Perl for Win32 - Basic Tutorial v1.0

Last update : 17th September 1997 12:45 CET

Introduction

What this document is...

A *basic* Perl tutorial for use on Win32 platforms. You won't find objects, closures and various types of references here. It is adapted from a course I ran for novice webmasters, and have now published on the web so a) others may benefit (hopefully) b) perl users can critique the course and suggest additions. The term "perl users" is defined as anyone with more than 20 seconds perl experience, or less. Another objective for writing this course was for me to learn more about Perl. That was certainly achieved :-)

What this document is not...

- A reference manual. You won't find all the regex stuff under Regex. I think it's more fun to learn the basics then add little extras along the way. Keeps you awake, and its a good excuse for not organising it better.
- A FAQ.
- Guaranteed to be 100% technically accurate. But It Works For Me ®
- Politically correct
- Gramatically correct

It doesn't cover CGI programming (I'm thinking of a seperate tutorial for that). Nor does it cover the multitude of modules for Perl. It is designed as a base, after which you can specialise in systems administration, CGI or whatever else.

How to Use..

Just work through from start to finish. All you need is a Win32 PC. When you finish, please send me a critique. In fact, send one even if you don't finish. I appreciate all input, especially error checks ! And money cheques !

All the code examples have been tested, and you can just cut'n'paste. I haven't listed the output of each example. You need to run it and see for yourself. Consider this course interactive.

What you need to know

You need to be able to differentiate between a PC and a toaster. No programming experience is necessary. You do need to understand the basics of PC operation. If you don't understand what directories and files are then you'll find this difficult. You might find it difficult even if you do :-) --
Robert Pepper
mailto:Robert@netcat.co.uk

What is Perl ?

Perl is a programming language. Perl stands for Pratical Report and Extraction Language. You'll notice people refer to 'perl' and "Perl". "Perl" is the programming language as a whole - 'perl' is the name of the core executable. Some of Perl's many strengths are :

- **Speed of development.** You edit a text file, and just run it. You can develop programs very quickly like this. No compiler needed.

- **Power.** Perl's regular expressions are some of the best available. You can work with objects, sockets...everything a systems administrator could want. And that's just the standard distribution. Add the wealth of modules available on CPAN and you have it all. Don't equate scripting languages with toy languages.

- **Usuability.** All that power and capability can be learnt in easy stages. If you can write a batch file you can program Perl. You don't have to learn object oriented programming, but you can write OO programs in Perl. If using incrementing nonexistent variables scares you, make perl refuse to let you. There is always more than one way to do it in Perl. You decide your style of programming, and Perl will accomodate you.

- **Portability.** On the Superhighway to the Portability Panacea, Perl's Porsche powers past Java's jaded jalopy. Many people develop Perl scripts on NT, or Win95, then just FTP them to a Unix server where they run. No modification necessary.

- **Editing tools** You don't need the latest Integrated Development Enviroment for Perl. You can develop Perl scripts with any text editor. Notepad, vi, MS Word 97, or even direct off the console. Of course, you can make things easy and use one of the many freeware or shareware programmer's file editors.

- **Price.** Yes, 0 guilders, pounds, dmarks, dollars or whatever. And the peer to peer support is also free, and often far better than you'd ever get by paying some company to answer the phone and tell you to do what you just tried several times already.

What is Win32 ?

Win32 refers to Microsoft Windows 32-bit operating systems. At the time of writing that's Windows 95 and Windows NT. Win32 does *not* mean Windows 3.11 running Win32s.

What is Perl for Win32 ?

Microsoft decided Perl would be a Good Thing to have on Win32. See, they're not so bad. So they employed Hip Commnications to port Perl to the Win32 platform (that is, take the source code for perl and change it so it runs on Windows). The main developer at Hip, Dick Hardt, later left the company and formed ActiveWare Internet Corp. Dick took Perl for Win32 with him, and continued development. In August 1997, ActiveWare changed their name to ActiveState Tool Corp. "Perl for Win32" is a trademark of ActiveState Tool Corp. However, the last release of the base (native) version will now compile directly for Win32. The latest native version at the time of writing is perl 5.004.

The ActiveState version includes some additional modules and features, not least of which are Perl for ISAPI (perlis.dll) and PerlScript. These don't work (yet) with the native version. But they will soon because ActiveState is merging their version with the native version. This should happen.....soon :-)

Perl is developed on the latest NT platform, and may or may not work on older versions of NT. The latest version of Perl will run on the latest version of Windows 95 (or 98 when it is released). Be aware that some things which work under Windows NT don't work under Windows 95 because Win95 just doesn't have the functionality. In the same way, some Perl features you can use under Unix either don't work, or work differently compared to the Win32 platform. Check the documentation !

What can you do with it ?

Just two popular examples :

The Internet

Go surf. Notice how many websites have dynamic pages with .pl or similar as the filename extension ? That's Perl. It is the most popular language for CGI programming for many reasons, most of which are mentioned above. In fact, there

are a great many more dynamic pages written with perl that may not have a .pl extension. Perl has spread across Internet.

Systems Administration

If you are an NT sysadmin, chances are you aren't used to programming. In which case, the advantages of Perl may not be clear. Do you need it ? Is it worth it ?

After you read this tutorial you will know more than enough to start using Perl productively. You really need very little knowledge to save time. Imagine driving a car for years, then realising it has five gears, not four. That's the sort of improvement learning Perl means. When you are proficient, you find the difference like realisng the same car has a revese gear and you don't have to push it backwards. Perl means you can be lazier. Lazy sysadmins are good sysadmins, as I keep telling my boss. You'll never touch a batch file again !

Support

There are six mailing lists for Perl for Win32. Read all about them on http://www.activestate.com/. Make sure you read the charter too. Many people put time and effort into the creation of those lists, so don't insult us by ignoring the guidelines. Anyone with an interest in Perl for Win32 should be subscribing to at least one of these lists. The charter also lists useful sites and newsgroups.

Setup

Three stages:

1. **Get the software**
2. **Install it**
3. **Run a test Script**

1. Getting the Software

An old version of Perl for Win32 is included with the Windows NT Resource Kit. Please don't use it. It is out of date. Follow the steps below to get a newer version.

The basic Perl for Win32 distribution kit is about 1.5Mb. This comprised of more than 250 files - the basic perl.exe interpreter, library modules (useful addons), documentation etc. Download times are about twice as long as for a 750Kb file. :-)

You might wish to create a root directory for the perl installation. The perl installation contains more than 250 files and it has its own directory structure. This tutorial will assume you are using `c:\perl` as your perl installation directory.

You can use FTP, HTTP (your web browser) or email.

Which file ?

You'll find three binaries for download. Don't worry about PerlScript or PerlIS. These are special versions of Perl for some web servers. If you run Microsoft Internet Information Server, they will be of use but you are *strongly advised* to work with perl a while before you start trying perlis and PerlScript.

As of 8th September 1997 the latest build is 310, so the file to get is pw32i310.exe.

Make sure you do not download pw32a310.exe. This is for a Alpha machine and will not work on an Intel PC. If you are an Alpha user you knew that already :-)

HTTP (Web browsers)

Go to http://www.activestate.com/ and follow the Download link.

FTP

ftp://ftp.linux.ActiveState.com/pub/Perl-Win32/Release

Email

Send an email to ListManager@ActiveState.com with the following commands in the body of the message:

```
GET perl-win32-announce Pw32i310.exe
```

and remember it is a 1.5Mb file, which is quite a large attachment. It is possible it won't make it to your machine for this reason.

---

2. Installation

So you now have pw32i310.exe and it is in c:\perl or whatever directory you are using. Installation is easy. We'll use a command prompt as you will be working with the command prompt later. Run your command prompt....now !

1. Switch to `c:\perl`
2. Run the install program thus : `pw32i310.exe`
3. The install program will offer to unzip into c:\perl. If you are not using c:\perl as your perl installation directory, change the path. Leave both checkboxes about overwriting and when done checked.
4. You'll see a command window. If you have followed these instructions perl has indeed been unpacked into its final destination directory, so you can just respond Y.
5. Allow the search path to be modified
6. Associate perl with .pl ? If you do this you can run a perl script just by doubleclicking it. Personally I prefer doubleclicking to start a text editor and load the script, so I always answer no to this and run scripts from the command line. Plus this way you can pass perl a plethora of command line arguments. So politely refuse the kind request and answer N. If you do decide to associate perl.exe with .pl, change the mapping so perl.exe accepts several parameters.
7. If you are running IIS you'll see a message about I/O redirection. Just say Y. It is a Good Thing. Trust me.
8. You'll need to logon/off as it says for the path to take effect.

---

3. Testing - Your First Perl Script

Assuming all has gone to plan, now create your first Perl script. I reccomend creating a new directory for your perl scripts, seperate to your data files and the perl installation. For example `c:\pscripts\`, which is what I'll assume you are using in this tutorial.

Start up whatever text editor you're going to hack Perl with. Notepad.Exe is just fine. Type in the following :

```
print "My first Perl script\n";
```

and save it to `c:\scripts\myfirst.pl`. You don't need to exit Notepad - keep it open, as we'll be making changes very soon. Switch to your command prompt, and change to the directory. Execute the script :

```
perl myfirst.pl
```

and you'll see the output. Welcome to the world of Perl ! See what I mean about it being easy to start ? However, it is difficult to finish with Perl once you begin :-)

Now we need to analyse what's going on here a little. First note that the line ends with a semicolon `;` . Almost all lines in Perl end with semicolons. Also note the `\n`. This the code to tell Perl to output a newline. If that's not clear, delete the `\n` from the program and run it again like so ;

```
print "My first Perl script";
```

NB - almost every Perl book is written for UN*X, which is a problem for Win32. This leads to scripts like :

```
#!c:/perl/perl.exe

print "I'm a cool Perl hacker\n";
```

The 'shebang' line is *not* necessary under Win32. It will be ignored. You may add one so your scripts run directly on Unix without modification, but Win32 will ignore it. Anyway, on with the lesson.

