Topicalization and Special Variables

Not all Perl variables are created as you run your code. Some useful information is provided in special variables before your code is even reached, and some of these special variables can be manipulated to affect the ongoing operation of your program. Perl even provides a facility that lets you leave parameters out of your source code, and it will assume a default current variable. This is known as topicalization.

Regular variables in Perl 1394
Special variables 1394
Topicalization 1395
Special information variables and the English module 1398
Special variables that provide controls 1401
Special variables provided by Perl operations 1404
Other special variables you may come across 1404
Command line options 1404

107.1 Regular variables in Perl

In Perl, most variables are created dynamically as you run your program, and if you refer to a variable which is part of a more complex structure, that more complex structure too will be created as necessary.

If you write:

```
$demo[16] = $abc123;
```

and the list **@demo** didn't previously exist, that list would be created with 17 elements (numbered 0 to 16). Element number 16 would be created large enough to hold a copy of whatever's already contained in the variable **$abc123**. There's no need to declare variables (although you might sometimes do so).

The rule for naming variables as described above are:

- They start with: **$** for a scalar (to hold a number, a string, etc.)
  @ for a list (to hold a series of scalars indexed by number)
  % for a hash(to hold a series of scalars by key)
  & for a code variable (a method or subroutine)
  [no start character] for a file handle
  ***** for a typeglob (a package deal - one of each)
- Followed by a letter
- Followed by as many (or as few) letters, digits and underscores as you wish

Variable names are case sensitive, and you can use the same name for one variable of each type if you wish and there won't be a conflict. For example, you could refer to a variable **$host** and a list **@host** if you wished. Anywhere you refer to **$host** it's a totally different variable to anything that's contained in **@host**; for example, the scalar **$host[0]**.

Important to note: Variable names are case sensitive.

In order to avoid any chance of your variable names conflicting with the special variables that we're about to talk about, we strongly suggest that you use a lower case letter directly after the **$** character. Others suggest a standard where you capitalise the first letter of each intermediate word, for example **$numberOfPeople**. In any case, please avoid the variable names **$a** and **$b** which have special significance in sorting.

Important to note: In naming, you are strongly advised to use a lowercase letter directly after the $character, and avoid using variable names "$a" and "$b".

107.2 Special variables

As well as regular variables, Perl uses a number of variables which are special in some way – the main subject of this training module. Some of these variables have names that follow the normal rules, but others have very curious names like **$"** or **$^O**.

There are a number of groups of special variables:

- Some are provided to you by Perl when you run a Perl program; they give you useful information about the environment in which your running.
- Others control how Perl performs certain operations. They're initially set to a default value, but you can change what the variables contain if you want different behaviour.
- Still others are set automatically if you perform certain operations. Operations return a result, and (sometimes) other information is put into special variables.

Let's see a first example of each:

```
#!/usr/bin/perl

# pre-defined special variables $0 and $^O

print "Example of special variables\n";
print "This program is $0 and the operating system is $^O\n";
```

```
# Using special variable $" to effect the "....." operator

@salad = qw(lettuce tomato dressing);
print "I am making a salad with @salad\n";
$" = " ..... ";
print "I am making a salad with @salad\n";

# Seeing how $& gets set by regular expression matching

$details = "The Perl programming language is great";
if ($details =~ /Perl.*language/) {
        print "I got: $&\n";
        }
```

When I ran that on my Apple laptop, I got the following result:

```
[localhost:~/p210] graham% ./special1
Example of special variables
This program is ./special1 and the operating system is darwin
I am making a salad with lettuce tomato dressing
I am making a salad with lettuce ..... tomato ..... dressing
I got: Perl programming language
[localhost:~/p210] graham%
```

and on one of our trainee workstations running Windows XP, I got:

```
C:\Documents and Settings\graham> perl special1
Example of special variables
This program is special1 and the operating system is MSWin32
I am making a salad with lettuce tomato dressing
I am making a salad with lettuce ..... tomato ..... dressing
I got: Perl programming language

C:\Documents and Settings\graham>
```

107.3 Topicalization

Let me describe my journey to our town centre. *"I go out of the house and get in the car. I turn it on the driveway and drive it out through the gate. At the end of our road, I turn right and then go straight ahead past the hospital..."*

What has this got to do with Perl? Perl is a language just like English, and certain of the grammars and use of language from English have been adopted by Perl. After all, English has been developed over many centuries but the first version of Perl wasn't around until 1988.

