Perl Programming

David Schweikert <dws@ee.ethz.ch>

IT Support Group / D-ITET
Swiss Federal Institute of Technology, Zurich

Version 1.6 – May 25, 2004

Abstract

Perl is a *"Practical Extraction and Report Language"* freely available for Unix, Windows, Macintosh, and other operating systems. Perl has powerful text manipulation functions, and combines features and purposes of many command languages. Perl is very popular for programming World Wide Web electronic forms and generally as glue and gateway between systems, databases, and users.

About this course

By completing this course, you should be able to:

- Locate reference materials and other resources related to Perl
- Express fundamental programming constructs such as variables, arrays, lo-ops, subroutines and input/output in Perl
- Understand several concepts relatively idiosyncratic to Perl, such as associative arrays and regular expressions

To keep this a short course, I won't explain object-oriented concepts and some other facilities appropriate for large projects. Perl, perhaps more than any other computer language, is full of alternative ways to do the same thing; I tend to show only one or two. I will try to stimulate by examples of useful bits of code, results, and questions. Turn to the reference materials for further explanation.

This text is almost a one-to-one conversion from HTML to LaTeX of the tutorial written by Nik Silver of the School of Computer Studies at the University of Leeds. Some parts were taken from the tutorial by Greg Johnson of the MU Information & Access Technology Services. In addition to the conversion to LaTeX I did adapt the text to our installation, added information about `perldoc`, Perl Modules, and safe coding style.

1 A basic program

Here is the basic perl program that we'll use to get started.

```
#!/usr/sepp/bin/perl -w
#
# Program to do the obvious

print 'Hello world.';   # Print a message
```

Each of the parts will be discussed in turn.

The first line

Every perl program starts off with this as its very first line:

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

although this may vary from system to system. This line tells the machine what to do with the file when it is executed (i.e. it tells it to run the file through Perl).

The `-w` option turns on diagnostic warnings. Bad things will happen if you don't use that option.

Comments and statements

Comments can be inserted into a program with the `#` symbol, and anything from the # to the end of the line is ignored (with the exception of the first line). The only way to stretch comments over several lines is to use a # on each line.

Everything else is a Perl statement which must end with a semicolon, like the last line above.

Simple printing

The `print` function outputs some information. In the above case it prints out the the literal string `Hello world.` and of course the statement ends with a semicolon.

Learn how to learn

There is online documentation covering all aspects of Perl programming. You can access it at any time with `perldoc`. `perldoc` is probably the most important thing to learn in this course, from there you can learn all you need.

```
perldoc perl
```

Try out some other pages you find there. Make yourself comfortable with it.

Another very useful way of calling `perldoc` is

You can use Perl modules by just writing `use Module-name` at the top of your program. `strict` mentioned earlier is for example a Perl Module which is part of the standard Perl distribution. `use` will import the subroutines in that module into your program. Every Perl Module should have documentation that you can access with `perldoc`.

If the module you need is not installed already, you can compile it yourself and use it. Compilation and installation of a Perl module normally happens as follows:

```
cd Module-source-directory
perl Makefile.PL PREFIX=/home/user
gmake
gmake install
```

This will install Perl modules into `/home/user/lib/site_perl`, any program in `/home/user/bin` and man-pages in `/home/user/man`

You can then with the following statement at the top of your script

```
use lib "/home/user/lib/site_perl";
```

tell Perl to look also in that place for modules.

Bonus Exercise

Print your full name in Morse.

```
        if(sometest) {
                # $x: 2
                my $y = 3;
                # $y: 3
        }
        # $x: 2
        # $y: undefined
}
# $x: 1
```

Good programming style

First of all:

Always use -w

Second, if you don't use my scoped variables, every variable is created in the global name space, which works ok for small programs, but can be a major source of errors because of conflicts.

```
use strict;
```

At the beginning of the program (after the first line with the interpreter), enforces every variable to be lexically scoped or marked specially otherwise. If a variable is used without being first declared, then Perl will make an error. You can also use my in the program not inside a block, in which case the scope will be the current file.

Exercise

Modify your programs to use -w (didn't you do it already?) and use strict.

Exercise

Modify your Fibonacci program to assign to a scalar variable the passed argument stored in $_[0].

15 Perl Modules

One of Perl's major strengths is it's incredible library of subroutines, called *Modules*. If you have to do some task, which is not very specific to your program but that can be generalized, there is a high probability that you will find the required functionality in form of a Module. Have a look at

```
http://search.cpan.org/
```



```
perldoc -f FUNCTION
```

Where FUNCTION is any Perl function such as open. Try it...
You can read documentation about *Perl modules* (libraries) this way

```
perldoc Module::Name
perldoc Name.pm
```

The first form searches the module Module::Name in Perl's search path and show it's documentation. Try it for example with File::Copy.
The second form does access the documentation from the file Name.pm.