Variables

So Perl is working, and you are working with Perl. Now for something more interesting than simple printing. Variables. Let's take simple scalar variables first. A **scalar variable is a single value**. Like `$var=10` which sets the variables `$var` to the value of 10. Later, we'll look at lists like arrays and hashes, where `@var` refers to more than one value. **Scalar is Singluar**.

If you've learnt any JavaScript or BASIC you'd be surprised by `$var=10`. With those languages, if you want to assign the value 10 to a variable called `var` you'd write `var=10`.

Not so in Perl. This is a Feature. All variables are prefixed with a symbol such as `$ @ %`. This has certain advantages, like making programs easier to read. You can see where the variables are quite easily. And not only that, what sort of variable it is. The human language German has a similar principle (except nouns are captalised, not prefixed with $ and Perl is easier to pronounce). You'll agree later....

Anyway, more hands-on. Time to try some variables :

```
$string="perl";
$num=20;
print "The string is $string and the number is $num\n";
```

A closer look...notice you don't have to say what type of variable you are declaring. In other languges you need to say if the variable is a string, array, or whatever. You might even have to declare what type of number it is. Yes, there are different types but you don't need to know about them with Perl. Typecasting ? That's not politcally correct any more !

Also notice the way the variables are used in the string. Sticking variables inside of strings has a technical term - **"variable interpolation"**. Now, if we didn't have the handy `$` prefix for we'd have to do something like the example below, which is pseudocode. Pseudocode is code to demonstrate a concept, not designed to be run. Like certain Microsoft software :-)

```
print "The string is ".string." and the number is ".num."\n";
```

which is much more work. Convinced about those prefixes yet ?

If you didn't already know, a tiny little comma out of can lead to completely unexpected results. If the above code didn't work, you haven't typed it in exactly as you should have done. Those are double quotes " , not singles ' .

Single quotes have their use. Try this :

```
$string="perl";
$num=20;
print "Doubles: The string is $string and the number is $num\n";
print 'Singles: The string is $string and the number is $num\n';
```

Double quotes allow the aforementioned variable interpolation. Single quotes do not. Both have their uses as you will see later, depending on wheter you wish to interpolate anything.

More on Variables

If you want to add 1 to a variable you can, logically, do this : `$num=$num+1;`. There is a shorter way to do this, which is `$num++`. This is an autoincrement. Guess what this is : `$num--`. Yes, an autodecrement.

This example illustrates it the above :

```
$num=10;
print "\$num is $num\n";

$num++;
print "\$num is $num\n";

$num--;
print "\$num is $num\n";

$num+=3;
print "\$num is $num\n";
```

The last example demonstrates that it doesn't have to be just the one you add/decrease by.

There's something else new in the code above. The `\` . You can see what this does - it **'escapes'** the special meaning of `$` . That means just the `$` symbol is printed instead of it referring to a variable. Actually `\` has a deeper meaning - it escapes all of Perl's special characters, not just `$` . Also, it turns some non-special characters into something special. Like what ? Like `\n` . Add the magic '\' and the humble 'n' becomes the mightly NewLine ! '\' can also escape itself. So if you want to print '\' try :

```
print "the MS-DOS path is c:\\scripts\\";
```

Of course, that's better done as single quotes, but it's just an example. Oh, '\' is also used for other things like references. But that's not even covered here.

Finally, try this :

```
$string="perl";
$num=20;
$mx=3;

print "The string is $string and the number is $num\n";

$num*=$mx;
$string++;
print "The string is $string and the number is $num\n";
```

Note the easy shortcut `*=` meaning 'mulitply $num by $mx' or, `$num=$num*$mx;`. Of course Perl supports the usual `+ - * / ** %` operators. The last two are exponentiation (to the power of) and modulus (remainder of x divided by y). Also note the way you can increment a string ! Is this language flexible or what ?

Subroutines

Let's take a another look at the example we used to show how the autoincrement system works. Messy, isn't it ? This is Batch File Writing Mentality. Notice how we use exactly the same code four times. Why not just put it in a subroutine ?

```
$num=10;                # sets $num to 10
&print_results;         # prints variable $num

$num++;
&print_results;

$num*=3;
&print_results;

$num/=3;
&print_results;

sub print_results {
        print "\$num is $num\n";
}
```

Easier and neater. The subrountine can go anywhere in your script, at the beginning, end, middle...makes no difference. Personally I put all mine at the bottom and reserve the top part for setting variables and main program flow.

A subroutine is defined by starting with `sub` then the name. After that you need a curly bracket {, then all the code for your subroutine. Finish it off with a closing brace. **The area between the two braces is called a block.** Remember this. There are such things as anonymous subrountines but not here. Everything here has a name.

Subroutines are usually called by prefixing their name with &, like so `&print_results;`. In most circumstances you can forget the `&` prefix but it is wise to leave it for the time being to avoid confusion.

If you are worrying about variable visbility, don't. All the variables we are using so far are visible everywhere. You can restrict visibility quite easily, but that's not important right now. If you weren't worrying about variable visibility, please don't start. (paranoid ?)

Notice a `#` crept in there. That's a comment. Everything after an `#` is ignored. You can't continue it onto a newline however, so if your comment won't fit on one line start a new one with `#` . There are ways to create Plain Old Documentation (POD) and more ways to comment but they are not detailed here.

Comparisons and If

An if statement is simple. `if day = sunday, lie in bed`. A simple test, with two outcomes. Perl conversion (don't run this);

```
if ($day eq "sunday") {
        &lie_in_bed;
}
```

You already know that `&lie_in_bed` is a call to a subroutine. We assume `$day` is set earlier in the program. If `$day` is not equal to weekend &lie_in_bed is not executed (pity). You don't need to say anything else. Try this :

```
$day="sunday";
```

```
if ($day eq "sunday) {
        print "Zzzzz....\n";
}
```

Note the syntax. The `if` statement requires something to test for Truth. This expression must be in (parens), then you have the braces to form a block.

There are many Perl functions which test for Truth. Some are `if, while, unless` . So it is important you know what truth is, as defined by Perl. Here are the three main rules :

1. Any string is true except for `""` and `"0"` .
2. Any number is true except for `0` .
3. Any undefined value is false.

Some example code to illustrate the point :

```
&isit;                  # $test1 is at this moment undefined

$test1="hello";         # a string, not equal to "" or "0"
&isit;

$test1=0.0;             # $test1 is now a number, effectively 0
&isit;

$test1="0.0";           # $test1 is a string, but NOT effectively 0 !
&isit;

sub isit {
        if ($test1) {                           # tests $test1 for truth or not
                print "$test1 is true\n";
        } else {                                # else statement if it is not true
                print "$test1 is false\n";
        }
}
```

The first test fails because `$test1` is undefined. This means it has not been created by assigning a value to it. So according to Rule 3 it is false. The last two tests are interesting. Of course, 0.0 is the same as 0 in a *numeric* context. But it is *not* the same as 0 in a string context, so it is true.

So here we are testing single variables. What's more useful is testing the result of an expression. For example, this is an expression : `$x * 2` and so is this `$day eq "Sunday"` . It is the end result of these expressions that is evaluated for truth.

Another example :

```
if (5 - 5) {
        print "Testnum is true\n";
} else {
        print "Testnum is false\n";
}

$day="Sunday";

$y=($day eq "Sunday");
$x=($day eq "Monday");

print "\$x is $x and \$y is $y\n";
```

The first test fails because 5-5 of course is 0, which is false. Next, we assign the result of comparing `$day` to two

different strings. The first returns the value 1, which is true. The second test doesn't seem to return anything (actually it returns ""), which is false.

Now pay close attention, otherwise you'll end up posting an annoying question somewhere. The symbol `=` is an **assignment operator, *not* a comparison operator.** Therefore :

- `if ($x = 10)` is always true, becuase $x has been *assigned* the value 10 successfully.
- `if ($x == 10)` *compares* the two values, which might not be equal.

There are two types of comparison operator - **numeric** and **string**. You've already seen two, `==` and `eq`. Run this :

```
$foo=291;
$bar=30;

if ($foo < $bar) {
        print "$foo is less than $bar (numeric)\n";
}

if ($foo lt $bar) {
        print "$foo is less than $bar (string)\n";
}
```

Alphabetically, that is in a string context, 291 comes before 30. It is actually decided by the ASCII value, but alphabetically is close enough. Change the numbers around a little. Notice how Perl doesn't care wheter it uses a string comparison operator on a numeric value, or vice versa. **This is typical of Perl's flexbility.** Bondage and discipline are alien concepts to Perl. This flexbility does have a drawback. If you're on a programming precipice, threatening suicide by jumping off, Perl won't talk you out of but will provide several ways of jumping, stepping or falling to your doom while silently watching your early conclusion. So be careful.

The Perl Motto is : "There is More Than One Way to Do It" or TIMTOWTDI. This tutorial doesn't try and mention all possible ways of doing everything. Write your Perl programs the way you want to.

The rest of the operators are ;

Comparison	*Numeric*	*String*
Equal	==	eq
Not equal	!=	ne
Greater than	>	gt
Less than	<	lt
Greater than or equal to	>=	ge
Less than or equal to	<=	le

Just remember :

- if you are testing a value **as a string** there should be only **letters** in your comparsion operator.
- if you are testing a value **as a number** there should only be **non-alpha** characters in your comparison operator
- note 'as a' above. You can test numbers as string and vice versa. Perl never complains.

There is another comparison operator to learn. A very useful pair, and of course there are numeric and string versions. The two are :

`<=>` and `cmp`

These operators return 0 if the two values are equal, 1 if the first is greater or -1 if the second is greater. An example, using the numeric operator (works the same for the string version) :

```
$x=10;
$y=20;

print $x <=> $y,"\n";
```

If you think about it, this operator is not a huge amount of use. Why is it there ? It is useful when sorting lists as you'll see later. It is typical of Perl to provide more than one route to achieve a desired objective. TIMTOWTDI.

