FUNCTIONS

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], the second is in $_[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 (explained in the next chapter) 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 (explained in the next chapter) 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
```

FILE HANDLING

The basics of handling files are simple: you associate a **filehandle** with an external entity (usually a file) and then use a variety of operators and functions within Perl to read and update the data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - **STDIN**, **STDOUT**, and **STDERR,** which represent standard input, standard output and standard error devices respectively.

Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any new or existing file in Perl.

```
open FILEHANDLE, EXPR
open FILEHANDLE

sysopen FILEHANDLE, FILENAME, MODE, PERMS
sysopen FILEHANDLE, FILENAME, MODE
```

Here FILEHANDLE is the file handle returned by the **open** function and EXPR is the expression having file name and mode of opening the file.

Open Function

Following is the syntax to open **file.txt** in read-only mode. Here less than < sign indicates that file has to be opend in read-only mode.

```
open(DATA, "<file.txt");
```

Here DATA is the file handle, which will be used to read the file. Here is the example, which will open a file and will print its content over the screen.

```
#!/usr/bin/perl

open(DATA, "<file.txt") or die "Couldn't open file file.txt, $!";
```

```
while(<DATA>) {
   print "$_";
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opend in the writing mode.

```
open(DATA, ">file.txt") or die "Couldn't open file file.txt, $!";
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it −

```
open(DATA, "+<file.txt"); or die "Couldn't open file file.txt, $!";
```

To truncate the file first −

```
open DATA, "+>file.txt" or die "Couldn't open file file.txt, $!";
```

You can open a file in the append mode. In this mode, writing point will be set to the end of the file.

```
open(DATA,">>file.txt") || die "Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it −

```
open(DATA,"+>>file.txt") || die "Couldn't open file file.txt, $!";
```

Following is the table, which gives the possible values of different modes

Sr.No.	Entities & Definition
1	**< or r** Read Only Access
2	**> or w** Creates, Writes, and Truncates
3	**>> or a** Writes, Appends, and Creates
4	**+< or r+** Reads and Writes

5	**+> or w+** Reads, Writes, Creates, and Truncates
6	**+>> or a+** Reads, Writes, Appends, and Creates

Sysopen Function

The **sysopen** function is similar to the main open function, except that it uses the system **open()** function, using the parameters supplied to it as the parameters for the system function −

For example, to open a file for updating, emulating the **+<filename** format from open −

```
sysopen(DATA, "file.txt", O_RDWR);
```

Or to truncate the file before updating −

```
sysopen(DATA, "file.txt", O_RDWR|O_TRUNC );
```

You can use O_CREAT to create a new file and O_WRONLY- to open file in write only mode and O_RDONLY - to open file in read only mode.

The **PERMS** argument specifies the file permissions for the file specified, if it has to be created. By default it takes **0x666**.

Following is the table, which gives the possible values of MODE.

Sr.No.	Entities & Definition
1	**O_RDWR** Read and Write
2	**O_RDONLY** Read Only
3	**O_WRONLY** Write Only
4	**O_CREAT** Create the file

5	**O_APPEND** Append the file
6	**O_TRUNC** Truncate the file
7	**O_EXCL** Stops if file already exists
8	**O_NONBLOCK** Non-Blocking usability

Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the **close** function. This flushes the filehandle's buffers and closes the system's file descriptor.

close FILEHANDLE
close

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns true only if it could successfully flush the buffers and close the file.

```
close(DATA) || die "Couldn't close file properly";
```

Reading and Writing Files

Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

The <FILEHANDL> Operator

The main method of reading the information from an open filehandle is the <FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle. For example −

```
#!/usr/bin/perl

print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array −

```
#!/usr/bin/perl

open(DATA,"<import.txt") or die "Can't open data";
@lines = <DATA>;
close(DATA);
```

getc Function

The getc function returns a single character from the specified FILEHANDLE, or STDIN if none is specified −

```
getc FILEHANDLE
getc
```

If there was an error, or the filehandle is at end of file, then undef is returned instead.

read Function

The read function reads a block of information from the buffered filehandle: This function is used to read binary data from the file.

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

print Function

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

```
print FILEHANDLE LIST
print LIST
print
```

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current output filehandle (STDOUT by default). For example −

```
print "Hello World!\n";
```

Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and generate another copy file file2.txt.

```
#!/usr/bin/perl

# Open file to read
open(DATA1, "<file1.txt");

# Open new file to write
open(DATA2, ">file2.txt");

# Copy data from one file to another.
while(<DATA1>) {
   print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```
#!/usr/bin/perl

rename ("/usr/test/file1.txt", "/usr/test/file2.txt" );
```

This function **renames** takes two arguments and it just renames the existing file.

Deleting an Existing File

Here is an example, which shows how to delete a file file1.txt using the **unlink** function.

```
#!/usr/bin/perl

unlink ("/usr/test/file1.txt");
```

Positioning inside a File

You can use to **tell** function to know the current position of a file and **seek** function to point a particular position inside the file.

tell Function

The first requirement is to find your position within a file, which you do using the tell function −

```
tell FILEHANDLE
tell
```

This returns the position of the file pointer, in bytes, within FILEHANDLE if specified, or the current default selected filehandle if none is specified.

seek Function

The seek function positions the file pointer to the specified number of bytes within a file −

```
seek FILEHANDLE, POSITION, WHENCE
```

The function uses the fseek system function, and you have the same ability to position relative to three different points: the start, the end, and the current position. You do this by specifying a value for WHENCE.

Zero sets the positioning relative to the start of the file. For example, the line sets the file pointer to the 256th byte in the file.