2 Running the program

Type in the example program using a text editor, and save it. Emacs is a good editor to use for this because it has its own Perl mode which formats lines nicely when you hit tab (use M-x cperl-mode). But as ever, use whichever you're most comfortable with.
After you've entered and saved the program make sure the file is executable by using the command
chmod +x *progname*
at the UNIX prompt, where *progname* is the filename of the program. Now to run the program just type any of the following at the prompt.

```
perl -w progname
./progname
```

When the file is executed Perl first compiles it and then executes that compiled version. So after a short pause for compilation the program should run quite quickly. This also explains why you can get compilation errors when you execute a Perl file which consists only of text.
Make sure your program works before proceeding. The program's output may be slightly unexpected - at least it isn't very pretty. We'll look next at variables and then tie this in with prettier printing.

3 Scalar variables

In Perl terminology a *scalar* is a single value, either a number or a string. The most basic kind of variable in Perl is the *scalar variable*. Scalar variables are always prefixed with the $ symbol. They can hold both strings and numbers, and are remarkable in that strings and numbers are completely interchangeable. For example, the statement

```
$priority = 9;
```

sets the scalar variable $priority to 9, but you can also assign a string to exactly the same variable:

```
$priority = 'high';
```

Perl also accepts numbers as strings, like this:

```
$priority = '9';
$default = '0009';
```

and can still cope with arithmetic and other operations quite happily.

In general variable names consists of numbers, letters and underscores, but they should not start with a number and the variable `$_` is special, as we'll see later. Also, Perl is case sensitive, so `$a` and `$A` are different.

Operations and Assignment

Perl uses all the usual C arithmetic operators:

```
$a = 1 + 2;     # Add 1 and 2 and store in $a
$a = 3 - 4;     # Subtract 4 from 3 and store in $a
$a = 5 * 6;     # Multiply 5 and 6
$a = 7 / 8;     # Divide 7 by 8 to give 0.875
$a = 9 ** 10;   # Nine to the power of 10
$a = 5 % 2;     # Remainder of 5 divided by 2
++$a;           # Increment $a and then return it
$a++;           # Return $a and then increment it
--$a;           # Decrement $a and then return it
$a--;           # Return $a and then decrement it
```

and for strings Perl has the following among others:

```
$a = $b . $c;   # Concatenate $b and $c
$a = $b x $c;   # $b repeated $c times
```

To assign values Perl includes

```
$a = $b;    # Assign $b to $a
$a += $b;   # Add $b to $a
$a -= $b;   # Subtract $b from $a
$a .= $b;   # Append $b onto $a
```

Note that when Perl assigns a value with `$a = $b` it makes a copy of `$b` and then assigns that to `$a`. Therefore the next time you change `$b` it will not alter `$a`.

Other operators can be found on the **perlop** manual page.

Type `perldoc perlop` (or `man perlop`) at the prompt.

You can (and should) also use the `return` statement, which returns from a subroutine. This results in much more readable code:

```
sub maximum
{
        ($a, $b) = @_;
        if ($a > $b) { return $a; }
        else         { return $b; }
}
```

Exercise

Write a program that outputs the Fibonacci number sequence. Hint: use directly the `$_[0]` variable, do not assign it to variables otherwise it will not work (try!).

14 Variable scope

The `@_` variable is local to the current subroutine (current *scope*), and so of course are `$_[0]`, `$_[1]`, and so on. Other variables can be made local too with the `my` statement. The following subroutine tests to see if one string is inside another, spaces not withstanding.

```
sub inside
{
        my ($a, $b) = @_;
        $a =~ s/ //g;
        $b =~ s/ //g;

        # Is $b inside $a or $a inside $b?
        return ($a =~ /$b/ || $b =~ /$a/);
}

inside("lemon", "dole money");        # true
```

When you declare a `my` variable, it is lexically scoped to the current block. It means that it is visible only inside that block or nested blocks. If a variable with the same name exists on the outside scope, it replaces it for the duration of the block. For example:

```
$x = 1;   # global variable

# $x: 1
sub a {
        # $x: 1
        my $x = 2;
        # $x: 2
```

Parameters

In the above examples parameters are accepted (enclosed in parentheses after the subroutine call) but ignored. When the subroutine is called any parameters are passed as a list in the special `@_` array variable (note that this variable has absolutely nothing to do with the `$_` scalar variable). The following subroutine merely prints out the list that it was called with. It is followed by a couple of examples of its use.

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

printargs("perly", "king");       # prints "perly king"
printargs("frog", "and", "toad"); # Prints "frog and toad"
```

The individual elements of `@_` can be accessed with the square bracket notation:

```
sub printfirsttwo
{
        print "Your first argument was $_[0]\n";
        print "and $_[1] was your second\n";
}
```

Again it should be stressed that the indexed scalars `$_[0]`, `$_[1]` and so on have nothing to with the scalar `$_` which can also be used without fear of a clash.

Returning values

Result of a subroutine is always the last thing evaluated. This subroutine returns the maximum of two input parameters.