Did you notice the new way of printing ? No variable interpolation there. You can guess why. If not, try this :

```
print "$x <=> $y\n";
```

and see how far you get. Perl's `print` operator accepts a list, and lists are delimited by default by commas. There are many more ways to print, and I'll introduce some as we go on. So with the comma we can do things like this :

```
print '$x is ',$x,' and $x * 2 is ',$x*2,"\n";
```

which can save a fair bit of typing. This was `$x * 2` is evaluated as an expression, which is what we want in this case. Notice how single quotes are used where no variable interpolation is needed. Double quotes are used at the end because the `\n` won't be noticed otherwise. There are many, many different ways to print that last example.

More about if statements. Run this :

```
$age=25;
$max=30;

if ($age > $max) {
        print "Too old !\n";
} else {
        print "Young person !\n";
}
```

It is easy to see what `else` does. If the expression is false then whatever in else is carried out. Simple. But what if you want another test ? Perl can do that too.

```
$age=25;
$max=30;
$min=18;

if ($age > $max) {
        print "Too old !\n";
} elsif ($age < $min) {
        print "Too young !\n";
} else {
        print "Just right !\n";
}
```

If the first test fails, the second is evaluated. This carries on until there are no more `elsif` statements, or an `else` statement is reached. An `else` statement is optional.

There is a big difference between the above example the the one below :

```
if ($age > $max) {
        print "Too old !\n";
}
```

```
if ($age < $min) {
        print "Too young !\n";
} else {
        print "Just right !\n";
}
```

If you run it, it will return the same result. However, it is Bad Programming Practice. In this case we are testing a number, but suppose we were testing a string to see if it contained R or S. It is possible that a string could contain *both* R and S. So it would pass both 'if' tests. Using an `elsif` avoids this. As soon as the first statement is true, no more `elsif` statements (and no `else` statement) are executed.

User Input

Sometimes you have to interact with the user. It is a pain, but sometimes necessary, especially for the live ones. To ask for input and do something with it try this :

```
print "Please tell me your name :";
$name=<STDIN>;
print "Thanks for making me happy $name !\n"
```

New things to learn here. Firstly, `<STDIN>` . STDIN is short for Standard Input, which is where all input normally comes from. In this case it is input from the keyboard. Also, the angle brackets <> read from a filehandle. Filehandles are what you use to interact with things such as files, socket connections and more.

So we are reading from the STDIN filehandle. The value is assigned to `$name` and printed. Any idea why the ! ends up on a new line ? on a *new line* on a **newline** ????

As you pressed Enter, you of course included a newline with your name. The easy way to get rid of it is to `chop` it like so :

```
print "Please tell me your name :";
$name=<STDIN>;
chop $name;
print "Thanks for making me happy $name !\n"
```

and that works as it should. chop removes the last character of whatever it is given to chop, in this case removing the newline for us. In fact, that can be shortened :

```
print "Please tell me your name :";
chop ($name=<STDIN>);
print "Thanks for making me happy $name !"
```

The parentheses () force `chop` to act on the result of what is inside them. So `$name=<STDIN>` is carried out first, then the result from that, which is $name, is chopped.

Arrays

Perl has two types of array, associative arrays (hashes) and arrays. Hashes are covered later. This is all about arrays.

An array is an ordered list of scalar variables. This list can be referred to as a whole, or you can refer to indviudal elements in the list. The program below defines a an array, called `@names` . It puts five values into the array.

```
@names=("Muriel","Gavin","Susanne","Sarah","Anna");

print "The elements of \@names are @names\n";
print "The first element is $names[0] \n";
print "The third element is $names[2] \n";
```

```
print 'There are ',scalar(@names)," elements in the array\n";
```

Firstly, notice how we define `@names` . **As it is in a list context, we are using parens.** Each value is **comma seperated, which is Perl's default list delimiter.** The double quotes are not necessary, but as these are string values it makes it easier to read and change later on.

Next, notice how we print it. Simply refer to it as a whole, that is in **list context.**. List context means referring to more than one element of a list at a time. The code `print @names ;` will work perfectly well too. If we want to do anything with the array as a list, that is doing something with all the values at once, refer to the array as `@names` or whatver you call your arrays. That's important. The `@` prefix is also used when you want to refer to more than one element, but not the entire array. That's called a **slice** . Cake analogies are appropiate, and somewhat tastier.

Arrays are not much use unless we can get to indvidual elements. Firstly, we are dealing with a single element of the list, so we cannot use `@` which refers to multiple elements of the array. **It is a single, scalar variable, so `$` is used.** Secondly, we must specify which element we want. That's easy - `[0]` for the first, `[1]` for the second and so forth. Array indexes start at 0, unless you do something which is so highly deprecated (that means allowed, usually for backwards compatbility, but disapproved of because there are better ways) I'm not even going to mention it.

Finally, we force what is normally list context (more than one element) into scalar context (single element) to give us the amount of elements in the array. Without the `scalar` , it would be the same as the second line of the program.

More on Arrays

The element number can be a variable.

```
print "Enter a number :";
chop ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah","Anna");

print "You requested element $x who is $names[$x]\n";

print "The index number of the last element is $#names \n";
```

This is useful. Notice the last line of the example. It returns the index number of the last element. Of course you could always just do this `$last=scalar(@names)-1;` but this is more efficient. It is an easy way to get the last element, as follows :

```
print "Enter a number :";
chop ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Trish","Simon");

print "The first two elements are @names[0,1]\n";
print "The first three elements are @names[0..2]\n";
print "You requested element $x who is $names[$x]\n";
print "The elements before and after are : @names[$x-1,$x+1]\n";
print "The first, second, third and fifth elements are @names[0..2,4]\n";

print "The last element is $names[$#names]\n";
```

It looks complex, but it is not. Notice you can have multiple values seperated by a comma. As many as you like, in whatever order. The range operator `..` gives you everything between and including the values. And finally look at how we print the last element - remember `$#names` gives us a number ? Simply enclose it inside square brackets and you have the last element.

Do also note that because element accesses such as [0,1] are more than one variable, we cannot use the scalar prefix,

namely the $ symbol. **We are accessing the array in list context, so we use the @ symbol.** Doesn't matter that it is not the entire array. Remember, accessing more than one element of an array but not the entire array is called a slice.

Foreach

All well and good, but what if we want to load each element of the array in turn ? Well, we could build a nasty looking for loop like this :

```
@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Trish","Simon");

for ($x=0; $x <= $#names; $x++) {
        print "$names[$x]\n";
}
```

which sets $x to 0, runs the loop once, then adds one to $x, checks it is less than $#names, if so carries on.

There is a prettier looking version :

```
for $x (0 .. $#names) {
        print "$names[$x]\n";
}
```

which takes advantage of the range operator `..` again but for true beauty we must use `foreach` .

```
foreach $person (@names) {
        print "$person";
}
```

This goes through each element ('iteriates', another good word to use) of `@names`, and assigns each element in turn to `$person`. Then you can do what you like with the variable. Much easier. You can use `for $person (@names) {` if you want. Makes no difference at all.

In fact, that gets shorter. And now I need to introduce you to `$_` , which is the **Default Input and Pattern Searching Variable.**

```
foreach (@names) {
        print "$_";
}
```

If you don't specify a variable to put each element into, `$_` is used instead as it is the default for this operation, and many, many others in Perl. Including the `print` function :

```
foreach (@names) {
        print;
}
```

As we haven't supplied anything to print, $_ is printed instead. You'll be seeing a lot of $_ in Perl.

Changing the Elements

So we have `@names`. We want to change it. Run this :

```
print "Enter a name :";
chop ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah");
```

```
print "@names\n";

push (@names, $x);

print "@names\n";
```

Fairly self explantaory. Push just pushes a value on to the end of the array. Of course, Perl being Perl, it doesn't have to be just the one value :

```
print "Enter a name :";
chop ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah");
@cities=("Brussels","Hamburg","London","Breda");

print "@names\n";

push (@names, $x, 10, @cities[2..4]);

print "@names\n";
```

So that's push. Now for some...

Jiggerypokery with Arrays

```
@names=("Muriel","Gavin","Susanne","Sarah");
@cities=("Brussels","Hamburg","London","Breda");

&look;

$last=pop(@names);
unshift (@cities, $last);

&look;

sub look {
        print "Names : @names\n";
        print "Cities: @cities\n";
}
```

Now we have two arrays. `pop` removes the last element of an array and returns it, which means you can do something like assign the returned value to a variable. `unshift` adds it to the beginning of the array. Hope you didn't forget that `&subroutinename` calls a subroutine.

push	Adds value to the end of the array
pop	Removes and returns value from end of array
shift	Removes and returns value from beginning of array
unshift	Adds value to the beginning of array

Now, accessing other elements of arrays. May I present the splice function ?

Splice

```
@names=("Muriel","Sarah","Susanne","Gavin");

&look;

@middle=splice (@names, 1, 2);
```

```
&look;

sub look {
        print "Names : @names\n";
        print "The Splice Girls are: @middle\n";
}
```

The arguments to `splice` are firstly an array, then the offset. This is the index number of the list element to begin splicing at. In this case it is 1. Then comes the number of elements to remove, which is sensibly 1 or more in this case. You can set it to 0 and perl, in true perl style, won't complain. Setting to 0 is handy because `splice` can add elements to the middle of an array, and if you don't want any deleted 0 is the number to use. Like so :

```
@names=("Muriel","Gavin","Susanne","Sarah");
@cities=("Brussels","Hamburg","London","Breda");

&look;

splice (@names, 1, 0, @cities[1..3]);

&look;

sub look {
        print "Names : @names\n";
        print "Cities: @cities\n";
}
```

Notice how the assignment to `@middle` has gone - it is no longer relevant. This is not to say nothing is returned from the function - you can test it for truth to see if it was successful - but it doesn't return a list of variables.

Regular Expressions

Or **regex** for short. These can be a little intimdating. But I'll be you have already used some regex in your computing life so far. Have you even said "I'll have any German beer ?" That's a regex which will match a Grolsch or Becks, but not a Budweiser, orange juice or cheese toastie. What about `dir *.txt` ? That's a regular expression too, listing any files ending in `.txt`

Perl's regex often look like this : `if ($name=~/piper/) { .` That is saying "If 'piper' is inside $name, then True.". Notice the way the expression you are searching for, in this case 'piper' is between slashes, and the `=~` operator to assign the target for the search.