Look back at my description. I established "the car" as my subject in the first sentence. I was then able to simply use the word "it" to refer to the car later on, and I was even able to leave it out completly if I so chose. Wouldn't it have appeared to be stilted if I had written: *"I go out of the house and get in the car. I turn* **the car** *on the driveway and drive* **the car** *out through the gate. At the end of our road, I turn* **the car** *right and then go* **in the car** *straight ahead past the hospital.."*?

And yet, this is exaclty the sort of thing that you'll often find written in a program.

Let's write a program that reads in a file, searches for all lines that match a pattern, and report on those lines:

```
#!/usr/bin/perl

open (FH,"hosts") or die ("No file called hosts\n");

while ($record = <FH>) {
        chop $record;
        $lineno++;
        if ($record =~ /www/) {
                print $record;
                $estring = ("th","st","nd","rd",("th") x 6)[$lineno%10];
                $lineno/10%10 == 1 and $estring = "th";
                print " # $lineno$estring line\n";
                }
        }
```

When we run that program, we get:

```
[localhost:~/p210] graham% perl topic1
192.168.200.130 lecht www.wellho.co.uk # 158th line
192.168.200.212 badger www.javatrainer.local # 241st line
192.168.200.213 mouse www.tcltraining.local # 242nd line
192.168.200.214 stoat www.wellhouseconsultants.local # 243rd line
192.168.200.215 squirrel www.spatraining.local # 244th line
192.168.200.216 vole www.perltraining.local  # 245th line
192.168.200.217 bat www.grahamellis.local # 246th line
192.168.200.218 rat www.avairpros.local # 247th line
[localhost:~/p210] graham%
```

It's a fine application, but did you notice how we had to keep repeating the name of the **$record** variable throughout the code?

- In Perl, if you read using the **<>** in a **while** condition and don't tell Perl where to place the line that it's read, it will place it automatically into a special variable known as the "default input and pattern matching space". If you need to refer to this variable explicitly, you can do so – it's **$_** ("Dollar Underscore").
- In Perl, many functions can have their last or final parameter left off, and Perl will use the contents of **$_**. Functions of this type include **chop** and **print**.
- If you specify a regular expression, but don't use the **=~** operator to tell Perl which string to match against, it will work on the contents of the **$_** variable.
- If you use a file status operator (for example **-e**) and don't specify a parameter, Perl will return the status of a file named in **$_**.

And this isn't a complete list either. We can take the previous program and remove all references to **$record**.

Perl will simply use **$_** and the code will work in exactly the same way:

```
#!/usr/bin/perl

open (FH,"hosts") or die ("No file called hosts\n");

while (<FH>) {
        chop;
        $lineno++;
        if (/www/) {
                print ;
                $estring = ("th","st","nd","rd",("th") x 6)[$lineno%10];
                $lineno/10%10 == 1 and $estring = "th";
                print " # $lineno$estring line\n";
                }
        }
```

Results:

```
[localhost:~/p210] graham% perl topic2
192.168.200.130 lecht www.wellho.co.uk # 158th line
192.168.200.212 badger www.javatrainer.local # 241st line
192.168.200.213 mouse www.tcltraining.local # 242nd line
192.168.200.214 stoat www.wellhouseconsultants.local # 243rd line
192.168.200.215 squirrel www.spatraining.local # 244th line
192.168.200.216 vole www.perltraining.local  # 245th line
192.168.200.217 bat www.grahamellis.local # 246th line
192.168.200.218 rat www.avairpros.local # 247th line
[localhost:~/p210] graham%
```

Important to note: Using $_ within short areas of the code is good practice, but don't set it early in a long program and rely on it being there later.

Newcomers to Perl often find that whilst using **$_** may reduce the length of their programs, initially it makes the programs much harder to follow. Using **$_ within short areas** of the code **is** good practice, but we suggest that you don't set it early in a long program and assume that it's still there much later.