```
sub maximum
{
        ($a, $b) = @_;
        if ($a > $b) { $a; }
        else         { $b; }
}
```

The `printfirsttwo` subroutine above also returns a value, in this case 1. This is because the last thing that subroutine did was a `print` statement and the result of a successful `print` statement is always 1.

Note that it is good coding style to immediately assign the values of the `@_` array containing the subroutine arguments. You can give your arguments names, to make the subroutine easier to read.

Interpolation

The following code prints `apples and pears` using concatenation:

```
$a = 'apples';
$b = 'pears';
print $a.' and '.$b;
```

It would be nicer to include only one string in the final print statement, but the line

```
print '$a and $b';
```

prints literally `$a` and `$b` which isn't very helpful. Instead we can use the double quotes in place of the single quotes:

```
print "$a and $b";
```

The double quotes force *interpolation* of any codes, including interpreting variables. This is a much nicer than our original statement. Other codes that are interpolated include special characters such as newline and tab. The code `\n` is a newline and `\t` is a tab.

Exercise

This exercise is to rewrite the Hello world program so that (a) the string is assigned to a variable and (b) this variable is then printed with a newline character.

4 Array variables

A slightly more interesting kind of variable is the *array variable* which is a list of scalars (i.e. numbers and strings). Array variables have the same format as scalar variables except that they are prefixed by an `@` symbol. The statement

```
@food  = ("apples", "pears", "eels");
@music = ("whistle", "flute");
```

assigns a three element list to the array variable `@food` and a two element list to the array variable `@music`.

The array is accessed by using indexes starting from 0, and square brackets are used to specify the index. The expression

```
$food[2]
```

returns `eels`. Notice that the `@` has changed to a `$` because the array element `eels` is a scalar.

List and scalar context

The interpretation of operations and values in Perl sometimes depends on the requirements of the context around the operation or value. There are two types of context: list and scalar. Certain operations return list values in contexts wanting a list, and scalar values otherwise.

For example if you assign an array variable, the assigned expression will be evaluated in list context and if you assign a scalar variable, the expression will be evaluated in scalar context.

```
@colors = ("blue", "red", "yellow");
@copy   = @colors;                    # list context
$size   = @colors;                    # scalar context
```

In the second line `@colors` is evaluated in list context because it is an assignment to a array variable. An array variable evaluated in list context just returns all its elements.

In the third line however, `@colors` is evaluated in scalar context. An array variable evaluated in scalar context returns its number of elements. Thus `$size` is assigned the value 3.

Sometimes it is useful to force a scalar context when it would be a list context. You can use for that the `scalar` function.

```
@colors = ("blue", "red", "yellow");
@test   = scalar @colors;             # $test[0] is 3
```

Array assignments

The first assignment below explodes the `@music` variable (because it is in a list context) so that it is equivalent to the second assignment:

```
@moremusic = ("organ", @music, "harp");
@moremusic = ("organ", "whistle", "flute", "harp");
```

This should suggest a way of adding elements to an array. A neater way of adding elements is to use the `push` function:

```
push(@food, "eggs");
```

which pushes `eggs` onto the end of the array `@food`. To push two or more items onto the array use one of the following forms:

```
push(@food, "eggs", "lard");
push(@food, ("eggs", "lard"));
push(@food, @morefood);
```

The `push` function returns the length of the new list.

To remove the last item from a list and return it use the `pop` function. From our original list the `pop` function returns `eels` and `@food` now has two elements:

This will only split the line in (maximally) two parts.

- To tidy up the line you'll need the `substr` function. Here are three examples of its use.

```
substr("Once upon a time", 3, 4);   # returns "e up"
substr("Once upon a time", 7);      # returns "on a time"
substr("Once upon a time", -6, 5);  # returns "a tim"
```

The first example returns a substring of length 4 starting at position 3. Remember that the first character of a string has index 0. The second example shows that missing out the length gives the substring right to the end of the string The third example shows that you can also index from the end using a negative index. It returns the substring that starts at the 6th character from the end and has length 5.

If you use a negative index that extends beyond the beginning of the string then Perl will return nothing or give a warning. To avoid this happening you can pad out the string by using the `x` operator mentioned earlier. The expression `" "x30` produces 30 spaces, for example.

13 Subroutines

Like any other programming language Perl allows the user to define their own functions, called *subroutines*. They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end. A subroutine has the form

```
sub mysubroutine
{
        print "Not a very interesting routine\n";
        print "This does the same thing every time\n";
}
```

regardless of any parameters that we may want to pass to it. The subroutine is then called just with `mysubroutine`.

The subroutine must be defined when it is first used the program. If the subroutine definition comes after it's use in the file, you can put at the top a *prototype declaration* like this:

```
sub b;                        # prototype
sub a { b; }                  # b is already used
sub b { print "foo-bar"; }    # definition
```

But this:

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:/);
```

would be like

```
@personal = ("Capes", "Geoff", "",
             "Shot putter", "", "", "Big Avenue");
```

A word can be split into characters, a sentence split into words and a paragraph split into sentences:

```
@chars = split(//, $word);
@words = split(/\s+/, $sentence);
@sentences = split(/\.\s*/, $paragraph);
```

In the first case the null string is matched between each character, and that is why the `@chars` array is an array of characters - i.e. an array of strings of length 1.