An example is called for. Run this, and answer it with 'the faq'. Then try 'my tealeaves' and see what happens.

```
print "What do you read before joining any Perl discussion ? ";
chop ($_=<STDIN>);

if ($_=~/the faq/) {
        print "Right !  Join up !\n";
} else {
        print "Begone, vile creature !\n";
}
```

So here $_ is searched for 'the faq'. Guess what we don't need ! The `~=` . This works just as well:
`if (/the faq/) {`
because if you don't specify a variable, then perl searches $_ by default.

But what if someone enters 'The FAQ' ? It fails, because the search is case sensitive. We can easily fix that :

```
if (/the faq/i) {
```

with the i switch, which specifies case-insensivity. Now it works for all variations.

Study this example just to clarify the above. Tabs and spaces have been added for asthetic beauty :

```
$_="perl for Win32";                              # sets the string to be searched

if ($_=~/perl/) { print "Found perl\n" };         # is 'perl' inside $_ ?  $_ is "perl for Win32".
if (/perl/)     { print "Found perl\n" };         # same as the regex above.  Don't need the =~ as
if (/PeRl/)     { print "Found PeRl\n" };         # this will fail because of case sensivitiy
if (/er/)       { print "Found er\n" };           # this will work, because there is an 'er' in 'p
if (/n3/)       { print "Found n3\n" };           # this will work, because there is an 'n3' in 'W
if (/win32/)    { print "Found win32\n" };        # this will fail because of case sensivitiy
if (/win32/i)   { print "Found win32 (i)\n" };   # this will *work* because of case insensivitiy

print "Found!\n"  if      / /;                   # another way of doing it, this time looking for
print "Found!!\n" unless !/ /;                   # same thing, but reversing the logic with unles
                                                 # don't do this, it will always never not confus

$find=32;                                        # Create some variables to search for
$find2=" for ";

if (/$find/) { print "Found '$find'\n" };         # you can search for variables like numbers
if (/$find2/) { print "Found '$find2'\n" };      # and of course strings !
```

As you can see from the last example, you can embed a variable in the regex too. Regular expressions could fill entire books (and they have done, see the book critiques at http://www.perl.com/) but here are some useful tricks:

```
@names=qw(Karlson Carleon Karla Carla Karin Carina Needanotherword);

foreach (@names) {
        if (/[KC]arl/) {                # this line will be changed a few times in the examples
                print "Match !  $_\n";
        } else {
                print "Sorry.   $_\n";
        }
}
```

This time `@names` is initalised using whitespace as a delimiter instead of a comma. `qw` refers to 'quote words', which means split the list by words.

The square brackets `[ ]` enclose **single characters to be matched**. Here either Karl or Carl must be in each element. It doesn't have to be two characters, and you can use more than one set. Change Line 4 in the above program to :

```
if (/[KCZ]arl[sa]/) {
```

matches if something begins with K, C, or Z, then arl, then either s or a. It does *not* match KCZarl. Negation is possible too, so try this :

```
 if (/[KCZ]arl[^sa]/) {
```

which returns things beginning with K, C or Z, then arl, and then anything EXCEPT s or a. The caret ^ has to be the first character, otherwise it doesn't work as the negation. Having said [ ] defines single characters only, I should mention than these two are the same ;

```
/[abcdeZ]arl/;
/[a-eZ]arl/;
```

if you use a hypen then you get the list of characters icluding the start and finish characters. And if you want to match a special character, you must escape it :

```
/[\-K]arl/;
```

matches Karl or -arl. Although the - character is represented by two characters, it is just the one character to match.

If you want to match at the end of the line, make sure a `$` is the last character in the regex. This one pulls out all those names ending in a. Slot it into the example above :

```
if (/a$/) {
```

And there is a corresponding character, the caret ^, which in this context matches at the beginning of the string. Yes, the caret also negates a character class like this `[^KCZ]arl` but in this case it **anchors** the match to the beginning of the string.

```
if (/n/i) {
if (/^n/i) {
```

The first one is true if the word contains an 'n' anywhere in it. The second specifies that the 'n' must be at the beginning of the string to be matched.

If you want to negate the entire regex change `=~` to `!~` (as `!` negates, remember 'not equal to' is `!=` ?

```
if ($_ !~/[KC]arl/) {
```

Of course, as we are testing `$_` this works too : `if (!/[KC]arl/) { .`

Now things get interesting. What if we want pull something out of a string ? So far all we have done is test for truth, that is say yea or nay, but not return what we found. Run this :

```
$_='My email address is <Robert@NetCat.co.uk>.';

/(<robert\@netcat.co.uk>)/i;

print "Found it ! $1\n";
```

Firstly, note the single quotes when $_ is assigned. **If there were double quotes, we'd need \@ instead of @**. Remember, double quotes "" allow variable interporlation, so Perl looks for an array called @NetCat which does not exist.

Secondly, look at the parens around the entire regex. If you use brackets, a side effect is that the first match is put into a variable called $1. We'll get to the main effect later. The second match goes into $2 and so on, up to $99 (I think). Also note that the @ has been escaped, so perl doesn't think it is an array. Remember `\` either escapes a special character, or gives a special meaning. Think of it as Superman's telephone box. Imagine Clark Kent walking around with with his magic partner Back Slash.

Notice how we specify in the regex case-insensitivty, and the regex returns the case-sensitive string - that is, exactly what it found.

Try the regex without parens. Then try this one :

```
/<(robert)\@netcat.co.uk>/i;
```

You can put the parens anywhere. More or less. Now, run this :

```
$_='My email address is <Robert@NetCat.co.uk>.';

/<(robert)(\@netcat.co.uk)>/i;
```

```
print "Found it ! $1 and $2\n";
```

See, you can have more than one ! Look at the above regex. Looks easy now, eh ? What about five minutes ago ? It would have looked like a mistake ! Well, there are some hairer regex to come, but you'll have a good barber.

What if we didn't know what the email address was going to be ?

```
$_='My email address is <webslave@work.com>.';

/(<.*>)/i;

print "Found it ! $1\n";
```

We'll discuss this. Firstly, we have the opening parens (. So everything from (to) will be put into $1 if the match is successful. Then the first character of what we are searching for, <. Then we have a dot, or period " . ". This matches any character at all, except \n, the newline (it can even match that if required using /s, but don't worry about it now).

So we are now matching < followed by any non \n character. The * means 0 or more of the previous character, up to the next character, >.

This is important. Get the basics right and all regex are easy (I read somewhere once). An example best illustrates the point. Slot this regex in instead :

```
/(<*>)/i;
```

Now, why does this not work ? Think about it. The * matches 0 or more of the preceding character. What's the preceding character here ? <. So it is looking for 0 or more < characters, up until >. It finds < but immediately after < is a 'w', not a ^;lt;. This doesn't match, because were are looking for 0 or more < chars until > . However, there are 0 < chars immediately before > so it matches there. Try this ;

```
$_='My email address is <<<<<>.';
```

and you will notice the match works. So why does `<.*>` work ? Because the regex first has to match an < followed by anything except \n. Then it matches unlimited anythings (that's the *) until it finds > .

Glad you followed that. Now, pay even closer attention ! Concentrate fully on the task at hand ! This should be straightforward now :

```
$_='HTML <I>munging</I> time !.';

/<I>(.*)<\/I>/i;

print "Found it ! $1\n";
```

Pretty much the same as the above, except the parens are moved so we return what's only inside the tags, not including the tags themselves. Also not how `/` is escaped like so : `\/` otherwise Perl thinks that's the end of the regex.

Now, suppose we change $_ to :

```
$_='HTML <I>munging</I> time is here <I>again</I> !.';
```

and run it again. Interesting effect, eh ? This is known as Greedy Matching. What happens is that when Perl finds the inital match, ie <I> it jumps right to the end of the string and works back from there to find a match, so the longest string matches. This is fine unless you want the shortest string. And there is a solution :

```
/<I>(.*?)<\/I>/i;
```

Just add a question mark and Perl does stingy matching. No nationalistic jokes. I have Dutch and Scottish friends I don't want to offend.

Suppose we didn't know what HTML tag we had to match ? It could be B, I, EM or whatever, and we want everything that is in between. Well, HTML container tags like B and EM have end tags which are the same as the start tag, except for the / . So what we could do is :

- find out what is inside < >
- search for exactly the same tag except for the closing /
- return whatever is in between.

Can this be done ? Does the pope wear a silly hat ? Of course. This is perl, all things are possible. Now, remember the side effect of parens. I promise I'll explain the primary effect at some point. If whatever is in (parens) matches, the result is stored in a variable called $1. So we can match `<.*?)>` which will find us < then as many anythings (the period and *) up to the next, not last > (the ? forces stingy matching).

The result is stored in $1 because we used parens. Next, we need everything up to the closing tag. That's easy : `(.*?)` matches everything up until the next character or set of characters. And how exactly do we define where to stop ?

Remember, we stored the first match in $1 because of the parens. You can use this even in the same regex, and in a regex it is not $1, but \1 . So we want to match </$1> which in perl code is `<\/\1>` . The / must be escaped because it is the end of the regex, and 1 is escaped so it refers to $1 instead of matching the number 1.

Still here ? This is what it looks like :

```
$_='HTML <I>munging</I> time is here <I>again</I> !.';
/<(.*?)>(.*?)<\/\1>/i;

print "Found it ! $2\n";
```

If you want to know how to return all the matches above, read on. But before that - How to Avoid Making Mountains while Escaping Special Characters.