Other uses of **$_**:

- If you don't specify a variable name before the "**(**" in a **foreach** loop, Perl puts each list element in turn into **$_**
- **grep** and **map** put each element of their input lists in turn into **$_ prior** to matching
- Functions that use **$_** unless told otherwise via a paramater include

abs	**alarm**	**chop**	**chr**
chroot	**cos**	**defined**	**eval**
exp	**glob**	**int**	**lc**
lcfirst	**length**	**log**	**lstat**
oct	**ord**	**pos**	**print**
printf	**quotemeta**	**readlink**	**require**
rmdir	**sin**	**split**	**sqrt**
stat	**study**	**uc**	**ucfirst**
unlink			

- The following operators also use **$_** if not instructed otherwise:

  tr///;
  s///;

- File operators return information about the file whose name is currently in $_ unless given a parameter. Thus

  if (-e) { .....}

Here's an example that uses some of those:

```
#!/usr/bin/perl

@info = (1..5,20,50,75,100);

print "@info\n";

foreach (@info) {
        print length," ";
        }
print "\n";

@odd = grep($_%2,@info);
print "@odd\n";

@teams = map(int($_/11),@info);
print "@teams\n";
```

Running, that gives:

```
[localhost:~/p210] graham% perl top3
1 2 3 4 5 20 50 75 100
1 1 1 1 1 2 2 2 3
1 3 5 75
0 0 0 0 0 1 4 6 9
[localhost:~/p210] graham%
```

Note, not all functions use **$_** if you leave off the last paramter.

rand	defaults to	**1**
srand	defaults to	**current time**
exit	defaults to	**0**
pop	defaults to work with	**@_**

As from version 6 of Perl, **$_** becomes a much more important feature of the language, so you'll be well advised to know it in Perl 5. The name "topicalization" is officially introduced at Perl 6, but we've included it here in this section as it's already here; it's just that the way it works will be greatly expanded.

107.4 Special information variables and the English module

We've divided the more useful special variables, a little artificially, into a number of groups. In this section, we'll tell you about the most useful of those which are provided by Perl when you run them, but that don't actually affect anything the program does unless they are used.

$0 tells you the name of the program that you're running.

Are you going to remember all these short special names, and which is which? Probably not. Perl is provided with a standard module called **English** that lets you define additional names for the special variables. **$PROGRAM_NAME** is the extra name for **$0**; you'll notice it's all capitals as are all such extra names. This is why we suggest you use lower case names for your own variables.

$0 or **$PROGRAM_NAME** is the name of your program.

Let's put these variables into a program:

```
#!/usr/bin/perl

use English;
```

```
print "$PROGRAM_NAME reports on special variables\n\n";

print "About the Perl process:\n";
print ("Process id:     $$  $PID  $PROCESS_ID\n");
print ("Real User id:   $<  $UID  $REAL_USER_ID\n");
print ("Real Group id:  $(  $GID  $REAL_GROUP_ID\n");

print ("\nAbout the Perl Environment\n");
# The Perl Version in $^V is a 3 character string ...
printf ("Perl Version:   %vd %vd\n",$^V,$PERL_VERSION);
print ("Old style:      $]\n");
print ("Operating Sys:  $^O  $OSNAME\n");
print ("Executable Name $^X  $EXECUTABLE_NAME\n");
print ("Script Started: $^T  $BASETIME\n");
```

Here's how the program ran on my Apple laptop:

```
[localhost:~/p210] graham% ./infovars
./infovars reports on special variables

About the Perl process:
Process id:     5429  5429  5429
Real User id:   501  501  501
Real Group id:  20 80 0 20  20 80 0 20  20 80 0 20

About the Perl Environment
Perl Version:   5.6.0 5.6.0
Old style:      5.006
Operating Sys:  darwin  darwin
Executable Name perl  perl
Script Started: 1053885796  1053885796
[localhost:~/p210] graham%
```

and on my WindowsXP box:

```
C:\Documents and Settings\graham>perl infovars
infovars reports on special variables

About the Perl process:
Process id:     2276  2276  2276
Real User id:   0  0  0
Real Group id:  0  0  0

About the Perl Environment
Perl Version:   5.8.0 5.8.0
Old style:      5.008
Operating Sys:  MSWin32  MSWin32
Executable Name C:\Perl\bin\perl.exe  C:\Perl\bin\perl.exe
Script Started: 1053886052  1053886052

C:\Documents and Settings\graham>
```