Bonus Exercise

A useful tool in natural language processing is *concordance*. This allows a specific string to be displayed in its immediate context wherever it appears in a text. For example, a concordance program identifying the target string `the` might produce some of the following output. Notice how the occurrences of the target string line up vertically.

```
discovered (this is the truth) that when he
t kinds of metal to the leg of a frog, an e
normous advances in the field of amphibian
ond -- almost.  But the greatest Electrical
ectrical Pioneer of them all was Thomas Edi
rrent developed and the frog's leg kicked,
 longer attached to the frog, which was dea
```

This exercise is to write such a program. Here are some tips:

- Read one line at a time from the file.
- Remove the line-feed with `chomp` from the line.
- Ignore lines that do not contain the target string and split the ones that do in two parts: what is before the target string and what is after. Use this form of split:

  ```
  ($a, $b) = split(/$target/, $line, 2);
  ```



```
$grub = pop(@food);  # Now $grub = "eels"
```

Arrays can also be used to make multiple assignments to scalar variables:

```
($a, $b) = ($c, $d);     # Same as $a=$c; $b=$d
($a, $b) = @food;        # $a,$b: first 2 items of @food
($a, @somefood) = @food; # $a is the first item of @food
                         # @somefood is a list of the others
(@bla, $a) = @food;      # @bla is @food and $a is undefined
```

The last assignment occurs because arrays are greedy, and `@somefood` will swallow up as much of `@food` as it can.

Displaying arrays

Since context is important, it shouldn't be too surprising that the following all produce different results:

```
print @food;        # By itself
print "@food";      # In double quotes (interpolation)
print scalar @food; # In a scalar context
```

The second line turns the list into a string with a space between each element. This space can be replaced by any other string by changing the value of the special `$"` variable.

Exercise

Try out each of the above three print statements to see what they do. Try to modify `$"` and see how it changes the output of array interpolation.

5 File handling

Here is a simple perl program which does the same as the UNIX `cat` command on a certain file.

```
#!/usr/sepp/bin/perl -w
#
# Program to open the password file, read it in,
# print it, and close it again.

$file = '/etc/passwd'; # Name the file
open(INFO, $file) or   # Open the file
    die "Can't open $file: $!\n";
@lines = <INFO>;       # Read it into an array
close(INFO);           # Close the file
print @lines;          # Print the array
```

The `open` function opens a file for input (i.e. for reading). The first parameter is the *file-handle* which allows Perl to refer to the file in future. Note that these special variables have no prefix and are traditionally written all up-case. The second parameter is an expression denoting the filename. If the filename was given in quotes then it is taken literally without shell expansion.

`open` returns false if an error occurred and the special variable `$!` is automatically set to the error in that case. Note on the second function the use of the `die` function which terminates the program with an error message. `or` (same as `||`, but with lower precedence) does evaluate the expression on the left and only if it is false it evaluates also the expression on the riht (in this case the `die`) to see if it is true.

The `close` function tells Perl to finish with that file.

There are a few useful points to add to this discussion on file handling. First, the `open` statement can also specify a file for output and for appending as well as for input. To do this, prefix the filename with a > for output and a >> for appending:

```
open(INFO, $file);     # Open for input
open(INFO, ">$file");  # Open for output
open(INFO, ">>$file"); # Open for appending
open(INFO, "<$file");  # Also open for input
```

Second, if you want to print something to a file you've already opened for output then you can use the print statement with an extra parameter. To print a string to the file with the INFO file-handle use

```
print INFO "This line goes to the file.\n";
```

To read from a file-handle you can use the *angle brackets* operator on it like this:

```
@lines = <INFO>;
```

this line reads the file denoted by the file-handle into the array `@lines`. Note that the <INFO> expression reads in the file entirely in one go. This because the reading takes place in the context of an array variable. If @lines is replaced by the scalar `$lines` then only the next one line would be read in. In either case each line is stored complete with its newline character at the end.

You can use the file-handles `STDIN` and `STDOUT` to read from standard input or write to standard output (`STDOUT` is implicit if no file-handle is given to `print`).