You want to match this : `http://language.perl.com/faq/` . That's a real (useful) URL by the way. To match it, you need to do this :

```
/http:\/\/language\.perl\.com\/faq\//;
```

which should make the awful metaphor above clearer, if not funnier. Fortunately, Perl allows you to pick your delimter if you prefix it with 'm' as this example shows :

```
m#http://language\.perl\.com/faq/#;
```

Which is a huge improvement, as we change `/` for `#` . We can go further with readability by quoting everything :

```
m#\Qhttp://language.perl.com/faq/\E#;
```

The `\Q` escapes everything up until `\E` or the regex delimiter (so we don't really need the \E above). In this case # will not be escaped, as it delimits the regex. Someone once posted a question about this to the Perl Win32 mailing list and I was so intruiged about this apparently undocumented trick I spent the next twenty minutes figuring it out by trial and error, and posted a reply. Next day I found lots of messages telling the poster to read the manual because it was clearly documented. <face colour='red' intensity='high'> My excuse was I didn't have the docs to hand....moral of the story - RTFM and RTF FAQs !

Subsitution

Suppose you want to replace bits of a string. For example, the awful 'alot' with 'many'.

```
$_='I like to use the word alot, like alot of my friends and alot of their relatives';

print "$_\n";

s/alot/many/;   # operates on $_, otherwise you need $foo=~s/alot/many/;

print "$_\n";
```

What happens here is that the pattern 'alot' is searched for, and when a match is found it is replaced with the right side of the expression, in this case many. Simple. You can also add /i for case insensivity.

You'll notice that only one substition was made. To match globally use /g which runs through the entire string, changing wherever it can. Try :

```
s/alot/many/g;
```

and notice everything is changed. Everything you have learn about regex can be used with `s` , like parens, character classes [ ], greedy and stingy matching and much more. Deleting things is easy too. Just specify nothing as as the replacement character, like so `s/alot//g;` .

Too easy. What about if we change $_ to :

```
$_='I like to use the word alot, like alot of my friends and alot of their relatives, who are rea
```

Notice how 'zealots' becomes mangled. How would you fix this ? What about `s/ alot[, ]/many/g;` which will match a space, 'alot', followed by either a comma or another space. That works. For this situation. But you'd have to add quite a bit of puncuation in to match every case. Perl has an easier way, as usual. You probably want to match `XalotX` , where X is not a-z, A-Z, 0-9 or the underscore '_' . So `[^a-zA-Z0-9_]` will work. Remember the caret negates, and '-' defines a range, eg all lowercase letters are a-z.

This easier way is simply \w, which a Word. A word in this case is defined as a-z, A-Z, 0-9 and _. **To negate this, and any other similar construct, captalise it : \W** . So we can use :

```
 s/\Walot\W/many/g;
```

Except that doesn't quite work. No spaces. What we need is to match **in between the two word boundaries**, ie between \w and \W. The \b construct (word boundary) is perfect for this.

```
 s/\balot\b/many/g;
```

There are several more constructs. We'll take a quick look at \d which means anything that is a digit, that is 0-9. First we'll use the negated form, \D, which is anything *except* 0-9 :

```
print "Enter a number :";
chop ($input=<STDIN>);

if ($input=~/\D/) {
        print "Not a number !!!!\n";
} else {
        print 'Your answer is ',$input x 3,"\n";

}
```

this checks that there are no non-number characters in $x. It's not perfect because it'll choke on decimal points, but it's just an example. I hope you trusted me and typed the above in exactly as it is show (or pasted it), becaus the `x` is not a

mistake, it is a feature. If you were too smart and s/x/*/ or something change it back and see what it does.

Of course, there is another way to do it :

```
unless ($input=~/\d/) {
        print 'Your answer is ',$input x 3,"\n";
} else {
        print "Not a number !!!!\n";
}
```

which reverses the logic with an `unless` statement.

More Matching

Assume we have :

```
$_='HTML <I>munging</I> time is here <I>again</I> !.';
```

and we want to find all the italic words. We know that /g will match globally, so surely this will work :

```
$_='HTML <I>munging</I> time is here <I>again</I> ! What <EM>fun</EM> !';

$match=/<i>.*?<\/i>/ig;

print "$match\n";
```

except it returns 1, and there were defintely two matches. The match operator returns true or false, not the number of matches. So you can test it for truth with functions like `if, while, unless` . Incidentally, the `s` operator does return the number of substutions.

To return what is matched, you need to supply a list.

```
($match) =~ /<i>.*?<\/i>/i;
```

which handily puts all the first matche into `$match`. The parens force a list context in this case. There is just the one element in the list, but it is still a list. The entire match will be assigned to the list, or whatever is in the parens. Try adding some parens :

```
$_='HTML <I>munging</I> time is here <I>again</I> ! What <EM>fun</EM> !';

($word1, $word2) = /<i>(.*?)<\/i>/ig;

print "Word 1 is $word1 and Word 2 is $word2\n";
```

In the example above notice /g has been added so a global search is done - this means perl carries on matching even after it finds the first match. Of course, you might not know how many matches there will be, so you can just use an array (or other type of list) :

```
$_='HTML <I>munging</I> time is here <I>again</I> ! What <EM>fun</EM> !';

@words = /<i>(.*?)<\/i>/ig;

foreach (@words) {
        print "Found $word\n";
}
```

and `@words` will be grown to the appropiate size for the matches.

There is more another trick worth knowing. Because a regex returns true each time it matches, we can test that and do something every time it returns true. The ideal operator is `while` which means 'do something while the condition I'm testing is true'. In this case, we'll print out the match every time it is true.

```
$_='HTML <I>munging</I> time is here <I>again</I> ! What <EM>fun</EM> !';

while (/<(.*?)>(.*?)<\/\1>/g) {
        print "Found the HTML tag $1 which has $2 inside\n";
}
```

So the while operator runs the regex, and if it is true, carries out the statements inside the block.

Try running the program above without the /g. Notice how it loops forever ? That's because the expression always evaluates to true. By using the /g we force the match to move on until it eventually fails.

Now we know this, an easy way to find the number of matches is :

```
$_='HTML <I>munging</I> time is here <I>again</I> ! What <EM>fun</EM> !';

$found++ while /<i>.*?<\/i>/ig;

print "Found $found times\n";
```

You don't need braces in this case as nothing apart from the expression to be evaluated follows the `while` function.

Parentheses Again

The real use for them. Precendence. Try this :

```
$_='One word sentences ? Eliminate. Avoid cliches like the plague.  They are old hat.';

while (/o(rd|ne|ld)/gi) {
        print "Matched $1\n";
}
```

Firstly, notice the subtle introduction of the or operator, in this case '|', the pipe. What I really want to explain however, is that this regex matches o followed by rd, ne or ld. Without the parens it would be `/ord|ne|ld/` which is defintely not what we want. That matches just plain ord, or ne or ld.

Finally, take a look at this :

```
$_='I am sleepy....zzzz....DING ! Wake Up!';

if (/(z{5})/) {
        print "Matched $1\n";
} else {
        print "Match failed\n";
}
```

The braces { } specify how many of the preceding character to match. So `/z{2}/` matches exactly two 'z's and so on. Change {5} to {4} and it works. And there's more...

/z{3}/	3 z only
/z{3,}/	At least 3 z
/z{1,3}/	1 to 3 z

/z{4,8}/	4 to 8 z
/z{4,8}?/	4 to 8 z , returns shortest match possible

And now on the Regex Programme Today, we have guest stars Prematch, Postmatch and Match. All of whom are going to slow our entire programme down, but are useful anyway :

```
$_='I am sleepy....snore....DING ! Wake Up!';

/snore/;

print "Postmatch: $'\n";
print "Prematch: $`\n";
print "Match: $&\n";
```

If you are wondering what the difference between match and using parens is you should remember than you can move the parens around, but you can't vary what `$&` returns. Also, using any of the above three operators does slow your entire program, whereas using parens will just slow the particular regex you use them for. However, once you've used one of the three matches you might as well use them all over the place as you've paid the speed penalty.

RHS Expressions

RHS means Right Hand Side. Suppose we have an HTML file, which contains :

```
$data="<FONT SIZE=2> <FONT SIZE=4> <FONT SIZE=6>";
```

and we wish to double the size of each font so 2 becomes 4 and 4 becomes 8 etc. What about :

```
$data="<FONT SIZE=2> <FONT SIZE=4> <FONT SIZE=6>";

print "$data\n";

$data=~s/(size=)(\d)/\1\2 * 2/ig;

print "$data\n";
```

which doesn't really work out. What this does is match `size=x`, where x is any digit. The first match, `size=`, goes into $1 and the second match, whatever the digit is, goes into $2. The second part of the regex simply prints $1 and $2 (referred to as \1 and \2), and attempts to multiply $2 by 2. Remember `/i` means case insensitve matching.

What we need to do is evaluate the right hand side of the regex as an expression - that is not just print out what it says, but actually evaluate it. Perl can do this :

```
$data=~s/(size=)(\d)/$1.($2 * 2)/eig;
```

A little explanation....the LHS is the same as before. We add `/e` so Perl evaluates the RHS as an expression. So we need to change \1 into $1 and so on. The parens are there to ensure that `$2 * 2` is evaluated, then joined to $1. And that's it ! Here's another example, which is a little more cunning :

```
$red="96000";
$white="FFFFF";
$yellow="FFFF33";

$data='<FONT COLOR=yellow> <FONT COLOR=white> <FONT COLOR=red>';

print "$data\n";

$data=~s/(color=)(\w+)/$1.${$2}/eig;
```

```
print "$data\n";
```

This one is interesting because it refers to a variable of the same name as the replaced string. The `{ }` is needed around the $2 to force Perl to realise that you mean a scalar variable called $2. The `\w+` section matches a word (same as `[a-zA-Z_0-9]`) and the `+` means one or more of the preceding character. In this case, one or more word characters. Of course, this regex does not consider quoted parameters to HTML tags but I have to leave something as an exercise for the reader...

It is even possible to have more than one `/e` . For example :

```
$data='important perl names are $names';
$names="Camel, Llama, ActiveState, Perl";

print "$data\n";

$data=~s/(\$[a-zA-Z]+)/$1/ee;

print "$data\n";
```

This is very useful. Notice that `\w` is not used. This is because `\w` will match `[a-zA-Z_0-9]` and Perl variables may not start with a number. This is because `$1, $2` etc are of course reserved for use by regex. You could write a more complicated regex to more precisely match variables, but that's a start.