Reading the command line in Perl

Parameters that are supplied after the program name (i.e. on the command line) are provided in a special list called **@ARGV**. You can find out how many command line parameters there were using **$#ARGV**, or **@ARGV** in a list context:

```
#!/usr/bin/perl

print "Command line reporter\n";
print "Called with ",$#ARGV+1," parameters which ",
        @ARGV == 1 ? "was" : "were" ,"\n";

print (++$n,": $_\n") foreach (@ARGV) ;
```

Results:

```
[localhost:~/p210] graham% perl commline
Command line reporter
Called with 0 parameters which were
[localhost:~/p210] graham% perl commline first second
Command line reporter
Called with 2 parameters which were
1: first
2: second
[localhost:~/p210] graham% ./commline first second
Command line reporter
Called with 2 parameters which were
1: first
2: second
[localhost:~/p210] graham% ./commline -h
Command line reporter
Called with 1 parameters which was
1: -h
[localhost:~/p210] graham% ./commline -h n*
Command line reporter
Called with 3 parameters which were
1: -h
2: nm
3: nn
[localhost:~/p210] graham%
```

Notes:

- The program name is not included in the list.[1]
- Options (as the **-h** of our example) are really just regular parameters in the list
- You can't tell from **@ARGV** whether the program was run in Perl because of a **#!** line or because Perl was specified on the command line.
- Wild carded file names are expanded by the shell on Unix- and Linux-based operating systems, as in our last example. This is not the case on Windows systems where the wild card is provided as part of the parameter, and if you wish you can use Perl'g **glob** function to expand it.

You could write your own command line handler subroutine to deal with options, but there's little point as the standard Perl is provided with three in the distribution.

[1] a difference to languages like C, where the program name is provided as being the first command line item

Here's an example of one of them: **getopts** in the **Getopt::Std** module. **@ARGV** is modified (filtered) by **getopts** so that after the call, it contains a list of what's left after option handling.

```
#!/usr/bin/perl

use Getopt::Std;

print "Raw parameters: @ARGV\n";
$status = getopts('vqo:');
($status == 0 or @ARGV < 1) and
die ("Usage: $0 [-v] [-q] [-o filename] file [file ...]\n");

print "Called correctly\n";
print "-v option ",$opt_v ? "selected\n" : "not given\n";
print "-q option ",$opt_q ? "selected\n" : "not given\n";
print "-o option ",$opt_o ? "given with $opt_o\n" : "not given\n";
print "To work on file(s) @ARGV\n";
```

The string parameter to **getopts** lists the valid letter options, and a colon is added after any option that also takes a parameter. Option values are returned in variables of the form **$opt_x** where "x" is the option letter – boolean if the option doesn't take a parameter, or the parameter value if it does.

Here are some examples of running that:

```
[localhost:~/p210] graham% perl gop -v -o abc123 def456 ghi789
Raw parameters: -v -o abc123 def456 ghi789
Called correctly
-v option selected
-q option not given
-o option given with abc123
To work on file(s) def456 ghi789
[localhost:~/p210] graham% perl gop -x -o abc123 def456 ghi789
Raw parameters: -x -o abc123 def456 ghi789
Unknown option: x
Usage: gop [-v] [-q] [-o filename] file [file ...]
[localhost:~/p210] graham%
```

107.5 Special variables that provide controls

Perl provides a number of special variables that affect the operation of specific functions, or a group of functions. They are initialised to a sensible default value for most applications, but can (with care) be changed. Why "with care"? Because you have to change them back to default if you don't want to upset code that follows.

Output control variables

$"** or **$LIST_SEPARATOR lets you change what's added between each element of a list referenced in double quotes, and defaults to a space character.

You may prefer to use the **join** function in new code. **$,** (or **$OFS** or **$OUTPUT_FIELD_SEPARATOR**) controls what's output between each list item in a **print** function (default is nothing), and **$** (or **$ORS** or **$OUTPUT_RECORD_SEPARATOR**) controls what's output at the end of a **print** function.

Once again, the default is nothing.

```
#!/usr/bin/perl

@starter = qw(soup pate);
@main = qw(salad);
@follow = ("ice cream","gateau");  # can't use qw - space after ice

$"=" or ";
$,=" followed by ";
$\="<br>\n";

print ("Choose @starter","@main","@follow");
```

Let's see what's on the menu today:

```
[localhost:~/p210] graham% perl spout
Choose soup or pate followed by salad followed by ice cream or gateau<br>
[localhost:~/p210] graham%
```

Perl also has a complete output formatting system[1] that uses a large number of special variables.

