Perl: Part 5

Thinking in Line Noise

In this month' article we examine how to create user defined functions, test and apply the finished functions in several separate scripts by creating libraries. BY FRANK FISH

Walter Nowak, visipix.com

User defined functions are an invaluable development tool enabling sections of code to be reused many times. Shrewd use of user functions can create generic functions that reduce repetition of code whilst increasing legibility and maintainability.

Functions

A function is a collection of statements that can be grouped together to perform a single task. The function can contain numerous calls to other perl functions or user defined functions, including itself. This allows functions to act as black-boxes that can be used without the knowledge of how it operates.

We declare a function by specifying its name and listing the statements as we would in a normal Perl script. In the example below a function being declared:

```
# Example 1
sub log_error {
  print STDERR "Oops!\n";
}
```

This example will write the message "Oops!" to wherever the standard error file handle has been pointed. The keyword 'sub' denotes that a function is about to be declared, next comes the name of the function, directly after is the code block which is enclosed in the curly braces. We will see later that there are several other arguments that can be passed. This form will now be sufficient to make a practical error logging function.

```
# Example 2
sub log_error {
  my @call = caller(1);
  print STDERR "Error: line ➥
$call [2]" . " of file $call ➥
[4]" . " in the function ➥
$call[3]\n";
}
```

In example 2 we use the Perl function 'caller' to give information on the calling subroutine where the error occurred. Caller returns an array of data pertaining to where it was called. The most common elements are listed below:

1. package
2. file-name
3. line number
4. subroutine / function

Items 2-4 should be familiar to you by now, packages will be discussed in the future. More information on the 'caller' function can now be found by using the 'perldoc -f caller' command from the shell prompt.

Invoking a user-defined function

As with most (possibly all) things in Perl: "There is more than one way of doing it", it follows then that there are numerous ways of calling a user-defined function.

The three most common methods are listed below. Each of these methods of calling the user defined functions has its own implicit characteristics.

```
# Example 3
log_error if $error == 1;
```

Example 3 calls the function 'log_error', provided that the function has been declared beforehand.

```
#Example 4
my $error = 1;
log_error() if $error;
```

Example 4 calls the function, regardless of where in the script the function was declared. The parenthesis indicate that the preceding word is a function. The parenthesis are used to pass values to the function, this is covered later.

```
# Example 5
&log_error if $error;
```

Example 5 calls the function, regardless of whether it was defined before the code section or not, using & is similar to using '$', '@' or '%', it clearly identifies the label as a function. However there are side-effects to using &, discussed later in this article.

Parameters

Parameters make user-defined functions very powerful and flexible. Arguments can be passed to user-defined functions in the same fashion as the predefined functions of Perl.

Values are copied to a function using parenthesis and the contents of the parenthesis are passed using the default array '@_'. This is the same variable that can be used throughout the program, however the value is stored elsewhere for the duration of the function and its value returned at the end of the script. This concept is called scope and is explained in more detail later.

Using parameters to provide values to a function enables the function to exist as a stand alone piece of code:

```
# Example 6
my $error_message;
my $file = '/etc/passwd';

sub log_error {
  my @call = caller(1);
  print STDERR "Error at line ➥
  $call[2] of file $call[1]" . ➥
   "in the function $call[4]\n";

  print STDERR $error_message if
    defined($error_message);
}

if (-e $file) {
  $error_message = "$file is ➥
  executable";
  log_error;
}
```

It seems comical to use this method, what would happen if you forgot to reset '$error_message', you'd give the wrong error message which would be extremely misleading putting you in the position of debugging your debug code. Modifying the previous example, we can give details as to the cause of an error as parameters to the argument:

```
# Example 7
sub log_error {
  my $error_message = shift;
  my @call = caller(1);
  print STDERR "Error at line ➥
  $call[2] of file $call[1]" .
   "in the function $call[4]\n";

  print STDERR ➥
  "$error_message\n" if
    defined($error_message);
}

my $file = '/dev/thermic_lance';

unless (-e $file) {
  log_error("$file doesn't ➥
  exist");
}
```

If you were particularly lazy you could then create the function to check for the existence of a file:

```
# Example 8
sub exist_file {
  my $file = shift;
  unless (-e $file) {
    log_error("$file doesn't ➥
    exist");
  }
  return 0;
}
```

This function in example 8 will call another user defined function, the code for which is shown previously. The code will now give a standardized explanation of the error that occurred, in a standard format using another user function to perform part of its task. The concept of splitting work among several user defined functions is called abstraction and has many benefits. An obvious one is that if you wanted to add a time-stamp then you would only need to add the time stamp code once and all existing calls to 'log_error' would reap the benefits.