Split and Join

While you are in the regex mood, a quick look at split and join. Destruction is always easier (just ask your car mechanic), so lets start with split.

```
$_='Piper:PA-28:Archer:OO-ROB:Antwerp';

@details=split /:/, $_;

foreach (@details) {
        print "$_\n";
}
```

Split is given two arguments here. The first one is a regex specifying what to split on. The next is what to split. Actually, I could leave the `$_` out because as usual it is the default if nothing is specifed.

The assignment can either be a scalar variable or a list like an array (or hash, but at this time 'hash' to you means what you think the Dutch do or a silly drinking event spoilt by some running). If it's a scalar variable you get the number of elements the split has splut. Should that be 'the split has splittered' or 'the split has splat'. Hmmm. Probably 'the split has split'. You know what I mean. I think I just generated a Fatal Error in English.dll. Whoops.

If the assigment is a list of some sort, then as you can see in the above example the array is created with the relevant elements in order. Note 'list' above, not array. For example :

```
$_='Piper:PA-28:Archer:OO-ROB:Antwerp';

($maker,$model,$name,$reg,$location)=split /:/, $_;

$number=split /:/ ;             # not bothering with the $_ at the end, as it is the default

print "$reg is a $maker $model $name based in $location\n";
print "There are $number details available on this aircraft\n";
```

This demonstrates that a list can be a list of scalar variables (which is bascially what an array is anyway), and that you

can easily see how many elements the expression can be split into.

The example below adds a third paramter to split, which is how many elements you want returned. If you don't want the extra stuff at the end `pop` it.

```
$_='Piper:PA-28:Archer:OO-ROB:Antwerp';

@details=split /:/, $_, 3;

foreach (@details) {
        print "$_\n";
}
```

In the example below we split on whitespace '\s' , which is a space, tab, newline, formfeed or carraige return. The whitespace split is specially optimised for speed. I've used spaces, double spaces, a tab and a newline in the list below. Also note the '+' which means one or more of the preceding character, so it will split on any combination of whitespace. And I think the final split is useful to know.

```
$_='Piper       PA-28  Archer           OO-ROB
Antwerp';

@details=split /\s+/, $_;

foreach (@details) {
        print "$_\n";
}

@chars=split //, $details[0];

foreach $char (@chars) {
        print "$char !\n";
}
```

So that's the fun stuff, destruction. Now to put it back together again with `join`.

Putting it back together

```
$w1="Mission critical ?";
$w2="Internet ready modems !";
$w3="J";
$w4="y2k compatible.";
$w5="We know the Web.";
$w6="...the leading product in an emerging market.";

$cool=join ' ', $w1,$w2,$w3,$w4,$w5,$w6;

print $cool;
```

Join takes a 'glue' operator, which is *not* a regular expression. It can be a scalar variable however. In this case it is a space. Then it takes a list, which can either be a list of scalar variables, an array or whatever as long as its a list. And you can see what the result is. You could assign it to an array, but you'd end up with everything in the first element of the array.

The example below adds an array into the list, and demonstrates use of a variable as the delimiter.

```
$w1="Mission critical ?";
$w2="Internet ready modems !";
$w3="J";
$w4="y2k approved, tested and safe !";
$w5="We know the Web.";
$w6="...the leading product in an emerging market.";
```

```
@morecool=("networkable","compatible");

$sep=" ";

$cool=join $sep, $w1,$w2,$w3,@morecool,$w4,$w5,$w6;

print $cool;
```

Aren't you wishing you could mix and match randomly so you too could get a job marketing vapourware ? Heh.

```
@cool=("networkable","compatible","Mission critical ?","Internet ready modems !",
"J","y2k approved, tested and safe !",
"We know the Web.","...the leading product in an emerging market.");
srand;

print "How many phrases would you like ?";
while (1) {
        chop ($input=<STDIN>);
        if ($input < $#cool && $input > 0) {
                last;
        }
        print 'Wrong.  Try again !';
}

for (1..$input) {
        $index=int(rand $#cool);
        print "$cool[$index] ";
        splice @cool, $index, 1;
}
```

A few things to explain. Firstly, `while (1) {` . We want an everlasting loop, and this one way to do it. 1 is always true, so round it goes. We could test `$input` directly, but that wouldn't allow `last` to be demonstrated.

Everlasting loops aren't useful unless you are a politician. We need to break out at some point. This is done by the `last` function. When `$input` is between 1 and the number of elements in @cool then out we go. (You can also break out to labels, in case you were wondering. Don't start now if you weren't.)

The `srand` operator initalises the random number generator. Works ok for us, but CGI programmers should think of something different because their programs are so frequently run (they hope :-).

`rand` generates a random number between 0 and 1, or 0 and a number it is given. In this case, the number of elements of @cool. `int` makes sure it is an integer, that is no messy bits after the decimal point.

The `splice` function removes the printed element from the array so it won't appear again. Don't want to stress the point.

Another Join Type Operator

There is another joining operator, this time the humble dot, or period : `.` . This **concatanates** (joins) variables :

```
$x="Hello";
$y=" World";
$z="\n";

print "$x\n";           # print $x and a newline

$prt=$x.$y.$z;          # make $y out of $x $y and $z

print $prt;

$x.=$y." again ".$z;    # add stuff to $x
```

```
print $x;
```

Files

Perl is very good at handling files. Create, in your perl scripts directory `c:\scripts`, a file called `stuff.txt`. Copy the following into it :

```
The Main Perl Newsgroup:comp.lang.perl.misc
The Perl FAQ:http://www.perl.com/faq/
```

Now, to open and do things with this file.

```
$stuff="c:\scripts\stuff.txt";

open STUFF, $stuff;

while (<STUFF>) {
        print "Line $. is : $_";
}
```

What this script does is fail. What is *should* do is open the file defined in $stuff, assign it to the filehandle STUFF and then, while there are still lines left in the file, print the line number `$.` and the current line.

It fails. That's not so bad, everything fails sometimes. What is unforgiveable is NOT CHECKING THE ERROR CODE !

This is a better line:

```
open STUFF, $stuff or die "Cannot open $stuff for read :$!";
```

If the open operation fails, the `or` operation is executed. Perl dies. This means it exits the script with a helpful error code. The error code is in `$!` , which is printed on request. The rest of the line is for clarity.

Always check your return codes !

The problem should now be apparent. The backslashes, being escape characters, are not displayed. There are two ways to fix this :

- Escape the backslashes, like so `$stuff="c:\\scripts\\stuff.txt";`
- Convert backslashes into forward slashes : `$stuff="c:/scripts/stuff.txt";`

The forward slashes are the preferred option, even under Win32, because you can then port the script direct to Unix or other platforms (assuming you don't use drive letters), and it is less typing. If you wish to use Perl to start external processes then you must use the \\ method, but this variable will be used only in a Perl program, not as a parameter to start an external program.

Changing the `$stuff` variable results in a working script. **Always check your return codes !**

```
$stuff="c:/scripts/stuff.txt";

open STUFF, $stuff or die "Cannot open $stuff for read :$!";

while (<STUFF>) {
        print "Line $. is : $_";
}
```

A little more detail on what is happening here. The file is opened for read. You can append and write too. You don't *have* to use a variable, but I always do because it is then easy to change and easy to insert into the `or die` section. `open STUFF, "c:/scripts/stuff.txt" or die "Cannot open stuff.txt for read :$!";` is just as good but more work.

The line input operator (that's the angle brackets <>) reads from the beginning of the file up until and including the first newline. The read data goes into $_, and you can do what you want with it there. On the next iteration of the loop data is read from where the last read left off, up to the next newline. And so on until there is no more data. When that happens the condition is false and the loop terminates. That's the default behaviour, but we can change this.

This means that you can open a 200Mb file in perl and run through it without having to load the entire file into memory. 200Mb of memory is quite a bit. If you really want to load the entire 200Mb file into one variable, Perl lets you. Limits are not the Perl Way.

`$.` is the current line number, starting at 1.

As usual, there is a quicker way to do the previous program.

```
$STUFF="c:/scripts/stuff.txt";

open STUFF or die "Cannot open $STUFF for read :$!";

while (<STUFF>) {
        print "Line $. is : $_";
}
```

and as that saves a little bit of typing I tend to use it. Reduces the possibility for eror too.

Writing to a File

```
$out="c:/scripts/out.txt";

open OUT, ">$out" or die "Cannot open $stuff for write :$!";

print OUT 'The time is now : ',scalar(localtime);
```

Note the addition of " > " to the filename. This opens it for writing. If we want to print to the filehandle, we now just specify the filehandle name. Filehandles don't have to be captalised, but it is wise. All Perl functions are lowercase, and Perl is case-sensitive. **So if you choose uppercase names they are guaranteed not to conflict with current or future function words.**

And a neat way to grab the date sneaked in there too. More on dates later.

```
$out="c:/scripts/out.txt";

&printfile;

open OUT, ">>$out" or die "Cannot open $out for append :$!";

print OUT 'The time is now : ',scalar(localtime),"\n";

close OUT;

&printfile;

sub printfile {
        open IN, $out or die "Cannot open $out for read :$!";
        while (<IN>) {
                print;
        }
```

```
        close IN;
}
```

This script demonstrates subrountines again, and how to append to a file, that is write additional data at the end. The `close` function is introduced here. This, well, closes a filehandle. You don't have to close a filehandle - just leave it open until the script finishes, or the next open command to the same filehandle will close it for you.

Perl has a special array called @ARGV. This is the list of arguments passed along with the script name on the command line. Run the following perl script as :
`perl myscript.pl hello world how are you`

```
foreach (@ARGV) {
        print "$_\n";
}
```

Another useful way to get parameters into a program - this time without user input. The relevance to filehandles is as follows. Run the following perl script as :
`perl myscript.pl stuff.txt out.txt`

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

Short and sweet ? If you don't specify anything in the angle brackets, whatever in ARGV is used instead. And after it finishes with the first file, it will carry on with the next and so on. You'll need to remove non-file elements from @ARGV before you use this.

