31008-2
The Java™ Tutorial Continued, The Rest of the JDK	1st	Campione, Walrath, Huml, Tutorial Team	0-201-48558-3
The Java™ Tutorial, Third Edition	3rd	Campione, Walrath, Huml, Tutorial Team	0-201-70393-9
Thinking in Java	2nd	Bruce Eckel	0-13-027363-5

Python

title	edition	author(s)	ISBN
Learn to Program Using Python	1st	Alan Gauld	0-201-70938-4

Learning Python	1st	Mark Lutz, David Ascher	1-56592-464-9
Programming Python	1st	Mark Lutz	1-56592-197-6
Python and Tkinter Programming	1st	John Grayson	1-884777-81-3
Python & XML	1st	Christopher Jones, Fred Drake, Jr.	0-596-00128-2
Python Annotated Archives	1st	Martin C. Brown	0-07-212104-1
Python Cookbook	1st	Alex Martelli, David Ascher	0-596-00167-3
Python in a Nutshell	1st	Alex Martelli	0-596-00188-6
Python Programming on Win32	1st	Mark Hammond, Andy Robinson	1-56592-621-8
Python Programming with Java Class Libraries	1st	Richard Hightower	0-201-61616-5
Python Standard Library	1st	Fredrik Lundh	0-596-00096-0
Python Visual Quickstart Guide	1st	Chris Fehily	0-201-74884-3
The Quick Python Book	1st	Daryl Harms, Kenneth McDonald	1-884777-74-0

C and C++

title	edition	author(s)	ISBN
Building Cocoa Applications: A Step-by-Step Guide	1st	Simson Garfinkel, Micheal Mahoney	0-596-00235-1
C By Example	1st	Greg Perry	0-88022-823-X
C++ for Linux	1st	Jesse Liberty, David B. Horvath	0-672-31895-4
C Programming in a Unix Environment	1st	Judy Kay, Bob Kummerfeld	0-201-12912-4
Learning Cocoa with Objective-C	2nd	James Davidson	0-596-00301-3
Objective-C Pocket Reference	1st	Andrew Duncan	0-596-00423-0
Standard C	1st	P. J. Plauger, Jim Brodie	1-55615-158-6
Turbo C++ Programmer's Guide	1st	Borland	part 14MN-CND02-10
Turbo C++ User's Guide	1st	Borland	part 14MN-CND01-10
Turbo Vision for C++ User's Guide	1st	Borland	part 14MN-TVC01

Other programming languages

title	edition	author(s)	ISBN
A Guide to Cobol Programming	2nd	Umberto Garbassi, Daniel D. McCracken	0-471-58243-3
Programming in Standard Fortran 77	1st	A. Balfour, D. H. Marwick	0-435-77486-7

Programming and Program design (not language specific)

title	edition	author(s)	ISBN
80386: A Programming & Design Handbook	2nd	Don Brumm, Penn Brumm	0-8306-3237-9
Bad Software	1st	Cem Kaner, David Pels	0-471-31826-4
Developing Bioinformatics Computer Skills	1st	Cynthia Gibas, Per Jambeck	1-56592-664-1
Object Solutions, Managing the Object-Oriented Project	1st	Grady Booch	0-8053-0594-7
Open Source Web Development with LAMP	1st	James Lee, Brent Ware	0-201-77061-X
Physics for Game Developers	1st	David Bourg	0-596-00006-5
Teach Yourself Beginning Programming in 24 Hours	1st	Greg Perry	0-672-31355-3
The Computer Desktop Encyclopedia	2nd	Alan Freedman	0-8144-7985-5
The Practice of Programming	1st	Brian W. Kernighan, Rob Pike	0-201-61586-X
UML Distilled	2nd	Fowler, Scott	0-201-65783-X
UML in a Nutshell	1st	Sinan Si Alhir	1-56592-448-7
UML Toolkit	1st	Hans-Erik Eriksson, Magnus Penker	0-471-19161-2