Exercise

Write a program that asks for the first name of the user and prints out “Hello” followed by that name. Hint: use the `STDIN` file-handle.

Bonus Exercise

For a slightly more interesting program you might like to try the following. Suppose your program is called `countlines`. Then you would call it with

```
./countlines
```

However, if you call it with several arguments, as in

```
./countlines first second etc
```

then those arguments are stored in the array `@ARGV`. In the above example we have `$ARGV[0]` is `first` and `$ARGV[1]` is `second` and `$ARGV[2]` is `etc`. Modify your program so that it accepts one argument and counts only those lines with that string. It should also put occurrences of this string in parentheses. So that

```
./countlines the
```

will output something like this line among others:

```
019 But (the) greatest Electrical Pioneer of (the)m all was
```

12 Split

A very useful function in Perl is `split`, which splits up a string and places it into an array. The function uses a regular expression and as usual works on the `$_` variable unless otherwise specified. It is used like this:

```
$info = "Caine:Michael:Actor:14, Leafy Drive";
@personal = split(/:/, $info);
```

which has the same overall effect as

```
@personal = ("Caine", "Michael", "Actor", "14, Leafy Drive");
```

If we have the information stored in the `$_` variable then we can just use this instead

```
@personal = split(/:/);
```

If the fields are divided by any number of colons then we can use the RE codes to get round this. The code

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:+/);
```

is the same as

```
@personal = ("Capes", "Geoff",
             "Shot putter", "Big Avenue");
```

Remembering patterns

It's often useful to remember patterns that have been matched so that they can be used again. It just so happens that anything matched in parentheses gets remembered in the variables `$1`,`$2`,...,`$n`. These strings can be used after the match or also in the substitution text. For example

```
$_ = "Lord Whopper of Fibbing";
s/([A-Z])/:$1:/g;
print "$_\n";
```

will replace each upper case letter by that letter surrounded by colons. The variables `$1`,`$2`,...,`$n` are read-only variables; you cannot alter them yourself.

The following swaps the first and last characters of a line in the `$_` variable:

```
s/^(.)(.*)(.)$/$3$2$1/
```

The `^` and `$` match the beginning and end of the line. The `$1` variable stores the first character; the `$2` variable stores everything else up the last character which is stored in the `$3` variable. Then that whole line is replaced with `$1` and `$3` swapped round.

You can refer the matched groups in the regular expression itself, but then you have to use `\1`, `\2`, etc. instead of `$1`, `$2`, etc. You can for example write:

```
/\b(\w+)\b.*\1/
```

This will match any line containing double words (word-boundary, word, word-boundary, any text, same word).

Finally on the subject of remembering patterns it's worth knowing that inside of the slashes of a match or a substitution variables are interpolated. So

```
$search = "the";
s/$search/xxx/g;
```

will replace every occurrence of `the` with `xxx`. If you want to replace every occurrence of `there` then you cannot do `s/$searchre/xxx/` because this will be interpolated as the variable `$searchre`. Instead you should put the variable name in curly braces so that the code becomes

```
$search = "the";
s/${search}re/xxx/;
```

Exercise

Your current program should count lines of a file which contain a certain string. Modify it so that it counts lines with double letters (or any other double character). Modify it again so that these double letters appear also in parentheses. For example your program would produce a line like this among others:

```
023 Amp, James Wa(tt), Bob Transformer, etc. These pion(ee)rs
```

Try to get it so that all pairs of letters are in parentheses, not just the first pair on each line.

Bonus Exercise

Modify the passwd-printing program so that the entire file is printed with a `#` symbol at the beginning of each line. You should only have to add one line and modify another. Use the `$"` variable.

6 Control structures

More interesting possibilities arise when we introduce control structures and looping. Perl supports lots of different kinds of control structures which tend to be like those in C, but are very similar to Pascal, too. Here we discuss a few of them.

foreach

To go through each line of an array or other list-like structure (such as lines in a file) Perl uses the `foreach` structure. This has the form

```
foreach $morsel (@food) # Visit each item of @food in
                        # turn and call it $morsel
{
        print "$morsel\n";  # Print the item
        print "Yum yum\n";  # That was nice
}
```

The actions to be performed each time are enclosed in a block of curly braces. The first time through the block `$morsel` is assigned the value of the first item in the array `@food`. Next time it is assigned the value of the second item, and so until the end. If `@food` is empty to start with then the block of statements is never executed.

Testing

The next few structures rely on an expression being true or false. In Perl any non-zero number and non-empty string is counted as true. The number zero, zero by itself in a string, and the empty string are counted as false. Here are some tests on numbers and strings.

