PERL - SUBROUTINES

A Perl subroutine or function is a group of statements that together performs a task. You can divide up your code into separate subroutines. How you divide up your code among different subroutines is up to you, but logically the division usually is so each function performs a specific task.

> *Perl uses the terms subroutine, method and function interchangeably.*

Define and Call a Subroutine

The general form of a subroutine definition in Perl programming language is as follows −

```
sub subroutine_name{
   body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows −

```
subroutine_name( list of arguments );
```

In versions of Perl before 5.0, the syntax for calling subroutines was slightly different as shown below. This still works in the newest versions of Perl, but it is not recommended since it bypasses the subroutine prototypes.

```
&subroutine_name( list of arguments );
```

Let's have a look into the following example, which defines a simple function and then call it. Because Perl compiles your program before executing it, it doesn't matter where you declare your subroutine.

```
#!/usr/bin/perl

# Function definition
sub Hello{
   print "Hello, World!\n";
}

# Function call
Hello();
```

When above program is executed, it produces the following result −

```
Hello, World!
```

Passing Arguments to a Subroutine

You can pass various arguments to a subroutine like you do in any other programming language and they can be acessed inside the function using the special array @_. Thus the first argument to the function is in $_[0], *thesecondisin*_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities. So we will use references *explainedinthenextchapter* to pass any array or hash.

Let's try the following example, which takes a list of numbers and then prints their average −

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

```
# Function definition
sub Average{
   # get total number of arguments passed.
   $n = scalar(@_);
   $sum = 0;

   foreach $item (@_){
      $sum += $item;
   }
   $average = $sum / $n;

   print "Average for the given numbers : $average\n";
}

# Function call
Average(10, 20, 30);
```

When above program is executed, it produces the following result −

```
Average for the given numbers : 20
```

Passing Lists to Subroutines

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_. If you have to pass a list along with other scalar arguments, then make list as the last argument as shown below −

```
#!/usr/bin/perl

# Function definition
sub PrintList{
   my @list = @_;
   print "Given list is @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);

# Function call with list parameter
PrintList($a, @b);
```

When above program is executed, it produces the following result −

```
Given list is 10 1 2 3 4
```

Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs. For example −

```
#!/usr/bin/perl

# Function definition
sub PrintHash{
   my (%hash) = @_;

   foreach my $key ( keys %hash ){
      my $value = $hash{$key};
      print "$key : $value\n";
   }
}
%hash = ('name' => 'Tom', 'age' => 19);

# Function call with hash parameter
```

```
PrintHash(%hash);
```

When above program is executed, it produces the following result −

```
name : Tom
age : 19
```

Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one array or hash normally causes them to lose their separate identities. So we will use references *explainedinthenextchapter* to return any array or hash from a function.

Let's try the following example, which takes a list of numbers and then returns their average −

```
#!/usr/bin/perl

# Function definition
sub Average{
   # get total number of arguments passed.
   $n = scalar(@_);
   $sum = 0;

   foreach $item (@_){
      $sum += $item;
   }
   $average = $sum / $n;

   return $average;
}

# Function call
$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

When above program is executed, it produces the following result −

```
Average for the given numbers : 20
```

Private Variables in a Subroutine

By default, all variables in Perl are global variables, which means they can be accessed from anywhere in the program. But you can create **private** variables called **lexical variables** at any time with the **my** operator.

The **my** operator confines a variable to a particular region of code in which it can be used and accessed. Outside that region, this variable cannot be used or accessed. This region is called its scope. A lexical scope is usually a block of code with a set of braces around it, such as those defining the body of the subroutine or those marking the code blocks of *if, while, for, foreach,* and *eval* statements.

Following is an example showing you how to define a single or multiple private variables using **my** operator −

```
sub somefunc {
   my $variable; # $variable is invisible outside somefunc()
   my ($another, @an_array, %a_hash); # declaring many variables at once
}
```

Let's check the following example to distinguish between global and private variables −

```
#!/usr/bin/perl

# Global variable
$string = "Hello, World!";

# Function definition
sub PrintHello{
   # Private variable for PrintHello function
   my $string;
   $string = "Hello, Perl!";
   print "Inside the function $string\n";
}
# Function call
PrintHello();
print "Outside the function $string\n";
```

When above program is executed, it produces the following result −

```
Inside the function Hello, Perl!
Outside the function Hello, World!
```

Temporary Values via local

The **local** is mostly used when the current value of a variable must be visible to called subroutines. A local just gives temporary values to global $meaning package$ variables. This is known as *dynamic scoping*. Lexical scoping is done with my, which works more like C's auto declarations.

If more than one variable or expression is given to local, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval.

Let's check the following example to distinguish between global and local variables −

```
#!/usr/bin/perl

# Global variable
$string = "Hello, World!";

sub PrintHello{
   # Private variable for PrintHello function
   local $string;
   $string = "Hello, Perl!";
   PrintMe();
   print "Inside the function PrintHello $string\n";
}
sub PrintMe{
   print "Inside the function PrintMe $string\n";
}

# Function call
PrintHello();
print "Outside the function $string\n";
```

When above program is executed, it produces the following result −

```
Inside the function PrintMe Hello, Perl!
Inside the function PrintHello Hello, Perl!
Outside the function Hello, World!
```

State Variables via state

There are another type of lexical variables, which are similar to private variables but they maintain their state and they do not get reinitialized upon multiple calls of the subroutines. These variables are defined using the **state** operator and available starting from Perl 5.9.4.

Let's check the following example to demonstrate the use of state variables −

```
#!/usr/bin/perl

use feature 'state';

sub PrintCount{
   state $count = 0; # initial value

   print "Value of counter is $count\n";
   $count++;
}

for (1..5){
   PrintCount();
}
```

When above program is executed, it produces the following result −

```
Value of counter is 0
Value of counter is 1
Value of counter is 2
Value of counter is 3
Value of counter is 4
```

Prior to Perl 5.10, you would have to write it like this −

```
#!/usr/bin/perl

{
   my $count = 0; # initial value

   sub PrintCount {
      print "Value of counter is $count\n";
      $count++;
   }
}

for (1..5){
   PrintCount();
}
```

Subroutine Call Context

The context of a subroutine or statement is defined as the type of return value that is expected. This allows you to use a single function that returns different values based on what the user is expecting to receive. For example, the following localtime returns a string when it is called in scalar context, but it returns a list when it is called in list context.

```
my $datestring = localtime( time );
```

In this example, the value of $timestr is now a string made up of the current date and time, for example, Thu Nov 30 15:21:33 2000. Conversely −

```
($sec,$min,$hour,$mday,$mon, $year,$wday,$yday,$isdst) = localtime(time);
```

Now the individual variables contain the corresponding values returned by localtime subroutine.