Web Server Content and Design

title	edition	author(s)	ISBN
Designing Web Usability	1st	Jakob Nielson	1-56205-810-X
Dynamic HTML: The Definitive Reference	1st	Danny Goodman	1-56592-494-0
Flash 4 for Windows and Macintosh - Visual Quickstart	1st	Katherine Ulrich	0-201-35473-X
Flash™ 4 Magic	1st	David J. Emberton, J. Scott Hamlin	0-7357-0949-1
Foundations of World Wide Web Programming with HTML & CGI	1st	Tittel, Gaither, Hassinger, Erwin	1-56884-703-3
Hip Pocket Guide to HTML 3.2	1st	Ed Tittel, James Michael Stewart	0-7645-8017-5
HTML 4 for the World Wide Web - Visual Quickstart	4th	Elizabeth Castro	0-201-35493-4
HTML for the World Wide Web - Visual Quickstart	2nd	Elizabeth Castro	0-201-68862-X
HTML Pocket Reference	1st	Jennifer Niederst	1-56592-579-3
HTML Sourcebook	2nd	Ian S. Graham	0-471-14242-5
HTML: The Definitive Guide	2nd	Chuck Musciano, Bill Kennedy	1-56592-235-2
HTML: The Definitive Guide	3rd	Chuck Musciano, Bill Kennedy	1-56592-492-4
HTML Web Magic	1st	Ardith Ibanez, Natalie Zee	1-56830-335-1
HTML & XHTML: The Definitive guide	5th	Chuck Musciano, Bill Kennedy	0-596-00382-X
Instant IE4 Dynamic HTML (IE4 Edition)	1st	Alex Homer, Chris Ullman	1-861000-68-5
JavaScript for the World Wide Web - Visual Quickstart	1st	Ted Gesing, Jeremy Schneider	0-201-68814-X
JavaScript for the World Wide Web - Visual Quickstart	3rd	Tom Negrino, Dori Smith	0-201-35463-2
JavaScript in easy steps	1st	Mike McGrath	1-874029-89-X
JavaScript & Netscape Wizardry	1st	Dan Shafer	1-883577-86-1
JavaScript Pocket Reference	1st	David Flanagan	1-56592-521-1
JavaScript The Definitive Guide	2nd	David Flanagan	1-56592-234-4
Learning WML & WMLScript	1st	Martin Frost	1-56592-947-0
Macromedia Dreamweaver for Windows & Macintosh	1st	J. Tarin Towers	0-201-84445-1
Permachart Quick Reference Guide: HTML 4	1st	M. Seringhaus	1-55080-382-4
Teach Yourself JavaScript in a Week	1st	Arman Danesh	1-57521-073-8
Teach Yourself Web Publishing with HTML		Laura Lemay	0-672-30667-0
The Book of Zope	1st	Beehive, L.L.C.	1-886411-57-3
Using HTML	special	Tom Savola	0-797-0236-3
WAP in easy steps	1st	Mike McGrath	1-84078-112-2
Web Design in a Nutshell	1st	Jennifer Niederst	1-56592-515-7
Webmaster in a Nutshell	1st	Stephen Spainhour, Valerie Quercia	1-56592-229-8
Zope Web Application Construction Kit	1st	Brockmann, Kirchner, Lühnsdorf, Pratt	0-672-32133-5

Web Client (Browser) topics - HTML, CSS, JavaScript

title	edition	author(s)	ISBN
CCS Pocket Reference	1st	Eric Meyer	0-596-00120-7
Content Management Systems	1st	Dave Addey, James Ellis, Phil Suh, David Thiemecke	1-904151-06-X
Creating Stores on the Web	2nd	Ben Sawyer, Dave Greely, Joe Cataudella	0-201-70005-0
Dreamweaver MX Magic	1st	various authors	0-7357-1179-8
Information Architecture for the World Wide Web	1st	Louis Rosenfeld, Peter Morville	1-56592-282-4
Information Architecture for the World Wide Web	2nd	Louis Rosenfeld, Peter Morville	0-596-00035-9
Looking Good Online	1st	Steve Bain, Daniel Gray	1-56604-469-3
Mining the Web	1st	Gordon Linoff, Michael Berry	0-471-41609-6
Practical Internet Groupware	1st	Jon Udell	1-56592-537-8
The Design of Sites	1st	Douglas Van Duyne, James Landay. Jason Hong	0-201-72149-X
Usable Forms for the Web	1st	Andy Beaumont, Jon James, Jon Stephens, Chris Ullman	1-904151-09-4
Web Design for Dummies	1st	Lisa Lopuck	0-7645-0823-7
Web Design Templates Sourcebook	1st	Lisa Schmeiser	1-56205-754-5
Web Navigation, Designing the User Experience	1st	Jennifer Fleming	1-56592-351-0
Web Security & Commerce	1st	Simpson Garfinkel with Gene Spafford	1-56592-269-7