One of the most frequent Perl tasks is to open a file, make some changes and write it back to the original filename. You already have enough knowledge to do this. The steps are :

1. Make a backup copy of the file
2. Open the file for read
3. Open a new temporary file for write
4. Go through the read file, and write it and any changes to the temp file
5. When finished, close both files
6. Delete the original file
7. Rename the temp file to the original filename

Phew. Perl of course has a much easier way. Make sure you have data in `c:\scripts\out.txt` then run this :

```
@ARGV="c:/scripts/out.txt";

$^I=".bk";              # let the magic begin

while (<>) {
        tr/A-Z/a-z/;    # another new function sneaked in
        print;          # this goes to the temp filehandle, ARGVOUT, not STDOUT as usual, so don
}
```

Now take a look at `out.txt`. Notice how all capital letters have been translierated into lowercase. This is the `tr` operator at work, which is more efficient than regex for changing single characters. You should also have an `out.txt.bk` file. And finally, notice the way @ARGV has been created. You don't have to create it from the command line arguments - it is an array just like any other.

Finally, what if your input file is doesn't look like this :

```
Beer
```

```
Wine
Pizza
Catfood
```

which is nicely delimited with a newline each time, but like this :

```
shorts
t-shirt
blouse

pizza
beer
wine
catfood

Viz
Private Eye
The Independent
Byte

toothpaste
soap
towel
```

which is delimited by TWO newlines, not one. Now, if you want each set of items as elements in an array you'll have to do something like this :

```
$SHOP="shop.txt";
$x=0;

open SHOP or die "Can't open $SHOP for read: $!\n";

while (<SHOP>) {
        if (/^\n/) {            # does line begin with newline ?
                $x++;           # if so, increment $x.  Rest of if statement not executed.
        } else {
                $list[$x].=$_;  # glue $_ on the end of whatever is in $list[$x]
        }
}

foreach (@list) {
        print "Items are:\n$_\n\n";
}
```

which works, but there is a much easier way to do it. You knew I was going to say that.

```
$SHOP="shop.txt";
$/="\n\n";

open SHOP or die "Can't open $SHOP for read: $!\n";

while (<SHOP>) {
        push (@list, $_);
}

foreach (@list) {
        print "Items are:\n$_\n\n";
}
```

The `$/` variable is a special variable (it even looks special). It is the **Default Input Record Seperator.** Remember the operation of the angle brackets being to read a file in up until the next newline ? Time to come clean. What the angle

bracket actually do is read up until whatever $/ is set to. It is set to a newline by default. So if we set it to two newlines, as above, then it reads up until it finds two consecutive newlines, then puts the data into $_. This makes the program a lot shorter and quicker. You can set $/ to just about anything, not just a newline. If you want to hack this list for example:

```
Tea:Beer:Wine:Pizza:Catfood:Coffee:Chicken:Salmon:Icecream
```

you could just set leave $/ as a newline and slurp it into memory in one go, but imagine the above item is a list of clothes than your girlfriend wants to buy or a list of clothes your boyfriend should have thrown away by now. Either are going to be really big files, and you might not want to read it all into memory in one go. So set `$/=":";` and all will be well. There are also `read` and `seek` functions, but they aren't covered here.

We'll go back to the last example for a moment. It is useful to know how to read just one line (well, up to $/) at a time :

```
$SHOP="shop.txt";
$/="\n\n";

open SHOP or die "Can't open $SHOP for read: $!\n";

$clothes=<SHOP>;        # everything up until the first occurence of $/ into $clothes

$food=<SHOP>;   # everything from first occurence of $/ to the second into into $food

print "We need...\n",$clothes,"...and\n",$food;
```

And now we know that, there is a even quicker way to achieve the aim of the original program :

```
$SHOP="shop.txt";
$/="\n\n";

open SHOP or die "Can't open $SHOP for read: $!\n";

@list=<SHOP>;   # dumps *all* of $SHOP into @list, not just one line.

foreach (@list) {
        print "Items are:\n$_\n\n";
}
```

and you don't need to grab it *all* : `@list[0..2]=<SHOP>` .

We haven't mentioned list context for a while. Wheter a the line input operator `<>` returns a single value or a list depends on the context you use it in. When you supply `@xxxx` then this must be a list. If you supply `$xxxx` then that's a scalar variable. You can force it into list context by using parens. There is no way to force list context because you never need to.

Edit the example above and run these two programs :

```
($first, $second)=<SHOP>;
$first, $second=<SHOP>;
```

Associative Arrays

Very, very useful. First, a quick recap on arrays. Arrays are an ordered list of scalar variables, which you access by their index number starting at 0. **Arrays always stay in the same order**.

Hashes are a list of scalars, but instead of being accessed by index number, **they are accessed by a key**. The tables below illustrate the point :

@myarray	
Index No.	*Value*
0	The Netherlands
1	Belgium
2	Germany
3	Monaco
4	Spain

%myhash	
Key	*Value*
NL	The Netherlands
BE	Belgium
DE	Germany
MC	Monaco
ES	Spain

So if we want 'Belgium' from `@myarray` and `%myhash`, it'll be :

```
print "$myarray[1]";
print "$myhash{nl}";
```

Notice that the `$` prefix is used, because it is a scalar variable. Despite the fact it is part of a list, it is still a scalar variable. The hash syntax is simply to use braces { } instead of square brackets.

So why use hashes ? When you want to look something up by a keyword. Suppose we wanted to create a program which returns the name of the country when given a country code. We'd input ES, and the program would come back with Spain.

You could do it with arrays. It would be messy however. One possible approach :

1. create @country, and give it values such as 'ES,Spain'
2. Iteriate over the entire array
3. Split the value, and check the first result to see if it matches the input
4. If so, return the index

```
@countries=('NL,The Netherlands','BE,Belgium','DE,Germany','MC,Monaco','ES,Spain');

print "Enter the country car code:";
chop ($find=<STDIN>);

foreach (@countries) {
        ($code,$name)=split /,/;
        if ($find=~/$code/i) {
                print "$name has the code $code\n";
        }
}
```

Complex and slow. We could also store a reference to another array in each element of @countries, but I'm not going to cover Lists of Lists here and that approach is not much more efficient. We could search the entire array with grep, but I haven't mentioned that yet and anyway you still need to search the whole thing. And what if @countries is a big array ? See how much easier a hash is :

```
%countries=('NL','The Netherlands','BE','Belgium','DE','Germany','MC','Monaco','ES','Spain');

print "Enter the country car code:";
chop ($find=<STDIN>);

$find=~tr/a-z/A-Z/;
print "$countries{$find} has the code $find\n";
```

Very easy. All we need to do is make sure everything is in uppercase with `tr` and we are there. Notice the way % countries is defined - exactly the same as a normal array, except that the values are put into the hash in key/value pairs.

So why use arrays ? One excellent reason is because when an array is created, its variables stay in the same order you created them in. With a hash, perl reorders elements for quick access. Add `print %countries;` to the end of that program above and run it. See what I mean ? No recognisable order at all. If you were writing code that stored a list of variables over time and you wanted it back in the order you found it in, don't use a hash.

Finally, you should know that each **key of a hash must be unique.** Stands to reason, if you think about it. You are accessing the hash via keys, so how can you have two keys named 'NL' or something ? If you do define a certain key twice, the second value overwrites the first. This is a feature, and useful. The values of a hash can be duplicates, but never the keys.

If you want to assign to a hash, there is of course no concept of push, pop and splice etc. Instead :

Assigning	`$countries{PT}='Portugal';`
Deleting	`delete $countries{NL};`

Accessing Your Hash

Assuming you keep the same %countries hash as above, here are some useful ways to access it :

All the keys	`print keys %countries;`
All the values	`print values %countries;`
How many elements ?	`print scalar(keys %countries);`
Does the key exist ?	`if exists$countries{NL} print "It's there !\n";`

Well, that last one is not an access but useful anyway. And it demonstrates that arguments to `if` don't have to be in parens if it is a simple test. I'm sure there's a rule, but I don't know it.

You may have noticed that `keys` and `values` return a list. And we can iteriate over a list, using `foreach` :

```
foreach (keys %countries) {
        print "The key $_ contains $countries{$_}\n";
}
```

which is useful. Note how any list can be fed to `foreach` , and off it goes. As usual, there is another way to do the above:

```
while (($code,$name)=each %countries) {
        print "The key $code contains $name\n";
}
```

The `each` function returns each key/value pair of the hash, and is slightly faster. In this example we assign them to a list (you spotted the parens ?) and away we go. Eventually there are no more pairs, which returns false to the while loop and it stops.

Sorting

If I was reading this I'd be wondering about sorting. Wonder no more, and behold :

```
foreach (sort keys %countries) {
        print "The key $_ contains $countries{$_}\n";
}
```

Spot the difference. Yes, `sort` crept in there. If you want the list sorted backwards, some cunning is called for. This is suitably foxy :

```
foreach (reverse sort keys %countries) {
        print "The key $_ contains $countries{$_}\n";
}
```

Perl is just so difficult at times, don't you think ? That's easy alphabetical sorting by the keys. If you had a hash of international access numbers like this one :

```
%countries=('976','Mongolia','52','Mexico','212','Morocco','64','New Zealand','33','France');

foreach (sort keys %countries) {
        print "The key $_ contains $countries{$_}\n";
}
```

You might want to sort numerically and then you need to understand how Perl's sort function works.