```
$a == $b  # Is $a numerically equal to $b?
          # Beware: Don't use the = operator.
$a != $b  # Is $a numerically unequal to $b?
$a eq $b  # Is $a string-equal to $b?
$a ne $b  # Is $a string-unequal to $b?
```

You can also use logical and, or and not:

```
($a && $b)  # Are $a and $b true?
($a || $b)  # Are $a or $b true?
!($a)       # Is $a false?
```

for

Perl has a `for` structure that mimics that of C. It has the form

```
for (initialize; test; inc)
{
        first_action;
        second_action;
        etc
}
```

First of all the statement *initialize* is executed. Then while *test* is true the block of actions is executed. After each time the block is executed *inc* takes place. Here is an example `for` loop to print out the numbers 0 to 9.

```
for ($i = 0; $i < 10; $i++)  # Start with $i = 0
                             # Do it while $i < 10
                             # Increment $i after each pass
{
        print "$i\n";
}
```

while and until

Here is a program that reads some input from the keyboard and won't continue until it is the correct password

```
#!/usr/sepp/bin/perl -w
print "Password? ";         # Ask for input
$a = <STDIN>;               # Get input
chomp $a;                    # Remove the newline at end
while ($a ne "fred")        # While input is wrong...
{
    print "sorry. Again? "; # Ask again
    $a = <STDIN>;           # Get input again
    chomp $a;               # Chop off newline again
}
```

The curly-braced block of code is executed while the input does not equal the password. `while` executes the block as long as its test argument is true. The `chomp` function removes any newline character at the end of the string.

To test the opposite thing we can use the `until` statement in just the same way. This executes the block repeatedly until the expression is true, not while it is true.

- the word `the` with or without a capital. Use `\b` to detect word boundaries.

In each case the program should print out every line, but it should only number those specified.

11 Substitution

As well as identifying regular expressions Perl can make substitutions based on those matches. The way to do this is to use the `s///` operator. Once again the match operator is used, and once again if it is omitted then the substitution is assumed to take place with the `$_` variable.

To replace an occurrence of `london` by `London` in the string `$sentence` we use the expression

```
$sentence =~ s/london/London/
```

and to do the same thing with the `$_` variable just

```
s/london/London/
```

Notice that the two strings (`london` and `London`) are surrounded by a total of three slashes. The first string is a regular expression matching a text and the second string is a text that will replace the regular expression.

The result of this expression is the number of substitutions made, so it is either 0 (false) or 1 (true) in this case.

Options

This example only replaces the first occurrence of the string, and it may be that there will be more than one such string we want to replace. To substitute all matches, in other words to make a *global* substitution the last slash is followed by a `g` as follows:

```
s/london/London/g
```

Again the expression returns the number of substitutions made, which is 0 (false) or something greater than 0 (true).

If we want to also replace occurrences of `lOndon`, `lonDON`, `LoNDoN` and so on then we could use

```
s/[Ll][Oo][Nn][Dd][Oo][Nn]/London/g
```

but an easier way is to use the `i` option (for *ignore case*). The expression

```
s/london/London/gi
```

will make a global substitution ignoring case. The `i` option can also be used in the basic `/.../` regular expression match.

```
\S       # Any non-whitespace character
\b       # A word boundary, outside [] only
\B       # No word boundary
```

See `perldoc perlre` for an exact definition of these, plus others. To search for `\b` do first initiate search with `/` and then type `\\b`. The program used to display man-pages (`less`) does also use regular expressions for searching, so you have to write a double backslash to get one literal backslash.

Clearly characters like $, |, [,), /, ., and so on are peculiar cases in regular expressions. If you want to match for one of those then you have to precede it by a backslash. So:

```
\|      # Vertical bar
\[      # An open square bracket
\)      # A closing parenthesis
\*      # An asterisk
\^      # A carat symbol
\/      # A slash
\\      # A backslash
```

and so on.

Some example REs

As was mentioned earlier, it's probably best to build up your use of regular expressions slowly. Here are a few examples. Remember that to use them for matching they should be put in `/.../` slashes

```
[01]        # Either "0" or "1"
\/0         # A division by zero: "/0"
\/ 0        # A division by zero with a space: "/ 0"
\/\s0       # A division by zero with a whitespace:
            # "/ 0" where the space may be a tab etc.
\/ *0       # A division by zero with possibly some
            # spaces: "/0" or "/ 0" or "/  0" etc.
\/\s*0      # A division by zero with possibly whitespace.
```

Exercise

Previously your program counted non-empty lines. Alter it so that instead of counting non-empty lines it counts only lines with

- the letter `x`
- the string `the`
- the string `the` which may or may not have a capital `t`

Another useful technique is putting the `while` or `until` check at the end of the statement block rather than at the beginning. This will require the presence of the `do` operator to mark the beginning of the block and the test at the end. If we forgot the '`sorry. Again?`' message in the above password program then it could be written like this:

```
#!/usr/sepp/bin/perl
do {
        print "Password? ";   # Ask for input
        $a = <STDIN>;   # Get input
        chop $a;        # Chop off newline
} while ($a ne "fred"); # Redo while wrong input
```

Exercise

Modify the passwd-printing program so that each line of the file is read in one by one and is printed with a line number at the beginning. You should get something like:

```
1 a b c
2 foo bar
3 example
...
```

You may find it useful to use the structure

```
while ($line = <INFO>) {
   ...
}
```

Bonus Exercise

When you have done this see if you can alter it so that line numbers are printed as 001, 002, ..., 009, 010, 011, 012, etc. Hint: Perl also has `printf`.

7 Conditionals

Of course Perl also allows `if`/`then`/`else` statements. These are of the following form:

```
if ($a) {
        print "The string is not empty\n";
}
else {
        print "The string is empty\n";
}
```

For this, remember that an empty string is considered to be false. It will also give an "empty" result if `$a` is the string 0.

It is also possible to include more alternatives in a conditional statement:

```
if (!$a) {
        # The ! is the negation operator
        print "The string is empty\n";
}
elsif (length($a) == 1) {
        # If above fails, try this
        print "The string has one character\n";
}
elsif (length($a) == 2) {
        # If that fails, try this
        print "The string has two characters\n";
}
else {
        # Now, everything has failed
        print "The string has lots of characters\n";
}
```

In this, it is important to notice that the elsif statement really does have an "e" missing.

Exercise

Find a fairly large file that contains some text and some blank lines. You can take this one:

```
/home/dws/checkouts/isg/courses/perl/2002-dws/electricity.txt
```

From the previous exercise you should have a program which prints out the password file with line numbers. Change it so that it uses that text file. Now alter the program so that line numbers aren't printed or counted with blank lines, but every line is still printed, including the blank ones. Remember that when a line of the file is read in it will still include its newline character at the end.

8 Hashes

Ordinary list arrays allow us to access their element by number. The first element of array `@food` is `$food[0]`. The second element is `$food[1]`, and so on. But Perl also allows us to create arrays which are accessed by string. These are called *associative arrays* or *hashes*.

To define a hash we use the usual parenthesis notation, but the array itself is prefixed by a `%` sign. The assigned elements must be in sequence the key and


```
t.e     # t followed by anything followed by e
        # This will match the, tre and tle
        # but not te or tale
^f      # f at the beginning of a line
^ftp    # ftp at the beginning of a line
e$      # e at the end of a line
tle$    # tle at the end of a line
und*    # un followed by zero or more d characters
        # This will match un
        #                 und
        #                 undd
        #                 unddd (etc)
.*      # Any string without a newline. This is because
        # the . matches anything except a newline and
        # the * means zero or more of these.
^$      # A line with nothing in it.
```

There are even more options. Square brackets are used to match any one of the characters inside them. Inside square brackets a `-` indicates "between" and a `^` at the beginning means "not":

```
[qjk]           # Either q or j or k
[^qjk]          # Neither q nor j nor k
[a-z]           # Anything from a to z inclusive
[^a-z]          # No lower case letters
[a-zA-Z]        # Any letter
[a-z]+          # Any non-zero sequence of lower case letters
```

At this point you can probably skip to the end and do at least most of the exercise. The rest is mostly just for reference.

A vertical bar `|` represents an "or" and parentheses `(...)` can be used to group things together:

```
jelly|cream     # Either jelly or cream
(eg|le)gs       # Either eggs or legs
(da)+           # Either da or dada or dadada or...
```

Here are some more special characters:

```
\n       # A newline
\t       # A tab
\w       # Any alphanumeric (word) character.
         # The same as [a-zA-Z0-9\_]
\W       # Any non-word character.
         # The same as [^a-zA-Z0-9\_]
\d       # Any digit. The same as [0-9]
\D       # Any non-digit. The same as [^0-9]
\s       # Any whitespace character: space, tab, newline, etc
```

The RE is case sensitive, so if

```
$sentence = "The quick brown fox";
```

then the above match will be false. The operator `!~` is used for spotting a non-match. In the above example

```
$sentence !~ /the/
```

is true because the string `the` does not appear in `$sentence`.

The $_ special variable

We could use a conditional as

```
if ($sentence =~ /under/) {
        print "We're talking about rugby\n";
}
```

which would print out a message if we had either of the following

```
$sentence = "Up and under";
$sentence = "Best winkles in Sunderland";
```

But it's often much easier if we assign the sentence to the special variable `$_` which is of course a scalar. If we do this then we can avoid using the match and non-match operators and the above can be written simply as

```
if (/under/) {
        print "We're talking about rugby\n";
}
```

The `$_` variable is the default for many Perl operations.

More on REs

In a RE there are plenty of special characters, and it is these that both give them their power and make them appear very complicated. It's best to build up your use of REs slowly; their creation can be something of an art form.

Here are some special RE characters and their meaning