Apache, IIS, Tomcat and Web Server Admin

title	edition	author(s)	ISBN
Apache Pocket Reference	1st	Andrew Ford	1-56592-706-0
Apache, The Definitive Guide	2nd	Ben Laurie, Peter Laurie	1-56592-528-9
HTTP Pocket Reference	1st	Clinton Wong	1-56592-862-8
Mastering Tomcat Development	1st	Peter Harrison, Ian McFarland	0-471-23764-7
Professional Apache Tomcat	1st	various authors	1-861007-73-6
Programming Web Services with SOAP	1st	Pavel Kulchenko, James Snell, Doug Tidwell	0-596-00095-2
Running a Perfect Web Site with Apache	1st	Brian Behlendorf, David Chandler	0-7897-0745-4
Web Services Essentials	1st	Ethan Cerami	0-596-00224-6

ASP, .NET, C# etc.

title	edition	author(s)	ISBN
ASP in a Nutshell	1st	A. Keyton Weissinger	1-56592-490-8
C# Essentials	1st	Ben Albahari, Peter Drayton, Brad Merrill	0-596-00079-0
C# Programming With the Public Beta	1st	Harvey, Robinson, Templeman, Watson	1-861004-87-7
Teach Yourself Active Server Pages 2.0 in 21 Days	1st	Sanjaya Hettihewa	0-672-31333-2

XML and XSLT

title	edition	author(s)	ISBN
Learning XML	1st	Erik T. Ray	0-596-00046-4
Permachart Quick Reference Guide: XML	1st	D. Gargaro	1-55080-454-5
Professional XML Web Services	1st	various	1-861005-09-1
Programming Web Services with XML-RPC	1st	Edd Dumbill, Joe Johnston, Simon St. Laurent	0-596-00119-3
XML Elements of Style	1st	Simon St. Laurent	0-07-212220-X
XML for the World Wide Web - Visual Quickstart	1st	Elizabeth Castro	0-201-71098-6
XML in a Nutshell	1st	Elliotte Rusty Harold, W. Scott Means	0-596-00058-8
XML Pocket Reference	1st	Robert Eckstein	1-56592-709-5
XML Schema	1st	Eric van der Vlist	0-596-00252-1
XPath and XPointer	1st	John E. Simpson	0-596-00281-2
XSLT	1st	Doug Tidwell	0-596-00053-7

Web overviews and general subjects

title	edition	author(s)	ISBN
Core Web Programming	1st	Marty Hall	0-13-625666-X
Fast Track Web Programming	1st	Dave Cintron	0-471-32426-4
Google Hacks: 100 Industrial-Strength Tips & Tools	1st	Tara Calishain & Rael Dornfest	0-596-00447-8
Internet & World Wide Web How to Program	2nd	Harvey Deitel, Paul Deitel, Tem Nieto	0-13-030897-8
.net All you need to know about The Internet	1st	Davey Winder	1-85870-064-7
NetLingo	1st	Erin Jansen	0-9706396-7-8
The Internet Yellow Pages	1st	Harley Hahn, Rick Stout	0-07-882-23-5
The Whole Internet User's Guide & Catalog	2nd	Ed Krol	1-56592-063-5
The World Wide Web Unleashed	1st	John December, Neil Randall	0-672-30617-4
The World-Wide Web, Mosaic and More	1st	Jason J. Manger	0-07-709132-9
Web Programming: Building Internet Applications	1st	Chris Bates	0-471-49669-3
Web Programming Languages Sourcebook	1st	Gordon McComb	0-471-17576-5

Linux

title	edition	author(s)	ISBN
Essential System Administration	3rd	Æleen Frisch	0-596-00343-9
Installing OpenLinux and StarOffice for Dummies	1st	Jon Hall, Nicholas Wells, Michael Meadhra	MM157
Just For Fun	1st	Linus Torvalds, David Diamond	1-58799-080-6
Learning the bash Shell	2nd	Cameron Newham, Bill Rosenblatt	1-56592-347-2
Linux!	2nd	Nicholas D. Wells	0-7821-2935-8
Linux in a Nutshell	3rd	Stephen Figgins, Jessica P. Hekman, Ellen Siever, Stephen Spainhour	0-596-00025-1
Linux System Administration Handbook		Mark Komarinski, Cary Collett	0-13-680596-5
Linux: The Complete Reference	2nd	Richard Petersen	0-07-882461-3
Linux Unleashed	2nd	Kamran Husain, Timothy Parker	0-672-30908-4
Permachart Quick Reference Guide: Linux	1st	D. Gargaro	1-55080-350-6
Rebel Code	1st	Glyn Moody	0-713-99520-3
Red Hat Linux	3rd	David Pitts, Bill Ball	0-672-31410-X
Red Hat Linux for Dummies		Jon Hall, Paul G. Sery	0-7645-0663-3
Running a Perfect Internet Site with Linux	1st	Dee-Ann LeBlanc	0-7897-0514-1