Default Parameters

A function does not mind how many parameters are passed to it by default. As with standard arrays, if you try to access an element that has no value, the value returned will be 'undef'. It is possible to make Perl strictly adhere to set function arguments as we will see.

```
# Example 9
sub error_log {
  my $error_message = shift || ➥
  'no message provided';
  my @call = caller(1);
  print STDERR "Error at line ➥
  $call[2] of file $call[1]" . ➥
   "in the function $call[4]\n";
  print STDERR ➥
  "$error_message\n";
}
```

In example 9 if a parameter is not passed, then the default value reads 'no message provided', so the area returned could be:

```
Error at line 20 of file ➥
script.pl in the function ➥
test no message provided
```

The '||' operator is usually seen in conditional operators but in Perl it's equally at

home in ordinary lines of code. It has a low order of precedence.

Many Happy returns

All functions return a value. The value that a function returns can be the results of an operation or a status flag to show success or failure of an operation.

The following example shows the results of an operation:

```
# Example 10

sub circ_area {
  my $radius = shift || 0;
  my $PI = 2.14;
  my $area = $PI * $radius * ➥
  $radius;
  return $area;
}

my $radius = 3;
my $area = circ_area($radius);
print "Area of a circle ➥
$radius in radius is $area\n";
```

Will be interpreted as "Set $radius to the value of the next element of the array @_, if there are no more values set the value to 'no message provided'.

The results of the user defined function are returned directly, the essence of the function is to return the data.

In large systems a function return value is used to convey success or failure of the function, this is extremely useful in tasks that use many sections.

```
# Example 11

sub get_index($$) {
  my ($line, $index) = @_;
  my $status = 0;

  if ($line =~ /^(\w+)!/ && $1 ➥
  ne '' ){
    $$index = $1;
    $status = 1;
  }
  else {
  log_error("No index on line: ➥
  $_\n");
  }

  return $status;
}

my @lines = (
  'fred!fred bloggs, 12 ➥
  Fictional Place, Some Town',
  'jdoe!john doe,Flat 184A 23rd➥
  Street, Some City',
  '!bad line', 'another bad ➥
  line.'
);

for (@lines) {
  my $index = '';

  # pass the line and a ➥
  reference to index.
  my $status = get_index($_,➥
  \$index);
  print "The line '$_' has an ➥
  index $index\n" if $status ➥
  == 1;
}
```

Example 11 will find the indexes for an array of items and return a status for each line, this status can then be used to decide if it is possible to continue with the process.

It is worth noting that by default Perl functions will return the value from the last statement in a function. It is not uncommon to see subroutines that don't have 'return …' as the last line but rather a variable, function or value by itself just before the function declaration ends: while it is only necessary to use a return value to explicitly leave a subroutine early it is good form to explicitly give a 'return' statement.

```
# Example 12

sub get_index($$) {
  my ($line, $index) = @_; ➥
  my $status = 0;

  if ($line =~ /^(\w+)!/ && $1 ➥
  ne '' ){
    $$index = $1;
        $status = 1;
  } else {
  log_error("No index.on line: ➥
  $_\n");
  }
  $status;
}
```

Something greatly frowned upon in some programming disciplines is having more than one exit point to a function. Since Perl acts as both a programming as well as a scripting language the popular interpretation of this rule is to bend it and use the scripting ethos of "Exit early".

Example 13, be;ow, is the "Exit Early" programming style.

```
# Example 13
sub circ_area {
  my $radius = shift or return ➥
  0;
  my $PI = 2.14;
  my $area = $PI * $radius * ➥
  $radius;
  return $area;
}
```

It is a foregone conclusion that a radius of zero will produce an area of zero, so rather than calculate this result as we did before, we return the result immediately. Since the rules of geometry are unlikely to change in the working life of this code (and perhaps even before Perl 6 is released) such an action can hardly be seen as cavalier.

We can return half-way through an assignment due to two key features of Perl. The 'or' operator has a greater precedence than the assignment operator, and more importantly Perl is a tolerant, stoic and syntactically gifted language.

Scope

In example 6, we called a function and it accessed a variable declared outside of the function

We assigned a particular message to the variable '$error_message' and this was used in the function 'log_error'.