```
seek DATA, 256, 0;
```

File Information

You can test certain features very quickly within Perl using a series of test operators known collectively as -X tests. For example, to perform a quick test of the various permissions on a file, you might use a script like this −

```
#/usr/bin/perl

my $file = "/usr/test/file1.txt";
my (@description, $size);
if (-e $file) {
   push @description, 'binary' if (-B _);
   push @description, 'a socket' if (-S _);
   push @description, 'a text file' if (-T _);
   push @description, 'a block special file' if (-b _);
   push @description, 'a character special file' if (-c _);
   push @description, 'a directory' if (-d _);
   push @description, 'executable' if (-x _);
   push @description, (($size = -s _)) ? "$size bytes" : 'empty';
   print "$file is ", join(', ',@description),"\n";
}
```

Here is the list of features, which you can check for a file or directory −

Sr.No.	Operator & Definition
1	**-A** Script start time minus file last access time, in days.
2	**-B** Is it a binary file?
3	**-C** Script start time minus file last inode change time, in days.

3	**-M** Script start time minus file modification time, in days.
4	**-O** Is the file owned by the real user ID?
5	**-R** Is the file readable by the real user ID or real group?
6	**-S** Is the file a socket?
7	**-T** Is it a text file?
8	**-W** Is the file writable by the real user ID or real group?
9	**-X** Is the file executable by the real user ID or real group?
10	**-b** Is it a block special file?
11	**-c** Is it a character special file?
12	**-d** Is the file a directory?
13	**-e** Does the file exist?
14	

	-f Is it a plain file?
15	**-g** Does the file have the setgid bit set?
16	**-k** Does the file have the sticky bit set?
17	**-l** Is the file a symbolic link?
18	**-o** Is the file owned by the effective user ID?
19	**-p** Is the file a named pipe?
20	**-r** Is the file readable by the effective user or group ID?
21	**-s** Returns the size of the file, zero size = empty file.
22	**-t** Is the filehandle opened by a TTY (terminal)?
23	**-u** Does the file have the setuid bit set?
24	**-w** Is the file writable by the effective user or group ID?
25	**-x**

	Is the file executable by the effective user or group ID?
26	**-z** Is the file size zero?

DIRECTORY

Following are the standard functions used to play with directories.

```
opendir DIRHANDLE, EXPR  # To open a directory
readdir DIRHANDLE        # To read a directory
rewinddir DIRHANDLE      # Positioning pointer to the begining
telldir DIRHANDLE        # Returns current position of the dir
seekdir DIRHANDLE, POS   # Pointing pointer to POS inside dir
closedir DIRHANDLE       # Closing a directory.
```

Display all the Files

There are various ways to list down all the files available in a particular directory. First let's use the simple way to get and list down all the files using the **glob** operator −

```
#!/usr/bin/perl

# Display all the files in /tmp directory.
$dir = "/tmp/*";
my @files = glob( $dir );

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

# Display all the C source files in /tmp directory.
$dir = "/tmp/*.c";
@files = glob( $dir );

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

# Display all the hidden files.
$dir = "/tmp/.*";
@files = glob( $dir );
foreach (@files ) {
   print $_ . "\n";
```

```
}

# Display all the files from /tmp and /home directories.
$dir = "/tmp/* /home/*";
@files = glob( $dir );

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

Here is another example, which opens a directory and list out all the files available inside this directory.

```
#!/usr/bin/perl

opendir (DIR, '.') or die "Couldn't open directory, $!";
while ($file = readdir DIR) {
   print "$file\n";
}
closedir DIR;
```

One more example to print the list of C source files you might use is −

```
#!/usr/bin/perl

opendir(DIR, '.') or die "Couldn't open directory, $!";
foreach (sort grep(/^.*\.c$/,readdir(DIR))) {
   print "$_\n";
}
closedir DIR;
```

Create new Directory

You can use **mkdir** function to create a new directory. You will need to have the required permission to create a directory.

```
#!/usr/bin/perl

$dir = "/tmp/perl";

# This creates perl directory in /tmp directory.
mkdir( $dir ) or die "Couldn't create $dir directory, $!";
print "Directory created successfully\n";
```

Remove a directory

You can use **rmdir** function to remove a directory. You will need to have the required permission to remove a directory. Additionally this directory should be empty before you try to remove it.

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

```
$dir = "/tmp/perl";

# This removes perl directory from /tmp directory.
rmdir( $dir ) or die "Couldn't remove $dir directory, $!";
print "Directory removed successfully\n";
```

Change a Directory

You can use **chdir** function to change a directory and go to a new location. You will need to have the required permission to change a directory and go inside the new directory.

```
#!/usr/bin/perl

$dir = "/home";

# This changes perl directory  and moves you inside /home directory.
chdir( $dir ) or die "Couldn't go inside $dir directory, $!";
print "Your new location is $dir\n";
```