SSH, The Secure Shell: The Definitive Guide	1st	Daniel Barrett, Richard Silverman	0-596-00011-1
The Cathedral & the Bazaar	1st	Eric S. Raymond	1-56592-724-9
The Complete Reference: Red Hat Linux	1st	Richard Petersen	0-07-212535-7
The Linux Web Server CD Bookshelf	1st	Stephen Figgins, Jessica P. Heckman, Ellen Siever, Stephen Spainhour	0-596-00208-4
vi Editor Pocket Reference	1st	Arnold Robbins	1-56592-497-5

Mac OS X

title	edition	author(s)	ISBN
Learning Unix for Mac OSX	1st	Dave Tylor, Jerry Peek	0-596-00342-0
Macintosh Trouble Shooting Pocket Guide	1st	David Lerner, Aaron Freimark	0-596-00443-5
Mac OS X for Unix Geeks	1st	Brian Jepson, Ernest Rothman	0-596-00356-0
Mac OS X in a Nutshell	1st	Jason McIntosh, Chuck Toporek, Chris Stone	0-596-00370-6
Mac OS X Pocket Guide	2nd	Chuck Toporek	0-596-00458-3
Mac OS X The Missing Manual	8th	David Pogue	0-596-00082-0
Mac OS X Unleashed	1st	John Ray, William Ray	0-672-32229-3
Unix for Mac OS X	1st	Matisse Enzer	0-201-79535-3

Unix

title	edition	author(s)	ISBN
Solaris System Administrator's Guide	2nd	Janice Winsor	1-57870-040-X
The Unix CD Bookshelf, version 2.1	2nd	various	0-596-00000-6
UNIX in a Nutshell: System V Edition	3rd	Arnold Robbins	1-56592-427-4
Unix in a Nutshell: System V & Solaris 2.0	2nd	Daniel Gilly and staff	1-56592-001-5
Unix Primer Plus	1st	Mitchell Waite, Donald Martin, Stephen Prata	0-672-22028-8
Unix System Security	2nd	Rik Farrow	0-201-57030-0
Unix System V Release 4 Administration	2nd	David Fiedler, Bruce Hunter, Ben Smith	0-672-22810-6

Windows

title	edition	author(s)	ISBN
Essential Windows NT System Administration	1st	Æleen Frisch	1-56592-274-3
Mastering Windows XP Professional	2nd	Mark Minasi	0-7821-4114-5

Networking

title	edition	author(s)	ISBN
An Introduction to ATM Technology	1st	Marc Boisseau, Michel Demange, Jean-Marie Munier	0-442-01994-7
ATM Networks	1st	Othmar Kyas	0-442-02000-2
Building Internet Firewalls	1st	D. Brent Chapman, Elizabeth D. Zwicky	1-56592-124-0
Cisco TCP/IP Routing Professional Reference	2nd	Chris Lewis	0-07-041130-1
DNS and Bind	3rd	Paul Albitz, Cricket Liu	1-56592-512-2
Managing Internet Information Services	1st	Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus, Adrian Nye	1-56592-062-7
Protecting Networks with SATAN	1st	Martin Freiss	1-56592-425-8
Running Weblogs with Slash	1st	chromatic, Brian Aker, Dave Krieger	0-596-00100-2
Samba Black Book	1st	Dominic Baines	1-57610-455-9
Sendmail	1st	Bryan Costales, Eric Allman, Neil Rickert	1-56592-056-2
Stopping Spam	1st	Alan Schwartz, Simson Garfinkel	1-56592-388-X
The Cuckoo's Egg	1st	Cliff Stoll	0-7434-1146-3
The Networking CD Bookshelf	1st	various	1-56592-523-8