Output buffering

When you print in your Perl program, output doesn't go straight to the screen or the output file, instead it's buffered. The buffers are only sent out when there's enough there for it to be worthwhile. The same thing happens in real life...you don't call someone up to collect every empty softdrink can for recycling one by one, do you? No, it's much more efficient to buffer up garbage and have it collected weekly.

Perl is excellent for knowing when it's worthwhile (or necessary) to flush buffered output; for example, it does so before reading user input which ensures that you can actually read the prompt to tell you what information to enter.

There are two circumstances where buffering can cause a problem. Both are specialist, but we're telling you about them here because if you come across them and don't realise what's happening, you can waste hours trying to find out.

q) If you're writing a program that runs for a long time with no apparrent activity, you may decide to print out a report from time to time to re-assure your user.

r) If you're writing a network program and talking to a remote service, you need to flush your buffer so that the remote service receives your full data before it can respond.

The **$|** (**$OUTPUT_AUTOFLUSH**) variable can be set to a true value[2] if you want your default output channel (**STDOUT** unless you change it) to flush after every **print** or **printf** statement.

[1] look up the write function and the format command if you're printing in a fixed width font on pre-printed stationary

[2] use the number 1

Here's an example:

```
#!/usr/bin/perl

$| = @ARGV;
$top = 20;

foreach (1..$top) {
        sleep 1;
        $now = localtime;
        printf "\r%20s %3d %3d%%",$now,$_,100*$_/$top;
        }

print "\n";
```

which always runs for 20 seconds. If it's run without any command line parameters, it appears to hang until the very end since **$|** is false. But if you give a command line parameter, **$|** becomes true and you'll see the clock ticking over, the counter increasing, and the percentage going up towards 100.

Notes:

- Use of **\r** and **printf** to overwrite a line on the output[1]
- Provision of clear feedback to the user; not only do we say how much of the task is done in this piece of code, but we also give information so that our user has some clue as to how much longer there is to go.

```
[localhost:~/p210] graham% perl ticker demo
Mon May 26 08:42:10 2003  20 100%
[localhost:~/p210] graham%
```

Input controls

When you read into a scalar using the **<>** operator, input terminates by default at the end of the line. If you read into a list, all remaining input on the given file handle is read, with the data being split into a series of scalars line by line. An excellent default behaviour, but if you're reading in data which is not inherently line-based (say XML or HTML for example), then you want something different.

$/** (or **$RS or **$INPUT_RECORD_SEPARATOR**) defaults to the new line character or sequence for your particular operating system.

You can change it to any other character or sequence to alter the delimiter if you wish. There are also two special cases:

$/="" Paragraph mode – read up to blank line

undef $/ No delimiter – read all available data to EOF

Here's an example to read a whole file into a single scalar:

```
#!/usr/bin/perl

$#ARGV and die "Usage: $0 regexp\n";
open (FH,"hosts") or die "No hosts file\n";

undef $/;
$info = <FH>;

print (($info =~ /$ARGV[0]/) ? "yes\n":"no\n");
```

[1] printf ensures that every line of output is the same length and you don't get a shorter line partially overwriting a longer line

A lot of the hosts in our data file are named after eatables, and not so many after things in the sky, so:

```
[localhost:~/p210] graham% ./allhosts
Usage: ./allhosts regexp
[localhost:~/p210] graham% ./allhosts toast
yes
[localhost:~/p210] graham% ./allhosts helicopter
no
[localhost:~/p210] graham%
```

107.6 Special variables provided by Perl operations

Variables in this section give you information about the most recent Perl operation of certain types. They're pure output variables, and if you change them you won't affect the further operations as you would have done with variables in the previous section.

Error information

In Perl 5 (to change in Perl 6), there's a variety of variables that contain information about errors encountered in performing code.

