Understanding Perl's Special Variables

By icarus

2003−07−10

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/Perl/PerlVariables/

Speaking Perl

You may find this hard to believe, but the Practical Extraction and Reporting Language, affectionately referred to as Perl by thousands of developers worldwide, was born primarily because its creator needed a tool to generate reports from raw text data. Over the years, though, Perl has grown from strength to strength; today, while it still boasts a very complete (and very impressive) text manipulation API, it can also be used to talk to databases, parse XML, execute SSL transactions and perform a wide variety of other (arcane?) functions.

Now, for newbie Perl programmers, one of the chief obstacles they have to overcome is the somewhat hard−to−understand shorthand used by experienced developers. This shorthand usually takes the form of built−in variables − my personal favourite is @_ though at one time I remember a particular fondness for $? − which have special meaning to the Perl interpreter and which, correctly used, can save you quite a few lines of code in your Perl program. Here, then, is a guide to the more interesting and commonly−used special variables in Perl, together with examples that demonstrate how they may be used.

In Default

* In Default

One of the more important special variables you'll encounter in your Perl forays is $_, which is used by many Perl functions and constructs as the so−called "default variable" when none other exists. In order to demonstrate, consider the following example:

```
#!/usr/bin/perl

# set array
%stuff = ("phone", "Nokia", "car", "BMW", "gun", "Smith & Wesson");

# iterate over array
foreach $s (keys(%stuff))
{
print "$s\n";
}
```

Here's the output:

```
car
gun
phone
```

In this case, the foreach() loop iterates over the hash, assigning the key at each iteration to the $s instance variable; this variable is then printed with the print() function. If, however, you omit the instance variable in the foreach() loop, Perl will assume that you want it to use the default $_ variable as the instance variable.

The following example, which is equivalent to the one above, demonstrates:

```
#!/usr/bin/perl

# set array
%stuff = ("phone", "Nokia", "car", "BMW", "gun", "Smith & Wesson");

# iterate over array
foreach (keys(%stuff))
{
print "$_\n";
}
```

Here, every time the loop iterates over the hash, since no instance variable is specified, Perl will assign the key to the default variable $_. This variable can then be printed via a call to print().

The $_ variable also serves as the default for both chop() and print() functions. Going back to the example above, you could also write it this way,

```
#!/usr/bin/perl

# set array
%stuff = ("phone", "Nokia", "car", "BMW", "gun", "Smith & Wesson");

# iterate over array
foreach (keys(%stuff))
{
print;
}
```

which would return

cargunphone

The $_ variable is also the default variable used by file and input handles. For example, consider the following simple Perl script, which prints back whatever input you give it:

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

```
# read from standard input
# print it back
while (<STDIN>)
{
print $_;
}
```

In this case, every line read by the standard input is assigned to the $_ variable, which is then printed back out with print().

Knowing what you now know about print() and $_, it's clear that you could also write the above as

```
#!/usr/bin/perl

# read from standard input
# print it back
while (<STDIN>)
{
print;
}
```

The $_ default variable is used in a number of different places: it is the default variable used for pattern−matching and substitution operations; the default variable used by functions like print(), chop(), grep(), ord(), etc; the default instance variable in foreach() loops; and the default used for various file tests.

Input...

Another very useful variable is the $/ variable, which Perl calls the input record separator. The $/ variable contains one or more characters that serve as line delimiters for Perl; Perl uses this variable to identify where to break lines in a file.

In order to illustrate, consider the following sample password file:

```
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

halt:x:7:0:halt:/sbin:/sbin/halt

mail:x:8:12:mail:/var/spool/mail:/sbin/nologin

news:x:9:13:news:/etc/news:

uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin

operator:x:11:0:operator:/root:/sbin/nologin

games:x:12:100:games:/usr/games:/sbin/nologin

gopher:x:13:30:gopher:/var/gopher:/sbin/nologin

ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

```
nobody:x:99:99:Nobody:/:/sbin/nologin
```

By default, the $/ variable is set to the newline character. Therefore, when Perl reads a file like the one above, it will use the newline character to decide where each line ends − as illustrated in the following example:

```
#!/usr/bin/perl

# read file into array
open (FILE, "dummy.txt");
@lines = <FILE>;

# iterate over file and print each line
foreach $line (@lines)
{
print "−−− " . $line;
}

# print number of lines in file
$count = @lines;
print "$count lines in file!\n";
```

Here's the output:

```
−−− shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
−−− halt:x:7:0:halt:/sbin:/sbin/halt
−−− mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
−−− news:x:9:13:news:/etc/news:
−−− uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
−−− operator:x:11:0:operator:/root:/sbin/nologin
−−− games:x:12:100:games:/usr/games:/sbin/nologin
−−− gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
−−− ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
−−− nobody:x:99:99:Nobody:/:/sbin/nologin
10 lines in file!
```

Now, if you alter the input record separator, Perl will use a different delimiter to identify where each line begins. Consider the following example, which sets a colon (:) to be used as the delimiter, and its output, which illustrates the difference:

```
#!/usr/bin/perl

# set input record separator
$/ = ":";
```

```
# read file into array
open (FILE, "dummy.txt");
@lines = <FILE>;

# iterate over file and print each line
foreach $line (@lines)
{
print "−−− " . $line;
}

# print number of lines in file
$count = @lines;
print "$count lines in file!\n";
```

Here's the new output:

```
−−− shutdown:−−− x:−−− 6:−−− 0:−−− shutdown:−−− /sbin:−−− /sbin/shutdown
halt:−−− x:−−− 7:−−− 0:−−− halt:−−− /sbin:−−− /sbin/halt
mail:−−− x:−−− 8:−−− 12:−−− mail:−−− /var/spool/mail:−−− /sbin/nologin
news:−−− x:−−− 9:−−− 13:−−− news:−−− /etc/news:−−−
uucp:−−− x:−−− 10:−−− 14:−−− uucp:−−− /var/spool/uucp:−−− /sbin/nologin
operator:−−− x:−−− 11:−−− 0:−−− operator:−−− /root:−−− /sbin/nologin
games:−−− x:−−− 12:−−− 100:−−− games:−−− /usr/games:−−− /sbin/nologin
gopher:−−− x:−−− 13:−−− 30:−−− gopher:−−− /var/gopher:−−− /sbin/nologin
ftp:−−− x:−−− 14:−−− 50:−−− FTP User:−−− /var/ftp:−−− /sbin/nologin
nobody:−−− x:−−− 99:−−− 99:−−− Nobody:−−− /:−−− /sbin/nologin
61 lines in file!
```

You can even undef() the record separator, which will result in Perl reading the entire file into a single string. Take a look:

```
#!/usr/bin/perl

# remove input record separator
undef($/);

# read file into array
open (FILE, "dummy.txt");
@lines = <FILE>;

# iterate over file and print each line
foreach $line (@lines)
{
print "−−− " . $line;
}

# print number of lines in file
$count = @lines;
print "$count lines in file!\n";
```

And here's the new output, which clearly displays that the file has been read as a single line:

−−− shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
1 lines in file!

...And Output

The reverse of the input record separator is the output record separator, quite logically found in the $\ variable. While the $/ variable deals with the delimiter used by Perl to break input into discrete records, the $\ variable controls which delimiter Perl uses to separate multiple print() invocations.

By default, the output record separator is null, which means that the output from every call to print() gets attached to the output from the previous call. Consider the following example, which demonstrates:

```
#!/usr/bin/perl

# print output
print "The";
print "cow";
print "jumped";
print "over";
print "the";
print "moon";
```

Here's the output:

Thecowjumpedoverthemoon

You can alter this by specifying a different value from the $\ variable − as the following example does:

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

```
# set output record separator
$\ = " −?− ";

# print output
print "The";
print "cow";
print "jumped";
print "over";
print "the";
print "moon";
```

Here's the result:

The −?− cow −?− jumped −?− over −?− the −?− moon −?−

Similar, though not identical, is the output field separator, which is used to specify the delimiter between the different values specified in a single print() command. This value is stored in the $, variable, and is usually null. Consider the following example, which demonstrates how it can be used:

```
#!/usr/bin/perl

# set output field separator
$, = ":";

# print output
print "The", "cow", "jumped", "over", "the", "moon";
```

Here's the output:

The:cow:jumped:over:the:moon

A common use of these two variables is to customize the output of the print() command − as in the following example:

```
#!/usr/bin/perl

# load module
use DBI();

# connect
my $dbh = DBI−>connect("DBI:mysql:database=test;host=localhost", "root",
"secret", {'RaiseError' => 1}) or die ("Cannot connect to database");

# query
my $sth = $dbh−>prepare("SELECT name, age, sex FROM users");
```

```
$sth−>execute();

# set separators
$, = ":";
$\ = "\r\n";

# print data as colon−separated fields
# each record separated by carriage return
while(my $ref = $sth−>fetchrow_hashref())
{
print $ref−>{'name'}, $ref−>{'age'}, $ref−>{'sex'};
}

# clean up
$dbh−>disconnect();
```

Here's the output:

john:34:M
jimmy:21:M
john:31:F
jane:27:F

Sure, this is a very complicated way of doing something really simple − but hey, it's an example. Don't take it too seriously!

Getting Into An Argument

Perl comes with some very interesting variables specifically designed to store input arguments, for both scripts and subroutines. The first of these is the special @ARGV array, which contains a list of all the command−line arguments passed to the Perl program; each argument is indexed as an element of the array. Consider the following example, which demonstrates:

```
#!/usr/bin/perl

# get length of argument list
$num = @ARGV;

# iterate and print arguments
for ($x=0; $x<$num; $x++)
{
print "Argument " . ($x+1) . " is $ARGV[$x]\n";
}
```

Here's an example of the output (I called this script with the command−line arguments "red 5px Arial"):

Argument 1 is red
Argument 2 is 5px
Argument 3 is Arial

Perl also comes with a variable named @_, which contains arguments passed to a subroutine, and which is available to the subroutine when it is invoked. The value of each element of the array can be accessed using standard scalar notation − $_[0] for the first element, $_[1] for the second element, and so on.

In order to illustrate, consider the following example:

```
#!/usr/bin/perl

# define a subroutine
sub add_two_numbers
{
$sum = $_[0] + $_[1];
return $sum;
}

$total = &add_two_numbers(3,5);
print "The sum of the numbers is $total\n";
```

In the example above, once the &add_two_numbers subroutine is invoked with the numbers 3 and 5, the numbers are transferred to the @_ variable, and are then accessed using standard scalar notation within the subroutine. Once the addition has been performed, the result is returned to the main program, and displayed on the screen via the print() statement.

The Right Path

A number of other special array variables also exist, in addition to @ARGV. One of the more commonly−used ones is the @INC variable, which sets up the "include paths" that Perl will look in when it encounters a call to require() or use(). This is analogous to the UNIX $PATH variable, which sets up default search paths for system binaries.

Let's take a look at this variable with the Data::Dumper module, used to "stringify" Perl data structure.

```
#!/usr/bin/perl

# use data dumper
use Data::Dumper;
```

```
# examine data structure
print Dumper @INC;
```

Here's what it looks like:

$VAR1 = '/usr/lib/perl5/5.8.0/i386−linux−thread−multi';
$VAR2 = '/usr/lib/perl5/5.8.0';
$VAR3 = '/usr/lib/perl5/site_perl/5.8.0/i386−linux−thread−multi';
$VAR4 = '/usr/lib/perl5/site_perl/5.8.0';
$VAR5 = '/usr/lib/perl5/site_perl';
$VAR6 = '/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi';
$VAR7 = '/usr/lib/perl5/vendor_perl/5.8.0';
$VAR8 = '/usr/lib/perl5/vendor_perl';
$VAR9 = '.';

In case you need to add a new search path to this, it's pretty simple − take a look:

```
#!/usr/bin/perl

# use data dumper
use Data::Dumper;

# add new path to include
push(@INC, "/usr/local/myapp/includes/");

# examine data structure
print Dumper @INC;
```

Here's the result:

$VAR1 = '/usr/lib/perl5/5.8.0/i386−linux−thread−multi';
$VAR2 = '/usr/lib/perl5/5.8.0';
$VAR3 = '/usr/lib/perl5/site_perl/5.8.0/i386−linux−thread−multi';
$VAR4 = '/usr/lib/perl5/site_perl/5.8.0';
$VAR5 = '/usr/lib/perl5/site_perl';
$VAR6 = '/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi';
$VAR7 = '/usr/lib/perl5/vendor_perl/5.8.0';
$VAR8 = '/usr/lib/perl5/vendor_perl';
$VAR9 = '.';
$VAR10 = '/usr/local/myapp/includes';

There's also a %INC hash, which lists all the files which have been included by the current script, together with their paths.

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

```
# use data dumper
use Data::Dumper;

# examine data structure
print Dumper [\%INC];
```

Here's the output:

```
$VAR1 = [
{
'warnings.pm' => '/usr/lib/perl5/5.8.0/warnings.pm',
'warnings/register.pm' =>
'/usr/lib/perl5/5.8.0/warnings/register.pm',
'bytes.pm' => '/usr/lib/perl5/5.8.0/bytes.pm',
'Carp.pm' => '/usr/lib/perl5/5.8.0/Carp.pm',
'XSLoader.pm' =>
'/usr/lib/perl5/5.8.0/i386−linux−thread−multi/XSLoader.pm',
'overload.pm' => '/usr/lib/perl5/5.8.0/overload.pm',
'Exporter.pm' => '/usr/lib/perl5/5.8.0/Exporter.pm',
'Data/Dumper.pm' =>
'/usr/lib/perl5/5.8.0/i386−linux−thread−multi/Data/Dumper.pm'
}
];
```

The difference between @INC and %INC is subtle but important − the former specifies the list of paths to search for files, while the latter specifies the files which have already been included in the current script, together with their paths.

There's also the %ENV hash, which contains a list of available environment variables.

```
#!/usr/bin/perl

# use data dumper
use Data::Dumper;

# examine data structure
print Dumper [\%ENV];
```

Take a look (this is an abridged output sample):

```
$VAR1 = [
{
'HOME' => '/home/me',
'SSH_CLIENT' => '192.168.0.241 1099 22',
'LESSOPEN' => '|/usr/bin/lesspipe.sh %s',
'MAIL' => '/var/spool/mail/me',
```

'PWD' => '/home/me',
'LANG' => 'en_US',
'USER' => 'me',
'G_BROKEN_FILENAMES' => '1',
'TERM' => 'xterm',
'SSH_TTY' => '/dev/pts/5'
}
];

Since %ENV is a hash, it's fairly easy to alter an environment setting − all you have to do is specify a new value for the corresponding hash key. Consider the following example, which shows you how:

```
#!/usr/bin/perl

# get value
print "Terminal is $ENV{'TERM'}\n";;

# change value
$ENV{'TERM'} = 'vt100';

# get new value
print "Terminal is now $ENV{'TERM'}\n";;
```

Here's the output:

Terminal is xterm
Terminal is now vt100

To Err Is Human

When it comes to dealing with errors, there are a number of variables you should know about. The first of them is the $? variable, which stores the error code returned by a call to an external binary, or to the system() function.

```
#!/usr/bin/perl

# add a user who already exists
`/usr/sbin/useradd root 2>/dev/null`;

# if error code, return it
if ($?)
{
print "Error code ", $? >> 8;
}
```

Here's the output:

Error code 9

In case you're wondering about the bitwise operation in the program above − the value stored in the $? variable is a 16−bit integer, of which the first 8 bits represent the error code returned by the invoked command.

You can also use the

```
$? & 127
```

operation to obtain information on the termination signal of the command, and

```
$? & 128
```

operation to get a Boolean value indicating whether or not the program dumped core.

As you may (or may not) know, Perl also allows you to trap errors in a syntax similar to Java's try−catch() blocks, by enclosing your code in an eval() block. In case the code within the eval() block produces an error, Perl stores the error in the $@ system variable without escalating it to the main program, from whence it may be retrieved for exception−handling purposes. The following example illustrates:

```
#!/usr/bin/perl

# attempt to use a file which does not exist
eval( "use Timezone;" );

# check for error
if ($@ ne "")
{
print "The following error occurred: ", $@;
}
```

In this case, since the call to use() is within an eval() block, the error returned when Perl is unable to locate the Timezone package will be trapped by the special $@ variable and will not be escalated upwards to the main program. You can then write your own exception−handling routine to inspect $@ and resolve the error appropriately.

Here's the output, with the $@ error−trapping above in action:

The following error occurred: Can't locate Timezone.pm in @INC (@INC contains: /usr/lib/perl5/5.8.0/i386−linux−thread−multi .) at (eval 1) line 1. BEGIN failed−−compilation aborted at (eval 1) line 1.

You can also catch error messages returned by die() within an eval() block with the $@ variable − as illustrated below:

```
#!/usr/bin/perl

# open file
eval( "open(FILE, '/tmp/dummy.txt') or die ('Could not open file');");

# check for error
if ($@ ne "")
{
print "The following error occurred: ", $@;
}
```

Here's the output:

The following error occurred: Could not open file at (eval 1) line 1.

In the case of Perl functions that use C library calls, you can also access the error returned by the underlying C library with the special $! variable. In order to illustrate, consider the Perl open() function, which uses the C open() call, in a variant of the example above:

```
#!/usr/bin/perl

# open file
eval( "open(FILE, '/tmp/dummy.txt') or die ('Could not open file');");

# check for error
print "The following error occurred: $!";
```

Note how, in this case, the error message displayed is the one returned by the C library, not Perl (compare it with the previous example to see the difference):

The following error occurred: No such file or directory

A Question Of Ownership

You can obtain information on the user and group the Perl script is running as, with the following four variables:

$< − the real UID of the process

$> − the effective UID of the process

$) − the real GID of the process

$(− the effective GID of the process

A difference between "real" and "effective" IDs appears when you use the setuid() or setgid() command to change the user or group. The "real" ID is always the one prior to the setuid() or setgid() call; the "effective" one is the one you've changed into following the call.

Consider the following example, which demonstrates:

```
#!/usr/bin/perl

# print real UID
print "Real UID: $<\n";

# print real GID
print "Real GID: $(\n";

# print effective UID
print "Effective UID: $>\n";

# print effective GID
print "Effective GID: $)\n";
```

Here's the output:

Real UID: 515
Real GID: 100 514 501 100
Effective UID: 515
Effective GID: 100 514 501 100

Notice that the $) and $(commands return a list of all the groups the user belongs to, not just the primary group.

Of course, most often this is not very useful by itself. What you really need is a way to map the numeric IDs into actual user and group names. And Perl comes with built−in functions to do this − consider the following example, which illustrates:

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

```
# set record separator
$\=" ";

# print user and group
print "This script is running as " . getpwuid($>) . " who belongs to the
following groups:";

foreach (split(" ", $))) { print scalar(getgrgid($_)); };
```

Here's the output:

This script is running as john who belongs to the following groups: users software apps

Rank And File

When reading data from files, Perl allows you to obtain the name of the file with the $ARGV variable, and the line number with the $. variable. This makes it easy to obtain information on which file (and which line of the file) is under the gun at any given moment. Consider the following example, which demonstrates:

```
#!/usr/bin/perl

# read a file and print its contents
while (<>)
{
# for each line, print file name, line number and contents
print $ARGV, ",", $., ": ";
print;
}
```

Here's an example of the output:

multiply.pl,1: #!/usr/bin/perl
multiply.pl,2:
multiply.pl,3: # multiply two numbers
multiply.pl,4: sub multiply()
multiply.pl,5: {
multiply.pl,6: my ($a, $b) = @_;
*multiply.pl,7: return $a * $b;*
multiply.pl,8: }
multiply.pl,9:
multiply.pl,10: # set range for multiplication table
multiply.pl,11: @values = (1..10);
multiply.pl,12:
multiply.pl,13: # get number from command−line

multiply.pl,14: foreach $value (@values)
multiply.pl,15: {
multiply.pl,16: print "$ARGV[0] x $value = " . &multiply($ARGV[0], $value) . "\n";
multiply.pl,17: }

Note, however, that the line number returned by $. does not automatically reset itself when used with multiple files, but rather keeps incrementing. In order to have the variable reset to 0 every time a new file is opened, you need to explicitly close() the previous file before opening a new one.

The variable returns the file name of the currently−running Perl script, as illustrated below:

```
#!/usr/bin/perl

# print filename
print "My name is ";
```

Here's the output:

My name is /tmp/temperature.pl

The $$ variable returns the process ID of the currently−running Perl process, as below:

```
#!/usr/bin/perl

# print process ID
print "This script is owned by process ID $$. Collect them all!";
```

Here's the output:

This script is owned by process ID 2209. Collect them all!

Finally, the special variable $] always contains information on the Perl version you are currently running. So, for example, the program

```
#!/usr/bin/perl

# print Perl version
print "Running Perl $]";
```

might return something like this:

This, coupled with the and $$ variables explained earlier, can come in handy when debugging misbehaving Perl programs,

```
#!/usr/bin/perl

# check for error
if ($error == 1)
{
# write to error log with script name, PID and Perl version
open (FILE, ">>/tmp/error.log");
print FILE "Error in (perl $] running as PID $$\n";
close (FILE);
}
```

or even to perform version checks to ensure that your code only works with a specific Perl version.

```
#!/usr/bin/perl

# check version
# display appropriate message
if ($] < 5.0)
{
die("You need a more recent version of Perl to run this program");
}
else
{
print "This is Perl 5 or better";
}
```

Calling For A Translator

Now, while the cryptic variable names discussed over the preceding pages are frequently−used by experienced Perl developers. novice users might find them a little disconcerting (at least in the beginning). That's why Perl also provides for alternative syntax, in the form of longer, more−readable English−language equivalents for the variables discussed previously.

In order to use the more−readable human−language names, simply add the line

```
use English;
```

to the top of your Perl script.

You should now be able to use Perl's longer names for the special variables discussed in this tutorial, thereby adding greater readability to your script. Here's a list mapping the special variables discussed above to their longer names:

$_ = $ARG

$/ = $INPUT_RECORD_SEPARATOR

$\ = $OUTPUT_RECORD_SEPARATOR

$, = $OUTPUT_FIELD_SEPARATOR

$? = $CHILD_ERROR

$@ = $EVAL_ERROR

$! = $OS_ERROR

$< = $REAL_USER_ID

$> = $EFFECTIVE_USER_ID

$(= $REAL_GROUP_ID

$) = $EFFECTIVE_GROUP_ID

$. = $INPUT_LINE_NUMBER

= $PROGRAM_NAME

$$ = $PROCESS_ID

$] = $PERL_VERSION

Consider the following variant of a previous example, which demonstrates how they may be used:

```
#!/usr/bin/perl

use English;

# set record separator
$OUTPUT_RECORD_SEPARATOR=" ";

# print user and group
print "This script is running as " . getpwuid($EUID) . " who belongs to the
following groups:";
```

```
foreach (split(" ", $))) { print scalar(getgrgid($ARG)); };
```

End Zone

And that's about it for this tutorial. Over the preceding few pages, I introduced you to the following special variables in Perl:

$_ (the default variable)

$/ (the input record separator)

$\ (the output record separator)

$, (the output field separator)

@ARGV (the command−line argument array)

@_ (the subroutine argument array)

@INC (the include path array)

%ENV (the environment variable array)

$? and $! (the last error code)

$@ (the last error in an eval() block)

$<, $>, $) and $((the real and effective UID/GIDs)

$. (the line number of an input file)

$ARGV (the name of an input file)

(the name of the current script)

$$ (the process ID of the current script)

$[(the Perl version number)

Of course, these are just some of the more commonly−used creatures you'll encounter in your travels through the Perl universe. Perl has a whole bunch more of these special variables which have not been discussed here − you can find them all in the "perlvar" manual page. Take a look for yourself, and until we meet again...stay healthy!

Note: Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!