```
.    # Any single character except a newline
^    # The beginning of the line or string
$    # The end of the line or string
*    # Zero or more of the last character
+    # One or more of the last character
?    # Zero or one of the last character
```

and here are some example matches. Remember that should be enclosed in `/.../` slashes to be used.

values of the hash. Suppose we want to create a hash of people and their ages. It would look like this:

```
%ages = ("Michael Caine", 39,
        "Dirty Den", 34,
        "Willy", "21 in dog years",
        "The Queen Mother", 108);
```

Now we can find the age of people with the following expressions

```
$ages{"Michael Caine"};      # Returns 39
$ages{"Willy"};              # Returns "21 in dog years"
```

Notice that like list arrays each `%` sign has changed to a `$` to access an individual element because that element is a scalar. Unlike list arrays the index (in this case the person's name) is enclosed in curly braces, the idea being that hashes are fancier than list arrays.

To make it more readable, you can also use `=>` instead of commas to separate keys from values. `=>` is like a comma, but also forces the interpretation of what is on its left as a string.

```
%me = (
        last_name  => "Schweikert",
        first_name => "David",
        email      => 'dws@ee.ethz.ch',
);
```

Note the unnecessary comma after the email: Perl will ignore it.

Operators

Hashes do not have any order to their elements but is it possible to access all the elements in turn using the `keys` function and the `values` function:

```
foreach $person (keys %ages)
{
        print "$person is $ages{$person}\n";
}

foreach $age (values %ages)
{
        print "Somebody is $age\n";
}
```

When `keys` is called it returns a list of the keys (indexes) of the hash. When `values` is called it returns a list of the values of the array. These functions return their lists in the same order, but this order has nothing to do with the order in which the elements have been entered.

When `keys` and `values` are called in a scalar context they return the number of key/value pairs in the hash.

Environment variables

When you run a perl program, or any script in UNIX, there will be certain environment variables set. These will be things like USER which contains your username and DISPLAY which specifies which screen your graphics will go to. All these variables and their values are stored in the special hash `%ENV` in which the keys are the variable names. `%ENV` is already set at the beginning and you can also use it to change the environment variables by changing the hash.

Exercise

Write a program that prints out all environment variables like the `setenv` command does.

9 References and Complex Data Structures

A "reference" in Perl is a scalar value that points to an array or a hash. It is a sort of placeholder through which its referenced array or hash can be accessed. Wherever you can use a scalar you can use a reference. The main use of references is building complex data structures such as multidimensional arrays or nested hashes.

Creating references

You can create a reference to a hash or array contained in a variable by prefixing the variable with a backslash `\` like this:

```
@colors     = ( 'red', 'green', 'blue' );
$colors_ref = \@colors;
%ages       = ( john => 30, alice => 35 );
%ages_ref   = \%ages;
```

You can also create references to arrays or hashes on the fly by using for arrays square brackets and for hashes curly brackets:

```
$colors_ref = [ 'red', 'green', 'blue' ];
$ages_ref   = { john => 30, alice => 35 };
```

Since references are scalars, you can put them in a array or hash:

```
@matrix = ( [ 1, 2, 3 ], [4, 5, 6] );
%people = (
     john  => { sex => 'm', age => 30 },
     alice => { sex => 'f', age => 35 }
);
```

Note the use of parentheses or square (curly) braces to create either arrays (hashes) or references to arrays (hashes).

Accessing referenced values

You can access the referenced array by using `{$ref}` everywhere you would put the name of the array or hash. For example:

```
@colors_copy = @{$colors_ref};      # @colors
$first_color = ${$colors_ref}[0];   # $colors[0]
%ages_copy   = %{$ages_ref};        # %ages
$johns_age   = ${$ages_ref}{'john'} # $ages{'john'}
```

Note that you can actually remove the curly braces if there is no ambiguity in the expression. You can also use the arrow `->` operator to access directly elements of a referenced array or hash:

```
$first_color = ${$colors_ref}[0];
$first_color = $colors_ref->[0];

$johns_age = ${$ages_ref}{'john'};
$johns_age = $ages_ref->{'john'};
```

If you access multiple levels of references, you can keep only one arrow and skip the others. For example:

```
$matrix_a = $matrix->[0]->[0];
$matrix_a = $matrix->[0][0];
```

Have a look at `perldoc perlref` for a full description on how references work and how they can be used to build complex data structures.

Exercise

Write a little address-book application where you have a list of pre-stored addresses in your program and you just print them out. Use a complex data structure such as an array of hashes.

10 String matching

One of the most useful features of Perl is its powerful string manipulation facilities. At the heart of this is the *regular expression* (RE) which is shared by many other UNIX utilities like `grep` or `sed`.

Regular expressions

A regular expression is contained in slashes, and matching occurs with the `=~` operator. The following expression is true if the string `the` appears in variable `$sentence`.

```
$sentence =~ /the/;
```