Computer Graphics

title	edition	author(s)	ISBN
An Introduction to Ray Tracing	1st	Andrew S. Glassner	0-12-286160-4
Computer Graphics	1st	Roy A Plastock, Gordon Kalley	0-07-050326-5
Computer Graphics	2nd	Foley, Van Dam, Feiner, Hughes	0-201-12110-7
Designing Web Graphics.4	4th	Lynda Weinman	0-7357-1079-1
Encyclopedia of Graphics File Formats	1st	James D. Murray, William vanRyper	1-56592-058-9
Encyclopedia of Graphics File Formats	2nd	James D. Murray, William vanRyper	1-56592-161-5
Fundamentals of Interactive Computer Graphics	1st	J.D. Foley, A. Van Dam	0-201-14468-9
Interactive Computer Graphics	1st	Peter Burger, Duncan Gillies	0-201-17439-1
PNG The Definitive Guide	1st	Greg Roelofs	1-56592-542-4
Power Graphics Programming	1st	Michael Abrash	0-88022-500-9
Procedural Elements for Computer Graphics	1st	David F. Rogers	0-07-053534-5
Programmer's Guide to the EGA and VGA Cards	1st	Richard F. Ferraro	0-201-12692-3
SVG Essentials	1st	J. David Eisenberg	0-596-00223-8
The Elements of Color	1st	Johannes Itten	0-471-28929-9

X Windows and Motif

title	edition	author(s)	ISBN
An X/Motif Programmer's Primer	1st	Fintan Culwin	0-13-101841-8
Building OSF/Motif Applications: A Practical Introduction	1st	Mark J. Sebern	0-13-122409-3
Introduction to the X Window System	1st	Oliver Jones	0-13-499997-5
The X Window System in a Nutshell	1st	O'Reilly staff	0-937175-24-2
X and Motif Quick Reference Guide	2nd	Randi J. Rost	1-5555-8118-8
X Windows System User's Guide Volume 3	2nd	Valerie Quercia, Tim O'Reilly	0-937175-36-6
Xlib Programming Manual Volume 1	1st	Adrian Nye	0-937175-27-7
Xlib Reference Manual Volume 2	1st	Adrian Nye	0-937175-28-5

Other books

title	edition	author(s)	ISBN
Computer Law	4th	Chris Reed, John Angel	1-84174-016-0
Dictionary of Computer and Internet Terms	6th	Douglas Downing, Michael Covington, Melody Covington	0-7641-0094-7
FileMaker Pro 5 for Windows & Macintosh	4th	Nolan Hester	0-201-70417-X

MP3 The Definitive Guide	1st	Scot Hacker	1-56592-661-7
Palm Pilot:The Ultimate Guide	2nd	David Pogue	1-56592-600-5
Type & Typography	1st	Phil Baines, Andrew Haslam	1-85669-244-2
Wired Style: Principles of English Usage in the Digital Age	1st	Constance Hale, Jessie Scanlon	0-7679-0372-2

Index

Behind Learning to Program in Perl: the author

Graham Ellis has a vast background in the computer industry. After receiving his degree in Computer Science in1976 from City University, London, he hasn't looked back.

Graham spent seven years leading a team developing cross-platform products (running on both PCs and Unix systems), from which he brings an appreciation of product specification, portability, standards and security.

Many highly technical staff prefer to remain "back room boys", but Graham has always enjoyed writing and presenting training courses. Since the first course he wrote and presented (on the Fortran programming language) in the late ’70s, training has gradually accounted for more and more of his working time. He does, though, take care to leave time aside for outside interests, family, and for time to undertake "real work" in the subject areas in which he teaches. Graham believes in practising what he preaches!

Graham was a user of the Internet before it ever hit big, and, with his training and support roles, was ideally placed both to make practical use of the technology and to help others do the same.

Well House Consultants, Ltd

In 1995, Graham founded Well House Consultants Ltd to present training courses for others under contract and to develop his own training material. Well House Consultants specialises in "niche" courses, the sort of topics that other training organisations can't provide economically themselves.

Well House Consultants is a partnership between Graham and his wife Lisa, whom he first noticed on the Internet. Lisa comes from a graphic arts background and looks after much of the web site maintenance work, as well as the production of sales material and manuals, such as this one, and sales enquiries and bookings. Lisa can be reached by email at *lisa@wellho.net*, via our main phone number 01225 708 225, or by fax on 01225 707 126.

In spite of being busy, Graham aims to remain approachable. Do feel free to email him (*graham@wellho.net*) if you have any questions on this material that you can't resolve through normal channels. If you're interested in booking places on courses, or seeing if we can help with your onsite requirements, please email enquiry@wellho.net (that will get you a quicker response if Graham's out training!) or call up our FAQ by sending a message to *faq@wellho.net*.

Thank you for attending this course!