$?	**$CHILD_ERROR**	Status from the latest pipe, **``**, etc.
$@	**$EVAL_ERROR**	Status from the latest **eval** call
$!	**$ERRNO / $OS_ERROR**	Most recent error from a system call

Regular expression match information

When you do a regular expression match, there's a whole series of extra variables that are provided for you, telling about how the match worked. Details of regular expression matching and these variables is outside the scope of this module, but we've provided you with a list of some of these variables here for completeness:

$'	**$POSTMATCH**	Part of incoming string after the match
$`	**$PREMATCH**	Part of incoming string before the match
$&	**$MATCH**	Part of incoming string that matched
$1, **$2** etc.		Part of string that match groups in regex
$+	**$LAST_PAREN_MATCH**	The last brackets than matched in regex

107.7 Other special variables you may come across

These special variables are integral to Perl. They don't have short names, and they're defined and available to you even if you don't **use English;**. Apart from **@ARGV**, these variables are not described in detail in this module; we cover them in other modules dedicated to the subects to which they relate.

@ARGV	Command line parameters
@INC	List of directories from which Perl loads modules
@ISA	List of classes from which the current class is to inherit
%INC	Hash of modules loaded, and where they were loaded from
%ENV	Hash of environment variables
%SIG	Hash of code references, what Perl is to do with external signals

107.8 Command line options

We'll conclude this module on topicalization and Perl special variables with a look at some command line options and associated capabilities.

Let's run a program:

```
[localhost:~/p210] graham% perl -w commline -n abcd
print (...) interpreted as function at commline line 7.
Name "main::n" used only once: possible typo at commline line 7.
Command line reporter
Called with 2 parameters which were
1: -n
2: abcd
[localhost:~/p210] graham%
```

In this example, the **-w** is a command line option and the **-n** is simply a parameter to the Perl program. Options that go in the same place as the **-w** are controlled by the Perl language itself, but options that go where the **-n** is are under the control of the programmer.

If we're going to run our Perl program using a **#!** line, then we can specify the options within the Perl course on that line:

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

print "Command line reporter\n";
print "Called with ",$#ARGV+1," parameters which ",
       @ARGV == 1 ? "was" : "were" ,"\n";

print (++$n,": $_\n") foreach (@ARGV) ;
```

And if we run that, we get:

```
[localhost:~/p210] graham% ./cl2 -n vv
print (...) interpreted as function at ./cl2 line 7.
Name "main::n" used only once: possible typo at ./cl2 line 7.
Command line reporter
Called with 2 parameters which were
1: -n
2: vv
[localhost:~/p210] graham%
```

Command line options to Perl include:

-v report version number and important details of Perl
-V report version number and many details of Perl
-w provide compile time and runtime warning messages
-c compile only, do not run (i.e. check the syntax only)
-T run in tainted mode (i.e. additional checks to assist security)

If you're writing a Perl program that you want to process every line in an incoming file, you can specify either of the following options:

-n assume a **while** (**<>**) loop around your program
-p assume a **while** (**<>**) loop with a print statement at its end

Reading from the empty file handle **<>** reads from any file(s) named on the command line, or if there are no named files, it reads from **STDIN**.

Using these command line options makes your programs look rather like **sed** or **awk** scripts.

```
[localhost:~/p210] graham% ./sample
brown
Brown
macdougall
MacDougall
obrien
O'Brien
[localhost:~/p210] graham%
```

Code:

```
#!/usr/bin/perl -p
tr/A-Z/a-z/;
s/(^|-)(.)/uc("$1$2")/eg;
s/^(Ma?c)(.)/"$1".uc("$2")/e;
s/(^|-)O(.)/"$1"."O'".uc("$2")/ieg;
s/^Davin/daVin/;
```

If you're using the **-n** or **-p** options, you can also use **-a**, which is "autosplit" mode. This will split each incoming line in turn into a list called **@F**. Let's find all the hosts in a data file that include the letters "tea" in their main name, i.e. the second field of the incoming line:

```
[localhost:~/p210] graham% ./hodo hosts
192.168.200.9
192.168.200.132
192.168.200.193
192.168.200.229
[localhost:~/p210] graham%
```

The code is as simple as:

```
#!/usr/bin/perl -na
print $F[0],"\n" if ($F[1] =~ /tea/);
```

Here's some of the data so you can see how it worked:

```
192.168.200.7      cod
192.168.200.8      perch
192.168.200.9      stanstead
192.168.200.10     golfer-gw lambourn
192.168.200.11     herring

192.168.200.131    x350
192.168.200.132    steamboat
192.168.200.133    vail
```

etc.