The sort function compares two variables, $a and $b. They must be called $a and $b otherwise it won't work. One chap published a book with stolen code, and he changed $a and $b to $x and $y. He obviously didn't test the program because it would have failed and he would have noticed. And this book was really published ! Don't believe everything you read in books - but web tutorials are always 100% truthful :-)

Back to sorting. $a and $b are compared, and the result is :

- 1 if $a is greater
- -1 if $b is greater
- 0 if $a and $b are equal

So as long as the sort operator gets one of those three values back it is happy. This means we can write our own sort routines, and feed them to sort. For example, we know the default sort is alphabetical. But if we write this :

```
%countries=('976','Mongolia','52','Mexico','212','Morocco','64','New Zealand','33','France');

foreach (sort supersort keys %countries) {
        print "$_ $countries{$_}\n";
}

sub supersort {
        if ($a > $b) {
                return 1;
        } elsif ($a < $b) {
                return -1;
        } else {
                return 0;
        }
}
```

then it works correctly. Of course, there is an easier way. Do you remember `<=>` ? It does exactly what the supersort subrountine does. So we can write the above much more easily as :

```
%countries=('976','Mongolia','52','Mexico','212','Morocco','64','New Zealand','33','France');

foreach (sort { $b <=> $a } keys %countries) {
        print "$_ $countries{$_}\n";
}
```

Notice the { } braces, which define the contents as the subroutine sort must use. Pretty short subroutine. There is a companion operator to `<=>` , namely `cmp` which does exactly the same thing but of course compares the values as strings, not numbers. Remember if you are comparing numbers, your comparison operator should contain non-alphas,

if you are comparing strings the operator should contains alphas only.

Anyway, you now have enough knowledge to sort a hash by value instead of keys. Suppose your pointy haired manager bounced up to you and demanded a hash sorted by value ? What would you do ? OK, what *should* you do ?

Well, we could just sort the values.

```
foreach (sort values %countries) {
```

But Pointy Hair wants the keys too. And if you have a value you can't find the key.

So we have to iteriate over the keys. But just because we are iterating over the keys doesn't mean to say we have to hand the keys over to the sort operator. What about :

```
%countries=('976','Mongolia','52','Mexico','212','Morocco','64','New Zealand','33','France');

foreach (sort { $countries{$a} cmp $countries{$b} } keys %countries) {
        print "$_ $countries{$_}\n";
}
```

beautifully simple. If you want a reverse sort swap $a and $b.

You can sort several lists at the same time :

```
%countries=('976','Mongolia','52','Mexico','212','Morocco','64','New Zealand','33','France');
@nations=qw(China Hungary Japan Canada Fiji);

@sorted= sort values %countries, @nations;

foreach (@nations, values %countries) {
        print "$_\n";
}


print "#----\n";

foreach (@sorted) {
        print "$_\n";
}
```

This sorts @nations and the values from %countries into a new array. The example also demonstrates that you can `foreach` over more than one list value - each list is processed in turn.

Grep and Map

Grep

If you want to search a list, and create another list of things you found, `grep` is one solution. This is an example, which also demonstrates `join` again :

```
@stuff=qw(flying gliding skiing dancing parties racing);

@new = grep /ing/, @stuff;

print join ":",@stuff,"\n";

print join ":",@new,"\n";
```

Remember `qw` means 'quote words', so word boundaries are used as delmiters intead. The `grep` function must be fed a list on the right hand side. On the left side, you may assign the results to a list or a scalar variable. The list gives you each actual element, and the scalar gives you the number of matches found :

```
@stuff=qw(flying gliding skiing dancing parties racing);

$new = grep /ing/, @stuff;

print join ":",@stuff,"\n";

print "Found $new elements of \@stuff which matched\n";
```

If you decide to modify the elements on they way through grep, you actually modify the original list.

```
@stuff=qw(flying gliding skiing dancing parties racing);

@new = grep s/ing//, @stuff;

print join ":",@stuff,"\n";
print join ":",@new,"\n";
```

To determine what actually matches you can either use an expression or a block. Up to now we've been using expressions, but when things become more complicated use a block :

```
@stuff=qw(flying gliding skiing dancing parties racing);

@new = grep { s/ing// if /^[gsp]/ } @stuff;

print join ":",@stuff,"\n";
print join ":",@new,"\n";
```

Try removing the braces and you'll get an error. Notice that the comma before the list has gone. It is now obvious where the expression ends, as it is inside a block delimited with { } . The regex says if the element begins with g, s or p, then remove ing. The result is only assigned to @new if the expression is completely true - 'parties' does begin with p, so that works, but s/ing// fails so the overall result in false, and the value is not assigned to @new.

Map

Map works the same way as grep, in that they both iteriate over a list, and return a list. There are two important differences however :

- grep returns the **value** of everything it **evaluates to be true**
- map returns the **results** of **everything** it evaluates

As usual, an example will assist the penny in dropping, clear the fog and turn on the light (if not make my metaphors easier to understand) :

```
@stuff=qw(flying gliding skiing dancing parties racing);

print join ":",@stuff,"\n";

@mapped  = map  /ing/, @stuff;
@grepped = grep /ing/, @stuff;

print join ":",@stuff,"\n";
print join ":",@mapped,"\n";
print join ":",@grepped,"\n";
```

You can see that @mapped is just a list of 1 or nothing. This is the *result* of map - in every case the expression /ing/ is

successful, except for 'parties'. Notice there is a null value returned in this case - false. Grep returns the actual value, and only if it is true. Try this :

```
@letters=(a,b,c,d,e);

@ords=map ord, @letters;
print join ":",@ords,"\n";

@chrs=map chr, @ords;
print join ":",@chrs,"\n";
```

This uses the ord function to change each letter into its ASCII equiavlent, then the chr function convert ASCII numbers to characters. If you change map to grep in the example above, you can see that nothing appears to happen. What is happening is that grep is trying the expression on each element, and if it succeeds (is true) it returns the element, not the result. The expression succeeds for each element, so each element is returned in turn. Another example :

```
@mapped  = map  { s/(^[gsp])/$1 x 2/e } @stuff;
@grepped = grep { s/(^[gsp])/$1 x 2/e } @stuff;
```

Recapping on regex, what that does is match any element beginning with g, s or p, and replace it with the same element twice. The caret ^ forces a match at the beginning of the string, the [square brackets] denote a character class, and /e forces Perl to evaluate the RHS as an expression.

The output from this is a mixture of 1 and nothing for map, and a three-element array called @grepped from grep. Yet another example :

```
@mapped  = map  { chop } @stuff;
@grepped = grep { chop } @stuff;
```

The `chop` function removes the last character from a string, and returns it. So that's what you get back from map, the result of the expression. The `grep` function gives you the mangled remains of the original value.

Finally, you can write your own functions :

```
@stuff=qw(flying gliding skiing dancing parties racing);

print join ":",@stuff,"\n";

@mapped  = map  { &isit } @stuff;
@grepped = grep { &isit } @stuff;

print join ":",@mapped,"\n";
print join ":",@grepped,"\n";

sub isit {
        ($word)=/(^.*)ing/;

        if (length $word == 3) {
                return "ok";
        } else {
                return 0;
        }
}
```

The subroutine `isit` first grabs everything up until 'ing', puts it into $word, then returns 'ok' if the there are three characters in $word. If not, it returns the false value 0. You can make these subroutines (think of them as functions) as complex as you like.

External Commands

Perl can start external commands. There are three main ways to do this :

- `system`
- `exec`
- Command Input, also known as `` `backticks` ``

We'll compare `system` and `exec` first.

Exec

Exec is broken on Perl for Win32. What it should do is stop running your Perl script and start running whatever you tell it to. If it can't start the external process, it should return with an error code. This doesn't work properly under Perl for Win32. Exec does work properly on the standard Perl distribution.

System

This runs an external command for you, then carries on with the script. It always returns, and the value it returns goes into `$?` . This means you can test to see if the program worked. Actually you are testing to see if it could be started, what the program does when it runs is outside your control (well, almost, see later on).

This demonstrates system in action. Run the 'vol' command from a command prompt first if you are not familiar with it. Then run the 'vole' command. I'm assuming you have no cute furry executables called vole on your system, or at least in the path. If you do have voles, be creative and change it.

```
system("vole");

print "\n\nResult: $?\n\n";

system("vol");

print "\n\nResult: $?\n\n";
```

As you can see, a successful system call returns 0. An unsucessful one returns a value which you need to divide by 256 to get the real return value. Also notice you can see the output. And because `system` returns, the code after the first system call is executed. Not so with `exec`, which will terminate your perl script if it is successful.

Backticks

These (``) are different again to system and exec. They also start external processes, but return the output of them. You can then do whatever you like with the output. If you aren't sure where backticks are on your keyboard, try the top left, just left of the 1 key. Often around there. Don't confuse single quotes " with backticks ``.

```
$volume=`vol`;

print "The contents of the variable \$volume are:\n\n";

print $volume;

print "\nWe shall regexise this variable thus :\n\n";

$volume=~m?Volume in drive \w is (.*)?;

print "$1\n";
```

As you can see here, the Win32 vol command is executed. We just print it out, escaping the $ in the variable name. Then a simple regex, using ? as a delimiter just in case you'd forgotten delimiters don't have to be / .

Before you get carried away with creating elaborate scripts based on the output from NT's `net` commands, note there are plenty of excellent modules out there which do a very good job of this sort of thing, and that any form of external process call slows your script. Also note there are plenty of built in functions such as readdir which can be used instead of `dir`. **You should use Perl functions where possible rather than calling external programs** because Perl's functions are :

- portable (usually, but there are exceptions). This means you can write a script on your Mac PowerBook, test it on an NT box and then use it live on your Unix box without modifying a single line of code.
- faster, as every external process significantly slows your program
- don't usually require regexing to find the result you want
- don't rely on output in a particular format, which might be changed in the next version of your OS or application
- are more likely to be understood by a Perl programmer - for example, `$files=`ls`;` on a Unix box means little to someone that doesn't know that ls is the Unix command for listing directories, as dir is in Windows.

Don't start using backticks all over the place when `system` will do. You might get a very large return value which you don't need, and will slurp lots of memory. Just use them when you want to check the return value.

Subroutines and Parameters

NT's Event Log

ChangeNotify

Thanks to...

Everyone that helped in the development of this tutorial. Specifically :

- **Katya de Vries** <k.devries@eurocities.be> for finding HTML errors and problems with the example code.
- **Steven Ham** for being picky about spelling errors. Good going, considering English is his second language !

--Robert Pepper mailto:Robert@netcat.co.uk