The variable we used is called a global variable, that is to say its name can be used anywhere and its value can be read or written to from anywhere within the program. The very fact that globals can be altered from anywhere is the biggest argument against using them, they lead to messy un-maintainable code and are considered a bad thing.

```
#Example 15 (Does not compile)
#!/usr/bin/perl
use strict;
use warnings;

my $global = 3;

sub functionX {
  my $private = 4;

  print "global is $global\n";
  print "private is $private\n";
}

functionX;

print "global is $global\n";

# This line will fail as
# $private doesn't exist
# outside of the function
# called functionX.
print "private is $private\n";
```

Use of global variables is best avoided, and should only be used to declare any constant values that will remain for the duration of the program. Better, even then, to make use of the fact that functions are always global, so no one can revise the code and knock out a global variable:

```
sub PI(){ 3.14 }
```

Even using functions to make constants it is wise to pass the values as parameters into the function, in case the function is placed directly into another script, where the constant has not been passed. Any function that can stand alone, can be unit tested, and its functionality vouched for.

My, Our and Local

There are three different ways to declare a variable in Perl, each affecting different aspects of the scope. As a rule 'my' is always used, failure to use 'our' or 'local' in the correct manner is considered an unforgivable sin.

'my' is the safest variety to use, this creates a variable and destroys it when it is no longer referred to, using the Perl garbage collection.

Any variable declared using 'my' within a scope (a looping structure or code block) exists only when that scope is called and its value is then reset after the iteration.

```
{
  my $v = 1; #  $v is 1;
  print "$v\n" #  $v is 1;
}  # $v no longer exists.
```

Online References

M-J. Dominus' excellent website:
perl.plover.com/FAQs/Namespaces.html
perl.plover.com/local.html

This can be especially useful in nested loops where the inner variable is each time automatically initialized.

```
for my $x ( 0..9 ) {
  my $y = 0;
  print "Coordinate ➥
  ( $x, $y )\n" while $y++ < 3;
}
```

'local' hi-jacks the value of a global variable, for the duration of its scope. This occurs at runtime rather than at compile time and is referred to as dynamic scope.

```
my $v = 5; {
  local $v = 1; #  $v is 1;
  print "$v\n" #  $v is 1;
}  # $v is 5 again.
```

As a rule 'local' should be avoided in preference to 'my'. It is primarily used to alter global variables for short spaces of time. Even then it is worth noting that any function calls made within this scope will also use the locally set value. If in doubt consult 'perldoc -f local' but remember 'my' is almost always what you want.

'our' allows the use of a variable within the lexical scope without initializing the value. 'our' becomes useful when we make packages, which we will investigate in the future.

Function Oddities

It is possible to establish a required set of values that the function must receive or make it fail to compile and exit with a runtime exception. This can be desirable in some cases and allows greater freedom in our use of user-defined functions.

We can declare the prototypes at the start of the code and then define the code block later in the program, in case we wish to use the extra features of prototyping to increase legibility of the code or force certain uses.

```
sub foo; # Forward declaration.➥
   sub foo(); # Prototype.
```

Using '&' does allow a programmer to overrule any prototyping on a function. A full description of prototyping with its strengths and weaknesses (Perl is after all a weakly typed language) will appear in the future.

```
sub debugm($) {
  print STDERR "\n\n****\n$_➥
  [0]\n***\n\n";
}

# Automatically uses $_
debugm;

# This only prints the first
# parameter but ignores the
# function prototype.
&debugm('beware','of this');
```

VARIABLES IN PERL

keyword	value	name
my	scoped	scoped
local	scoped	global
our	global	scoped

Further Reading

Perl docs: perldoc perlfunc

Perl Documentation

Perl has a wealth of good documentation coming with the standard distribution, It covers every aspect of the Perl language and is viewed using your computer system's default pager program. The pages of Perldoc resemble the man pages in that they cite examples of use and give pertinant advice.

There are a great many parts to the Perl documentation. To list the categories available, type the following command at the shell prompt:

```
perldoc perl
```

This then displays a page which has two columns. The left hand column lists the mnemonic title while the right column shows a description of the topic:

```
Perlsyn    Perl syntax
Perldata   Perl data structures
Perlop     Perl operators and
           precedence
Perlsub    Perl subroutines
```

Simply type perldoc and the mnemonic for that subject on the command line:

```
perldoc perlsyn
```

This shows the Perl syntax information. ■